Roberto Baldoni
Paola Flocchini
Ravindran Binoy (Eds.)

LNCS 7702

Principles of Distributed Systems

16th International Conference, OPODIS 2012
Rome, Italy, December 2012
Proceedings

Springer

Lecture Notes in Computer Science 7702

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

Roberto Baldoni Paola Flocchini
Ravindran Binoy (Eds.)

Principles of Distributed Systems

16th International Conference, OPODIS 2012
Rome, Italy, December 18-20, 2012
Proceedings

Springer

Volume Editors

Roberto Baldoni
Università degli Studi di Roma “La Sapienza”
Dipartimento di Informatica, Automatica e Gestionale "Antonio Ruberti"
Via Ariosto 25, 00168 Rome, Italy
E-mail: baldoni@dis.uniroma1.it

Paola Flocchini
University of Ottawa
School of Electrical Engineering and Computer Science
800 King Edward Street, K1N 6N5, Ottawa, ON, Canada
E-mail: flocchin@site.uottawa.ca

Ravindran Binoy
Virginia Technical University
Electrical and Computing Engineering Department
302 Whittemore Street, Blacksburg, VA 24061, USA
E-mail: binoy@vt.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-35475-5 e-ISBN 978-3-642-35476-2
DOI 10.1007/978-3-642-35476-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012953317

CR Subject Classification (1998): C.2.4, C.2, F.2, D.2, I.2.11, G.2.2

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

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

Printed on acid-free paper

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

Preface

OPODIS, the International Conference on Principles of Distributed Systems, is an international forum for the exchange of state-of-the-art knowledge on distributed computing and systems among researchers from around the world. The 16th edition of OPODIS was held during December 18–20, 2012, in Rome, Italy.

Papers were sought soliciting original research contributions to the theory, specification, design, and implementation of distributed systems. In response to the call for papers, 89 submissions were received, out of which 24 papers were accepted, after a rigorous reviewing process that involved 31 Program Committee members and at least three reviews per paper.

We would like to thank the Program Committee members, as well as the external reviewers, for their fundamental contribution in selecting the best papers.

In addition to the technical papers, the program included three invited presentations by: Giuseppe Ateniese (Sapienza University of Rome), Pierre Fraigniaud (University of Paris 7), and Antony Rowstron (Microsoft Research Cambridge).

This event would not have been possible without the technical support of Adriano Cerocchi and the administrative support of Carola Aiello and Gabriella Caramagno. We would like to express our gratitude to our sponsors and particularly to Sapienza University of Rome, Over Technologies, and the Sapienza Research Center of Cyber Intelligence and Information Security.

December 2012

Roberto Baldoni
Paola Flocchini
Binoy Ravindran

Organization

Program Committee

Bjorn Andersson	Polytechnic Institute of Porto, Portugal
James Aspnes	Yale, USA
Hagit Attiya	Technion, Israel
Roberto Baldoni	University of Rome "La Sapienza", Italy
Xavier Defago	Japan Advanced Institute of Science and Technology (JAIST)
Carole Delporte-Gallet	University of Paris Diderot, France
Stefan Dobrev	Slovak Academy of Sciences, Bratislava, Slovakia
Shlomi Dolev	Ben-Gurion University of the Negev, Israel
Paola Flocchini	University of Ottawa, Canada
Hacene Fouchal	Université de Reims Champagne-Ardenne, France
Shelby Funk	University of Georgia, USA
Vijay Garg	University of Texas at Austin, USA
David Ilcinkas	LaBRI, CNRS & Université de Bordeaux, France
Boris Koldehofe	University of Stuttgart, Germany
Eric Koskinen	New York University, USA
Fabian Kuhn	University of Freiburg, Germany
Xu Li	University of Waterloo, Canada
Bernard Mans	Macquarie University, Australia
Alessia Milani	University of Bordeaux 1-IPB, France
Sotiris Nikoletseas	University of Patras and CTI, Greece and The Netherlands
Marina Papatriantafilou	Chalmers University of Technology, Sweden
Marta Patiño	Universidad Politecnica de Madrid, Spain
Giuseppe Prencipe	Università di Pisa, Italy
Leonardo Querzoni	Università degli Studi di Roma "La Sapienza", Italy
Binoy Ravindran	Virginia Tech, USA
Paulo Romano	INESC-ID, Portugal
Nicolas Schiper	Cornell University, USA
Michael Spear	Lehigh University, USA
Sebastien Tixeuil	Université Pierre et Marie Curie, France
Roman Vitenberg	University of Oslo, Norway
Jennifer Welch	Texas A&M University, USA

Wenyuan Xu	University of South Carolina, USA
Masafumi Yamashita	Kyushu University, Japan
Shmuel Zaks	Technion, Israel

Additional Reviewers

Abdurusul, Kudireti
Alglave, Jade
Ayaida, Marwane
Balasubramanian, Bharath
Barbalace, Antonio
Benzing, Andreas
Bernard, Thibault
Bonnet, François
Bonomi, Silvia
Bridgman, John
Callau-Zori, Mar
Carabelli, Ben
Casteigts, Arnaud
Chang, Yen-Jung
Chatterjee, Bapi
Chung, Hyun Chul
Cohen, Asaf
Devismes, Stephane
Drachsler, Dana
Fauconnier, Hugues
Fernandez Anta, Antonio
Fodor, Viktoria
Fouchal, Hacene
Fu, Zhang
Georgiadis, Giorgos
Golab, Wojciech
Gulisano, Vincenzo
Harris, Tim
He, Liang
Herlihy, Maurice
Hung, Wei-Lun
Johnen, Colette
Kamei, Sayaka
Kamiyama, Naoyuki
Kijima, Shuji
Klasing, Ralf
Koutsopoulos, Andreas
Kralovic, Rastislav
Larsson, Andreas
Le Merrer, Erwan
Li, Ximing
Liang, Xiaohui
Liu, Zhenhua
Mei, Yongguo
Michael, Maged
Michail, Othon
Nesterenko, Mikhail
Nikolakopoulos, Ioannis
Ottenwälder, Beate
Pagli, Linda
Peluso, Sebastiano
Platania, Marco
Potop-Butucaru, Maria
Provensi, Lucas
Raptopoulos, Christoforos
Ridge, Tom
Rinetsky, Noam
Samanta, Roopsha
Santoro, Nicola
Setty, Vinay
Sharma, Shantanu
Singh, Abhishek
Sohier, Devan
Sorriente, Claudio
Steffenel, Luiz Angelo
Tan, Yongmin
Tariq, M. Adnan
Travers, Corentin
Tripp, Omer
Tudor, Valentin
Viglietta, Giovanni
Viqar, Saira
Wade, Ahmed
Widder, Josef
Widmayer, Peter
Xu, Miao
Yamauchi, Yukiko

The Cloud Was Tipsy and Ate My Files!
(Invited Talk)

Giuseppe Ateniese

Department of Computer Science
Sapienza - University of Rome
ateniese@di.uniroma1.it

Cloud computing is shaping the future of computer science and it is affecting the way we perform business and operate daily. Our entire digital life is stored on remote storage servers such as Amazon S3, Microsoft Azure, Google, iCloud, etc. Our emails, pictures, calendars, documents, music/video playlists, and generic files are readily available, anytime and anywhere.

Not everyone, however, is ready to move to the Cloud. Businesses and organizations are still reluctant to outsource their databases for fear of losing control on their files or, worse, releasing sensitive information to third parties. While encryption can help, it is not yet clear how to operate efficiently on encrypted data stored remotely. In addition, encrypted data can still be intentionally lost or damaged.

In Cloud storage, the major stumbling block is that there is no local copy of data anymore. Thus, there is nothing in our hands that can be used to check against the version of our files stored remotely. This is somehow exacerbated by the fact that outsourced data could be very large and thus impossible to retrieve in its entirety. In this scenario: How can we be certain that Cloud providers are storing the entire database intact, even portions that are rarely accessed? Can we check the integrity of files without downloading them from the Cloud?

In this invited talk, we will provide answers to the questions above. We will introduce some novel cryptographic tools that allow users to check the integrity of their files in the Cloud while keeping local storage and bandwidth consumption *essentially* constant. These new cryptographic primitives are efficient and scalable and may help persuade skeptics to adopt full-fledged Cloud computing solutions.

Distributed Local Decision and Verification (Invited Talk)

Pierre Fraigniaud*

CNRS and University Paris Diderot, France

Abstract. Distributed *decision* refers to the task in which every process p_i, $i = 1, \ldots, n$, is given some input x_i, and the processes have to collectively decide whether $x = (x_1, \ldots, x_n)$ satisfies some prescribed property, i.e., belongs to some language $\mathcal{L}$. For instance, one may want to decide whether the x_is provide a proper coloring of the actual network, or one may want to decide whether the x_is are all identical, and equal to a proposed value. A typical application of distributed decision is actually distributed *checking*, in which the processes have to check whether the result of a computation performed by some black box is correct. In the above examples, the issue was checking proper coloring, and checking consensus.

Distributed *verification* refers to the task in which every process p_i is given some input x_i, together with a *certificate* y_i, and the processes have to collectively verify, with the help of the certificate $y = (y_1, \ldots, y_n)$, whether $x = (x_1, \ldots, x_n)$ belongs to some language $\mathcal{L}$, in the following sense: if $x \in \mathcal{L}$ then there must exist y such that the processes collectively accept x; and if $x \notin \mathcal{L}$ then for every y the processes must collectively reject x. A typical application of distributed verification is to certify the correctness of some data structure, e.g., x is a spanning tree of the actual network.

This talk will survey our recent results about distributed decision and distributed verification. It will mostly focus on the $\mathcal{LOCAL}$ model. In this latter context, one expects each node to take its decision after having inspected just a restricted neighborhood around itself in the network. If time permits, the talk will also provide a brief survey of recent results in other distributed models, including the $\mathcal{CONGEST}$ model, the *wait-free* model, and mobile agent computing.

* Additional support from ANR project DISPLEXITY, and INRIA project GANG.

Converged Data Centers
(Invited Talk)

Antony Rowstron

Microsoft Research, Cambridge, UK

Abstract. We have been exploring what happens when you take the best ideas from distributed systems, networking, high-performance computing (HPC) and recent advances in hardware and apply them to commodity data center clusters. The motivation is that as a distributed systems builder I have often had to build distributed systems that need to handle problems that are really simply consequences of design choices of the underlying hardware platform. When running distributed systems across the Internet it is hard to change the hardware platform, but when running inside a data center it is very feasible. This led us to start build a number of different clusters with very different properties from the clusters traditional used in data centers.

The talk will use two motivating examples to demonstrate the concepts. The first example is the based on the CamCube project which explores using different interconnects, inspired by the HPC world, to run distributed applications like Map Reduce. The second example is looking at how hardware trends should be changing the way we think of implementing some services in the data center. This should be driving us to close the gap between hardware and software, leading to converged data centers.

Table of Contents

FixMe: A Self-organizing Isolated Anomaly Detection Architecture for Large Scale Distributed Systems

Emmanuelle Anceaume[1], Erwan Le Merrer[2], Romaric Ludinard[3], Bruno Sericola[3], and Gilles Straub[2]

[1] IRISA / CNRS, France
firstname.name@irisa.fr
[2] Technicolor Rennes, France
firstname.name@technicolor.com
[3] Inria Rennes - Bretagne Atlantique, France
firstname.name@inria.fr

Abstract. Monitoring a system is the ability of collecting and analyzing relevant information provided by the monitored devices so as to be continuously aware of the system state. However, the ever growing complexity and scale of systems makes both real time monitoring and fault detection a quite tedious task. Thus the usually adopted option is to focus solely on a subset of information states, so as to provide coarse-grained indicators. As a consequence, detecting isolated failures or anomalies is a quite challenging issue. In this work, we propose to address this issue by pushing the monitoring task at the edge of the network. We present a peer-to-peer based architecture, which enables nodes to adaptively and efficiently self-organize according to their "health" indicators. By exploiting both temporal and spatial correlations that exist between a device and its vicinity, our approach guarantees that only isolated anomalies (an anomaly is isolated if it impacts solely a monitored device) are reported on the fly to the network operator. We show that the end-to-end detection process, *i.e.*, from the local detection to the management operator reporting, requires a logarithmic number of messages in the size of the network.

1 Introduction

The number of IP-enabled devices keeps on growing in a steady manner, often reaching millions of units managed by a single operator. If those devices are able to provide a service to the user in their intended running state, deviations in behavior or hardware/software problems are generally detected offline by human intervention. The technical barrier for efficient online monitoring and analysis is the size of the devices set to operate, together with the huge amount of parameters and states to consider. Network operators deploy helpdesk in order to support their customers when they are facing problems. In the last years the cable and telecom industry have developed different remote management standards [1] to better support the helpdesk operator via dedicated protocols and

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 1–15, 2012.

tools. As a consequence, the helpdesk operation represents an important part of the overall operating cost of a network provider. Reducing the number of calls as well as their duration is an important key for every network operator to sustain profitability and reduce the total cost of ownership. Nevertheless both telecom and cable industries came up with client-server architectures where a single server (or a farm of servers) is in charge of managing a set of devices. Such architectures are typically used for management tasks (*e.g.*, service provisioning, device firmware upgrading) rather than for real time monitoring activities, essentially because of scaling issues. Indeed, the massive scale we are considering calls for efficient monitoring algorithms. A first option is to gather all the devices logs in a single place, and to analyze collected data using for instance the MapReduce paradigm [2] to detect the causes of the anomalies. This nevertheless implies a significant detection latency and processing cost at the cloud architecture level.

The second option is to push monitoring procedures on devices. Actually, standardized procedures exist at devices level to autonomously trigger asynchronous alarms in presence of anomalies. However, these procedures are never used for practical reasons. Indeed if the cause of the anomaly lies in the network itself (*e.g.*, at routers, links or data center outages) this may impact a very large number of devices, and thus letting thousands of impacted devices reporting the problem to the helpdesk operator may quickly become a disaster due to the volume of generated messages. On the other hand, it is of utmost importance to minimize the overall network footprint by giving each device the capability to self distinguish network-based anomalies from *isolated* ones – anomalies that only impact the device itself – so that only isolated anomalies are reported on the fly to the helpdesk. This is the problem that we address in this paper. Specifically, we propose a novel distributed monitoring tool, called FixMe, that enjoys the following properties.

- FixMe is self-managing: all the monitored devices self-organize according to their "health" indicators so that they can detect any correlation between their state and the one of their neighbors,
- FixMe is dynamic: *(i)* monitored devices may join the system or may be removed from it at any time, and *(ii)* there is no assumption regarding the QoS repartition of the monitored nodes (*i.e.*, we do not assume that the repartition is uniform),
- FixMe does not rely on any complex bootstrap procedure. In contrast to most of the monitoring tools, devices do not need to be prearranged into a predefined number of clusters (as required for in instance in k-means based solutions),
- FixMe is scalable: the end-to-end detection process, *i.e.* from the local detection to the management operator reporting, requires a logarithmic number of messages.

The remaining of the paper is organized as follows. Section 2 provides an overview of existing monitoring approaches. Section 3 presents the model of the system, and defines the addressed problem. Section 4 describes the FixMe overlay and

its associated operations, while in Section 5 its efficiency is analyzed. Section 6 describes the algorithm that solves the addressed problem. Section 7 concludes and presents future works.

2 Related Work

This Section provides an overview of the existing techniques used in large scale systems to continuously and automatically monitor time-varying metrics. The authors in [3] exploit temporal and spatial correlations [4,5,6] among groups of monitored nodes to decrease monitoring communication costs, *i.e.*, the cost incurred by the periodic reporting of the updated metrics values from the monitored nodes to the management node. The idea is to prevent any reporting message from occurring when such a reporting would contain metrics values that could be directly inferred by the management node. This is achieved by giving each monitored node the capability to locally detect whether the current values of its monitored metrics are in accordance with predicted ones (through Kalman filters tools [7] installed at both monitored nodes and the management node), and by gathering nodes into clusters (such that, for each monitored metric, a set of clusters group together nodes that share correlated values of the considered metric according to the Pearson correlation coefficient). At clusters level, an elected leader is in charge of communicating with the management system when the current metric values of its group members differ from each others. Although close to our objectives, the main drawback of this solution lies on the centralized clustering process. All the nodes of the system are continuously organized into clusters computed through the k-means algorithm exclusively run by the management node, which is a clear impediment to the scalability of their approach. Other works aim at minimizing the processing cost for continuous monitoring [8,9,10] in the light of the theoretical results of [11], however similarly to [3], all these approaches suffer from a centralized handling of the clustering process.

In contrast, our objective is a fine-grain detection tool capable of accurately and efficiently detecting isolated events. As will be described in the remaining of the paper, we combine clustering and structured peer-to-peer architectures to tend toward this objective.

3 Model of the System

We consider a set of N nodes that communicate among each other through the standard synchronous message-passing model. Each node in the system is assigned a unique random identifier derived from a standard hash function (*e.g.* MD5, SHA-1). Each node has access to D services numbered $1, \ldots, D$. At any time t, the QoS of each service is locally measured with an end-to-end performance measurement function

$$\begin{aligned} Q_i : \{1, \ldots, N\} \times \mathbb{N} &\longrightarrow [a_i, b_i] \\ (p, t) &\longmapsto \text{Quality of service } i \text{ at node } p \text{ at time } t \end{aligned}$$

Without loss of generality we suppose that the QoS range $[a_i, b_i]$ of service i is equal to $[0, 1]$. We define the *position* of a node p at time t by the vector $Q(p, t)$ defined as

$$Q(p, t) = (Q_1(p, t), \ldots, Q_D(p, t)). \tag{1}$$

For each monitored service $i = 1, \ldots, D$, we split interval $[0, 1]$ into n_i disjoint intervals $[x_i^{(j-1)}, x_i^{(j)})$, $1 \leq j \leq n_i$, with $x_i^{(0)} = 0$ and $x_i^{(n_i)} = 1$, the last interval being closed. Integer n_i is a parameter of the system. These n_i intervals can be thought as n_i QoS classes of service i. For instance, one can consider a division of $[0, 1]$ for service i such that $|x_i^{(j)} - x_i^{(j-1)}| \geq |x_i^{(j+1)} - x_i^{(j)}|$. Such a division could be used to reflect the increasing sensitivity of users regarding QoS variations. A user is more sensitive to a very small variation of a high QoS than to a large variation of a low QoS. Without loss of generality, we suppose a regular division into identical length intervals and we define $\rho_i = |x_i^{(j)} - x_i^{(j-1)}| = 1/n_i$. In the following these intervals are named *buckets* (a more precise definition is given in Section 4).

In addition to the functions $Q_1, \ldots, Q_D$, each node has access to D anomaly detection functions $A_1, \ldots, A_D$. At each time t, each function A_i is fed with the sequence of the $\ell_i \geq 1$ last QoS values $Q_i(p, t - \ell_i + 1), \ldots, Q_i(p, t)$ and provides some meaningful prediction of what should be the next QoS value. Note that ℓ_i is a parameter of A_i. These functions are implemented to cope with the specific variations of their input values, and thus different kinds of anomaly detection functions exist, ranging from a simple threshold based functions, to more sophisticated ones like the Holt-Winters forecasting or Cusum method. In this paper, we suppose that the output of these anomaly detections are boolean. At time t, $A_i(p, t) = \texttt{true}$ if the sequence $Q_i(p, t - \ell_i + 1), \ldots, Q_i(p, t)$ is considered as an anomaly, it is `false` otherwise. Implementation of both Q_i and A_i functions are out of the scope of the paper.

Finally, suppose that a node locally detects an anomaly whose origin comes from a network/service dysfunction or failure. Then this anomaly will have an impact on the QoS of other nodes, and thus these nodes will locally detect it. On the other hand, we suppose that if a node locally detects an anomaly whose origin is local (hardware or software), then this anomaly will only impact its QoS, and thus no other nodes will be impacted by this specific anomaly.

Prior to defining the addressed problem, let us consider the following simple scenario presented in Fig. 1. The QoS of a single service monitored by two nodes a and b is represented by interval $[0, 1]$. At time t the quality positions $Q(a, t)$ and $Q(b, t)$ of both nodes lie in bucket j, while at time $t + 1$, at least one of the two nodes experience a QoS change. Five situations can be observed. In situations (1), (2) and (4) node a is the only node that observes a QoS change. In situation (1), this change does not push a position outside bucket j, while in situation (2) and (4) it does. However in both situations (1) and (2), the anomaly detection function $A_1(a, t+1) = \texttt{false}$, thus a does not consider this move as an anomaly, therefore does not do any more investigation. In the other hand, in situation (4), $A_1(a, t+1) = \texttt{true}$, and thus node a triggers a FixMe message. Now observe the two last situations (3) and (5). Both nodes observe a QoS change considered as an anomaly by their function A (*i.e.*, $A_1(a, t+1) = A_1(b, t+1) = \texttt{true}$). However

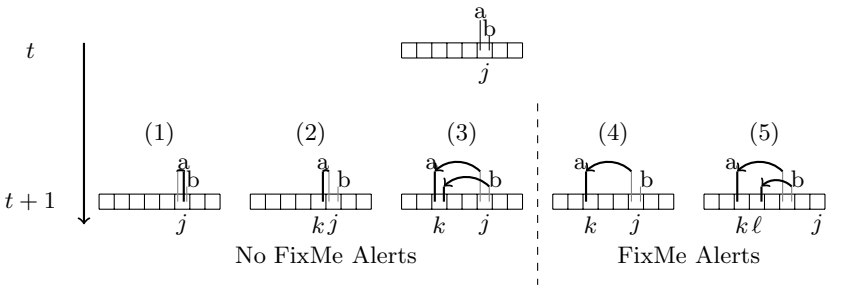


Fig. 1. Isolated anomaly detection of one monitored service. Node a triggers FixMe message in both cases (4) and (5), while node b triggers it only in case (5).

in situation (3) the QoS degradation is the same for both nodes ($Q(a, t+1)$ and $Q(b, t+1)$ lie in bucket k) and thus neither a nor b consider this anomaly as isolated, while in situation (5) $Q(a, t+1)$ and $Q(b, t+1)$ respectively lie in buckets k and ℓ. Thus both nodes trigger a FixMe message. We now formally define the problem we address in this work.

Definition 1 (The Isolated Anomaly Detection Problem). *Let $\mathcal{S} = \{1, \ldots, N\}$ be the set of monitoring nodes, and an additional node named the management operator with which any of the N nodes communicate. Let $\mathcal{S}^t_{j,k} \subseteq \mathcal{S}$ be such that $\forall p \in \mathcal{S}^t_{j,k}$, p has moved from bucket j to bucket k from time $t-1$ to time t and there exists a service i such that $A_i(p, t) =$ `true`. Then at time $t+1$, an alert is raised at the management operator if and only if $|\mathcal{S}^t_{j,k}| \leq \tau$, with τ a parameter of the system. In Fig. 1, $\tau = 1$.*

4 FixMe Framework

4.1 Rationale

In this Section, we describe how we address the Isolated Anomaly Detection problem in a distributed system composed of N monitored nodes. FixMe framework orchestrates the monitored nodes into an overlay network, named in the following FixMe overlay. An overlay network is actually a virtual network built on top of the physical network within which nodes communicate among each other along the edges of the overlay by using the communication primitives provided by the underlying network (*e.g.* IP network service). The algorithms nodes use to choose their neighbors and to route their messages define the overlay topology. The topology of unstructured overlays conforms with random graphs (*i.e.*, relationship among nodes are mostly set according to a random process which reveals to be inefficient to find a particular node or set of nodes in the overlay). On the other hand, structured overlays build their topology according to structured graphs (*e.g.*, tree, torus, hypercube). Most of the structured overlays are

based on Distributed Hash Tables (*e.g.*, [12,13]). The efficiency and scalability of all these proposed DHTs rely on the uniform distribution of the nodes in the identifiers space at the expense of breaking the application logic. This is why, for specific applications such as streaming applications, broadcast spanning trees structures, that support the application-level broadcast, have been proposed [14]. Our concern is to exploit the QoS relationship among monitored nodes, which make all the aforementioned solutions non adapted. As a consequence, we propose to organize nodes so that at any time t the neighbors of any node p are the nodes q whose QoS (*i.e.* $Q(q,t)$) are closer to the QoS of p (*i.e.* $Q(p,t)$). The description of such an organization is done in Section 4.2. From the application point of view, three operations are provided by the system: the `lookup`, the `join`, and `leave` operations that allow nodes to respectively find a position in the overlay, join the overlay or `leave` it. From the topological structure point of view, two operations are provided: the `split` and `merge` operations that guarantee the scalability of FixMe overlay when some regions of the overlay become too dense or too sparse. All these operations are described in Section 4.3. Finally, when too many monitored nodes share exactly the same QoS (or equivalently sit at the same position in the overlay), nodes within the bucket self-organize into an hypercube as described in Section 4.4.

4.2 Overview of FixMe Overlay

The FixMe overlay is a virtual multi-dimensional cartesian coordinate space on a multi-torus. The entire coordinate space is tessellated into a collection of *buckets*. A bucket is the cartesian product of D intervals of respective length $\rho_1, \ldots, \rho_D$ (*cf.* Fig. 2, where FixMe overlay is made of 16^2 buckets). When a node p `joins` FixMe at time t, p joins the bucket which corresponds to its quality position (or simply its position) $Q(p,t)$. When a bucket is populated by more than $S_{\min}$ nodes this bucket is called a *seed*. The entire coordinate space is dynamically partitioned into distinct zones, named *cells*, such that a cell contains at most one seed (*cf.* Fig. 2, where FixMe overlay on the left is made of four cells, and the one on the right is made of five cells). More formally,

Definition 2 (Cell). *A cell is defined as an hyper-rectangle of buckets, among which at most one is a seed. A cell is fully and uniquely characterized by a set of 2^D buckets called the corners of the cell, sorted using the lexicographic order.*

Figure 2 shows these different elements for $D = 2$ and $\rho_1 = \rho_2 = 1/16$. The buckets are elementary squares, the seeds are represented by the black squares, and the cells are depicted by the coloured rectangles. Note that neither cell 4 (on the figure on the left) nor cell 5 (on the figure on the right) have a seed. The reasons will be detailed in the following.

4.3 FixMe Operations

`Lookup` operation. We describe how a node locates the seed that is in charge of a given bucket b through the `lookup` operation. In FixMe, routing is exclusively

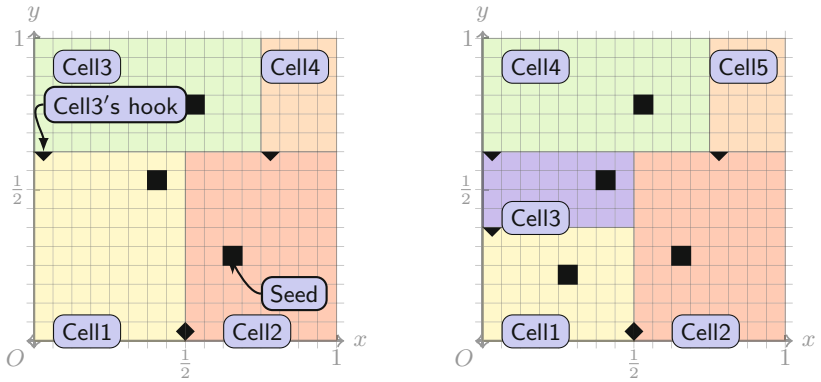


Fig. 2. FixMe overlay before (on the left) and after (on the right) a `split` operation

handled by seeds. Each seed maintains a *routing table* that contains an entry for each of its $2D$ neighboring seeds in the coordinate space. An entry contains the IP address and the virtual coordinate of the seed. A `lookup` message contains the destination coordinates. Using the neighbor coordinate, a seed routes a `lookup` message toward its destination using a simple greedy forwarding to the neighbor seed that is closest to the destination address. CAN [12] uses this routing to cross its zones. However, as such the lookup operation needs to cross in average $\mathcal{O}(DN^{1/D})$ zones. We combine the multidimensional routing of CAN with Chord-fashioned long-range neighbors [13,15] to improve the lookup operation cost. Specifically, in addition to its $2D$ neighboring seeds, each seed associates a location key to each neighbor seed of its routing table. Hence, if the seed coordinates are $(x_1, \ldots, x_d, \ldots, x_D)$, then the $+i$th (respectively the $-i$th) key location for the dth axis is defined by $(x_1, \ldots, x_{(d,+i)}, \ldots, x_D)$, where $x_{(d,+i)} = x_d + 2^i \rho_d$ (respectively $x_{(d,-i)} = x_d - 2^i \rho_d$). In addition, the distance between the seed and the location key is bounded by R_d where R_d is a system parameter corresponding to the absolute farthest location to be accessed in one hop in the dth axis. Each seed s also maintains a predecessors table that contains couples (s', l), where s' is a seed pointing on location l in s cell. The predecessors table is used when a `split` or `merge` operation are triggered to update the predecessors routing table.

***Join** operation.* When some new node p wants to `join` the system at time t, it contacts some node q already in the system. This bootstrap node q sends a lookup request for the incoming node position $Q(p, t)$ to find the seed s responsible for the cell in which p must be inserted. Once p gets s address, it asks s to `join` the bucket that matches its position $Q(p, t)$. If that bucket is the seed s itself, then the procedure described in Section 4.4 is run. Otherwise, s updates its *cell routing table* by inserting p address and its position $Q(p, t)$. Similarly, p keeps a pointer to s (as described above, routing is handled by seeds, thus p only needs to point to s). Now, if the number of nodes that sit in p bucket exceeds $S_{\min}$

Algorithm 1. p.join(t,q=None)

```
1  begin
2      if q = None then
3          q ← getBootstrapNode() ;
4      end
5      seed ← q.lookup(Q(p,t));
6      if p ∈ seed then
7          seed.insert(p);
8      else
9          bucket ← seed.findBucket(p);
10         bucket.insert(p);
11         if | bucket |≥ S_min then
12             cells ← seed.split(bucket);
13         end
14     end
15 end
```

Algorithm 2. cell.merge(bucket)

```
   Data: bucket such that| bucket |< S_min
1  begin
2      seed ← cell.seed;
3      seed.addOrphanCell(bucket.cell);
4      seed.mergeSiblingsCells();
5      seed.notifyPredecessors(bucket.cell);
6  end
```

Algorithm 3. cell.split(bucket)

```
   Requires: | bucket |≥
             S_min ∧ ¬bucket.isSeed()
   Ensures : bucket.isSeed()
1  begin
2      matchingCell ← findCell(bucket);
3      if matchingCell ∈ orphanCells then
4          bucket.cell ← matchingCell;
5          orphanCells.remove(matchingCell);
6      else
7          matchingCell.split(bucket);
8      end
9      bucket.notifyPredecessors(matchingCell);

10     bucket.updateRoutingTable();
11     bucket.setSeed(True);
12 end
```

Algorithm 4. p.leave()

```
1  begin
2      bucket ← p.bucket;
3      isSeed ← bucket.isSeed();
4      bucket.removePeer(p);
5      if isSeed ∧ | bucket |< S_min then
6          bucket.setSeed(False);
7          cell ← bucket.cell.getHook();
8          cell.merge(bucket.cell);
9      end
10 end
```

then this bucket becomes a seed, and a `split` operation is triggered by s (see below). The pseudo-code of the `join` operation is presented in Algorithm 1.

`Split` operation. A cell `splits` into two smaller cells when the population of one of its buckets exceeds $S_{\min}$ nodes and the cell has already one seed. The cell `splits` along the dimension that corresponds to the largest distance between the two seeds. More precisely, let s_1 and s_2 be the two seeds whose coordinates are $s_1 = (x_1^{(1)}, \ldots, x_D^{(1)})$ and $s_2 = (x_1^{(2)}, \ldots, x_D^{(2)})$. Let $i_0 = \text{argmax}_{1 \leq i \leq D} |x_i^{(1)} - x_i^{(2)}|$. Then the cell is split along the hyperplane orthogonal to i_0 axis and passing through the point $\lfloor (x_{i_0}^{(1)} + x_{i_0}^{(2)})/2\rho_{i_0} \rfloor \rho_{i_0} e_{i_0}$ where e_{i_0} is the D dimensional vector with $e_{i_0}(i) = 1_{\{i=i_0\}}$. Both seeds s_1 and s_2 update their respective cell routing tables to point to the nodes whose bucket falls in respectively s_1 and s_2 cells, as well as their routing table to point to their respective neighboring seeds. Figure 2 depicts the `split` operation of cell 1.

`Leave` operation. Let p be a node, c be the cell node p sits in, and s be the seed in charge of c. When node p `leaves` the overlay (either voluntarily or not) then seed s simply discards p from its cell routing table. As presented in Algorithm 4, if p was sitting in s and the population of s undershoots $S_{\min}$ nodes, then p departure provokes the `merging` of cell c with another cell c' as described in the sequel.

Prior to describing the `merge` operation, we introduce the notion of *cell hook* represented in Fig. 2 by black triangles.

Definition 3 (Cell hook). *Let c be a cell in a D-dimensional FixMe overlay. Each corner of c has $2D$ neighbors buckets. The hook of c is the first bucket (in the lexicographic order) of these neighboring buckets that does not belong to c.*

Proposition 1. *For a non-initial cell, the hook exists and is unique.*

Proof. Consider a cell c. By definition of a cell, c has 2^D corners. Each corner has $2D$ neighboring buckets. Among these neighboring buckets, the set B of buckets belonging to a neighboring cell has ℓ elements, with $\ell \in \{0\} \cup \{D, \ldots, 2D\}$. If c is the initial cell, $\ell = 0$ and thus $B = \emptyset$. Otherwise, c has at least one neighbor. In this case, $B \neq \emptyset$ and thus the hook exists. By definition, it is the first element of B in the lexicographic order. Thus it is unique. ■

`Merge` operation. A cell c `merges` with one of its neighbors c' when the population of its seed undershoots $S_{\min}$ nodes, and thus reverts to a default bucket. The cell c' with which c `merges` is determined as follows. If both c and c' share at least one face (we say that both cells are *sibling*), then both c and c' merge together in a single cell c'. On the other hand, if c has no sibling, then the cell that contains c hook takes in charge cell c. Thus a single seed may be in charge of several cells. In Fig. 2 on the right, the seed of cell 2 is also in charge of cell 5.

4.4 Self-organizing Nodes in Dense Seeds

In the context of QoS monitoring, it is not unusual to observe that a very large number of nodes perceive a quite similar QoS for a set of services. In such cases, FixMe would show cells with very dense seeds, that is seeds with a quite large number of nodes. Thus to keep the scalability property of FixMe, we propose to self-organize these nodes into a structured graph so that the routing cost among them remains logarithmic in their population size. Any structured graph proposed in the literature can be chosen. In this work we use PeerCube [16] essentially because each vertex of the hypercube gather from $S_{\min}$ to $S_{\max}$ nodes, which makes this cluster-based DHT highly robust to churn. Thus, in FixMe overlay as shown in Fig. 3, all the seeds are organized as follows. The first $S_{\min}$ nodes that are in a seed form the *root* of the hypercube, and upon new nodes arrivals, the dimension of the hypercube increases [16]. From the point of view of the neighboring seeds of any other seed s, only the root of the hypercube is visible.

5 Analysis

In this section, we evaluate the complexity of FixMe operations. There is trade-off between the number of seeds in the overlay and the number of nodes in the seeds. The two distributions that illustrate this trade-off are the uniform

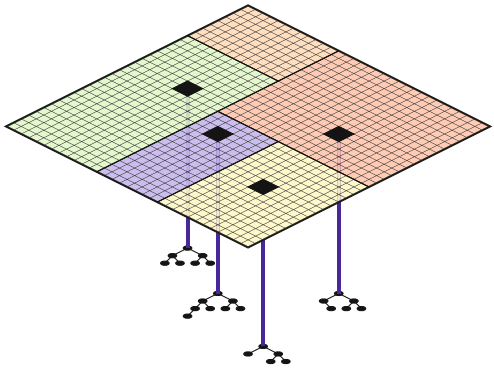


Fig. 3. FixMe cell-layer overlay and the embedded clusterized overlays

distribution and the Dirac one. The Uniform distribution maximizes the seeds number, and the Dirac distribution, which concentrates all the nodes in the same bucket, maximizes the dimension of the underlying hypercube.

Proposition 2. *The seed routing table has* $2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor$ *entries.*

Proof. As explained in Section 4.3, the distance between the cell centre and each entry $x_{(d,k)}$ of the routing table is bounded by R_d. Let $K_+^{(d)} = \max\{k \geq 1 \mid |x_d - x_{(d,+k)}| \leq R_d\}$ and let $K_-^{(d)} = \max\{k \geq 1 \mid |x_d - x_{(d,-k)}| \leq R_d\}$. Since $|x_d - x_{(d,+k)}| = |x_d - x_{(d,-k)}| = 2^k\rho_d$, we have $K_+^{(d)} = K_-^{(d)} = \lfloor\log_2(R_d/\rho_d)\rfloor$. Let $K^{(d)}$ be this common value. For each dimension d, the routing table has $2K^{(d)}$ entries. Thus, the routing table has $\sum_{d=1}^{D} 2K^{(d)} = 2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor$ entries. ■

Proposition 3 (Node `join`). *If an incoming node is inserted in a seed then, the insertion complexity is* $\Theta(\log H)$*, with* H *the number of nodes populating the underlying hypercube.*

Proof. If the incoming node belongs to the seed, there is no change in the cell layer. Thus, the complexity of this operation is only driven by the insertion in the underlying hypercube. It is well known that the complexity of this operation is $\Theta(\log H)$, where H is the number of nodes in the underlying hypercube. ■

Proposition 4 (Cell `split`). *If an incoming node insertion leads to a seed creation then, the complexity of this operation is* $\mathcal{O}(D)$*.*

Proof. As previously seen, the insertion of a node p might lead to a split operation (*cf.* `join` operation). This operation triggers only one write operation in the routing table of the concerned seed s_1. From the created seed s_2 point of view, node p is inserted in the hypercube root node. This operation is performed in constant time. Nevertheless, seed s_2 needs to build its routing table that will contain its neighboring seeds. As the routing table has $2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor$ entries, and as seed s_1 is necessarily a neighbor of s_2, then s_2 routing table creation will generate $2\sum_{d=1}^{D}\lfloor\log_2(R_d/\rho_d)\rfloor - 1$ `lookup` operations. ■

Proposition 5 (Node `leave`). *The `leave` of a node without topological change requires $\Theta(\log(H))$ messages number, with H the number of nodes in the underlying hypercube.*

Proof. When a node `leaves` its bucket, it is simply removed from its cluster in the underlying hypercube. Two cases are possible: either its cluster remains sufficiently populated, or its cluster has to merge with another one. In the former case, the cluster nodes simply update their view of the cluster. In the later case, $\Theta(\log(H))$ messages have to be sent to merge both clusters (See [16]). ■

When the hypercube has a single cluster populated by exactly $S_{\min}$ nodes (*i.e.*, the root cluster), a node `leave` makes the seed undershoot its population lower bound. Thus the corresponding cell c must `merge` with the cell containing c hook.

Proposition 6. *Let c be a cell, and p_i be the number of seeds that point to c along the ith axis. We have*

$$p_i \leq 2\log_2(n_i/2)\prod_{j=1,j\neq i}^{D} n_j$$

Proof. Let ℓ_j be the length of cell c along the jth axis. In the case where each neighboring bucket of c is a seed, c has at most ℓ_j/ρ_j immediate neighbors on each side. Thus, we have $p_i \leq 2K^{(i)}\prod_{j=1,j\neq i}^{D}\ell_j/\rho_j$. As shown in Proposition 2, we have $K^{(i)} = \lfloor\log_2(R_i/\rho_i)\rfloor$. Moreover, $\forall i \in \{1,\ldots,D\}$ we have $R_i \leq 1/2$ since in a torus unitary space, the farthest point is located at distance $1/2$. By definition $\rho_i = 1/n_i$, thus we have $p_i \leq 2\log_2(n_i/2)\prod_{j=1,j\neq i}^{D} n_j\ell_j$. The space being unitary, $\forall j \in \{1,\ldots,D\}$ we have $\ell_j \leq 1$, and thus we get $p_i \leq 2\log_2(n_i/2)\prod_{j=1,j\neq i}^{D} n_j$. ■

Proposition 7 (Cell `merge`). *If a cell merges, then the* `merge` *operation requires at most*
$2Dn^{D-1}\log_2(n/2)$ *messages, with* $n = \max_{1\leq i\leq D} n_i$.

Proof. By assumption of the proposition, the size of the corresponding seed s is equal to $S_{\min}$. Thus, when a node `leaves` seed s, this triggers a merge operation. The remaining nodes in s must contact the seed that will take in charge their seedless cell, and notify their predecessors. The number of predecessors p, which is given by $p = \sum_{i=1}^{D} p_i$, satisfies by Proposition 6, $p \leq 2\sum_{i=1}^{D} \log_2(n_i/2) \prod_{j=1,j\neq i}^{D} n_j$. By definition of n, we have $n^{D-1} \geq \prod_{j=1,j\neq i}^{D} n_j$, it follows that $p \leq 2n^{D-1}\sum_{i=1}^{D} \log_2(n_i/2)$ and thus $p \leq 2Dn^{D-1}\log_2(n/2)$. ■

Proposition 8. *The Uniform distribution and the Dirac distribution give rise to a* `lookup` *operation requiring* $\mathcal{O}(\log N)$ *messages.*

Proof. Suppose that the quality position of the nodes are uniformly distributed. Then the nodes will join fairly all the buckets. By construction, this maximizes the number of seeds and minimizes the dimension of each underlying hypercube. The dimensions of the hypercubes are equivalent and depend on the expected population H in each one. For N large enough, each bucket is a seed, and thus there are $\prod_{i=1}^{D} n_i$ seeds. Thus, since $n_i = 1/\rho_i$, we have $H = N\prod_{i=1}^{D} \rho_i$. As described in Section 4.3, a `lookup` operation is decomposed into three parts, namely, the hypercube traversal at the source of the `lookup` operation, the cells traversal and, the hypercube traversal at the destination of the `lookup` operation). As shown in [16], the dimension of an hypercube populated by H nodes equals to $\log(H/S_{max})$. By setting $S_{\max}$ to $\log N$, the number of messages required is equal to traverse an hypercube is equal to $\log(N\prod_{i=1}^{D} \rho_i) - \log\log N = \mathcal{O}(\log N)$. The cells traversal requires $\mathcal{O}(D)$ messages. Thus the total number of messages required for a `lookup` operation is $\mathcal{O}(\log N)$.

Suppose now that the quality position of the nodes follows a Dirac distribution. Then all the nodes will join the same bucket. The overlay is thus equal to the unique initial cell, and all the nodes belong to the same underlying hypercube. Its dimension is maximal, and thus $H = N$. By an argument similar to the previous one, the total number of messages required for a `lookup` operation is $\mathcal{O}(\log N)$. ■

6 Solving the Isolated Anomaly Detection Problem

We now propose an algorithm that solves the isolated anomaly detection problem. The algorithm, whose pseudo code is presented in Fig. 4, is cyclically run

by any node p, and is made of the following three tasks. Briefly, in Task 1, node p changes its position in FixMe overlay according to the QoS change of its monitored services (if necessary). If this QoS change is diagnosed as an anomaly by its function A, then p determines whether this anomaly is isolated or not (Task 2), and in the affirmative sends a FixMe message to the management operator (Task 3).

Let r be the current round of the algorithm. In Task 1 node p computes its current position $Q(p, r)$. Let b_r be the bucket that corresponds to this position, c_r be the cell that contains b_r, and s_r be the seed in charge of cell c_r. If $Q(p, r)$ differs from p position at time $r - 1$ (we note b_{r-1} the bucket that corresponds to this position), then p `leaves` bucket b_{r-1} and `joins` bucket b_r. If there exists a service i for which $A_i(p, r) =$ `true` then p runs Task 2. The goal of Task 2 is to enable node p to determine whether there are other nodes in the overlay that have experienced the same QoS change as p, that is, nodes that `left` bucket b_{r-1} at the beginning of round $r - 1$ and `join` bucket b_r at the beginning of round r. This is achieved as follows. By construction of FixMe, an hypercube is embedded in the seed s of each cell (see Section 4.4), and all the nodes in that cell point to the cluster root of seed s (see Section 4.3). Let H_r be the hypercube embedded in seed s_r. Then p computes a random key h that depends on both round $r - 1$ and its previous position b_{r-1} (see line 14 of Algorithm 5), and asks the node in H_r that is in charge of key h (by construction of any DHT, such a node always exists) to increment a counter v (initially set to 0 at the beginning of round r). After T time units, Task 3 starts. Node p reads counter v, and if it strictly less than τ (*i.e.*, no more than τ nodes have jump from bucket b_{r-1} to bucket b_r) then p sends a FixMe message to the management node, which ends Task 3.

Theorem 1. *Algorithm 5 solves the isolated anomaly detection problem.*

Proof. The proof is made by contradiction. Suppose that at round r, *(i)* $k \leq \tau$ nodes experience the same change in their monitored qualities, *(ii)* such a change is large enough to be diagnosed as an anomaly, and *(iii)* none of these k nodes send a FixMe message at the end of the round r.

Let p be one of these nodes. At each round, p executes Algorithm 5, and in particular round r. By assumption (i), p has experienced a quality change and thus moves in the FixMe overlay from its current bucket b_1 to the new one b_2. By assumption (ii), p runs Task 2, and thus the counter tracking jumps from b_1 to b_2 is incremented. By assumption (i) $k - 1$ other nodes proceed as p. Thus at the end of Task 2, the counter value is less than or equal to τ. By Task 3, p (and all the other $k - 1$ nodes) sends a FixMe message to the coordinator. Which is a contradiction with assumption (iii). This completes the proof of the theorem. ■

Proposition 9. *Algorithm 5 described in Fig. 4 requires* $\mathcal{O}(\log N)$ *messages.*

Proof. Straightforward from Property 3. ■

Algorithm 5. p.updatePosition(r:round)

```
Data: T: delay such that all nodes have moved to their new bucket (if
      necessary).
Output : The positioning of p in the appropriate bucket, and the
         sending of a FixMe message if p detects an isolated failure
1  begin
2    Task 1
3      r← r+1;
4      oldposition ← p.bucket ;
5      newposition ← Q(p,r);
6      newbucket ← p.lookup(newposition);
7      if newbucket ≠ p.bucket then
8        p.leave();
9        p.join(r,p);
10     end
11   EndTask
12   if ∃i, 1 ≤ i ≤ D, A_i(p, r)=true then
13     Task 2
14       h ← H(oldposition, r − 1);
15       p.incrementValue(h);
16     EndTask
17     Wait Until T;
18     Task 3
19       n ← p.cell.seed.get(h);
20       if n ≤ τ then
21         send FixMe msg to Management Operator;
22       end
23     EndTask
24   end
25 end
```

Fig. 4. Isolated Anomaly Detection algorithm run by any node p

7 Conclusion

In this work, we have formalized the isolated anomaly detection problem. Such a problem is recurrent in various large scale monitoring applications, and in particular in the cable and telecom industry where it is of utmost importance to make the difference between isolated anomalies and network based anomalies. One of the reasons being a financial one. In this context we have proposed the FixMe tool that pushes monitoring to end devices, and by combining local algorithms to detection functions provides a scalable and efficient solution to the isolated anomaly detection problem. As a future work, we first plan to analyze the evolution of FixMe in a stochastic model to study, in particular, the influence of the distributions on the cells repartition and their sizes. The long term objective is the implementation, and deployment of FixMe.

References

1. Broadband Forum: TR-069 CPE WAN Management Protocol Issue 1, Amend.4 (2011)
2. Rabkin, A., Katz, R.: Chukwa: a system for reliable large-scale log collection. In: Proceedings of the International Conference on Large Installation System Administration, LISLA (2010)

3. Zhao, Y., Tan, Y., Gong, Z., Gu, X., Wamboldt, M.: Self-correlating predictive information tracking for large-scale production systems. In: Proceedings of the International Conference on Autonomic Computing, ICAC (2009)
4. Desphand, A., Guestrin, E., Madden, S.: Model-driven data acquisition in sensor networks. In: Proceedings of the International Conference on Very Large Databases, VLDB (2002)
5. Krishnamurthy, S., He, T., Zhou, G., Stankovic, J.A., Son, S.H.: RESTORE: A Real-time Event Correlation and Storage Service for Sensor Networks. In: Proceedings of the International Conference on Network Sensing Systems, INSS (2006)
6. Vuran, M.C., Akyildiz, I.F.: Spatial correlation-based collaborative medium access control in wireless sensor networks. IEEE/ACM Transactions on Networking (TON) 14(2), 316–329 (2006)
7. Kalman, R.E.: A New Approach to Linear Filtering and Prediction Problems. Journal of Basic Engineering 82(1), 35–45 (1960)
8. Xiong, X., Mokbel, M., Aref, W.: SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Databases. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE (2005)
9. Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y.: A Threshold-Based Algorithm for Continuous Monitoring of K Nearest Neighbors. IEEE Transactions on Knowledge and Data Engineering 17(11), 1451–1464 (2005)
10. Zhang, Z., Yang, Y., Tung, A.K.H., Papadias, D.: Continuous k-means monitoring over moving objects. IEEE Transactions on Knowledge and Data Engineering 20(9), 1205–1216 (2008)
11. Har-Peled, S., Sadri, B.: How fast is the k-means method? Algorithmica 41(3), 185–202 (2005)
12. Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., Shenker, S.: A scalable content-addressable network. In: Proceedings of the SIGCOMM Conference (2001)
13. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the SIGCOMM Conference (2001)
14. Lin, J.: Broadcast scheduling for a p2p spanning tree. In: Proceedings of the IEEE International Conference on Communications (2008)
15. Kovacs, B., Vida, R.: An adaptive approach to enhance the performance of content-addressable networks. In: Proceedings of the International Conference on Network and Computer Science, ICNS (2007)
16. Anceaume, E., Ludinard, R., Ravoaja, A., Brasileiro, F.V.: Peercube: A hypercube-based p2p overlay robust against collusion and churn. In: Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO (2008)

Analyzing Global-EDF for Multiprocessor Scheduling of Parallel Tasks

Björn Andersson and Dionisio de Niz

Software Engineering Institute, Carnegie Mellon University

Abstract. Consider the problem of scheduling a set of constrained-deadline sporadic real-time tasks on a multiprocessor where (i) all processors are identical, (ii) each task is characterized by its execution requirement, its deadline and its minimum inter-arrival time, (iii) each task generates a (potentially infinite) sequence of jobs and (iv) the execution requirement of a job and its potential for parallel execution is described by one or many stages with a stage having one or many segments such that all segments in a stage have the same execution requirement and segments in the same stage are permitted to execute in parallel and a segment is only allowed to start execution if all segments of previous stages have finished execution. We present a schedulability test for such a system where tasks are scheduled with global-EDF. This schedulability test has a resource-augmentation bound of two, meaning that if it is possible for a task set to meet deadlines (not necessarily with global-EDF) then our schedulability test guarantees that all deadlines are met when tasks are scheduled with global-EDF, assuming that the system analyzed with our schedulability test is provided processors of twice the speed.

Keywords: Real-time systems, Scheduling, Multiprocessors, Multicores.

1 Introduction

Today, a multiprocessor implemented on a single chip (a.k.a multicore) is the norm with the trend being that the number of processors on a chip increases exponentially while the clock frequency stays constant. This trend makes it increasingly common that the only way for a job requiring C units of execution to be performed by its deadline D is to execute some of these C units of execution in parallel. Therefore, we believe software practitioners benefit from a scheduling theory allowing task parallelism and it should fulfill the following requirements:

R1. It should be possible to schedule tasks with short deadline but long minimum inter-arrival time efficiently. The constrained-deadline sporadic model is suitable for this. The rationale for this requirement is that many real-time systems must perform processing to handle rare but critical events and serving them with polling is very inefficient.

R2. It should offer a performance guarantee which should be as tight as possible. A suitable performance guarantee is the so-called *resource augmentation*

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 16–30, 2012.

Table 1. Summary of the state of art

Algorithm	fulfills requirements		has resource	Comments
	R1	R2	aug. bound	
[1]	No	Yes	3.42	implicit deadline only
[2]	No	No		
[3] G-EDF	No	Yes	4	implicit deadline only
[3] P-DM	No	Yes	5	implicit deadline only
[4] G-EDF	Yes	No		soft real-time only
[6] G-EDF	Yes	No		
[7]	Yes	No		distributed system, task migration is not allowed
[8,9,10]	Yes	No		single processor only
This paper: G-EDF	Yes	Yes	2-1/m < 2	multiprocessor, task migration is allowed.

bound. A scheduling algorithm A and corresponding schedulability analysis S, together having resource-augmentation bound f has the property that if it is possible that a task set will meet deadlines then the scheduling algorithm A together with the schedulability test S guarantees that all deadlines are met as well if given processors f times as fast. Clearly, the lower the resource augmentation bound is, the better the performance is.

The research community has produced scheduling algorithms and schedulability analyses for parallel tasks. Unfortunately, they do not fulfill the requirements R1 and R2.

Therefore, in this paper, we present a new schedulability test that fulfills the requirements R1 and R2. Specifically, we consider global-EDF and present a schedulability analysis for it. With respect to R1, we note that our new schedulability test considers the model constrained-deadline sporadic tasks. This model of describing arrival times and deadlines is at least as general as the ones used in all previously mentioned works on task scheduling with parallelism. With respect to R2, we note that our new schedulability test has a resource augmentation bound two; this is better than all previous works and it is the same resource augmentation bound as the actual scheduling algorithm [11]. So from the perspective of resource augmentation bound, it is impossible to create a schedulability test for global-EDF that is better than the one we present in this paper. The main idea of our new schedulability test is to use the schedulability analysis framework previously proposed by Baruah et al. [12] for global-EDF but modify it for parallel tasks.

The remainder of this paper is structured as follows. Section 2 presents the system model. Section 3 defines the concept *ff-dbf*. Section 4 presents a schedulability test. Section 5 proves the resource augmentation bound. Section 6 gives conclusions.

2 System Model and Terminology

We assume that the computer system is composed of m processors each of speed s and the software system is described by a task set τ where each task τ_i in τ is characterized by T_i, D_i and the values $c_{i,j}$ with the interpretation that the task τ_i generates a (potentially infinite) sequence of jobs where the arrival time of two consecutive jobs of the same task τ_i are separated by at least T_i and a job of task τ_i needs to finish execution within at most D_i time units after its arrival.

We describe the execution requirement of a task as a number of stages where nstages_i denotes the number of stages of a job of task τ_i and $\text{nseg}_{\text{i,j}}$ denotes the number of segments of stage j of a job of task τ_i. Two segments in a stage can execute in parallel but they do not have to. In order for a segment in the j:th stage of a job of task τ_i to finish, it must perform a certain number of units of execution. This amount is called the execution requirement of the segment and it is in $[0,\ c_{i,j}]$. Segments of the first stage of a job of a task become eligible when the job arrives. Segments of a later stage of the job become eligible when all segments of all previous stages of the job have finished.

We assume that preemptive global-EDF is used to schedule tasks. It works as follows. Let eligiblejobs(t) denote the set of jobs that have arrived at instant t but not yet finished execution. Let eligiblesegments(t) denote the set of segments that have arrived at instant t but not yet finished. Then, the $\min(m, |\text{eligiblesegments(t)}|)$ highest priority segments in eligiblesegments(t) are selected for execution on $\min(m, |\text{eligiblesegments(t)}|)$ processors at instant t. We ignore preemption and migration costs and hence we do not specify which processor a segment actually executes on at a given time. A job of task τ_i has its absolute deadline D_i time units after its arrival. Global-EDF assigns priorities to jobs so that a job with earlier absolute deadline is given a higher priority than a job with later absolute deadline. An eligible segment is assigned the same priority as the job it was generated from.

We say that an assignment of values to the arrival times of jobs and execution requirements of segments is legal if (i) it holds that for each two consecutive jobs of the same task τ_i, the arrival times are separated by at least T_i time units and (ii) for each job of task τ_i, for each stage j of the job, it holds for each of the segments in the j:th stage, the execution requirement of the segment is in $[0,\ c_{i,j}]$. One legal assignment that we will find to be particularly important is the ASAP-CMAX assignment which we define as follows (i) it holds that for each two consecutive jobs of the same task τ_i, the arrival times are separated by *exactly* T_i time units and (ii) for each job of task τ_i, for each stage j of the job, it holds for each of the segments in the j:th stage, the execution requirement of the segment is $c_{i,j}$.

We say that a task set τ is S-schedulable if for each legal assignment of arrival times and execution times it holds that all deadlines are met when the scheduling algorithm S is used. Clearly, we are particularly interested in global-EDF schedulability. A task set τ is feasible if for each legal assignment of arrival times and execution times it holds that it is possible to create a schedule so that

all deadlines are met. A task set τ is schedulable by S according to schedulability test A if it holds that A proves that τ is S-schedulable.

We say that a segment executed for L time units on a processor of speed s performs s^*L units of execution. If a job is executing exactly d (where $d \geq 1$) segments in parallel in a time interval of duration L then we say that the job performs d^*s^*L units of execution in the time interval and it performs s^*L units of elapsed execution in the time interval. For a time interval of duration L where a job J executes with different degrees of parallelism, we can subdivide the time interval into sub time intervals where in each sub time interval the degree of parallelism of J is constant and calculate the number of units of execution performed in these sub time intervals and then add them up; this gives us the number of units of execution performed during the time interval of duration L. If a job is executing exactly d (where $d \geq 1$) segments in parallel in a time interval of duration L such that during this time interval, no other job executes then we say that the job performs d^*s^*L units of alone execution in this time interval and it performs s^*L units of alone elapsed execution in this time interval.

We assume that T_i parameters are rational non-negative numbers; this assures us that P=lcm(T_1, T_2, ..., T_n) exist. We assume that D_i, $c_{i,j}$, and s are real non-negative numbers. In order to simplify our discussion later in the paper, we now introduce additional concepts. We define η_i and C_i as:

$$\eta_i = \sum_{j=1}^{\text{nstages}_i} \left(\lceil \frac{\text{nseg}_{i,j}}{m} \rceil * c_{i,j} \right) \qquad C_i = \sum_{j=1}^{\text{nstages}_i} (\text{nseg} * c_{i,j}) \tag{1}$$

Intuitively, η_i denotes the number of elapsed units of execution performed by a job of task τ_i for the case that the job executes with its maximum execution time. And C_i denotes the number of units of execution performed by a job of task τ_i for the case that each segment of the job executes with its maximum execution time.

Let δ_i denote the elapsed density of task τ_i. Formally it is defined as: $\delta_i = \eta_i / \min(D_i, T_i)$. Since we consider constrained-deadline sporadic tasks (where $\forall \tau_i \in \tau : D_i \leq T_i$), we obtain: $\delta_i = \eta_i / D_i$.

Lemma 1. *Consider a task set τ. If there is a task τ_i in τ such that $\delta_i > s$ then the task set τ is infeasible.*

Proof. If $\delta_i > s$ then $\eta_i / D_i > s$ which can be rewritten as $\eta_i / s > D_i$. Consider the ASAP-CMAX assignment. Then, the expression η_i / s indicates the number of time units that a job of τ_i must execute on a processor of speed s in order to meet its deadline. Since this exceeds the deadline, it implies that a deadline is missed.

Let δ^{max} be defined as $\delta^{max} = \max_{i=1..n} \delta_i$. Let u_i denote the utilization of task τ_i. Formally, it is defined as $u_i = C_i / T_i$. Let U be defined as: $U = \sum_{i=1}^{n} u_i$. Let idxt(τ_i) be a function that returns the index of task τ_i. Let task(J) denote

a function that takes a job as input and returns the index of the task that generated the job J.

3 Defining FF-DBF

We define $\mathrm{WJ}(\tau_i,\, t,\, s)$ as follows:

$$\mathrm{WJ}(\tau_i, t, s) = \begin{cases} 0 & \text{if t} < 0 \\ \mathrm{WJS}(idxt(\tau_i), t, 1, s) & \text{if } 0 \leq \text{t} < \frac{\eta_{\mathrm{idxt}(\tau_i)}}{s} \\ C_{\mathrm{idxt}(\tau_i)} & \text{if } \frac{\eta_{\mathrm{idxt}(\tau_i)}}{s} \leq \text{t} \end{cases} \tag{2}$$

where

$$\mathrm{WJS}(i, t, j, s) = \begin{cases} t * m * s & \text{if } 0 \leq t < \mathrm{bsp}_{i,j} \\ \mathrm{bsp}_{i,j} * m * s + (t - \mathrm{bsp}_{i,j}) * (\mathrm{nseg}_{i,j} - \lfloor \frac{\mathrm{nseg}_{i,j}}{m} \rfloor * m) * s & \text{if } \mathrm{bsp}_{i,j} \leq t < \mathrm{sp}_{i,j} \\ c_{i,j} * \mathrm{nseg}_{i,j} + \mathrm{WJS}(i, t - \mathrm{sp}_{i,j}, j+1, s) & \text{if } \mathrm{sp}_{i,j} \leq \text{t} \end{cases} \tag{3}$$

where

$$\mathrm{bsp}_{i,j} = \frac{c_{i,j}}{s} * \lfloor \frac{\mathrm{nseg}_{i,j}}{m} \rfloor \qquad \mathrm{sp}_{i,j} = \frac{c_{i,j}}{s} * \lceil \frac{\mathrm{nseg}_{i,j}}{m} \rceil \tag{4}$$

Intuitively, $\mathrm{WJ}(\tau_i,\, t,\, s)$ denotes the units of execution "work" that a job of task τ_i performs in a time interval of duration t such that (i) this job arrives when this time interval begins and (ii) for each stage j of the job, it holds for each of the segments in the j:th stage, the execution requirement of the segment is $c_{i,j}$. These equations express that we can compute $\mathrm{WJ}(\tau_i,\, t,\, s)$ by iterating over all stages (left-to-right order) and express the iteration recursively. Note that the way WJ is defined, it considers just a single job. Also, note that a job of task τ_i can perform more than $\mathrm{WJ}(\tau_i,\, t,\, s)$ units of execution if there is a stage j of the job, for which it holds that there is a segment in the j:th stage where the execution requirement of the segment is less than $c_{i,j}$.

Lemma 2. *For each $f>0$, it holds that:* $\mathrm{WJ}(\tau_i,\, t,\, s) = \mathrm{WJ}(\tau_i,\, t*f,\, s/f)$

Proof. Follows from the definition of WJ.

Lemma 3. *For each $f\geq 1$, it holds that:* $\mathrm{WJ}(\tau_i,\, t,\, s) \leq \mathrm{WJ}(\tau_i,\, t,\, s*f)$

Proof. Follows from the definition of WJ.

Note that $\mathrm{WJ}(\tau_i,\, t,\, s)$ is a step-wise linear function with respect to t. Therefore, we define:

$$\mathrm{nhcWJ}(\tau_i, t, s) = \begin{cases} 0 & \text{if t} < 0 \\ \mathrm{nhcWJS}(idxt(\tau_i), t, 1, s) & \text{if } 0 \leq \text{t} < \frac{\eta_{\mathrm{idxt}(\tau_i)}}{s} \\ T_{\mathrm{idxt}(\tau_i)} & \text{if } \frac{\eta_{\mathrm{idxt}(\tau_i)}}{s} \leq \text{t} \end{cases} \tag{5}$$

and

$$\text{nhcWJS}(i,t,j,s) = \begin{cases} \text{bsp}_{i,j} & \text{if } 0 \leq t < \text{bsp}_{i,j} \\ \text{sp}_{i,j} & \text{if } \text{bsp}_{i,j} \leq t < \text{sp}_{i,j} \\ \text{sp}_{i,j} + \text{nhcWJS}(i, t - \text{sp}_{i,j}, j+1, s) & \text{if } \text{sp}_{i,j} \leq \text{t} \end{cases} \tag{6}$$

Intuitively, $\text{nhcWJ}(\tau_i, t, s)$ denotes the lowest value greater than t such that the derivative of WJ with respect to t might change. Let $q_i(t)$ and $r_i(t)$ be defined as

$$q_i(t) = \lceil \frac{t}{T_i} \rceil \qquad r_i(t) = t \ mod \ T_i \tag{7}$$

Let us define $\text{ff} - \text{dbf}(\tau_i, t, v, s)$ as follows.

$$\text{ff} - \text{dbf}(\tau_i, t, v, s) = q_i(t) * C_i + C_i - \text{WJ}(\tau_i, (D_i - r_i(t)) * v, s) \tag{8}$$

Intuitively, $\text{ff} - \text{dbf}(\tau_i, t, v, s)$ indicates the number of units of execution that jobs of task τ_i can perform in a time interval of duration t assuming that if there is a job that arrives before the time interval and this job has a deadline after the beginning of the time interval and it holds that this job performs exactly $(D_i\text{-}r_i(t))^*v^*s$ elapsed units of execution before the start of the time interval of duration t.

Lemma 4. *For each $f>0$, it holds that:*

$$\text{ff} - \text{dbf}(\tau_i, t, v, s) = \text{ff} - \text{dbf}(\tau_i, t, v * f, s/f) \tag{9}$$

Proof. Follows from the definition of $\text{ff} - \text{dbf}$ and WJ.

4 Schedulability Test

In this section, we will discuss two variants of our task model. The SEGSEQ-variant refers to the model in Section 2 assuming that a segment must execute sequentially. Note that with this model, it is still possible for two or more segments in a stage to execute in parallel or partially in parallel. The SEGPAR-variant refers to the model in Section 2 assuming that a segment can be broken into any number of subsegments — we assume that the breakup into subsegments is not done under the control of the scheduling algorithm.

We say that time t is a decision instant of the SEGPAR-variant if (i) there is a segment that arrives at time t or (ii) there is a subsegment that finishes at time t. In the SEGPAR-variant, at a decision instant t, a segment is broken into any number of subsegments so that the sum of the execution requirement of these subsegments equals the remaining execution requirement of the segment. A subsegment can be broken up into new subsegments just like a segment. Note that even in the SEGPAR-variant, it still holds that a segment is only allowed to start execution if all segments in the previous stage has finished. Figure 1 illustrates the two variants.

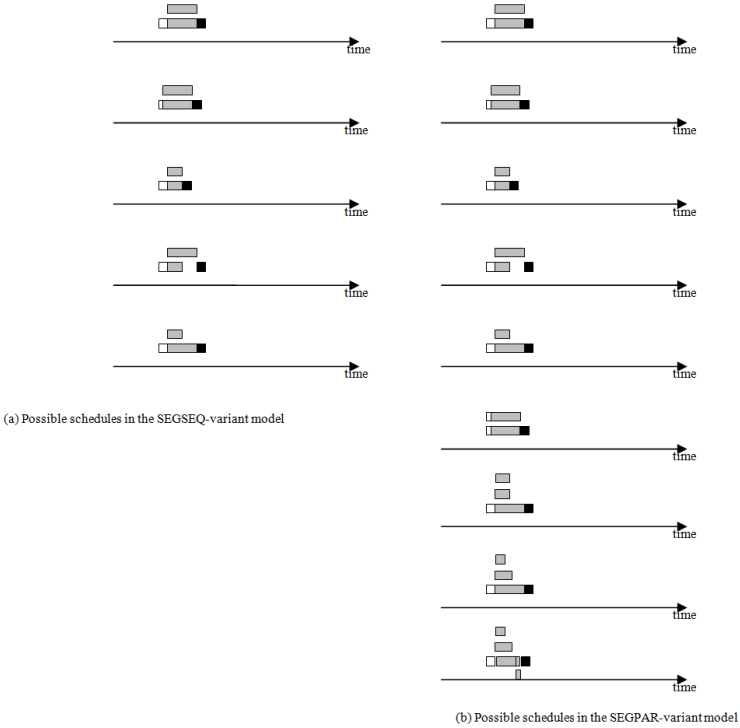


Fig. 1. An example of the SEGPAR and SEGSEG variants of our model. Consider a task τ_i with nstages_i=3 and $\text{nseg}_{i,1}$=1 and $\text{nseg}_{i,2}$=2 and $\text{nseg}_{i,3}$=1 and $c_{i,1} = 2$ and $c_{i,2} = 6$ and $c_{i,3} = 2$. Each subfigure shows schedules that can be generated with each variant model. A rectangle shows execution on a processor; if at a certain instant, there are two rectangles, then it indicates that two segments execute on two processors in parallel.

Definition 1. *A task set is schedulable in the SEGPAR variant of our task model if for all possible scenarios that can happen in this model, all deadlines are met.*

Definition 2. *A task set is schedulable in the SEGSEQ variant of our task model if for all possible scenarios that can happen in this model, all deadlines are met.*

Lemma 5. *If a task set τ is schedulable in the SEGPAR variant of our task model then task set τ is schedulable in the SEGSEQ variant of our task model.*

Proof. Follows from the fact that each possible scenario in SEGPAR is also a possible scenario in SEGSEQ.

Lemma 6. *Consider the SEGPAR variant of our task model. Consider a task set and a time t where there is a job J_x released by task $\tau_{taskidx}$ where the arrival time of job J_x is A_x and the deadline of the job is D_x (with $D_x = D_{taskidx}$) and $A_x < t$ and $A_x + D_x > t$. Assume that for the resulting schedule, it holds that when job J_x executes, no other job executes. Let* work_alone_elapsed*$([a,b], J_x)$ indicate the number of alone elapsed units of execution that job J_x performed in a time interval [a,b] and* work*$([a,b], J_x)$ indicates the number of units of execution that job J_x performed in a time interval [a,b). Let z be a non-negative number. We claim that* work_alone_elapsed*$([A_x, t), J_x) \geq z*(t- A_x)$ implies* work*$([t, \infty), J_x) \leq$* $C_{taskidx}$-WJ*$(\tau_{taskidx}, z*(t- A_x)/s, s)$*

Proof. The proof is by contradiction. Suppose that the lemma would be false. Then, it holds that it is possible that work_alone_elapsed$([A_x, t), J_x) \geq z*(t- A_x)$ and work$([t, \infty), J_x) >$ $C_{taskidx}$-WJ$(\tau_{taskidx}, z*(t- A_x)/s, s)$. We can modify this scenario to set the execution requirement of each segment of J_x to its maximum as specified by its $c_{i,j}$ parameters. In this way, the number of segments of J_x that executes in $[t, \infty)$ will increase or stay the same because the segments of J_x executing before t will push later segments of J_x to start later and some of them may be pushed into partially or fully execute in the time interval $[t, \infty)$. Also, for the segments of J_x that execute in $[t, \infty)$ the number of units of execution they perform will increase or stay the same. In addition, the number of units of alone elapsed execution before t will increase or stay the same. Hence, we obtain that:

$$\text{work\_alone\_elapsed}([A_x, t), J_x) \geq z * (t - A_x) \tag{10}$$

and

$$\text{work}([t, \infty), J_x) > \text{C}_{\text{taskidx}} - \text{WJ}(\tau_{taskidx}, z * (t - A_x)/s, s) \tag{11}$$

and

$$\text{execution requirement of each segment of } \text{J}_\text{x} \text{ is its maximum as specified by } \text{c}_{\text{i,j}} \text{ parameters.} \tag{12}$$

Clearly, it holds that work$([A_x, t), J_x)$ + work$([t, \infty), J_x) \leq$ $C_{taskidx}$. Applying it on Equation 11 gives us:

$$\text{work}([A_x, t), J_x) < \text{WJ}(\tau_{taskidx}, z * (t - A_x)/s, s) \tag{13}$$

From Equation 10 and 12 we obtain

$$\text{work}([A_x, t), J_x) \geq \text{WJ}(\tau_{taskidx}, z * (t - A_x)/s, s) \tag{14}$$

This is a contradiction and hence the lemma is correct.

Lemma 7. *Consider the SEGPAR variant of our task model. Consider a single job J_x released by task $\tau_{taskidx}$ that meets its deadline and whose absolute deadline is r'_x time units after t and where the job arrives before t and the deadline of the job is after t and such that before time t, job J_x performed at least $z*(D_x - r'_x)/s$ elapsed units of execution. Assume that for the resulting schedule, it holds that when job J_x executes, no other job executes. We claim that it holds that job J_x performs at most:* $C_{taskidx}$-WJ*$(\tau_{taskidx}, z*(D_x - r'_x)/s, s)$ units of execution in the time interval $[t, d_x)$*

Proof. Follows from Lemma 6 and applying $A_x + D_x - r'_x = t$.

Lemma 8. *Consider the SEGPAR variant of our task model. Consider that* $r'_x \leq r_x$ *and consider a single job* J_x *released by task* $\tau_{taskidx}$ *that meets its deadline and whose absolute deadline is* r'_x *time units after* t *and where the job arrives before* t *and the deadline of the job is after* t *and such that before time* t*, job* J_x *performed at least* $z^*(D_x - r_x)/s$ *alone elapsed units of execution. Assume that for the resulting schedule, it holds that when job* J_x *executes, no other job executes. We claim that it holds that job* J_x *performs at most:* $\mathrm{C}_{\mathrm{taskidx}}$-$\mathrm{WJ}(\tau_{taskidx}, z^*(D_x - r_x)/s, s)$ *units of execution in the time interval* $[t, d_x)$

Proof. Follows from Lemma 7 and the fact that the function WJ is monotonic with respect to its second parameter.

Theorem 1. *Consider the SEGPAR variant of our task model. If a task set* τ *is global-EDF unschedulable on* m *processors of speed* s *then for each* σ*,* $\sigma \geq \delta^{\max}$*, there is a time interval of duration* $t \geq 0$ *such that:*

$$\sum_{\mathrm{taskidx}=1}^{n} \mathrm{ff-dbf}(\tau_{\mathrm{taskidx}}, t, \frac{\sigma}{s}, s) > (m - (m-1) * \frac{\sigma}{s}) * t * s \tag{15}$$

Proof. The proof is by contradiction. Suppose that Theorem 1 is false. Then there must exist a task set τ that is global-EDF unschedulable in the SEGPAR-variant on m processors of speed s and there exist a z, $z \geq \delta^{\max}$, such that for each time interval of duration $t \geq 0$ it holds:

$$\sum_{\mathrm{taskidx}=1}^{n} \mathrm{ff-dbf}(\tau_{\mathrm{taskidx}}, t, \frac{z}{s}, s) \leq (m - (m-1) * \frac{z}{s}) * t * s \tag{16}$$

Hence, for this task set τ, there is a scenario (assignment of arrival times of jobs and actual execution times of segments and their potential splitting into subsegments) where a deadline is missed by global-EDF in the SEGPAR-variant on m processors of speed s and there exist a z, $z \geq \delta^{\max}$, such that for each time interval of duration $t \geq 0$ it holds:

$$\sum_{\mathrm{taskidx}=1}^{n} \mathrm{ff-dbf}(\tau_{\mathrm{taskidx}}, t, \frac{z}{s}, s) \leq (m - (m-1) * \frac{z}{s}) * t * s \tag{17}$$

Let t_0 denote the first instant at which a deadline miss occurs. Let JOB_1 denote the job that misses a deadline at t_0. If there are more than one job that missed a deadline at time t_0 then let JOB_1 denote the highest priority job among the jobs that missed a deadline at time t_0. (Tie-breaking rule of global-EDF determines that.) Let us delete all jobs with lower priority than JOB_1. Since these jobs do not impact the timeliness of JOB_1, it still holds that JOB_1 misses a deadline. Also, observe that deleting these jobs does not impact $\delta^{\max}$.

Let t_1 denote job JOB_1's arrival time. Since (i) we consider the SEGPAR model and (ii) a job can only be prevented from executing entirely if all m processors are occupied executing jobs of higher priority, we can rearrange the

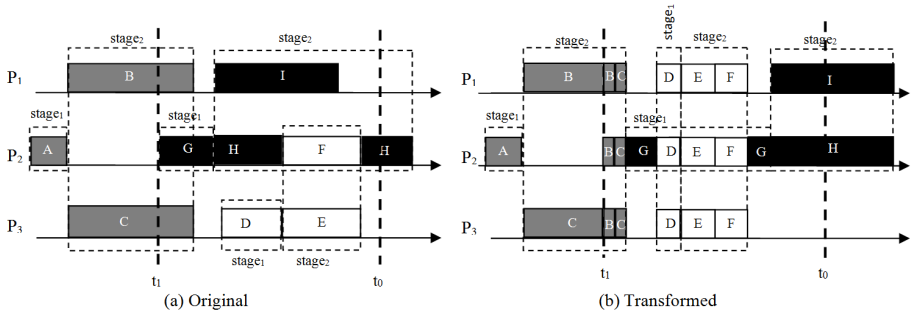


Fig. 2. Packed SEGPAR Execution obtained from transformation

execution of higher-priority jobs such that either (C1) they execute on either all the m processors simultaneously or (C2) they do not execute at all. We call this rearrangement a Packed SEGPAR execution. One way to arrange this is by splitting each segment into m equal parts and filling up all processors with each part of a segment. This can be seen in Figure 2.

Because at each instant, the schedule satisfies either (C1) or (C2), we say that the schedule in $[t_1,t_0)$, satisfies the Packed SEGPAR execution property. Note that with this rearrangement, there is no instant where a segment of JOB_1 executes and a job of a higher-priority than JOB_1 executes.

Hence, we have that there is a task set τ, for which there is a scenario (assignment of arrival times of jobs and actual execution times of segments and their potential splitting into subsegments) where a deadline is missed by global-EDF in the SEGPAR-variant on m processors of speed s and there exist a z, $z \geq \delta^{\max}$, such that for each time interval of duration $t \geq 0$ it holds:

$$\sum_{\text{taskidx}=1}^{n} \text{ff} - \text{dbf}(\tau_{\text{taskidx}}, t, \frac{z}{s}, s) \leq (m - (m-1) * \frac{z}{s}) * t * s \tag{18}$$

and where the Packed SEGPAR execution property holds for the schedule $[t_1,t_0)$, where t_0 is the earliest time of a deadline miss and t_1 is the arrival time of the job that missed a deadline at time t_0 and there are no jobs with deadline greater than t_0.

Since (by definition) job JOB_1 misses its deadline, it follows that the number of units of alone elapsed execution performed by JOB_1 in the time interval $[t_1, t_0]$ is strictly less than $\tau_{\text{task(JOB1)}}$. Since, $z \geq \delta^{\max} \geq \delta_{\text{task(JOB1)}} = \eta_{\text{task(JOB1)}} / D_{\text{task(JOB1)}}$ and this can be rewritten as: $z^* D_{\text{task(JOB1)}} \geq \eta_{\text{task(JOB1)}}$ and since $D_{\text{task(JOB1)}} = t_0 - t_1$, we have: $z^*(t_0\text{-}t_1) \geq \eta_{\text{task(JOB1)}}$. Knowing that the number of units of alone elapsed execution performed by JOB_1 in the time interval $[t_1, t_0]$ is strictly less than $\eta_{\text{task(JOB1)}}$ and that $z^*(t_0\text{-}t_1) \geq \eta_{\text{task(JOB1)}}$ we obtain that the number of units of alone elapsed execution performed by JOB_1 in the time interval $[t_1, t_0]$ is strictly less than $z^*(t_0\text{-}t_1)$.

We generate a sequence of jobs with index k denoting the job in the sequence with the earliest arrival time as follow:

for i := 2, 3, 4,. . . **do**
  let JOB_i denote a job that
    arrives at some time instant before t_{i-1}; let t_i denote the arrival time of JOB_i.
    has a deadline after t_{i-1};
    has not completed execution by t_{i-1}; and
    performed strictly less than $z^*(t_{i-1}- t_i)$, alone elapsed units of execution in the time interval $[t_i, t_{i-1}]$, that is, executed during $[t_i, t_{i-1}]$ for strictly less than $z^*(t_{i-1}- t_i)/s$ time units and during this time, no other jobs executed.
  if there is no such job **then** $k := i$ - 1 break (out of the loop) **end if**
end for

Let L denote the length of the interval $[t_k, t_0)$, that is, $L{=}t_0{-}t_k$. For each i, $1 \leq i \leq k$, let W_i denote the total amount of execution that occurs over the interval $[t_i,t_{i-1})$. For each i, $1 \leq i \leq k$, for each $1 \leq \text{taskidx} \leq n$, let $W_{i,\text{taskidx}}$ denote the number of units of execution that a job of task $\tau_{taskidx}$ performs in the time interval $[t_i,t_{i-1})$. Note that W_i and WJ are different; W_i refers to a time interval for a scenario when a deadline miss has occurred but WJ is a function computed on static task parameters.

Lemma 1.1. $\forall \text{taskidx} \in [1, n] : \text{ff} - \text{dbf}(\tau_{taskidx}, L, z/s, s) \geq \sum_{i=1}^{k} W_{i,taskidx}$.

Proof: Recall that for all jobs that execute in $[t_k,t_0)$ have deadlines at t_0 or earlier and hence all jobs that perform work during $[t_k,t_0)$ have deadlines at t_0 or earlier. Let us consider different cases:

Case 1: There is no job of task τ_{taskidx} such that the job arrives before t_k.
For this case, there are at most $\lfloor L/T_{\text{taskidx}} \rfloor$ jobs of τ_{taskidx} with arrival time in $[t_k,t_0)$ and deadline in $[t_k,t_0)$. Hence, we have: $\lfloor L/T_{\text{taskidx}} \rfloor * C_{\text{taskidx}} \geq \sum_{i=1}^{k} W_{i,taskidx}$. Applying the definition of $\text{ff} - \text{dbf}$ on the left-hand side of this inequality yields: $\text{ff} - \text{dbf}(\tau_{taskidx}, L, z/s, s) \geq \sum_{i=1}^{k} W_{i,taskidx}$. End of Case 1.

Case 2: There is a job of task τ_{taskidx} such that the job arrives before t_k.

Case 2.1: Of those jobs of task τ_{taskidx} with arrival time before t_k, none of them have deadlines in $[t_k,t_0)$.
With the same reasoning as in Case 1, we obtain that $\text{ff} - \text{dbf}(\tau_{taskidx}, L, z/s, s) \geq \sum_{i=1}^{k} W_{i,taskidx}$. End of Case 2.1.

Case 2.2: Of those jobs of task τ_{taskidx} with arrival time before t_k, at least one of them has its deadline in $[t_k,t_0)$.
Since we consider constrained-deadline sporadic tasks, it holds that there is exactly one job of task τ_{taskidx} such that the job has arrival time before t_k and deadlines in $[t_k,t_0)$. Let $\text{LATEJ}_{\text{taskidx}}$ denote this job.

Case 2.2.1: The job $\text{LATEJ}_{\text{taskidx}}$ has finished execution at time t_k.
With the same reasoning as in Case 1, we obtain that $\text{ff} - \text{dbf}(\tau_{taskidx}, L, z/s, s) \geq \sum_{i=1}^{k} W_{i,taskidx}$. End of Case 2.2.1.

Case 2.2.2: The job $\text{LATEJ}_{\text{taskidx}}$ has not finished execution at time t_k.
Let $\text{ENTIRE}_{\text{taskidx}}$ denote the set of jobs of task τ_{taskidx} with arrival time in

$[t_k,t_0)$ and deadline in $[t_k,t_0)$. Clearly, $\text{LATEJ}_{\text{taskidx}}$ is not in $\text{ENTIRE}_{\text{taskidx}}$. Let $\text{WENTIRE}_{\text{taskidx}}$ denote the units of execution jobs in $\text{ENTIRE}_{\text{taskidx}}$ perform in $[t_k,t_0)$. Let $\text{WLATEJ}_{\text{taskidx}}$ denote the units of execution that jobs in $\text{LATEJ}_{\text{taskidx}}$ perform in $[t_k,t_0)$. Clearly, $\text{WENTIRE}_{\text{taskidx}} \leq |\text{ENTIRE}_{\text{taskidx}}| * \text{C}_{\text{taskidx}}$ and $\text{WLATEJ}_{\text{taskidx}} \leq \text{C}_{\text{taskidx}}$. Also, clearly, we have: $\sum_{i=1}^{k} W_{i,taskidx} = \text{WLATEJ}_{\text{taskidx}} + \text{WENTIRE}_{\text{taskidx}}$.

Case 2.2.2.1: $|\text{ENTIRE}_{\text{taskidx}}| < \text{q}_{\text{taskidx}}(\text{L})$
From this case, we obtain that $|\text{ENTIRE}_{\text{taskidx}}| \leq \text{q}_{\text{taskidx}}(\text{L})$ -1 and using it gives us that $\sum_{i=1}^{k} W_{i,taskidx} \leq (\text{q}_{\text{taskidx}}(\text{L})-1)*\text{C}_{\text{taskidx}}+\text{C}_{\text{taskidx}} = (\text{q}_{\text{taskidx}}(\text{L}))*\text{C}_{\text{taskidx}}$. Rewriting and relaxing gives us: $\sum_{i=1}^{k} W_{i,taskidx} \leq \text{ff} - \text{dbf}(\tau_{taskidx}, L, z/s, s)$. End of Case 2.2.2.1.

Case 2.2.2.2: $|\text{ENTIRE}_{\text{taskidx}}| \geq \text{q}_{\text{taskidx}}(\text{L})$
It is easy to see that it is impossible for $|\text{ENTIRE}_{\text{taskidx}}| > \text{q}_{\text{taskidx}}(\text{L})$. Hence we obtain that: $|\text{ENTIRE}_{\text{taskidx}}| = \text{q}_{\text{taskidx}}(\text{L})$. Let $\text{ALATEJ}_{\text{taskidx}}$ denote the arrival time of $\text{LATEJ}_{\text{taskidx}}$. Let $\text{rLATEJ}_{\text{taskidx}}$ be the time from t_k to the deadline of $\text{LATEJ}_{\text{taskidx}}$. Formally, this is expressed as: $\text{rLATEJ}_{\text{taskidx}} = \text{ALATEJ}_{\text{taskidx}} + \text{D}_{\text{taskidx}} - t_k$. From $|\text{ENTIRE}_{\text{taskidx}}| = \text{q}_{\text{taskidx}}(\text{L})$, it follows from the definition of $q_i(L)$ and $r_i(L)$ that $\text{rLATEJ}_{\text{taskidx}} \leq r_{\text{taskidx}}(L)$. Note that before time t_k, job $\text{LATEJ}_{\text{taskidx}}$ performed at least $z^*(\text{C}_{\text{taskidx}} - \text{rLATEJ}_{\text{taskidx}})/s$ alone elapsed units of execution (otherwise $\text{LATEJ}_{\text{taskidx}}$ would have been included in the sequence generated by the algorithm above.). Also, note that because of Case 2.2.2, we have that job $\text{LATEJ}_{\text{taskidx}}$ has not yet finished execution at time t_k. Hence, the assumptions in Lemma 8 are fulfilled. This gives us that it holds that job $\text{LATEJ}_{\text{taskidx}}$ performs at most: $\text{C}_{\text{taskidx}}\text{-WJ}(\tau_{taskidx}, z^*(\text{D}_{\text{taskidx}} - \text{r}_{\text{taskidx}}(\text{L}))/s, s)$ units of execution in the time interval $[t_k,\infty)$. Since the time interval $[t_k, t_0)$ is in the time interval $[t_k,\infty)$, it holds that that job $\text{LATEJ}_{\text{taskidx}}$ performs at most: $\text{C}_{\text{taskidx}}\text{-WJ}(\tau_{taskidx}, z^*(\text{D}_{\text{taskidx}} - \text{r}_{\text{taskidx}}(\text{L}))/s, s)$ units of execution in the time interval $[t_k, t_0)$. Observing that there are $\text{q}_{\text{taskidx}}(\text{L})$ jobs of task τ_{taskidx}, each executing at most $\text{C}_{\text{taskidx}}$ units of execution in time interval $[t_k,t_0)$ we obtain task that τ_{taskidx} performs at most $\text{q}_{\text{taskidx}}(\text{L}) * \text{C}_{\text{taskidx}} + \text{C}_{\text{taskidx}}\text{-WJ}(\tau_{taskidx}, z^*(\text{D}_{\text{taskidx}} - \text{r}_{\text{taskidx}}(\text{L}))/s, s)$ units of execution during the time interval $[t_k, t_0]$. End of Case 2.2.2.2.

Hence, for each of the cases, we obtain that the lemma is true.

Lemma 1.2. $\sum_{\text{taskidx}=1}^{n} \text{ff} - \text{dbf}(\tau_{taskidx}, L, z/s, s) \geq \sum_{i=1}^{k} W_i$.

Proof: Follows from adding the inequalities of Lemma 1.1 and observing that $\sum_{\text{taskidx}=1}^{n} \sum_{i=1}^{k} W_{i,\text{taskidx}} = \sum_{i=1}^{k} W_i$

Lemma 1.3. For each i, $1 \leq i \leq k$, it holds that: $W_i > (m - (m-1) * z/s) * (t_{i-1} - t_i) * s$.

Proof: Let x denote the total length of the time intervals over $[t_i,t_{i-1}]$ during which at least one segment of JOB_i executes. By choice of job JOB_i, it is the case that: $x < (t_i\text{-}t_{i-1})*z/s$ By choice of job JOB_i, it has not completed execution

by time instant t_{i-1}. Hence, over $[t_i,t_{i-1})$, all m processors must be executing whenever job JOB_i does not execute any segment at all. Hence during $[t_i,t_{i-1})$, the number of units of execution performed by jobs other than JOB_i is at least: $m^*(\ t_{i-1}\ -\ t_i\ -\ x)^*s$. During the x time units that job JOB_i executes, we know that at least one processor was busy. Hence during this time, at least x^*s units of execution is performed. Adding it up gives us that: $W_i \geq m^*(t_{i-1}\ -\ t_i\ -\ x)^*s + x^*s = m^*(t_{i-1}\ -\ t_i)^*s\ -\ (m\text{-}1)^*x^*s > m^*(t_{i-1}\ -\ t_i)^*s\ -\ (m\text{-}1)^*(t_i\text{-}t_{i-1})^*(z/s)^*s = (t_{i-1}\ -\ t_i)^*s^*(m\ -\ (m\text{-}1)^*(z/s)) = (m\text{-}(m\text{-}1)^*\ (z/s))^*(t_{i-1}\ -\ t_i)^*s$. Hence, Lemma 1.3 is proved.

Using Lemma 1.3 and rewriting gives us:

$$\sum_{i=1}^{k} W_i > \ \sum_{i=1}^{k}(m-(m-1)*z/s)*(t_{i-1}-t_i)*s = (m-(m-1)*z/s)*s*(\sum_{i=1}^{k}(t_{i-1}-t_i)) = (m-(m-1)*z/s)*s*L \tag{19}$$

Combining Equation 19 with Lemma 1.2 gives us:

$$\sum_{\text{taskidx}=1}^{n} \text{ff}-\text{dbf}(\tau_{taskidx}, L, z/s, s) > (m-(m-1)*z/s)*s*L \tag{20}$$

But this contradicts Equation 18. Hence Theorem 1 is correct.

Theorem 2. *Consider a task set τ with the SEGPAR-variant model scheduled by global-EDF on a computer platform with m processors each of speed s. If there is a σ, $\sigma \geq \delta^{\max}$, such that for each time interval of duration $t \geq 0$ it holds that:*

$$\sum_{i=1}^{n} \text{ff}-\text{dbf}(\tau_i, t, \frac{\sigma}{s}, s) \leq (m-(m-1)*\frac{\sigma}{s})*t*s \tag{21}$$

then task set τ is schedulable.

Proof. Follows from Theorem 1.

Theorem 3. *Consider a task set τ with the SEGSEQ-variant model scheduled by global-EDF on a computer platform with m processors each of speed s. If there is a σ, $\sigma \geq \delta^{\max}$, such that for each time interval of duration $t \geq 0$ it holds that:*

$$\sum_{i=1}^{n} \text{ff}-\text{dbf}(\tau_i, t, \frac{\sigma}{s}, s) \leq (m-(m-1)*\frac{\sigma}{s})*t*s \tag{22}$$

then task set τ is schedulable.

Proof. Follows from Theorem 2 and Lemma 5.

5 Proving the Resource Augmentation Bound

Theorem 4. *Consider a task set τ. Let Q denote 2-1/m. If τ is feasible on m processors of speed s/Q then τ is global-EDF schedulable according to the schedulability test of Theorem 3 on m processors of speed s.*

Proof. The proof is by contradiction. Suppose that the theorem would be false. Then there exist a task set τ such that τ is feasible on m processor of speed s/Q but global-EDF is unschedulable according to the schedulability test of Theorem 3 on m processors of speed s. Because of feasibility, Lemma 1 give us:

$$\delta^{\max} \leq \frac{s}{Q} \tag{23}$$

Because of feasibility, it holds that all deadlines are met for the ASAP-CMAX assignment of arrival times and execution requirements. Hence, for all t such that $t > 0$, it holds that:

$$\sum_{i=1}^{n} \text{ff} - \text{dbf}(\tau_i, t, 1, \frac{s}{Q}) \leq m * \frac{s}{Q} * t \tag{24}$$

Because Theorem 3 could not guarantee schedulability of global-EDF, we have that for each σ, $\sigma \geq \delta^{\max}$, there is a $t > 0$ such that:

$$\sum_{i=1}^{n} \text{ff} - \text{dbf}(\tau_i, t, \frac{\sigma}{s}, s) > (m - (m-1) * \frac{\sigma}{s}) * t * s \tag{25}$$

Because of Equation 23, we can apply $\sigma = s/Q$ on Equation 25 and using Lemma 4 yields that there is a $t > 0$ such that:

$$\sum_{i=1}^{n} \text{ff} - \text{dbf}(\tau_i, t, 1, \frac{s}{Q}) > (m - (m-1) * \frac{1}{Q}) * t * s \tag{26}$$

Combing Equation 26 and Equation 24 yields:

$$(m - (m-1) * \frac{1}{Q}) * t * s < m * \frac{s}{Q} * t \tag{27}$$

Rewriting yields: Q<2-1/m. But this contradicts the definition of Q. Hence, it holds that the theorem is true.

6 Conclusions

We have presented a new algorithm for performing schedulability analysis of parallel tasks scheduled by global-EDF. This algorithm assumes the constrained-deadline sporadic model which is at least as general as the ones used in previous work and it has better resource augmentation bound than previously known schedulability analyses.

The formulation of our new schedulability analysis in this paper states that a condition is tested for all t and for all σ. That is an infinite number of inequalities to check and it is therefore tempting to believe that the method is impractical. One can use techniques in previous research [12] and our function nhcWJ(τ_i, t, s) to show that the same result can be computed by checking only a finite number of inequalities though.

7 Legal Notices

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References

1. Lakshmanan, K., Kato, S., Rajkumar, R.: Scheduling Parallel Real-Time Tasks on Multi-core Processors. In: RTSS 2010 (2010)
2. Fauberteauy, F., Midonnety, S., Qamhiehy, M.: Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems. SIGBED Review (2011)
3. Saifullah, A., Agrawal, K., Lu, C., Gill, C.: Multi-core Real-Time Scheduling for Generalized Parallel Task Models. In: RTSS 2011 (2011)
4. Cong, L., Anderson, J.H.: Supporting Soft Real-Time DAG-Based Systems on Multiprocessors with No Utilization Loss. In: RTSS 2010 (2010)
5. Lupu, I., Goossens, J.: Scheduling of Hard Real-Time Multi-Thread Periodic Tasks. In: RTNS 2011 (2011)
6. Kato, S., Ishikawa, Y.: Gang EDF Scheduling of Parallel Task Systems. In: RTSS 2009 (2009)
7. Jayachandran, P., Abdelzaher, T.: Reduction-based schedulability analysis of distributed systems with cycles in the task graph. Journal Real-Time Systems (2010)
8. Goddard, S.: On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs. PhD thesis (1998)
9. Gerber, R., Hong, S., Saksena, M.: Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In: RTSS 1994 (1994)
10. Audsley, N.C., Burns, A., Richardson, M.F., Wellings, A.J.: Data Consistency In Hard Real-Time Systems. Informatica (1993)
11. Philips, C., Stein, C., Torng, E., Wein, J.: Optimal time-critical scheduling via resource augmentation. In: STOC 1997 (1997)
12. Baruah, S.K., Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S.: Improved multiprocessor global schedulability analysis. In: Real-Time Systems (2010)

Range Queries in Non-blocking k-ary Search Trees

Trevor Brown[1] and Hillel Avni[2]

[1] Dept. of Computer Science, University of Toronto
tabrown@cs.toronto.edu
[2] Dept. of Computer Science, Tel-Aviv University
hillel.avni@gmail.com

Abstract. We present a linearizable, non-blocking k-ary search tree (k-ST) that supports fast searches and range queries. Our algorithm uses single-word compare-and-swap (CAS) operations, and tolerates any number of crash failures. Performance experiments show that, for workloads containing small range queries, our k-ST significantly outperforms other algorithms which support these operations, and rivals the performance of a leading concurrent skip-list, which provides range queries that cannot always be linearized.

1 Introduction and Related Work

The ordered set abstract data type (ADT) represents a set of keys drawn from an ordered universe, and supports three operations: INSERT(key), DELETE(key), and FIND(key). We add to these an operation RANGEQUERY(a, b), where $a \leq b$, which returns all keys in the closed interval $[a, b]$. This is useful for various database applications.

Perhaps the most straightforward way to implement this ADT is to employ software transactional memory (STM) [13]. STM allows a programmer to specify that certain blocks of code should be executed atomically, relative to one another. Recently, several fast binary search tree algorithms using STM have been introduced [7,2]. Although they offer good performance for INSERTs, DELETEs and FINDs, they achieve this performance, in part, by carefully limiting the amount of data protected by their transactions. However, since computing a range query means protecting all keys in the range from change during a transaction, STM techniques presently involve too much overhead to be applied to this problem.

Another simple approach is to lock the entire data structure, and compute a range query while it is locked. One can refine this technique by using a more fine-grained locking scheme, so that only part of the data structure needs to be locked to perform an update or compute a range query. For instance, in leaf-oriented trees, where all keys in the set are stored in the leaves of the tree, updates to the tree can be performed by local modifications close to the leaves. Therefore, it is often sufficient to lock only the last couple of nodes on the path to a leaf, rather than the entire path from the root. However, as was the case for STM,

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 31–45, 2012.

a range query can only be computed if every key in the range is protected, so typically every node containing a key in the range must be locked.

Persistent data structures [11] offer another approach. The nodes in a persistent data structure are immutable, so updates create new nodes, rather than modifying existing ones. In the case of a persistent tree, a change to one node involves recreating the entire path from the root to that node. After the change, the data structure has a new root, and the old version of the data structure remains accessible (via the old root). Hence, it is trivial to implement range queries in a persistent tree. However, significant downsides include contention at the root, and the duplication of many nodes during updates.

Brown and Helga [9] presented a k-ST in which each internal node has k children, and each leaf contains up to $k-1$ keys. For large values of k, this translates into an algorithm which minimizes cache misses and benefits from processor pre-fetching mechanisms. In some ways, the k-ST is similar to a persistent data structure. The keys of a node are immutable, but the child pointers of a node can be changed. The structure is also leaf-oriented, meaning that all keys in the set are stored in the leaves of the tree. Hence, when an update adds or removes a key from the set, the leaf into which the key should be inserted, or from which the key should be deleted, is simply replaced by a new leaf. Since the old leaf's keys remains unmodified, range queries using this leaf need only check that it has not been replaced by another leaf to determine that its keys are all in the data structure. To make this more efficient, we modify this structure by adding a *dirty*-bit to each leaf, which is set just before the leaf is replaced.

Braginsky and Petrank [5] presented a non-blocking B$^+$tree, another search tree of large arity. However, whereas the k-ST's nodes have immutable keys, the nodes of Braginsky's B$^+$tree do not. Hence, our technique for performing range queries cannot be efficiently applied to their data structure.

Snapshots offer another approach for implementing range queries. If we could quickly take a snapshot of the data structure, then we could simply perform a sequential range query on the result. The snapshot object is a vector V of data elements supporting two operations: UPDATE(i, val), which atomically sets V_i to val, and SCAN, which atomically reads and returns all of the elements of V. SCAN can be implemented by repeatedly performing a pair of COLLECTs (which read each element of V in sequence and return a new vector containing the values it read) until the results of the two COLLECTs are equal [1]. Attiya, et al. [3] introduced *partial snapshots*, offering a modified SCAN$(i_1, i_2, ..., i_n)$ operation which operates on a subset of the elements of V. Their construction requires both CAS and fetch-and-add.

Recently, two high-performance tree structures offering $O(1)$ time snapshots have been published. Both structures use a lazy copy-on-write scheme that we now describe.

Ctrie is a non-blocking concurrent hash trie due to Prokopec et al. [12]. Keys are hashed, and the bits of these hashes are used to navigate the trie. To facilitate the computation of fast snapshots, a sequence number is associated with each node in the data structure. Each time a snapshot is taken, the root is copied and

its sequence number is incremented. An update or search in the trie reads this sequence number *seq* when it starts and, while traversing the trie, it duplicates each node whose sequence number is less than *seq*. The update then performs a variant of a double-compare-single-swap operation to atomically change a pointer while ensuring the root's current sequence number matches *seq*. Because keys are ordered by their hashes in the trie, it is hard to use Ctrie to efficiently implement range queries. To do so, one must iterate over all keys in the snapshot.

The second structure, Snap, is a lock-based AVL tree due to Bronson et al. [6]. Whereas Ctrie added sequence numbers, Snap *marks* each node to indicate that it should no longer by modified. Updates are organized into *epochs*, with each epoch represented by an object in memory containing a count of the number of active updates belonging to that epoch. A snapshot marks the root node, ends the current epoch, and blocks further updates from starting until all updates in the current epoch finish. Once updates are no longer blocked, they copy and mark each node they see whose parent is marked. Like Ctrie, this pushes work from snapshots onto subsequent updates. If these snapshots are used to compute small range queries, this may result in excessive duplication of unrelated parts of the structure.

If we view shared memory as a contiguous array, then our range queries are similar to *partial snapshots*. We implement two optimizations specific to our data structure. First, when we traverse the tree to perform our initial Collect, we need only read a pointer to each *leaf* that contains a key in the desired range (rather than reading each key). This is a significant optimization when k is large, e.g., 64. Second, instead of performing a second Collect (which would involve saving the parent of each leaf or traversing the tree again), we can simply check the *dirty*-bit of each node read by the first Collect. As a further optimization, range queries can return a sequence of leaves, rather than copying their keys into an auxiliary structure.

Contributions of this work:

- We present a new, provably correct data structure, and demonstrate experimentally that, for two very different sizes of range queries, it significantly outperforms data structures offering $O(1)$ time snapshots. In many cases, it even outperforms a non-blocking skip-list, whose range queries cannot always be linearized.
- We contribute to a better understanding of the performance limitations of the $O(1)$ time snapshot technique for this application.

The structure of the remainder of this paper is as follows. In Sec. 2, we describe the data structure, how updates are performed, and our technique for computing partial snapshots of the nodes of the tree. We give the details of how range queries are computed from these partial snapshots of nodes in Sec. 3. A sketch of a correctness proof is presented in Sec. 4. (The full version of this paper [8] contains a detailed proof.) Experimental results are presented in Sec. 5. Future work and conclusions are discussed in Sec. 6.

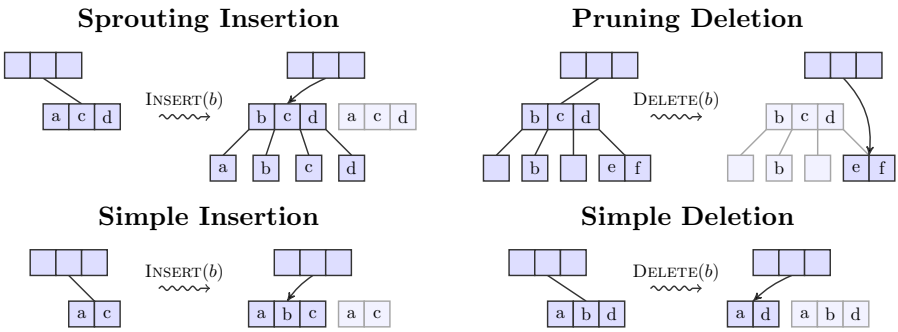


Fig. 1. The four k-ST update operations

2 Basic Operations and Validate

The k-ST is a linearizable, leaf-oriented search tree in which each internal node has k children and each leaf contains up to $k-1$ keys. Its non-blocking operations, Find, Insert and Delete, implement the set ADT. Find(key) returns True if key is in the set, and False otherwise. If key is not already in the set, then Insert(key) adds key and returns True. Otherwise it returns False. If key is in the set, then Delete(key) removes key and returns True. Otherwise it returns False. The k-ST can be extended to implement the dictionary ADT, in which a value is associated with each key (described in the technical report for [9]). Although a leaf-oriented tree occupies more space than a node-oriented tree (since it contains up to twice as many nodes), the expected depth of a key or value is only marginally higher, since more than half of the nodes are leaves. Additionally, the fact that all updates in a leaf-oriented tree can be performed by a local change near the leaves dramatically simplifies the task of proving that updates do not interfere with one another.

Basic *k*-ST Operations. These operations are implemented as in [9]. Find is conceptually simple, and extremely fast. It traverses the tree just as it would in the sequential case. However, concurrent updates can prevent its termination (see [10]). Insert and Delete are each split into two cases, for a total of four update operations: sprouting insertion, simple insertion, pruning deletion and simple deletion (see Fig. 1). An Insert into leaf l will be a sprouting insertion when l already has $k-1$ keys. Since l cannot accommodate any more keys, it is atomically replaced (using CAS) by a newly created sub-tree that contains the original $k-1$ keys, as well as the key being inserted. Otherwise, the Insert will be a simple insertion, which will atomically replace l with a newly created leaf, containing the keys of l, as well as the key being inserted. A Delete from leaf l will be a pruning deletion if l has one key and exactly one non-empty sibling. Otherwise, the Delete will be a simple deletion, which will atomically replace

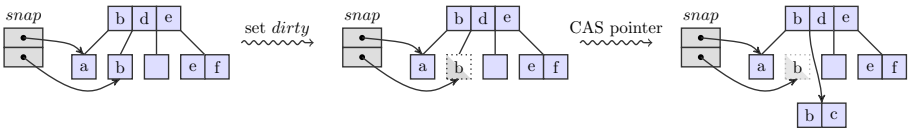


Fig. 2. A VALIDATE is in progress on the leaves containing a and b. Before it can check the *dirty* bits of these leaves at line 2, an update INSERT(c) sets the *dirty* bit of the leaf containing b. After this, the VALIDATE is doomed to return FALSE. The INSERT(c) then changes its child pointer and finishes.

l with a newly created leaf, containing the keys of l, except for the key being deleted. Note that if l has one key, then l will be empty after a simple deletion. With this set of updates, it is easy to see that the keys of a node never change, and that internal nodes always have $k-1$ keys and k children.

The non-blocking progress property is guaranteed by a helping scheme that is generalized from the work of Ellen et al. [10], and is somewhat similar to the cooperative technique of Barnes [4]. Every time a process p performs an update U, it stores information in the nodes it will modify to allow any other process to perform U on p's behalf. When a process is prevented from making progress by another operation, it helps that operation complete, and then retries its own.

Validate. We include a VALIDATE subroutine, which is used in the RANGEQUERY algorithm in Sec. 3. It is closely related to a partial snapshot operation when memory is viewed as an array of locations. VALIDATE takes as argument a sequence of pointers to leaves which were reached by following child pointers from the root.

```
1  VALIDATE(p_1, p_2, ..., p_n) {
2    check the dirty bit of each leaf pointed to by an element of {p_1, ..., p_n}
3    if any dirty bit is TRUE {
4      return FALSE
5    } else {
6      return TRUE
7  } }
```

The *dirty* field is a single bit included in each leaf l that is initially FALSE, and is irrevocably set to TRUE just before any CAS that will remove or replace l. (See Fig. 2 for an illustration of how a *dirty* bit causes a snapshot to retry.) Thus, a FALSE *dirty* bit in a leaf l that was visited in the tree implies that l is in the tree. Hence, if each $l \in \{p_1, ..., p_n\}$ satisfies $l.dirty =$ FALSE, then we know each l was in the tree when the its *dirty* bit was read at line 2 by the algorithm. Furthermore, since the keys of a node never change, l's FALSE *dirty* bit means that all of the keys in l are in the tree. Hence, if VALIDATE returns TRUE, all of the keys contained in the leaves in the snapshot are in the tree when the VALIDATE began. If VALIDATE returns FALSE, then there is a leaf that has been or is in the process of being replaced.

```
8   type Node {
9     Key ∪ {∞}  a_1, ..., a_{k-1}
10  }

11  subtype Internal of Node {
12    Node  c_1, ..., c_k
13  }

14  subtype Leaf of Node {
15    boolean dirty
        ▷ (initially FALSE)
16  }

17  RANGEQUERY(Key lo, Key hi) : List of Nodes {
      ▷ Precondition: lo, hi ≠ ∞, and lo ≤ hi
18    List snap := new List()
      ▷ DFS to populate snap with all leaves that could possibly contain a key in [lo, hi]
19    Stack s := new Stack()
20    s.push(root.c_1)
21    while |s| > 0 {
22      Node u := s.pop()
23      if u is a Leaf then do snap.add(u)
        ▷ Determine which children of u to traverse
24      int i := 1
25      while i < k and u.a_i ≤ lo {          ▷ Find first sub-tree that intersects [lo, ∞)
26        i := i + 1
27      }
28      if i = 1 {                            ▷ Boundary case: don't test u.a_0 below
29        s.push(u.c_i)
30        i := i + 1
31      }
32      while i ≤ k and u.a_{i-1} ≤ hi {      ▷ Push sub-trees until all keys are in (hi, ∞)
33        s.push(u.c_i)
34        i := i + 1
35    } }
      ▷ Validate (check the nodes in snap have not changed)
36    if not VALIDATE(snap) then retry (i.e., go back to line 18)
      ▷ Return all leaves in snap that contain some key in range [lo, hi]
37    List result := new List()
38    for each u in snap {
39      if at least one of u's keys is in range [lo, hi] then do result.add(u)
40    }
41    return result
42  }
```

Fig. 3. Abridged type definitions and pseudocode for RANGEQUERY. RANGEQUERY accepts two keys, *lo* and *hi*, as arguments, and returns all leaves that (a) were in the tree at the linearization point, and (b) have a key in the closed interval $[lo, hi]$.

3 Range Queries in a *k*-ST

An abridged description of the type definitions of the data structure and Java-like pseudocode for the RANGEQUERY operation are given in Fig. 3. We borrow the concept of a *reference* type from Java. In this psuedocode, variables of any type $E \notin \{\text{int, boolean}\}$ are references to objects of type E. A reference x is like pointer, but is automatically dereferenced when a field of the object is accessed with the (.) operator, as in: $x.field$ (which means the same as `x->field` in C). References take on the special value NULL when they do not point to any object. (However, no field of any node is ever NULL.)

We now take a high-level tour through the RANGEQUERY algorithm. The algorithm begins by declaring a list *snap* at line 18 to hold pointers to all leaves which may contain a key in $[lo, hi]$. In lines 19-35 the algorithm traverses the

tree, saving pointers in *snap*. It uses a depth-first-search (DFS), implemented with a stack (instead of recursion), except that it may prune some of the children of each node, and avoid pushing them onto the stack. The loop at line 25 prunes those children that are the roots of sub-trees with keys strictly less than lo. The loop at line 32 then pushes children onto the stack until it hits the first child that is the root of a sub-tree with keys strictly greater than hi. Both of these loops use the fact that keys are maintained in increasing order within each node. It follows that all paths that could lead to keys in $[lo, hi]$ are explored, and all terminal leaves on these paths are placed in *snap* at line 23.

RangeQuery then calls Validate (described in the previous section). If this validation is successful, then each element of *snap* that points to a leaf containing at least one key in $[lo, hi]$ is copied into *result* by the loop at lines 38-40. The range query can be modified to return a list of keys instead of nodes simply by changing the final loop (since the keys of a node never change).

4 Correctness

We now provide a proof sketch. The interested reader can find the details of the following results in the full version of this paper [8].

Observation 1. *Apart from child pointers, nodes are never modified. To insert or delete a key,* Insert *and* Delete ***replace*** *affected node(s) with newly created node(s).*

Lemma 2. *If no* Insert *or* Delete *operations are executing, then there are no dirty leaves reachable by following child pointers from root.*

The proof of this lemma is quite laborious, but the intuition is simple. Leaves only become dirty (have their dirty bit set) by an update (Insert or Delete) right before the update executes a *Child CAS*, which changes a child pointer to remove the leaf from the tree.

Next, we prove progress. Since RangeQuery does not write to shared memory, it cannot affect the correctness or progress of Find, Insert or Delete. The proof in [9] still applies, and Find, Insert and Delete are all non-blocking. Hence, it is sufficient to prove that, if a RangeQuery operation is performed, it eventually terminates if no Insert or Delete operations are executing. If a RangeQuery is being performed and no Insert or Delete operations are executing then, by Lemma 2, RangeQuery will eventually encounter no dirty leaf and, hence, will eventually terminate.

Theorem 3. *All operations are non-blocking.*

Definition 4. *At any configuration C, let T_C be the k-ary tree formed by the child references. We define the* ***search path*** *for key a in configuration C to be the unique path in T_C that would be followed by the ordinary sequential k-ST search procedure.*

Definition 5. *We define the* ***range*** *of a leaf u in configuration C to be the set R of keys such that, for any key $a \in R$, u is the terminal node on the search path for a in configuration C. (Consequently, if u is not in the tree, its range is the empty set.)*

To simplify the statements of upcoming results, we also make two more simple definitions. A node is ***in the tree*** if it is reachable by following child pointers from the root. A node is ***initiated*** if it has ever been in the tree.

Lemma 6. *If an initiated leaf is not in the tree, then it is dirty.*

This simple result follows from the fact that any leaf removed from the tree is removed by a *Child CAS*, just prior to which the leaf's dirty bit is set.

Lemma 7. *If a key is inserted into or deleted from the range of a leaf u, and u is in the tree just before the linearization point of the* Insert *or* Delete*, then u is no longer in the tree just after the linearization point.*

This lemma relies heavily on results from [9]. We first prove that a successful update on key a must remove a leaf whose range contained a at the linearization point of a Find which the update invokes. We then prove that this leaf's range must still contain a just before the update is linearized. Finally, we invoke the k-ary search tree property to argue that a can only be in the range of one leaf just before the update is linearized, which implies the result.

Now we can prove the correctness of RangeQuery. We linearize each completed invocation of RangeQuery immediately before performing Validate for the last time.

Theorem 8. *Each invocation of* RangeQuery(lo, hi) *returns a list containing precisely the set of leaves in the tree that have a key in the range $[lo, hi]$ at the time the* RangeQuery *is linearized.*

This final result is proved with the help of Lemma 6, Lemma 7, and the following sub-claims:

(a) In every configuration C, every internal node has exactly k children and satisfies the k-ary search tree property.
(b) The DFS in the first loop of RangeQuery traverses the relevant part of the tree and adds every leaf it visits to *snap*.
(c) The only sub-trees that are not traversed by the DFS are those that cannot ever contain a key in range $[lo, hi]$.

5 Experiments

In this section, we present the results of experiments comparing the performance of our k-ST (for $k = 16, 32, 64$) with Snap, Ctrie, and SL, the non-blocking, randomized skip-list of the Java Foundation Classes. We used the authors' implementations of Snap and Ctrie. Java code for our k-ST is available on-line [8].

All of these data structures implement the dictionary ADT, where INSERT(key) returns FALSE if an element with this key is already in the dictionary. For SL, a RANGEQUERY is performed by executing the method `subSet(lo, true, hi, true)`, which returns a reference to an object permitting iteration over the keys in the structure, restricted to $[lo, hi]$, and then copying each of these keys into an array. This does not involve any sort of snapshot, so SL's range queries are not always linearizable. For Snap, a RANGEQUERY is performed by following the same process as SL: i.e., executing `subSet`, and copying keys into an array. However, unlike SL, iterating over the result of Snap's `subSet` causes a snapshot of the data structure to be taken. Since keys are ordered by their hashed values in Ctrie, it is hard to perform range queries efficiently. Instead, we attempt to provide an approximate lower bound on the computational difficulty of computing RANGEQUERY(lo, hi) for any derivative of Ctrie which uses the same fast snapshot technique. To do this, we simply take a snapshot, then iterate over the first $(hi-lo+1)/2$ keys in the snapshot, and copy each of these keys into an array. We explain below that $(hi - lo + 1)/2$ is the expected number of keys returned by a RANGEQUERY. To ensure a fair comparison with the other data structures, our k-ST's implementation of RANGEQUERY returns an array of keys. If it is allowed to return a list of leaves, its performance improves substantially.

Our experiments were performed on two multi-core systems. The first is a Fujitsu PRIMERGY RX600 S6 with 128GB of RAM and four Intel Xeon E7-4870 processors, each having 10 × 2.4GHz cores, supporting a total of 80 hardware threads (after enabling hyper-threading). The second is a Sun SPARC Enterprise T5240 with 32GB of RAM and two UltraSPARCT2+ processors, each having 8 × 1.2GHz cores, for a total of 128 hardware threads. On both machines, the Sun 64-bit JVM version 1.7.0_3 was run in server mode, with 512MB minimum and maximum heap sizes. We decided on 512MB after performing preliminary experiments to find a heap size which was small enough to regularly trigger garbage collection and large enough to keep standard deviations small. We also ran the full suite of experiments for both 256MB and 15GB heaps. The results were quite similar to those presented below, except that the 256MB heap caused large standard deviations, and the total absence of garbage collection with the 15GB heap slightly favoured Ctrie.

For each experiment in {5i-5d-40r-size10000, 5i-5d-40r-size100, 20i-20d-1r-size10000, 20i-20d-1r-size100}, each algorithm in {KST16, KST32, KST64, Snap, Ctrie, SL}, and each number of threads in $\{4, 8, 16, 32, 64, 128\}$, we ran 3 trials, each performing random operations on keys drawn uniformly randomly from the key range $[0, 10^6)$ for ten seconds. Operations were chosen randomly according to the experiment. Experiment "xi-yd-zr-size$s$" indicates x% probability of a randomly chosen operation to be an INSERT, y% probability of a DELETE, z% probability of RANGEQUERY($r, r + s$), where r is a key drawn uniformly randomly from $[0, 10^6)$, and the remaining $(100 - x - y - z)$% probability of a FIND. Our graphs do not include data for 1 or 2 threads, since the differences between the throughputs of all the algorithms was very small. However, we did include an extra set of trials at 40 threads for the Intel machine, since it has 40

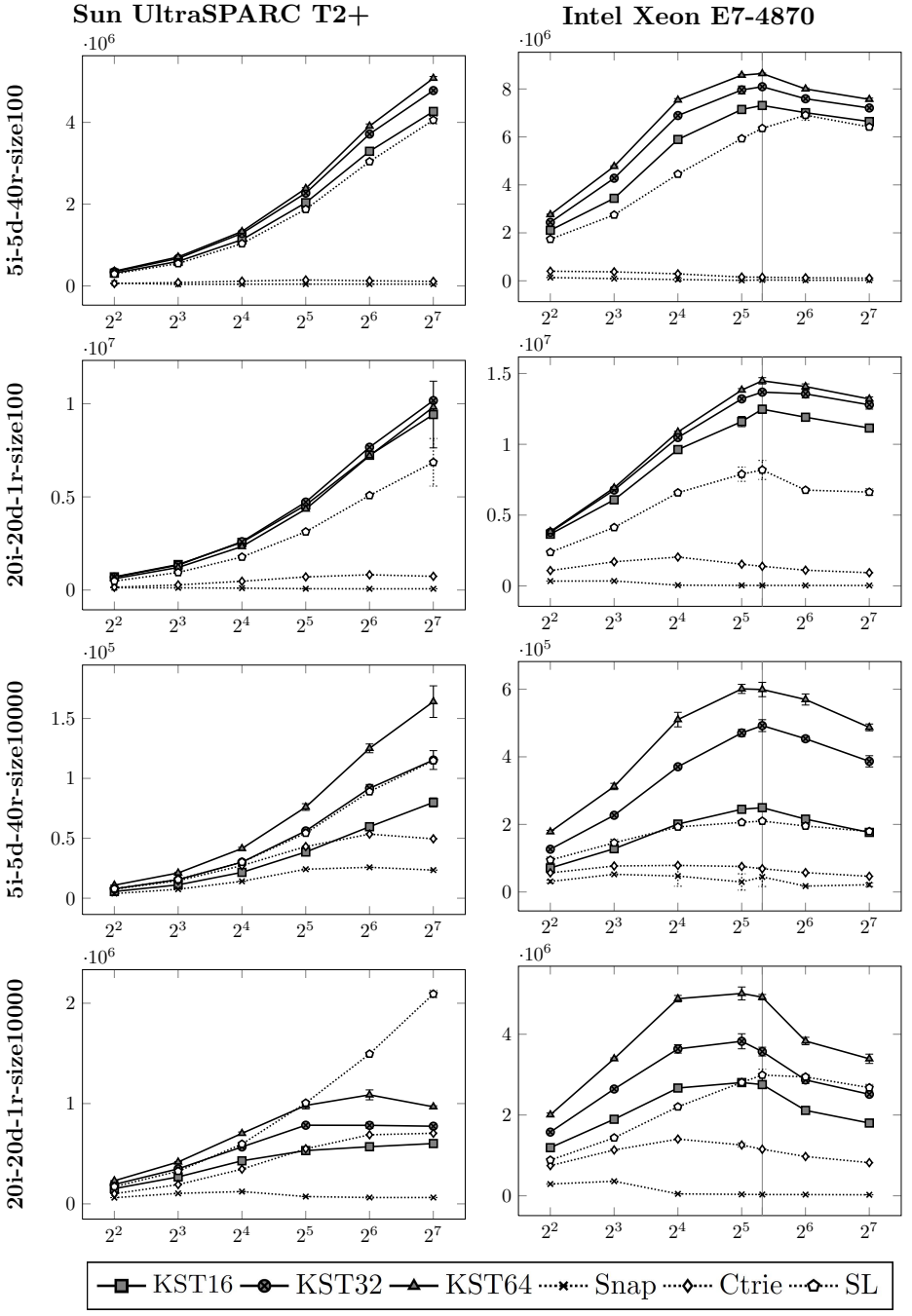


Fig. 4. Experimental results for various operation mixes (in rows) for two machines (in columns). The x-axes show the number of threads executing, and the y-axes show throughput (ops/second). The Intel machine has 40 cores (marked with a vertical bar).

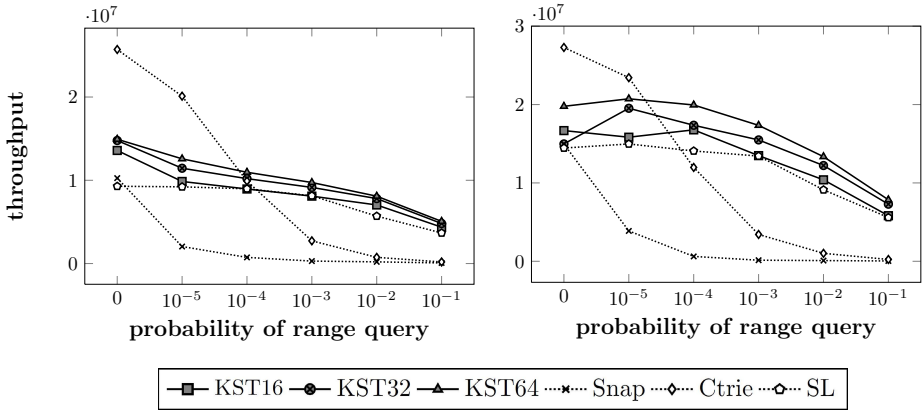


Fig. 5. Sun (left) and Intel (right) results for experiment 5i-5d-?r-100 wherein we vary the probability of range queries. Note: as we describe in Sec. 5, Ctrie is merely performing a partial snapshot, rather than a range query. The Sun machine is running 128 threads, and the Intel machine is running 80 threads.

cores. Each data structure was pre-filled before each trial by performing random INSERT and DELETE operations, each with 50% probability, until it stabilized at approximately half full (500,000 keys). Each data structure was within 5% of 500,000 keys at the beginning and end of each trial. This is expected since, for each of our experiments, at any point in time during a trial, the last update on a particular key has a 50% chance of being an INSERT, in which case it will be in the data structure, and a 50% chance of being a DELETE, in which case it will not. Thus, $(hi - lo + 1)/2$ is the expected number of keys in the data structure that are in $[lo, hi]$. In order to account for the "warm-up" time an application experiences while Java's HotSpot compiler optimizes its running code, we performed a sort of pre-compilation phase before running our experiments. During this pre-compilation phase, for each algorithm, we performed random INSERT and DELETE operations, each with 50% probability, for twenty seconds. Our experiments appear in Fig. 4. Error bars are drawn to represent one standard deviation. A vertical bar is drawn at 40 threads on the graphs for the Intel machine, marking the number of cores in the machine.

Broadly speaking, our experimental results from the Sun machine look similar to those from the Intel machine. If we ignore the results on the Intel machine for thread counts higher than 40 (the number of cores in the machine), then the shapes of the curves and relative orderings of algorithms according to performance are similar between machines. A notable exception to this is SL, which tends to perform worse, relative to the other algorithms, on the Intel machine than on the Sun machine. This is likely due to architectural differences between the two platforms. Another Intel Xeon system, a 32-core X7560, has also shown the same scaling problems for SL (see [9] technical report).

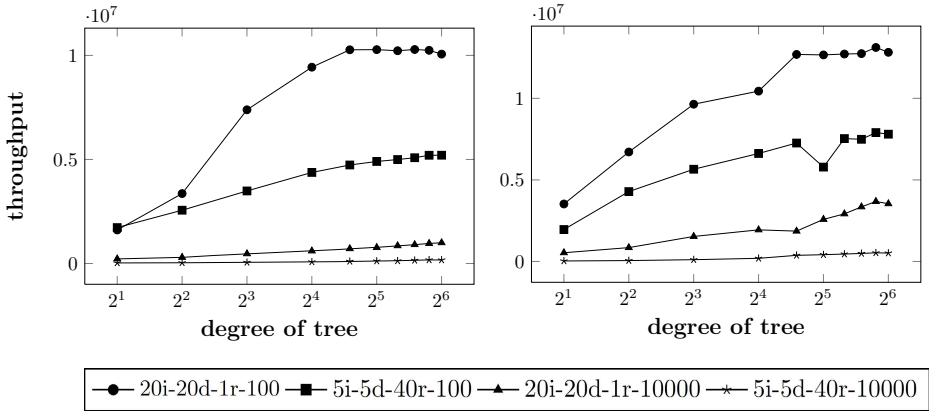


Fig. 6. Sun (left) and Intel (right) results showing the performance of the k-ST for many values of k, and for various operation mixes. The Sun machine is running 128 threads, and the Intel machine is running 80 threads.

We now discuss similarities between the experiments 5i-5d-40r-size100 and 20i-20d-1r-size100, which involve small range queries, before delving into their details. The results from these experiments are highly similar. In both experiments, all k-STs outperform Snap and Ctrie by a wide margin. Each range query causes Snap (Ctrie) to take a snapshot, forcing all updates (updates and queries) to duplicate nodes continually. Similarly, SL significantly outperforms Snap and Ctrie, but it does not exceed the performance of any k-ST algorithm. Ctrie always outperforms Snap, but the difference is often negligible. In these experiments, at each thread count, the k-ST algorithms either perform comparably, or are ordered KST16, KST32 and KST64, from lowest to highest performance.

Experiment 5i-5d-40r-size100 represents the case of few updates and many small range queries. The k-ST algorithms perform extremely well in this case. On the Sun machine (Intel machine), KST16 has 5.2 times (5.3 times) the throughput of Ctrie at four threads, and 38 times (61 times) the throughput at 128 threads. The large proportion of range queries in this case allows SL, with its extremely fast, non-linearizable RangeQuery operation, to nearly match the performance of KST16 on the Sun machine.

Experiment 20i-20d-1r-size100 represents the case of many updates and few small range queries. The k-STs are also strong performers in this case. On the Sun machine (Intel machine), KST16 has 4.7 times (3.4 times) the throughput of Ctrie at four threads, and 13 times (12 times) the throughput at 128 threads. In contrast to experiment 5i-5d-40r-size100, since there are few range queries, KST32 and KST64 do not perform significantly better than KST16. Similarly, with few range queries, the simplicity of SL's non-linearizable RangeQuery operation does not get a chance to significantly affect SL's throughput. Compared to experiment 5i-5d-40r-size100, the throughput of SL significantly decreases, relative to the k-ST algorithms. Whereas KST16 only outperforms SL by 5.2% at

128 threads on the Sun machine in experiment 5i-5d-40r-size100, it outperforms SL by 37% in experiment 20i-20d-1r-size100.

We now discuss similarities between the experiments 5i-5d-40r-size10000 and 20i-20d-1r-size10000, which involve large range queries. In these experiments, at each thread count, the k-ST algorithms are ordered KST16, KST32 and KST64, from lowest to highest performance. Since the size of its range queries is fairly large (5,000 keys), Ctrie's fast snapshot begins to pay off and, for most thread counts, its performance rivals that of KST16 or KST32 on the Sun machine. However, on the Intel machine, it does not perform nearly as well, and its throughput is significantly lower than that of SL and the k-ST algorithms. Ctrie always outperforms Snap, and often does so by a wide margin. SL performs especially well in these experiments, no doubt due to the fact that its non-linearizable RangeQuery operation is unaffected by concurrent updates.

Experiment 5i-5d-40r-size10000 represents the case of few updates and many large range queries. In this case, SL ties KST32 on the Sun machine, and KST16 on the Intel machine. However, KST64 outperforms SL by between 38% and 43% on the Sun machine, and by between 89% and 179% on the Intel machine. On the Sun machine, Ctrie's throughput is comparable to that of KST16 between 4 and 64 threads, but KST16 outperforms Ctrie by 61% at 128 threads. KST64 outperforms Ctrie by between 44% and 230% on the Sun machine, and offers between 3.1 and 10 times the performance on the Intel machine.

Experiment 20i-20d-1r-size10000 represents the case of many updates and few large range queries. On the Sun machine, SL has a considerable lead on the other algorithms, achieving throughput as much as 116% higher than that of KST64 (the next runner up). The reason for the k-ST structures' poor performance relative to SL is two-fold. First, SL's non-linearizable range queries are not affected by concurrent updates. Second, the extreme number of concurrent updates increases the chance that a range query of the k-ST will have to retry. On the Intel machine, KST64 still outperforms SL by between 21% and 136%. As in the previous experiment, Ctrie ties KST16 in throughput on the Sun machine. However, KST64 achieves 127% (270%) higher throughput than Ctrie with four threads, and 37% (410%) higher throughput at 128 threads on the Sun machine (Intel machine).

As we can see from Fig. 5, in the total absence of range queries, Ctrie outperforms the k-ST structures. However, mixing in just one range query per 10,000 operations is enough to bring it in line with the k-ST structures. As the probability of an operation being a range query increases, the performance of Ctrie decreases dramatically. Snap performs similarly to the k-ST structures in the absence of range queries, but its performance suffers heavily with even one range query per 100,000 operations.

We also include a pair of graphs in Fig. 6 for the Intel and Sun machines, respectively, which show the performance of the k-ST over many different values of k, for each of the four experiments. Results for both machines are similar, with larger values of k generally producing better results. On both machines, the curve for experiment 20i-20d-1r-size100 flattens out after $k = 24$,

and 5i-5d-40r-size100 begins to taper off after $k = 32$. Throughput continues to improve up to $k = 64$ for the other experiments. The scale of the graphs makes it difficult to see the improvement in 5i-5d-40r-size10000 but, on the Sun (Intel) machine, its throughput at $k = 64$ is 6 times (16 times) its throughput at $k = 2$. This seems to confirm our belief that larger degrees would improve performance for range queries. Surprisingly, on the Intel machine, experiment 20i-20d-1r-size10000 sees substantial throughput increases after $k = 24$. It would be interesting to see precisely when a larger k becomes detrimental for each curve.

6 Future Work and Conclusion

Presently, the k-ST structure is unbalanced, so there are pathological inputs that can yield poor performance. We are currently working on a general scheme for performing atomic, non-blocking tree updates, and we believe the results of that work will make it a simple task to design and prove the correctness of a balancing scheme for this structure.

Another issue is that, in the presence of continuous updates, range queries may starve. We may be able to mitigate this issue by having the RangeQuery operation write to shared memory, and having other updates help concurrent range queries complete. It is possible to extend the flagging and marking scheme used by the k-ST so that range queries flag nodes, and are helped by concurrent updates. While this will likely alleviate starvation in practice, it is an imperfect solution. First, it will not eliminate starvation, for the same reason that updates in the k-ST are not wait-free. More specifically, if helpers assist in flagging all of the nodes involved in a RangeQuery, they must do so in a consistent order to avoid deadlock. Moreover, the nodes of the tree do not have parent pointers, so only top-down flagging orders make sense. Therefore, it is possible to continually add elements to the end of the range and prevent the RangeQuery from terminating. Second, if range queries can be helped, we must answer questions such as how helpers should avoid duplicating work when a RangeQuery involves many nodes. Since nodes must be flagged in a consistent order by all helpers, one cannot simply split helpers up so they start flagging at different nodes. One possibility is to have helpers collaborate through a work-queue.

Despite the potential for starvation, we believe our present method of performing range queries is practical in many cases. First, range queries over small intervals involve few nodes, minimizing the opportunity for concurrent updates to interfere. Second, for many database applications, a typical workload has many more queries (over small ranges) than updates. For example, consider an airline's database of flights. Only a fraction of the queries to their database are from serious customers, and a customer may explore many different flight options and date ranges before finally purchasing a flight and updating the database.

It would be interesting to measure and compare the amount of shared memory consumed by each data structure over the duration of a trial, as well as the amount of local memory used by our range query algorithm.

In this work, we described an implementation of a linearizable, non-blocking k-ary search tree offering fast searches and range queries. Our experiments show

that, under several workloads, this data structure is the only one with scalable, linearizable range queries. When compared to other leading structures, ours exhibits superior spatial locality of keys in shared memory. This makes it well suited for NUMA systems, where each cache miss is a costly mistake.

Acknowledgements. We would like to thank Faith Ellen for her extensive help in organizing and editing this paper. Our thanks also go out to the anonymous OPODIS reviewers for their helpful comments. Finally, we thank Michael L. Scott at the University of Rochester for graciously providing access to the Sun machine. This research was supported, in part, by NSERC.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM 40(4), 873–890 (1993)
2. Afek, Y., Avni, H., Shavit, N.: Towards Consistency Oblivious Programming. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 65–79. Springer, Heidelberg (2011)
3. Attiya, H., Guerraoui, R., Ruppert, E.: Partial snapshot objects. In: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 336–343. ACM, New York (2008)
4. Barnes, G.: A method for implementing lock-free data structures. In: Proc. 5th ACM Symposium on Parallel Algorithms and Architectures, pp. 261–270 (1993)
5. Braginsky, A., Petrank, E.: A lock-free b+tree. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012, pp. 58–67. ACM, New York (2012)
6. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming, pp. 257–268 (2010)
7. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: Transactional predication: high-performance concurrent sets and maps for stm. In: PODC, pp. 6–15 (2010)
8. Brown, T., Avni, H.: Range queries in non-blocking k-ary search trees, `http://www.cs.toronto.edu/~tabrown/kstrq`
9. Brown, T., Helga, J.: Non-blocking k-ary search trees. In: Proc. 15th International Conference on Principles of Distributed Systems, pp. 207–211 (2011), Complete proof and code available at `http://www.cs.toronto.edu/~tabrown/ksts`, more details in Tech. Report CSE-2011-04, York University
10. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proc. 29th ACM Symposium on Principles of Distributed Computing, pp. 131–140 (2010), Full version in Tech. Report CSE-2010-04, York University
11. Okasaki, C.: Purely functional data structures. Cambridge University Press, New York (1998)
12. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent tries with efficient non-blocking snapshots. To appear in Proc. 17th ACM Symposium on Principles and Practice of Parallel Programming (2012)
13. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1995, pp. 204–213. ACM, New York (1995)

On the Polling Problem for Social Networks*

Bao-Thien Hoang and Abdessamad Imine

Lorraine University and INRIA Nancy – Grand-Est, France
{bao-thien.hoang,abdessamad.imine}@inria.fr

Abstract. We tackle the polling problem in social networks where the privacy of exchanged information and user reputation are very critical. Indeed, users want to preserve the confidentiality of their votes and to hide, if any, their misbehaviors. Recent works [7,8] proposed polling protocols based on simple secret sharing scheme and without requiring any central authority or cryptography system. But these protocols can be deployed safely provided that the social graph structure should be transformed into a ring-based structure and the number of participating users is perfect square. Accordingly, devising polling protocols regardless these constraints remains a challenging issue.

In this paper, we propose a simple decentralized polling protocol that relies on the current state of social graphs. More explicitly, we define one family of social graphs and show their structures constitute necessary and sufficient condition to ensure vote privacy and limit the impact of dishonest users on the accuracy of the output of the poll. In a system of N users with $D \leq N/5$ dishonest ones (and similarly to the works [7,8] where they considered $D < \sqrt{N}$), a *privacy parameter* k enables us to obtain the following results: (i) the probability to recover one vote of honest node is bounded by $\sum_{m=k+1}^{2k} \left(\frac{D}{N}\right)^m \cdot \left(\frac{1}{2}\right)^{2k+1-m}$; (ii) the maximum number of votes revealed by dishonest nodes is $2D$; and, (iii) the maximum impact on the output is $(6k+4)D$. Despite the use of richer social graph structures, we succeed to detect the misbehaving users by manipulating verification procedures based on shortest path scheme and routing tables. An experimental evaluation demonstrates that the dishonest coalition never affects the outcome of the poll outside the theoretical bound of $(6k+4)D$.

Keywords: Social networks, Polling protocol, Secret sharing, Privacy.

1 Introduction

In this work, we approach to one of the current practical, useful but sensitive topic in Online Social Networks (OSN), the *polling* process. In general, polling is the way to determine the most favorite choice amongst some options from the participants. Each participant can distribute his preference by submitting vote, and after aggregating all votes, the majority option will be chosen as the final result. For instance, one company of mobile phone has just launched a new product and may want to ask customers whether or not its features are

* Funded by ANR Streams project.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 46–60, 2012.

comfortable, and user will choose one option between "Yes" or "No". We here consider simply a binary polling with only two options "+1" or "-1" for the concerning question.

The main objective in such a polling protocol is performing a secure and accurate process to sum up the initial votes with the presence of dishonest users, who try to bias the final result and reveal the votes of honest ones. The polling problem is simple but it takes an important role in incorporating user's opinion online. Thus, currently, there are some studies and solutions for this problem in two approaches, centralized and distributed networks. In the centralized OSN, a central server is used to collect the users' votes and sum up all values to obtain output. Facebook Pool[1] and Doodle[2] are well illustrative examples. However, this approach suffers from server failures and particularly privacy problems: it is not guaranteed the central server will not bias and disclose the user votes.

In our work, we are interested in polling protocol based on decentralized OSN, where privacy of user is improved as information is not concentrated in one place. Recently, Guerraoui et al. [7,8] proposed, DPol, a simple decentralized polling protocol based on secret sharing scheme (without using cryptography) where both honest and dishonest participants are considered. In DPol, participants care about their reputation. Indeed, they do not want their votes to be disclosed nor their misbehaviors, if any, to be publicly exposed. To dissuade user misbehaviors, distributed verification procedures are manipulated to detect with a non-zero probability these misbehaviors and enable honest users to tag profile of dishonest ones. Moreover, DPol ensures privacy of votes and final result accuracy by limiting the impact of dishonest users. However, DPol has practically some disadvantages. Firstly, DPol relies on a structured overlay, cluster-ring-based structure, which is on top and really apart from the normal social graph. It does not take into account the social links among users in the sense that it builds the uniform distribution of users into groups. This is not practical as we have to target a special case using notion of group instead of reserving the normal structure of the graph. This construction would be necessarily based on centralized solution. Hence we lose the benefit of a fully decentralized polling protocol. Second, the number of users should be a perfect square number such that one graph with N users are divided into $\sqrt{N}$ groups of size $\sqrt{N}$. It should be noted designing decentralized polling protocol without cryptography and constraints (overlay structure and perfect square number of users) imposed in [7,8] remains a challenge problem.

Contributions. Our objective is to keep the natural property of the graph in the sense user and social links should be preserved, and each individual can perform the voting process privately and securely without resorting to the group division.

Inspired from [7,8], first, we propose a design of a simple decentralized polling protocol that uses social graphs. Second, we describe properties required for the social graph to ensure the correctness of the protocol. Furthermore, we cover a general case for the graph topology on which the protocol can run properly. Despite the use of richer social graph structures, one node can receive/send so

[1] `http://apps.facebook.com/opinionpolls/`
[2] `http://www.doodle.com/`

many duplicated messages from/to other nodes. This can lead to flooding the local storage. By thoroughly using the graph structure, our protocol enables one node to deal with only necessary messages. Instead of accepting all messages, a node stores only the ones passed by the optimal paths. To prevent user misbehaviours, we introduce verification procedures based on shortest path scheme and routing tables. Using the same notion of privacy parameter k in [7,8], we get the following results in a system of size N with D dishonest users: one vote of honest node is recovered with the probability at most $\sum_{m=k+1}^{2k} \left(\frac{D}{N}\right)^m \cdot \left(\frac{1}{2}\right)^{2k+1-m}$; and up to $2D$ votes can be revealed by the dishonest coalition, and the impact from the dishonest coalition to the final result is at most $(6k+4)D$. We validate our solution with a performance evaluation which shows that our protocol is accurate and close to the theoretical average impact, that is $4k + 2\alpha + 2$, where α is the proportion of users correctly voting. Our result encourages the use of polling protocol without transforming the social graphs into other overlay structures.

Outline. This paper is organized as follows. Section 2 describes our polling model, and introduces a family of social graphs. Section 3 presents our polling protocol with its correctness properties, establishes formally the relation between the protocol and the family of social graphs, and analyses different complexities to perform the polling. Section 4 illustrates our experimental results. We review related work in Section 5 and conclude the paper with future research direction in Section 6. Proofs for the correctness of our solution are given in [14].

2 Polling and Social Network Models

This section introduces what are ingredients of polling models and presents the graph models to describe social networks. It should be noted that we consider the same assumptions like the work [7,8].

2.1 Polling Model

The polling problem consists of a system with N uniquely identified nodes representing users of a social network. Each participant u_n (or simply, n) expresses its opinion by giving a vote $v_n \in \{-1, 1\}$. After collecting the votes of all nodes, the expected outcome is $\sum_n v_n$. In this work, we consider the following assumptions:

Each node is able to communicate with its neighbors (e.g., direct friends), and is either honest or dishonest. The honest node completely complies with the protocol and takes care about its privacy and reputation in the sense that the vote value is not disclosed. All dishonest nodes can form a coalition to get the full knowledge of the network and try to do everything to achieve these goals without being detected: (i) bias the result of the election by promoting their votes or changing the values they received from other honest nodes; (ii) infer the opinions of other nodes. However, they also want to protect their reputation from being affected. In order to unify the opinions and not give compensating effects, all dishonest nodes make the single coalition $\mathcal{D}$ of size D. Nevertheless, they are still selfish in the sense that each dishonest node prefers to take care about its own reputation to covering up each other.

In order to prevent and reduce the incorrect behaviors, there is an activity affected to profile of concerned node. In particular, if node u is detected as misbehavior one by v then u's profile is tagged with statement "u has been detected bad behavior by v" and in v's profile has statement "u is bad guy". Furthermore, we do not take into account the situation that dishonest nodes wrongfully blame honest ones, or do Sybil attacks and spam since that kinds of misbehavior can be detected by some tools or several existing systems such as SybilLGuard[17], SybilLimit [16], [11,15] (for filtering wrongful blames), and [11,13] (for mitigating spam).

2.2 Social Network as a Graph Models

We present the social network in our problem as the form of models of social graph. In this section, firstly, we define the terms and notations of graph used throughout our work. Later, we demonstrate the family of graphs including the ideal case (network without dishonest nodes) and normal case (network with the presence of dishonest nodes).

Notations. Let $G = (V, E)$ be an undirected graph where $V = \{u_0, u_1, ..., u_{N-1}\}$ is a set of uniquely identified nodes of size N, and E is an edge set. Each node is either honest or dishonest. We represent $\mathcal{H}(X)$ and $\mathcal{D}(X)$ as the set of honest nodes and dishonest nodes of size $D = |\mathcal{D}(\mathcal{X})|$ in graph X. For a node $u_n \in V$, let us identify the following notations: d_n as a degree (a number of neighbors) of u_n; $\mathcal{R}(u_n)$ (or simply, $\mathcal{R}_n$) as the set of neighbors of u_n; $\mathcal{F}_n$ and $\mathcal{Q}_n$ ($\mathcal{F}_n, \mathcal{Q}_n \subseteq \mathcal{R}_n$) are respectively set of neighbors that u_n sends and receives messages.

Paths and Distances. Given two nodes $u, v \in V$, they can connect directly or not. We denote by function $e(u, v)$ this kind of relation, namely, $e(u, v) = 1$ if there is a link between u and v, otherwise $e(u, v) = 0$.

A path p of length $l \in \mathbb{N}$ in the graph is an ordered sequence of $l + 1$ nodes such that there exists an edge connecting two consecutive nodes in the sequence: $p = \langle u_{k_1}, u_{k_2}, \ldots, u_{k_{l+1}} \rangle$ with $u_{k_i} \in V$, $e(u_{k_i}, u_{k_{i+1}}) = 1$, $1 \leq i \leq l$. We write $l(p)$ to refer the length of path p, i.e., number of the edges of p. As $e(u_{k_i}, u_{k_{i+1}}) = 1$ then $l(\langle u_{k_i}, u_{k_{i+1}} \rangle) = 1$. If path p contains only one node, $l(p) = 0$.

For two nodes $u, v \in V$, let $p(u, v)$ be a path connecting between u and v and $Pa(u, v)$ be the set of all such paths. We write $x \in p(u, v)$ if path $p(u, v)$ contains node x. For two paths $p(u_1, v_1)$, $p(u_2, v_2)$, we define the intersection of them as follows: $p(u_1, v_1) \cap p(u_2, v_2) = \{x \in V \mid x \in p(u_1, v_1) \text{ and } x \in p(u_2, v_2)\}$.

Additionally, each node is either honest or dishonest. Thus, to transmit messages between two nodes u and v, it is important to consider the honesty property of each node (i.e., checking whether node is honest or dishonest) in the paths connecting them. Particularly, if u and v are directly connected, i.e., $e(u, v) = 1$, we should investigate the honesty property of u and v. The transmission is secure only if they are all honest and is unsecured in other case. If $e(u, v) = 0$, we should examine all paths connecting between u and v. For a path $p(u, v) = \langle u \equiv u_{k_1}, u_{k_2}, ..., u_{k_m} \equiv v \rangle$ (where $e(u_{k_i}, u_{k_{i+1}}) = 1$, $1 \leq i \leq m-1$), we have to check honest property of each intermediate node u_{k_i}. The transmission in that path is secure only if all nodes are honest and we call it "honest path".

If there exists at least one honest path between u and v, it guarantees the correct information from u (or v) will approach to v (or u).

For a graph G, there exists, for all pairs of honest nodes u, v, at least one honest path between them, then G is called "honest graph".

Shortest Paths. We illustrate by $p_S(u, v)$ and $Pa_S(u, v)$ the shortest path and the set of all shortest paths between two nodes u and v. In [6], a simple but fast and accurate algorithm for the approximation of shortest paths between pair of nodes in the real-world graph is presented. We can use this method to determine the shortest paths and distances between two nodes. The length of the shortest path between u and v, is denoted by $\delta(u, v)$, i.e., $\delta(u, v) = l(p_S(u, v))$.

Graph Model. So far, we presented the polling model in which participants are either honest or dishonest. Like [7,8], we use a predefined parameter $k \in \mathbb{N}$ (this parameter will be detailed in section 3.1) to present the features of our social graphs. Let $G = (V, E)$ be a social graph with the following properties:

Property 1 (P_{g_1}). $d_n \geq 2k + 1$ and $|\mathcal{F}_n| = |\mathcal{Q}_n| = 2k + 1$, for every $u_n \in V$.

Property 2 (P_{g_2}). G is a honest graph, i.e., for every honest nodes u, v, there exists a path $p(u, v)$ containing only intermediate honest nodes.

Property 3 (P_{g_3}). $D < N/2$.

From these properties, we characterize two families of graphs:

(i) $\mathcal{G}_1 = \{G \mid \mathcal{D}(G) = \varnothing$ and G satisfies $P_{g_1}\}$.
(ii) $\mathcal{G}_2 = \{G \mid \mathcal{D}(G) \neq \varnothing$ and G satisfies P_{g_1}, P_{g_2} and $P_{g_3}\}$.

Graphs in $\mathcal{G}_1$ contain no dishonest nodes and in $\mathcal{G}_2$ are normal graphs with the existence of dishonest nodes. According to Property P_{g_1}, each node has a set of receivers ($\mathcal{F}_n$) and a set of senders ($\mathcal{Q}_n$) to establish communication and they have the same size and may be disjoint. Property P_{g_2} ensures each honest node always obtains one correct version of data from other honest ones. Property P_{g_3} enables us to limit the control of dishonest users in the whole system.

3 Protocol

In this section, we first present our polling protocol and give some properties of this protocol. We assume there is no crash and message loss.

3.1 Description

Generally, the polling protocol includes three phases (see Algorithm 1): (*i*) *Sharing*, (*ii*) *Broadcasting* and (*iii*) *Aggregating*. Phase *Sharing* describes the generation, distribution of a set of shares of each node to its neighbors as well as collecting these shares from its neighbors. In the *Broadcasting* phase, each node broadcasts messages containing the total shares, which are collected in the Sharing phase, to its direct and indirect neighbors. The last phase, *Aggregating*, shows the process that each node decides data received from other nodes and computes the final outcome.

Algorithm 1. POLLING ALGORITHM AT NODE u_n, $n \in \{0, 1, ..., N-1\}$

Input:
v_n: A vote of node, value in $\{-1, 1\}$
d_n: degree of node
k : privacy parameter
$\mathcal{R}_n$: set of direct neighbors
$\mathcal{F}_n$: set of neighbors to send shares
$\mathcal{Q}_n$: set of neighbors to receive shares

Variables:
c_n: collected data, $c_n = 0$
$\mathcal{C}_n$: set of possible collected data
$\mathcal{C}_n[\{0, 1, ..., N-1\} \to \varnothing]$
h_n: set of final choosing collected data
$h_n[\{0, 1, ..., N-1\} \to \perp]$
Γ_n: routing table
$\Gamma_n[\{0, 1, ..., N-1\} \to \varnothing]$

Output: result

Algorithm

```
1 Share(v_n, F_n) | ReceiveShareEvent
2 Broadcast(n, c_n, 1, R_n) | ReceiveDataEvent
3 Aggregate()
```

Procedure Share(v_n, $\mathcal{F}_n$)

```
4  P_n ← ∅
5  for i ← 1 to k do
6      P_n ← P_n ∪ {v_n} ∪ {−v_n}
7  end
8  P_n ← P_n ∪ {v_n}
9  μ_n ←_rand P_n
10 for i ← 0 to 2k do
11     send(SHARE, μ_n[i], F_n[i])
12 end
```

Procedure ReceiveShareEvent(SHARE, p, r)

```
13 if (r ∈ Q_n ∧ p ∈ {−1, 1}) then
14     c_n ← c_n + p
15 end
```

Procedure Broadcast(n, c_n, l_n, $\mathcal{R}_n$)

```
16 foreach (r ∈ R_n) do
17     send(DATA, n, c_n, l_n, r)
18 end
```

Procedure ReceiveDataEvent(DATA, s, c_s, l_s) from neighbor identity t

```
19 if (s = n or l_s > δ_L(s, n)) then exit
20 if (c_s ∉ C_n[s]) then
21     ν_s ← c_s
22     C_n[s] ← C_n[s] ∪ {c_s}
23     Broadcast(s, ν_s, l_s + 1, R_n \ {t})
24 else
25     ν_s ← ⊥
26 end
27 Γ_n[s] ← Γ_n[s] ∪ {(t, c_s, ν_s, l_s)}
```

Procedure Aggregate()

```
28 result ← 0
29 for s ← 0 to N − 1 do
30     if (s ≠ n) then
31         h_n[s] ← CheckInconsistency(s)
32     else
33         h_n[s] ← c_n
34     end
35     result ← result + h_n[s]
36 end
```

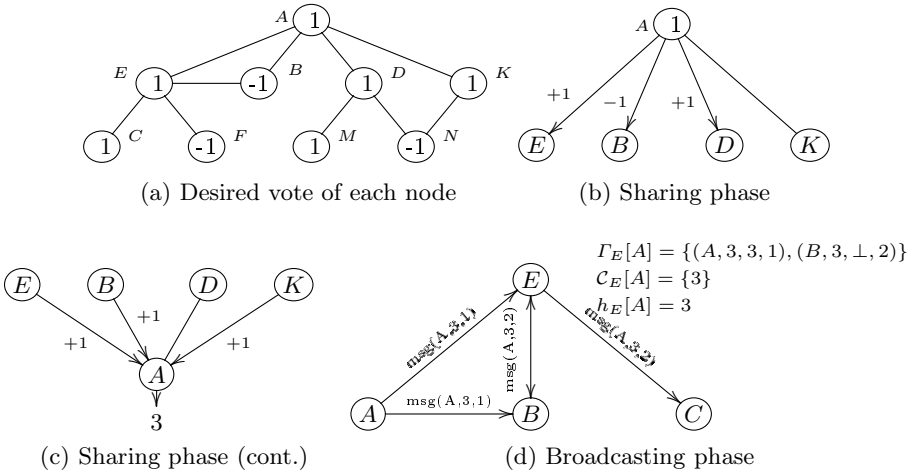
Procedure CheckInconsistency(s)

```
37 if (|C_n[s]| = 1) then
38     return C_n[s][0]
39 else
40     return correct value after verifying Γ[s]
           of neighbors
41 end
```

Sharing. In this phase, each node u_n contributes its opinion by sending a set of shares expressing its vote $v_n \in \{-1, 1\}$ to its neighbors. We inspired the sharing scheme proposed in [4] to generate shares. Namely, u_n generates $2k+1$ shares $\mathcal{P}_n = \{p_1, p_2, ..., p_{2k+1}\}$ where $p_i \in \{-1, 1\}$, $i = 1, 2, ..., 2k+1$ including: $k+1$ shares of value v_n, and k shares of opposite v_n's value. Later it generates randomly a permutation of $\mathcal{P}_n$, and sends these $2k+1$ messages to $2k+1$ direct neighbors. Lines 4–12 in Algorithm 1 describe this phase. Node also receives exactly $2k+1$ messages from its direct neighbors. We can target this strict number of nodes in the set of receivers and senders by the following approach: each node u_n will determine all possible receiver sets $\mathcal{F}_n \subseteq \mathcal{R}_n$ such that $|\mathcal{F}_n| = 2k+1$ by choosing $2k+1$ arbitrary elements from $\mathcal{R}_n$, i.e., $\mathcal{F}_n = \{m_{n_1}, m_{n_2}, ..., m_{n_{2k+1}}\}$, $\forall m_{n_i} \in \mathcal{R}_n$. Node u_n also knows all other possible set $\mathcal{F}_p$ of any other node u_p. From this it can identify all possible tuples of N sets of the form $(\mathcal{F}_0, \mathcal{F}_1, ..., \mathcal{F}_{N-1})$. For each tuple $(\mathcal{F}_0, \mathcal{F}_1, ..., \mathcal{F}_{N-1})$, it will check whether the following condition is satisfied: each element $m_{n_i} \in \mathcal{F}_n$,

$0 \leq n < N$ must belong to exactly other $2k$ sets $\mathcal{F}_{i_1}$, $\mathcal{F}_{i_2}$, ..., $\mathcal{F}_{i_{2k}}$, $n \neq i_j$, $j = 1, 2, ..., 2k$. If there exists a tuple $(\mathcal{F}_0, \mathcal{F}_1, ..., \mathcal{F}_{N-1})$ fulfilled above conditions, the requirement about number of receivers and senders at each node will be satisfied. For instance, specially, each node u_n defines the set of receivers $\mathcal{F}_n = \{u_{(n+1) \bmod N}, u_{(n+2) \bmod N}, \ldots, u_{(n+2k+1) \bmod N}\}$ of size $2k + 1$. Besides, wee see that $\mathcal{Q}_n = \{u_{(n-1) \bmod N}, u_{(n-2) \bmod N}, \ldots, u_{(n-2k-1) \bmod N}\}$ of size $2k + 1$.

After all nodes collect $2k+1$ shares from its neighbors, and sums into *collected data* c_n (lines 13–15 in Algorithm 1), this phase is complete. Figures 1 illustrates an example of the protocol for $k = 1$. Figure 1a presents desired vote of each node, whereas Figure 1b depicts the sharing phase at node A. Node A would like to vote $+1$, thus, it generates a set of $2k+1 = 3$ shares $\{+1, -1, +1\}$ which total equals to $v_A = 1$. Figure 1c shows node A collects the shares from its neighbors and computes the collected data $c_A = 3$.



(a) Desired vote of each node (b) Sharing phase

(c) Sharing phase (cont.) (d) Broadcasting phase

Fig. 1. Polling algorithm for $k = 1$

Broadcasting. In this phase, each node u_n encapsulates the collected data c_n with its identity n and length counter l_n, which expresses the length of the path message has passed (initially, $l_n = 1$), into message msg and disseminates it to all neighbors (lines 2 and 16–18 in Algorithm 1). This action is depicted in Figure 1d. When u_n receives from u_t message $msg(s, c_s, l_s)$ emitted from the source u_s, it performs the following actions (see ReceiveDataEvent() in Algorithm 1):

1. *Loop detection*: u_n checks contents of msg and detects the loop based on the source's identity (line 19 in Algorithm 1). If this message is the one u_n has emitted earlier, i.e., $s = n$, then u_n simply drops the message and does not need to inform u_t. Otherwise, u_n accepts msg.

2. *Message Forwarding*: For a message passing the loop detection, u_n should get data c_s and forward to its friends except u_t.

We see that, naively approaching, u_n can receive c_s from many disjoint paths (without loop) connecting between u_s and u_n. However, the number of paths can be blown up to exponential value. More specifically, the worst case is when G is clique and one node has $N-1$ friends. Each message passes through all nodes in the network, and thus, the number of possible paths between u_s and u_n is $(N-1)(N-2)...1 = (N-1)!$. This motivates us to find out an optimal solution to bound the number of messages emitted from u_s that u_n should receive without losing any necessary information.

Instead of using naive approach, we propose other technique which is small but very useful and much more optimal: node receives messages which passed by paths with the limited length rather than accepting all. Here, for messages broadcasting from u_s, we use bread-first expansion with the assumption that transmitting message in one edge takes one time unit. Hence, we see that firstly u_n receives messages from u_s in the shortest path $p_S(u_s, u_n)$, and then from other paths of greater length. By the way, the content of messages can be changed by some intermediate dishonest nodes in the path $p \in Pa(u_s, u_n)$. Thus, we should take care the intermediate nodes. For each intermediate node x, it receives message from u_s in $p_S(u_s, x)$ first and from the longer path later. Node u_n also receives message, which passed x, from the shortest path $p_S(x, u_n)$ first and then from other longer paths $p(x, u_n)$. Therefore, u_n receives messages, which are broadcast from u_s and passed x, from the paths with length $\delta(u_s, x) + \delta(x, u_n)$ first, and from other longer paths later. To take care of all possible changes in contents, u_n should receive all messages which already passed all intermediate nodes. And so, the maximum length of the paths passing message u_n should receive is $max_x\{\delta(u_s, x) + \delta(x, u_n)\}$.[3] In case that for all node x, $p_S(u_s, x)$ and $p_S(x, u_n)$ have some common nodes (different from x), u_n should not receive messages from the paths of length $\delta(u_s, x) + \delta(x, u_n)$ since they have a loop inside. It should receive messages from paths of length $\delta(u_s, u_n)$ instead. So, we combine all of these results, and define one value which u_n (resp. u_s) could use to determine the maximum length of paths which deliver messages from u_s (resp. u_n) to u_n (resp. u_s) as follows:

$$\delta_L(u_s, u_n) = \begin{cases} \max_{x \in U_{sn}}\{\delta(u_s, x) + \delta(x, u_n)\} & \text{if } u_s \neq u_n \wedge |U_{sn}| > 0 \\ \delta(u_s, u_n) & \text{otherwise} \end{cases} \quad (1)$$

where $U_{sn} = \big\{x \in V | x \neq u_s, u_n \text{ and } \exists p_1 \in Pa_S(u_s, x), p_2 \in Pa_S(x, u_n) \text{ s.t. } p_1 \cap p_2 = \{x\}\big\}$.

For message with $l_s \in \big[\delta(u_s, u_n), \delta_L(u_s, u_n)\big]$, node u_n accepts and does the following activities, otherwise it simply eliminates that message. Line 19 in Algorithm 1 shows this verification.

The activities in the case $l_s \in \big[\delta(u_s, u_n), \delta_L(u_s, u_n)\big]$ are as follows (lines 20–27 in Algorithm 1): u_n checks $\mathcal{C}_n[s]$, a set of possible values emitted from the source with identity s, to determine whether c_s is already presented in it. If c_s is not stored in $\mathcal{C}_n[s]$, u_n will add it into $\mathcal{C}_n[s]$, and then forward message

[3] See [14, Lemma 5] for the correctness of this consideration.

$msg(s, \nu_s, l_s + 1)$, where ν_s is value to be sent (in this case $\nu_s = c_s$), to other direct neighbors except u_t. All information about the messages from source u_s is stored in the *routing table* $\Gamma_n[s]$ which is used for checking inconsistency later. This table contains the following fields: first field is neighbor identity from which it received message (e.g., t), second one is the receiving value (e.g., c_s), third one is the value to be forwarded (e.g., ν_s), and last field is the length of the path passing message from the source u_s (i.e., l_s). In this case, u_n adds tuple (t, c_s, ν_s, l_s) into $\Gamma_n[s]$. In other case that $\mathcal{C}_n[s]$ has value c_s inside, u_n does not need replicating that value in $\mathcal{C}_n[s]$, as well as forwarding it to other friends as it already did earlier. It just stores information in the routing table, by setting the sent value as null, i.e., $\nu_s = \perp$ (null).

Figure 1d depicts the process when node E receives message emitted from A. When $msg(A, 3, 1)$ with length 1 arrives to E, it stores $c_A = 3$ into set of possible collected data of source A, that is $\mathcal{C}_E[A]$. It then forwards $msg(A, 3, 2)$ with length 2 to B and C and adds a tuple $(A, 3, 3, 1)$ into routing table of source A, i.e., $\Gamma_E[A]$. Notice that A also sends $msg(A, 3, 1)$ to B with the same length as the one to E, thus, B gets the same message and does the same actions like E. Node E gets forwarded message with length 2 from B. Since that is the second message having the same source and collected data, but higher length, E does not forward it. Node E just inserts one more tuple $(B, 3, \perp, 2)$ expressing the information received from B into routing table $\Gamma_E[A]$.

Once there is no broadcasting messages in the network, this phase is over. Since each node just sends and receives a finite number of messages, and all messages eventually arrives, it is guaranteed this phase terminates correctly.

Aggregating. In this phase, u_n has to decide the collected data of other nodes before calculating the final result. To make decision for node u_s, it checks $|\mathcal{C}_n[s]|$ (lines 37–41 in Algorithm 1): if $|\mathcal{C}_n[s]| = 1$, the single element in $\mathcal{C}_n[s]$ is chosen as a correct collected data, otherwise there exists an inconsistency and it should do the verification: requesting all routing tables $\Gamma[s]$ of neighbors and indirect neighbors to check information received and forwarded by them. If one node is detected that it already sent different values of its data or its receiving information, then an alarm is raised and that node is tagged in its profile. By doing this, u_n also gets the correct collected data of source u_s.[4] So, in any case, u_n achieves the correct copy of collected data of source u_s. It then stores that value as one item $h_n[s]$ in the array h_n, which contains collected data of other nodes, and adds into result (lines 29–36 in Algorithm 1). After checking and summing up all collected data of nodes (including its own collected data c_n), u_n obtains the final result (that is $\mathsf{result} = c_n + \sum_{i \neq n} h_n[i]$).

For instance, we consider Figure 1d again. From formula (1), we see that $\delta_L(A, E) = \delta(A, B) + \delta(B, E) = 2$. After receiving message $msg(A, 3, 2)$ from node B, and updating routing table, node E makes final decision to choose value from source A. As the set $\mathcal{C}_E[A]$ is singleton, it will set $h[A] = \mathcal{C}_E[A][0] = 3$. This value will be used to compute final outcome of polling later.

[4] See [14, Lemma 8] and Section 3.3 for the detail of this verification.

3.2 Properties of Protocol

In section 2.1, we already introduced the characteristics of the polling model. It implies that our protocol should have some properties such that the system can run correctly with (or without) the existence of dishonest nodes. Namely, each honest node outputs the correct polling result, controls the impact from the dishonest nodes, and not disclose its private information, whereas the dishonest coalition could not control the polling process or fool an entire network without being detected. In this section, we clarify those desirable properties by stating what protocol should achieve with (or without) the existence of dishonest nodes such as accuracy and privacy. We denote by $u \vDash x$ (or $\mathcal{D} \vDash x$, resp.) node u (or coalition $\mathcal{D}$, resp.) reveals vote x.

Privacy. The privacy property expresses the ability of the system to prevent the private information from being leaked to the dishonest nodes. In other words, the coalition could not reveal any information of particular honest node beyond what it can deduce from its own vote, the output of computation and the shares of votes.

Definition 1 (Privacy). *The protocol is said* private *if the dishonest nodes cannot learn anything about the vote of honest node. More formally, for any honest node u_n with vote v_n, there exists a negligible function $\xi(k)$ such that: $Pr[\mathcal{D} \vDash v_n] \leq \xi(k)$.*

Accuracy. We define the impact of dishonest nodes as the difference between the output and the expected result. In our case, vote is either "+1" or "-1", and thus, with the system of N nodes, the maximum and minimum final results are N and $-N$ respectively. This implies the maximum difference amongst the final outputs is $2N$. As defined in [5], accuracy is given by the maximum impact with respect to the maximum difference of the final outputs:

$$\Lambda = \frac{1}{2N} \cdot \max_{n \in \mathcal{H}(G)} \Delta(\mathsf{result}_n, \sum_{i=0}^{N-1} v_i) \tag{2}$$

where result_n is the output of the poll (see Algorithm 1). Here and throughout this work, we denote by $\Delta(x, y)$ the difference between value x and y, i.e., $\Delta(x, y) = |x - y|$.

Definition 2 (Accuracy). *The protocol is said* accurate *if there exists a negligible function $\xi(k)$ such that $\Lambda \leq \xi(k)$.*

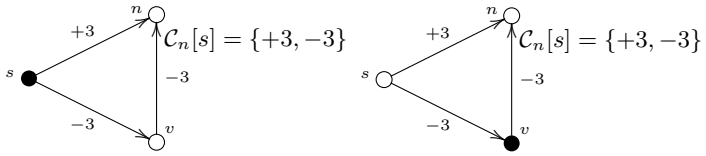
3.3 Protocol and Graph with Dishonest Nodes

We consider graphs of family $\mathcal{G}_2$ (see graph model on page 50) and analyze the correctness (including privacy, accuracy and termination) of our protocol when deployed with graphs of $\mathcal{G}_2$. Next we give spatial, message and time complexities. Finally, we show properties of $\mathcal{G}_2$ are necessary and sufficient condition to ensure the correctness of our protocol. All proofs are given in full details in [14].

Privacy. When deploying protocol in graphs of family $\mathcal{G}_2$, we have the following results: (i) A vote of one node is revealed with certainty if and only if $k+1$ direct neighbors receive the shares corresponding to that vote (see [14, Lemma 3]); (ii) the probability that dishonest coalition reveals vote of a given honest node is bounded by $\sum_{m=k+1}^{2k} \left(\frac{D}{N}\right)^m \cdot \left(\frac{1}{2}\right)^{2k+1-m}$, protocol is private for $k \ll N$ or $D \ll N$ (see [14, Lemma 4]); (iii) the maximum number of votes revealed by coalition $\mathcal{D}$ is $2D$ with $D \leq N/5$ (see [14, Corollary 1]).

Accuracy. We shows that in our protocol, there is no lost information in sending/receiving messages by using shortest path scheme. Even in the network with privacy conscious settings in which each node u_n has no knowledge to calculate bounds for the path lengths between it and source u_s, our protocol can be adapted by the following approach: in the broadcasting phase, node u_n does action 2 without checking condition $l_s \in \left[\delta(u_s, u_n), \delta_L(u_s, u_n)\right]$ (see [14, Lemma 5, Corollary 3]). In our protocol, one honest node affects at most $2k+2$ and $4k+2$ to the final result in the sharing and broadcasting phases, respectively. And thus, the maximum impact from the dishonest coalition $\mathcal{D}$ is $(6k+4)D$ (see [14, Lemmas 6-9] for more details). Moreover, the accuracy is preserved by the verification process in the protocol. The final result of our protocol is based on the information stored in $\mathcal{C}_n[s]$. In normal case, this set contains one and only one correct value. However, dishonest nodes can promote their votes by modifying the content of the broadcast/forwarded messages and thus, $\mathcal{C}_n[s]$ can contain distinct values. In that case, we start the verification process to get correct value (see [14, Lemma 8] for more details). For example, in Figure 2, node n receives two values from source s, one directly from s (+3) and one from v (-3). In this case n does not know whether s sent two different values or v modified s's value, and thus it asks for routing tables of its neighbors. As motivated in Section 2.1, we do not take into account the Sybil attacks, spam, and wrongfully blaming since these kinds of attacks are already detected by several practical systems. Therefore, without Sybil attacks, dishonest nodes can change contents of the message, except identity of the source and cannot create any forged messages containing identity of other nodes. Moreover, without wrongfully blame, dishonest nodes cannot spoof incorrect routing table. In Figure 2a (and similar explanation in Figure 2b), this situation cannot occur: s replies to n that s sent to v value +3 (same as the one it sent to n). Otherwise, s will indirectly wrongfully accuse v as dishonest node, because according to information from s, v later forwards to n different value from the one it received from s, and thus, v must be dishonest and tagged. Actually, in a system that honest nodes are majority, the probability for one dishonest nodes to be exposed when wrongly accusing honest ones is high. Like in the example, if v is wrongly accused only by a small number of nodes, the allegation would be in doubt and not be considered, and the accuser s would be finally backfired. By the way, we do not allow this kind of blame in the system, and assume that no node would like to be tagged as dishonest which does not wrongly blame other nodes.

Termination. In the sharing phase, each node sends and receives a finite number of messages, that is $2k+1$. In the broadcasting phase, each node u_n receives



(a) Case 1 (b) Case 2

Fig. 2. Intention of the dishonest nodes

and forwards the finite number of messages from source u_s, because it just receives message passed by the path of length inside the interval $\left[\delta(s,n), \delta_L(s,n)\right]$, not all messages departing from s. Moreover, as there is no loop, each phase terminates correctly and the algorithm has a finite number of steps. Therefore, it is guaranteed our protocol terminates.

Complexities. For a node u_n: the number of messages it sends and forwards is $O(N.D.d_n + k)$, and the total space it must hold is $O(N.D.d_n)$. In addition, if we assume the system is synchronous one (computation is performed under one or many rounds), then the protocol operates in $O(k + N^2)$rounds. The details of these results are given in [14, Propositions 4-6]).

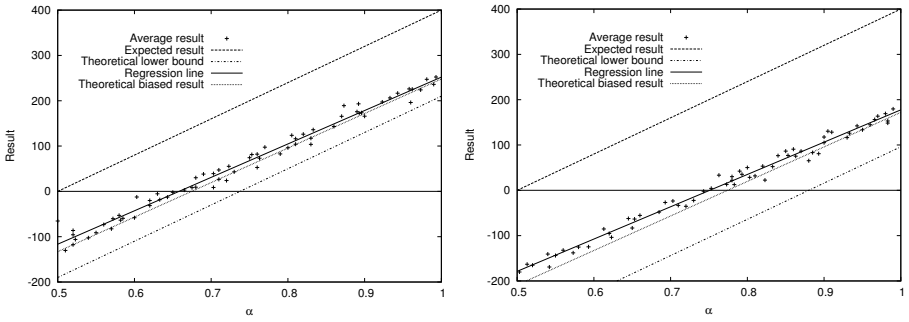
Main Result. From the results above about privacy, accuracy and termination, we observe that when deployed the protocol in the graphs of family $\mathcal{G}_2$, the protocol performs correctly. Furthermore, considering a graph G, to deploy correctly the protocol, G must satisfy the condition of the properties P_{g_1}, P_{g_2} and P_{g_3} (more details in [14, Theorem 2]). This implies the properties of $\mathcal{G}_2$ are the necessary and sufficient condition for the polling protocol to be performed correctly in the system containing dishonest nodes.

4 Experimental Evaluation

We perform this evaluation to analyse the correctness of the protocol by observing the difference between experiment output and the theoretical bounds. In the experiments, we use UDP and asynchrony for exchanging messages without crash or message loss. We implement protocol by using framework YALPS[5] to demonstrate the communication amongst nodes and facilitate the development and testing of the applications. In our experiments, we consider the worst case for the system: each dishonest node sends $2k+1$ shares of value "-1" and converts all receiving shares of "1" into ones of "-1". Thus, it affects to the final result at most $6k+4$ (including the impact of $2k+2$ in the sharing phase and $2(2k+1) = 4k+2$ in the broadcasting phase). If we denote by α number of nodes voting "+1", then the expected result will be $\alpha N - (1-\alpha)N = (2\alpha - 1)N$. So theoretically, the biased final outcome should be inside the interval $[(2\alpha - 1)N - (6k + 4)D; (2\alpha - 1)N]$.

We examine the experiment with same value of N and D as [7,8]. W.l.o.g., we consider α value in the interval $[0.5, 1.0]$. Figure 3 depicts our results for

[5] `http://yalps.gforge.inria.fr/`



(a) $k = 1$ (b) $k = 2$

Fig. 3. Experiment with $N = 400$, $D = 19$ to check the accuracy of protocol

the network with $N = 400$, $D = 19$ in two subcases $k = 1$ (in Figure 3a) and $k = 2$ (in Figure 3b). In each test case, we compute the average output of all nodes and represent it as a point in the figure. We see that all experimental results are inside two theoretical bounds (thick-dashed line and dot-dashed line) and the average impact from dishonest coalition is less than $6k + 4$. Moreover, there is no data point having the expected value. The reason we obtained this consequence comes from the fact the amount of average impact depends on the number of shares "+1" one dishonest node receives from neighbors. Namely, a dishonest node gets average $k + \alpha$ shares of "+1" and turns into shares of "-1", and affects $2(k + \alpha)$ in total. It also gives impact of value $2k + 2$ in the sharing phase. Consequently, the total impact is $2k+2+2(k+\alpha) = 4k+2\alpha+2$ and the average biased outcome is $(2\alpha - 1)N - (4k + 2\alpha + 2)D$ (a thin-dotted line). We try to fit our data points with a regression line $a(2\alpha - 1) - b(4k + 2\alpha + 2)$ and obtain these values (depicted as a solid line in Figure 3): for $k = 1$: $a = 385$ and $b = 17$, and for $k = 2$: $a = 373$ and $b = 16$. These parameters are quite accurate comparing to conditions $N = 400$ and $D = 19$.

In Figure 3, we also discover that the impact from the dishonest nodes in case $k = 2$ is greater than in case $k = 1$. This result is reasonable since we know that the higher value k is, the higher privacy can be hold but the higher impact dishonest nodes can enforce, and so, the worse the final outcome is. Besides, all nodes output the correct results (or the final result greater than 0): for $k = 1$ when $\alpha \geq 0.67$, and for $k = 2$ when $\alpha \geq 0.75$. It means the dishonest nodes confuse the majority of nodes for $k = 1$ when $\alpha < 0.67$, and for $k = 2$ when $\alpha < 0.75$. Comparing to other recent polling protocols like [7,8], that value of α in our experiment is similar to them.

5 Related Work

We present here some recent works related to distributed polling protocols. We focus on the ones which are not based on any overlay structure and cryptographic

technique. Secret sharing schemes with homomorphisms property in [1] can be used for polling with respect to addition. Nevertheless, it does not give protection for the initial shares with the existence of dishonest nodes, and thus, the final result is likely impacted. Under the assumption about the majority of the honest nodes in the system, *Verifialable Secret Sharing Scheme* (VSS) and *Multi-party Computation protocol* (MPC) [12] privately compute the node's shares and get the output with small error. However, these techniques use cryptography and do not control the initial input. Thus, a dishonest node can share an arbitrary data, and bias the output. Other later researches based on MPC such as [2,3] have improved the time and communication complexity, but they still use cryptography. AMPC [10] provided users anonymity without using cryptography but this structure used the notion of group. E-voting protocol [9], based on AMPC and enhanced check vectors, is the information-theoretically secure protocol. But it defines different roles for users and thus, is different from our direction.

DPol [7,8] is a simple distributed polling protocol in a social network without using cryptography where nodes are concerned about their reputation. It ensures privacy and accuracy despite the presence of dishonest nodes by means of combination of secret sharing and verification procedures. By the way, DPol also remains some shortcomings. Firstly, DPol relies on a cluster-ring-based structure, which is on top and really apart from the social graph. It does not take into account any social links between nodes in the sense that it uses the uniform assignment of nodes to group. This is not practical as we have to target a special one using notion of group instead of reserving the normal structure of the graph. Moreover, the number of nodes should be a perfect square such that a graph with N nodes can be divided into $\sqrt{N}$ groups of size $\sqrt{N}$. On the contrary, we propose a protocol deployed in a more general structure. We can observe that the graphs of family $\mathcal{G}_2$ includes the overlay structure presenting DPol (see full details in [14]). Likewise, *AG-S3* [5] can be used for polling in a scalable and secure way, but it uses the same structure as DPol.

6 Conclusion

In this paper, we proposed a design of a distributed polling protocol and defined a family of social graphs. We proved the structures of our family of graphs constitute necessary and sufficient condition to assure privacy and accuracy properties of the protocol with the presence of dishonest nodes. To detect dishonest nodes' misbehaviors, we presented verification procedures by using routing table and shortest path scheme. Furthermore, a small but useful technique based on shortest path scheme was introduced to prevent a node from receiving/sending so many duplicated messages without losing any necessary information. Unlike other works, we considered a protocol with a more general family of graphs, but obtained some similar results. More specifically, we achieved the same maximum number of votes that dishonest coalition can reveal, and the same impact from the coalition to the final output. In the future work, we plan to design an efficient polling protocol that can be deployed in the real-world network with the presence of failure and message loss.

References

1. Benaloh, J.C.: Secret Sharing Homomorphisms: Keeping Shares of a Secret Secret. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 251–260. Springer, Heidelberg (1987)
2. Damgård, I., Ishai, Y., Krøigaard, M., Nielsen, J.B., Smith, A.: Scalable Multiparty Computation with Nearly Optimal Work and Resilience. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 241–261. Springer, Heidelberg (2008)
3. Damgård, I., Nielsen, J.B.: Scalable and Unconditionally Secure Multiparty Computation. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 572–590. Springer, Heidelberg (2007)
4. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Ruppert, E.: Secretive Birds: Privacy in Population Protocols. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 329–342. Springer, Heidelberg (2007)
5. Giurgiu, A., Guerraoui, R., Huguenin, K., Kermarrec, A.-M.: Computing in Social Networks. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 332–346. Springer, Heidelberg (2010)
6. Gubichev, A., Bedathur, S.J., Seufert, S., Weikum, G.: Fast and accurate estimation of shortest paths in large graphs. In: CIKM, pp. 499–508 (2010)
7. Guerraoui, R., Huguenin, K., Kermarrec, A.-M., Monod, M.: Decentralized Polling with Respectable Participants. In: OPODIS, pp. 144–158 (2009)
8. Guerraoui, R., Huguenin, K., Kermarrec, A.-M., Monod, M., Vigfusson, Y.: Decentralized polling with respectable participants. J. Parallel Distrib. Comput. 72(1), 13–26 (2012)
9. Malkhi, D., Margo, O., Pavlov, E.: E-voting without 'Cryptography' (Extended Abstract). In: Blaze, M. (ed.) FC 2002. LNCS, vol. 2357, pp. 1–15. Springer, Heidelberg (2003)
10. Malkhi, D., Pavlov, E.: Anonymity without 'Cryptography' (Extended Abstract). In: Syverson, P.F. (ed.) FC 2001. LNCS, vol. 2339, pp. 117–135. Springer, Heidelberg (2002)
11. Mislove, A., Post, A., Druschel, P., Gummadi, P.K.: Ostra: Leveraging trust to thwart unwanted communication. In: Crowcroft, J., Dahlin, M. (eds.) NSDI, pp. 15–30. USENIX Association (2008)
12. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In: Johnson, D.S. (ed.) STOC, pp. 73–85. ACM (1989)
13. Sirivianos, M., Kim, K., Yang, X.: Socialfilter: Introducing social trust to collaborative spam mitigation. In: INFOCOM, pp. 2300–2308 (2011)
14. Thien, H.B., Imine, A.: On the Polling Problem for Social Networks. Research Report RR-8055, INRIA (2012)
15. Tran, D.N., Min, B., Li, J., Subramanian, L.: Sybil-resilient online content voting. In: Rexford, J., Sirer, E.G. (eds.) NSDI, pp. 15–28. USENIX Association (2009)
16. Yu, H., Gibbons, P.B., Kaminsky, M., Xiao, F.: Sybillimit: A near-optimal social network defense against sybil attacks. IEEE/ACM Trans. Netw. 18(3), 885–898 (2010)
17. Yu, H., Kaminsky, M., Gibbons, P.B., Flaxman, A.D.: Sybilguard: defending against sybil attacks via social networks. IEEE/ACM Trans. Netw. 16(3), 576–589 (2008)

Non-deterministic Population Protocols

Joffroy Beauquier[1,2], Janna Burman[1,2], Laurent Rosaz[1], and Brigitte Rozoy[1,2]

[1] LRI, Université Paris Sud XI, France
[2] INRIA Saclay - Ile de France, Grand Large project
{jb,burman,rozoy,rosaz}@lri.fr

Abstract. In this paper we show that, in terms of generated output languages, non-deterministic *population protocols* are strictly more powerful than deterministic ones. Analyzing the reason for this negative result, we propose two slightly enhanced models, in which non-deterministic population protocols can be *exactly* simulated by deterministic ones. First, we consider a model in which interactions are not only between couples of agents, but also between triples and in which non-uniform initial states are allowed. We generalize this transformation and we prove a general property for a model with interactions between any number of agents. Second, we simulate any non-deterministic population protocol by a deterministic one in a model where a *configuration* can have an *empty output*.

Non-deterministic and deterministic population protocols are then compared in terms of inclusion of their output languages, that is, in terms of solvability of problems. We present a transformation realizing this inclusion. It uses (again) the natural model with interactions of triples, but does not need non-uniform initial states. As before, this result is generalized for the natural model with interactions between any number of agents.

Note that the transformations in the paper apply to a whole class of non-deterministic population protocols (for a proposed model), in contrast with the transformations proposed in previous works, which apply only to a specific sub-class of protocols (satisfying a so called "elasticity" condition).

1 Introduction

Population protocols have been introduced [2] as a computation model (of functions or predicates) for asynchronous networks of simple (anonymous, resource limited) mobile agents, interacting pairwise. A characterization of what can be computed in this model is given in [4], namely the first order predicates in Presburger arithmetic. There, the protocols are assumed to be deterministic, meaning that, when two agents interact, there is a unique executable transition. The computational power of non-deterministic population protocols has been only partially studied in [1,5].

The question concerning the comparison, in terms of computability and expressiveness, of deterministic and non-deterministic machines is a natural question in all computation models. Concerning population protocols, this question appears at different levels.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 61–75, 2012.

As population protocols were originally introduced in the context of function and predicate computation [2], at the first level, the question is whether or not deterministic and non-deterministic population protocols compute the same functions and the same predicates (in the sense of [2,4]).

The second level concerns expressiveness in general. Some common method to define *any problem* (and not only a problem of function or predicate computation) is to define a set of (correct) *execution sequences* (see, e.g., [10]). An execution of a population protocol generates an *output sequence*, each configuration being associated to an output value. Thus, a problem can be also defined by the set of (correct) output sequences. Then, a population protocol can be defined *to solve a problem*, if its (non-empty) set of output sequences is included in the set of output sequences characterizing the problem.[1] At this level, the question is whether or not non-deterministic population protocols solve the same family of problems as the deterministic ones. In other words, are they equivalent in terms of the problems they can solve? This issue is not only theoretical. Indeed, implementing non-determinism is usually made by randomization. However, non-determinism is not randomization. Why using randomization, if a deterministic solution for the same problem is available? At the same time, designing a non-deterministic solution is sometimes easier and more elegant than the equivalent deterministic one. Thus, the availability of automatic transformers of non-deterministic protocols into deterministic ones, could be of some help for a developer. Note that such transformers are generally a by-product of the study on expressiveness.

The third level concerns the generating power of population protocols. In finite automata and language theory, a label is associated to each transition, so that an execution generates a word, and an automaton produces a language. The Rabin and Scott construction [9] shows that, in terms of generated languages, non-deterministic and deterministic finite automata are equivalent, both generating the family of regular languages. However, language theory is related to programming language analysis and compilation, so its tools and outcomes hardly apply in a model of mobile agents. For instance, the definitions of non/determinism in population protocols are very different. Even if the rules defining a population protocol are deterministic, the resulting global transition system is not, because of the unpredictable interactions assumed between the agents. What is more relevant for the study of the generating power of population protocols is to consider and compare (for *equivalence* or *inclusion*) the generated languages of output sequences.

To motivate the study on this third level, note that proving *inclusion* of the output language of a protocol in the output language of another one implies that the former protocol solves the same problem as the latter. This may appear useful in practice, as already explained before in context of solvability of problems.

[1] One can see the terms "problem", "output sequence" and "solving a problem" as equivalent to the terms "behavior", "output trace" and "implementing a behavior", respectively. These terms are used in the literature about population protocols as well (see, e.g., [5,7]).

At the same time, having *equivalence* of generated output languages can be also of practical help. For instance, if an implementation of a deterministic version of a protocol is preferable to that of a non-deterministic one (e.g., due to some development cost reasons), it can be useful and even necessary to have the same set of output sequences generated by the corresponding deterministic protocol. For instance, one reason may be efficiency in terms of time complexity. That is, e.g., the average complexity or the complexity of prevalent execution scenarios could be much better when concerning the larger set of executions/output sequences. Another reason may be the necessity to perform statistics over the whole set of execution/output sequences that can be generated by the non-deterministic protocol. Thus, it will be helpful to study whether the deterministic version of the protocol generates the same language.

Now, we summarize what is already known and what are the new results in this paper about population protocols in terms of the three types of questions explained above. First, the question about the computational power (in terms of predicate or function computability) of non-deterministic population protocols has been already raised in [3]. One can consider it received an answer in [1], where it is actually only stated that the non-deterministic population protocols compute exactly the Presburger predicates, exactly like the deterministic ones.[2]

In the context of problem solvability in general, (rather than in the context of computability of predicates or functions), [5] proves that the protocols solving the, so called, *elastic problems* (*elastic behaviors*, in terms of [5]), have a deterministic counterpart solving the same problem. To define elastic problem, first, define the repetition closure of a sequence t as the set of sequences obtainable from t by repeating each element of t one or more times. Extend this definition to a set of sequences O by taking the union of repetition closures of every sequence $t \in O$. Then, O is said *elastic* if it is closed by repetition closure. An elastic problem is a problem characterized by an elastic set of output sequences. Note however that this result of [5] does not imply that the output language of the deterministic counterpart is included in the output language of the given non-deterministic protocol. Still, one can deduce the following different result for some smaller class of protocols that we call *strongly elastic*.

A population protocol is strongly elastic, if for every rule $(p, q) \rightarrow (p', q')$ of the protocol, there is an idempotent rule $(p, q) \rightarrow (p, q)$. Then, it can be easily deduced from [5] that, if a problem is elastic and if there exists a strongly elastic non-deterministic population protocol solving this problem, then there exists a transformation giving a deterministic population protocol solving the same problem and moreover, with an output language included in the output language of the non-deterministic protocol.

However, the transformation in [5] does not provide the equality between the output languages of the strongly elastic non-deterministic population protocol and of its deterministic transformed version. In this paper, we study a way to

[2] In some unpublished submitted version, one can find only the sketch of proof of the statement. There, the proof uses a transformation technique also used in [5] and one can understand how a complete equivalence proof would use this transformation.

obtain such an equality for population protocols in general (Sec. 3). Unfortunately, we come with a counter example (Sec. 3.1). When studying carefully the reason for this negative result, it appears that a natural way for simulating the non-determinism in the transitions of a non-deterministic population protocol is to use the non-determinism in the interactions between the agents. The negative result comes from the fact that, when there are not enough possible interactions between agents, a high degree of non-determinism in the transitions cannot be simulated.

In order to circumvent this negative result, we propose (in Sec. 3.2) to increase the number of possible interactions by allowing interactions between more than two agents. Without changing the total number of agents, this allows more non-determinism. As a matter of fact, we prove that a non-deterministic population protocol with pairwise interactions can be exactly simulated by a deterministic population protocol with three agent interactions, under the assumption that the initial states of the agents may be different. We show how this result can be generalized to k agent interactions, for any integer $k > 1$.

A second attempt to obtain equality of output languages consists in modifying slightly the definition of what can be an output value of a configuration (Sec. 3.3). Thus, an *empty output* value for a configuration is introduced such that, when it appears in the output sequence, it is taken as an identity element. We show that, in this extended model, the equality of output languages of non-deterministic and deterministic population protocols is obtained.

The results about equality of output languages yield also results about inclusion. However, we try to weaken the assumptions (that are made to obtain equality) in order to obtain stronger results about inclusion. It happens that, when considering interactions with more than two agents, we do not need non-uniform initial agent states, as we assume to obtain equality (see Sec. 4). This involves that, if the model does not restrict the number of agents in the interactions, non-deterministic and deterministic population protocols are equivalent, in terms of solvability of problems.

Due to the lack of space, some of the proofs are omitted or sketched. Complete proofs can be found in the extended version of this paper [6].

2 Basic Model and Notations

As a basic model, we use the model of population protocols, as defined in [5,7]. A *population* $\hat{\mathcal{A}}$ consists of a set $\mathcal{A}$ of $\mathbf{n}$ agents together with a weakly connected directed graph $G(\mathcal{A}, E)$. An agent represents a finite state sensing device and $\mathbf{n}$ is unknown to the agents. $G(\mathcal{A}, E)$ is called the *interaction (or communication) graph*, where $E \subseteq \mathcal{A} \times \mathcal{A}$. An edge $(u, v) \in E$ represents the possibility of a communication (an interaction) between u and v in which u is the initiator and v is the responder.

Population protocols can be modeled as *transition systems*. Thus, each agent is represented by the same finite transition system. The states of agents are from a finite set Q. Each agent has a constant *input value* and different agents may have

different input values. For simplicity and as we assume constant input values, we consider the inputs as a part of the state of an agent. There is an *output value* associated to each state of an agent. A transition function δ maps each element of $Q \times Q$ to a subset of $Q \times Q$. Let $(p, q) \in Q \times Q$. If $(p', q') \in \delta(p, q)$, then $(p, q) \to (p', q')$ is called a *transition*, and $(p, q) \to \delta(p, q)$ is called a *rule*. When, two agents u, in state p, and v, in state q, interact (meet), respectively playing the roles of initiator and responder, they execute a transition $(p, q) \to (p', q')$ such that $(p', q') \in \delta(p, q)$. As a result, u changes its state from p to p' and v from q to q'. It is possible that $p = p'$ and/or $q = q'$. The transition function and the protocol are *deterministic*, if $\delta(p, q)$ always contains just one pair of states (in other words, if each rule provides just one transition). Otherwise, if $|\delta(p, q)| = k > 1$, δ and the protocol are said *non-deterministic* (then u, v execute one of the k transitions in $\delta(p, q)$ chosen non-deterministically). Let us call k the *degree of non-determinism* of the rule $(p, q) \to \delta(p, q)$. Let $d = \max_{(p,q)\in Q\times Q}\{|\delta(p, q)|\}$ be the degree of non-determinism of δ and of the protocol. For simplicity, for any non-deterministic protocol, if for some $(p, q) \in Q \times Q, |\delta(p, q)| < d$, we duplicate some pairs of states in $\delta(p, q)$ in order to obtain $|\delta(p, q)| = d$. Thus, w.l.o.g., we assume that $\forall (p, q) \in Q \times Q, |\delta(p, q)| = d$.

A population protocol is also a finite transition system whose states are called *configurations*. A *configuration* is a mapping $C : \mathcal{A} \to Q$. A subset of configurations $\mathcal{C}_0$ defines the *initial configurations*. We say that C *goes to* C' via pair (interaction) $\pi = (u, v)$, denoted $C \xrightarrow{\pi} C'$, if the pair $(C'(u), C'(v))$ is in $\delta(C(u), C(v))$ and for all $w \in \mathcal{A}\setminus\{u, v\}$, $C'(w) = C(w)$. We say that C can go to C' in one step (or C' is *reachable in one step* from C), denoted $C \to C'$, if $C \xrightarrow{\pi} C'$ for some edge $\pi \in E$. If there is a sequence of configurations $C = C_0, C_1, \ldots, C_k = C'$, such that $C_i \to C_{i+1}$ for all $i, 0 \leq i < k$, we say that C' *is reachable* from C, denoted $C \xrightarrow{*} C'$.

An *execution* is an infinite sequence of configurations $C_0, C_1, C_2, \ldots$ such that $C_0 \in \mathcal{C}_0$ and for each i, $C_i \to C_{i+1}$. The *output of a configuration* C is the multiset of the output values of agents in C. The *output word (or the output trace)* of an execution $e = C_0, C_1, C_2 \ldots$ is a sequence $O_0, O_1, O_2, \ldots$ resulting from the concatenation of the successive outputs of the configurations of e. That is, for all $i \geq 0$, O_i is the output of the configuration C_i. The set of output words of a protocol P is called the *(generated) output language* of the protocol and denoted by $L(P)$.

Let P_1 and P_2 be two protocols with sets of states Q_1 and $Q_1 \times Q'$ respectively, for some set Q'. For a state $s_2 = [s_1\ s'] \in Q_1 \times Q'$ of P_2, where $s_1 \in Q_1$ and $s' \in Q'$, $\Pi_{P_1}(s_2) = s_1$. That is, $\Pi_{P_1}(s_2)$ denotes the state of P_1 which is the projection of s_2 on P_1 (in other words, which is the mapping of s_2 to the state component of P_1). We extend the notation of Π in the natural way to configurations, sets of states or configurations, rules, transitions and executions.

A *problem* is defined by some conditions on executions, or equivalently by the sub-set of the executions that satisfy the conditions. As an output word associated to an execution can be defined to be the execution sequence itself (by defining the output of each agent as being the whole state), a problem can be

well defined by giving conditions only on output words. Thus, w.l.o.g. and for the sake simplicity, *we assume that a problem is defined by conditions on output words, i.e., by a sub-set $\mathcal{B}$ of output words.* A population protocol is said to *solve a problem*, if and only if the set of its output words $\mathcal{O}$ is non-empty and each output word $o \in \mathcal{O}$ satisfies the conditions defining the problem, i.e., $o \in \mathcal{B}$ or equivalently, $\mathcal{O} \subseteq \mathcal{B}$ (see, e.g., [10]).

The *transition graph* $G(P, \hat{\mathcal{A}})$ of a protocol P running in population $\hat{\mathcal{A}}$ is a directed graph whose nodes are all possible population configurations and whose edges are all possible transitions on those nodes. A strongly connected component of a directed graph is *final* iff no edge leads from a node in the component to a node outside.

As originally for population protocols, we assume a strong fairness condition on the executions that is called *global fairness*. An execution is said globally fair, if for every two configurations C and C' such that $C \to C'$, if C occurs infinitely often in the execution, then C' also occurs infinitely often in the execution.

3 Results about Equality of Output Languages

In this section, we study the strong relation of equality between the sets of output languages of deterministic and non-deterministic population protocols. First, we give a negative result (Theorem 1, Sec. 3.1) showing that in the basic model of Sec. 2, in terms of the equality between the sets of generated output languages, non-deterministic protocols are more powerful. Then, in sections 3.2 and 3.3, we propose two model extensions that allow to circumvent this negative result.

3.1 A Negative Result

The following example provides some simple preliminary intuition for the result stated in Theorem 1 below. Consider a population of two agents in the initial configuration (q_0, q_0) and the non-deterministic protocol P with two rules $(q_0, q_0) \to \{(q_0, q_0), (q_1, q_1)\}$ and $(q_1, q_1) \to (q_0, q_0)$. Assume that the output of an agent in P is its state. Then, each output word is an infinite concatenation of the output sequences of the form $(q_0, q_0)^k, (q_1, q_1)$, for any positive integer k. Thus, P has an output language of infinite size. However, the output language of any deterministic protocol executing on two agents (with finite states) has a finite size. This example proves the theorem below for two agent populations. Note that the same argument is wrong in larger populations, since then, intuitively, there exists a non-determinism in the choice of interactions that can lead to an infinite output language size. In the proof of the theorem, we give an example that works for a population of any size $\mathbf{n}$.

Theorem 1. *Given a population of size* $\mathbf{n}$*, the set of output languages of non-deterministic population protocols strictly contains the corresponding set of deterministic population protocols.*

Proof. It consists in exhibiting an example of a non-deterministic protocol whose output language is not equal to the output language of any deterministic protocol.

Consider a population of $\mathbf{n}$ agents and a non-deterministic population protocol P. Let $t = \mathbf{n} \cdot (\mathbf{n} - 1)$. Let P to have a single non-deterministic rule $(p_0, p_0) \rightarrow \{(p_1, p_1), (p_2, p_2), \ldots, (p_{t+1}, p_{t+1})\}$, and $t+1$ deterministic rules $(p_1, p_1) \rightarrow (p_0, p_0)$, $(p_2, p_2) \rightarrow (p_0, p_0)$, $\ldots$, $(p_{t+1}, p_{t+1}) \rightarrow (p_0, p_0)$. Let o_i be the output value associated to a state p_i. We choose $o_i = p_i$. The initial configuration of P is $C_0 = (p_0, p_0, \ldots, p_0)$. Note that the output of C_0 is the multi-set $M_0 = \{p_0, p_0, \ldots, p_0\}$. Assume, for the sake of contradiction, that there is a deterministic population protocol P' such that $L(P') = L(P)$.

In P, consider all the prefixes of execution of a type $e\_ = (C_0, C)$. There are exactly $t + 1$ different configuration output prefixes corresponding to these execution prefixes. Now, consider the concatenation of two such prefixes $e\_e\_$, which is also a prefix of a possible execution in P. The number of different configuration output prefixes for $e\_e\_$ is $(t + 1)^2$, and more generally, for the concatenation of k prefixes of $e\_$, the number is $(t+1)^k$. Denote by H_k the set of these output prefixes ($|H_k| = (t + 1)^k$). Since $L(P') = L(P)$, all the prefixes in H_k are also output prefixes of P'. However, if P' has a single configuration with *output* $\{p_0, p_0, \ldots, p_0\} = M_0$, P' can "generate" only t output prefixes of length 2, starting from C_0. More generally, P' can "generate" only t^k output prefixes composed by concatenation of k output prefixes of length 2, starting from C_0. The number $t = \mathbf{n} \cdot (\mathbf{n} - 1)$ is the maximum number of different pairs of states that $\mathbf{n}$ agents can have (with the distinction between initiator and responder).

Thus, since $|H_k| = (t + 1)^k > t^k$, but $L(P') = L(P)$, P' has necessarily more than one configuration with output $\{p_0, p_0, \ldots, p_0\} = M_0$. Assume then that P' has r different configurations with output M_0. Each of them "generates" at most t^k different concatenated (k times) output prefixes of length 2 starting with C_0. That is, P' can "generate" at most $r \cdot t^k$ different such output prefixes. However, since $L(P') = L(P)$, $r \cdot t^k$ must be at least as large as $(t+1)^k$. This involves that r is at least as large as $\frac{(t+1)^k}{t^k}$ and that, for *every* integer $k \geq 1$. A contradiction arises from the fact that r is bounded by the number of the configurations of P' which is finite. □

An immediate corollary from the proof of Theorem 1, is that one of the reasons for the theorem correctness is the assumption that each agent in population protocols has only a finite size state. One can also notice that the negative result comes from the fact that, when there are not enough possible interactions between agents, a high degree of non-determinism in the transitions cannot be simulated by any deterministic protocol. That is why, increasing the number of agents in an interaction, as in Sec. 3.2, allows to overcome the negative result.

3.2 Equality with Interactions of More than Two Agents

One way to increase the degree of non-determinism through the interactions of agents is to consider a more general population protocol model, where the interactions concern more than only two agents. The issue of considering such a

generalization was raised already in [2], but to our knowledge, it was not dealt in the literature in the context of non-deterministic protocols as in this work.Thus, to obtain the desired equality of output languages, we consider interactions involving more than two agents. Roughly, the idea is to assign a different integer from $[1, d]$ to each agent, where d is the degree of non-determinism of the given non-deterministic protocol. Then, we simulate deterministically the k^{th} ($k \in [1, d]$) choice of a non-deterministic transition between two agents u and v, by an interaction between three agents u, v and an agent holding the integer k.

Let us denote by PP_k the model of population protocols in which possible interactions are between k agents or less. The definition of PP_k follows the definition of the basic model of population protocols in Sec. 2. However, for PP_k, we should generalize the definition of the transition function δ and the notion of initiator and responder. During an interaction of k' agents, $u_1, u_2, \ldots, u_{k'}$, $2 \leq k' \leq k$, we say that u_1 is the initiator, u_2 is the primary responder, u_3 is the secondary responder, and so on. Now, δ maps each element in $Q^{k'}$, for each $2 \leq k' \leq k$, to a subset of $Q^{k'}$. We first prove the following result.

Theorem 2. *Consider a non-deterministic population protocol P_1 with the degree of non-determinism d. Let $\hat{\mathcal{A}}$, be any population with $\mathbf{n} \geq d + 2$ and a complete interaction graph. Given a protocol P_1 executing on $\hat{\mathcal{A}}$ in PP_2 (the basic model of Sec. 2), there exists a deterministic population protocol P_2 executing on $\hat{\mathcal{A}}$ in PP_3 (with $d + 2$ non-uniform initial states)*[3] *and generating the same output language as P_1.*

The proof consists in, first, constructing for any population protocol P_1 a deterministic population protocol P_2 and then, in proving that $L(P_1) = L(P_2)$. Thus, we first construct P_2. As explained in Sec. 2, we assume, w.l.o.g., that all the rules of P_1 have the same degree d of non-determinism. The state of an agent in P_2 is a couple $[p\ c]$, where p is a state of P_1 (p is the projection of $[p\ c]$ on P_1, denoted $\Pi_{P_1}([p\ c]) = p$), and c is an input value of P_2 which is an integer in $[1, m], m = d + 2$. The purpose of c is to serve as a *switch* value to decide deterministically, in P_2, on a transition of a non-deterministic rule of P_1. For every initial configuration C_0 of P_1, there is one initial configuration C_0' of P_2 such that $\Pi_{P_1}(C_0') = C_0$. We make an important assumption about the input value c. During an execution, each value in $[1, m]$ is the input value c of at least one agent (in all this section, we assume $\mathbf{n} \geq m$). The output of the state $[p\ c]$ in P_2 is the output of p in P_1. To each rule $(p, p') \to \{(p_1, p_1'), (p_2, p_2'), ..., (p_d, p_d')\}$ of P_1, the construction associates three types of deterministic rules of P_2:

i. For c in $[1, d]$, $([p\ c_1], [p'\ c_2], [q\ c]) \to ([p_c\ c_1], [p'_c\ c_2], [q\ c])$
ii. For c_1 in $[1, d]$, $([p\ c_1], [p'\ c_2]) \to ([p_{c_1}\ c_1], [p'_{c_1}\ c_2])$
iii. For c in $\{d + 1, d + 2\}$, $([p\ c_1], [p'\ c_2], [q\ c]) \to ([p_{c_2}\ c_1], [p'_{c_2}\ c_2], [q\ c])$

[3] Note that this assumption cannot be used to assign identifiers to agents, if $\mathbf{n} \gg d+2$. As for population protocols, it is generally assumed that $\mathbf{n} \gg |Q|$, that implies that $\mathbf{n} \gg d + 2$. In any case, we show in the sequel (Sec. 4) that this assumption can be dropped to obtain a weaker property of inclusion for the output languages.

The intuition behind the rules of P_2 is, for any pair of states (p_2, p'_2) with $\Pi_{P_1}(p_2, p'_2) = (p, p')$, to be able to simulate any possible transition in the set $\delta(p, p')$ of P_1, and this by executing only *one* transition of P_2. For obtaining the equality of output languages, it is important to be able to execute exactly one transition for this purpose.[4] Thus, the rule of type i. serves to execute a projected transition $(p, p') \overset{(u,v)}{\rightarrow} (p_c, p'_c)$, for two agents u, v, in the case where there exists another agent holding the switch input value c. Otherwise, the rules of type ii. and iii. are provided for the case where the same "needed" switch value is unique and held either by the initiator or by the primary responder (respectively). Below, we prove the equality of the output languages of P_1 and P_2. For that, we first prove the following basic lemma that actually validates the intuitive ideas explained above.

Lemma 1. *Let C_2 be a configuration of the deterministic protocol P_2 given by the construction above. Let $C_1 = \Pi_{P_1}(C_2)$ and let C'_1 be any configuration of P_1 such that $C_1 \rightarrow C'_1$. Then, there exist a configuration C'_2 of P_2 and $C_2 \rightarrow C'_2$ such that $C'_1 = \Pi_{P_1}(C'_2)$.*

Proof. Let C'_1 be reachable from C_1 by executing a transition $(p, p') \rightarrow (p_i, p'_i)$, corresponding to the i^{th} choice in $\delta(p, p')$. As C_1 is a projection of C_2, in C_2, there are two agents, one in a state $[p\ c_p]$, and another one in a state $[p'\ c_{p'}]$. If $c_p = i$, then applying rule ii. of P_2, in configuration C_2, gives a configuration C'_2 whose projection is C'_1. Otherwise, if $c_{p'} = i$, then, by the construction of P_2, there are at least two agents with switch value equal to either $d + 1$ or $d + 2$. Then, rule iii. can be applied in C_2, which results in configuration C'_2 whose projection is C'_1. In case neither c_p, nor $c_{p'}$ is equal to i, there is at least one additional agent with a switch value equal to i. Then, rule i. can be applied in C_2, which results in configuration C'_2 whose projection is C'_1. Thus, in all cases, C'_2 is reachable from C_2. □

Proof (of Theorem 2). To prove the theorem, we first show that given any globally fair execution e_2 of P_2, its projection $e_1 = \Pi_{P_1}(e_2)$ is a globally fair execution of P_1 and thus the output word of e_2 is in the output language of P_1. Then, we show that for every globally fair execution e_1 of P_1, there is a globally fair execution of P_2 whose projection on P_1 is e_1 and thus, the output word of e_1 is in the output language of P_2.

Thus, let e_2 be a globally fair execution of P_2. The projection $e_1 = \Pi_{P_1}(e_2)$ is an execution of P_1, since, by construction, the projection of each transition of P_2 is a transition of P_1. In the following, we show that e_1 is globally fair. Let C_1 be a configuration of P_1 appearing infinitely often in e_1, and let C'_1 be a configuration reachable in one step from C_1, $C_1 \rightarrow C'_1$. Then, since e_1 is the projection of e_2, there are infinitely many configurations appearing in e_2, whose projection is C_1. Thus and by the finiteness of the states of the agents,

[4] Note, however, that by changing the model definitions, e.g., for the output words of a protocol, as in Sec. 3.3, it is possible to drop this requirement when still having the equality of output languages for non/deterministic protocols.

there is such a configuration C_2, appearing infinitely often in e_2. By Lemma 1, a configuration C_2' of P_2 whose projection is C_1' is reachable from C_2 in one step. As e_2 is globally fair, C_2' appears infinitely often in e_2. Thus C_1' appears infinitely often in e_1. That proves that e_1 is globally fair.

Now consider a globally fair execution e_1 of P_1. Consider the prefix of e_1 of length r, e_1^r, for some integer $r \geq 1$ and assume (by induction on r) that there exists a segment e_2^r , prefix of an execution of P_2, with projection e_1^r on P_1 (the basis of the induction, for $r = 1$, holds by construction). Assume that $e_1^{r+1} = (e_1^{r-1}, C_1, C_1')$ and $e_2^r = (e_2^{r-1}, C_2)$. By Lemma 1, a configuration C_2' of P_2 whose projection is C_1' is reachable from C_2 in one step. Thus, there is a prefix of an execution of P_2, e_2^{r+1}, whose projection on P_1 is e_1^{r+1}. Thus, by induction, an execution e_2 whose projection is e_1 can be built. As e_1 is globally fair and as the switch values are constant, e_2 is also globally fair. □

The result of Theorem 2 can be generalized for any k.

Theorem 3. *Consider any population $\hat{\mathcal{A}}$ with $\mathbf{n} \geq d + k^2$ and complete interaction graph. For any non-deterministic population protocol on $\hat{\mathcal{A}}$ in PP_k, there exists a deterministic population protocol on $\hat{\mathcal{A}}$ in PP_{k+1} (with $d + k^2$ non-uniform initial states) with the same output language.*

Proof Sketch. For the general case, we propose two kinds of transformation protocols in PP_{k+1}. One, denoted P_3, is a generalization of the protocol P_2 (given above for $k = 2$). Thus, in P_3, for any $k > 1$, $m = d + k^2$. During any execution, each value in $[1, m]$ is the input value c of at least one agent. To each rule of P_1, $(p_1, p_2, \ldots, p_{k'}) \rightarrow \{(p_{11}, p_{12}, \ldots, p_{1k'}), (p_{21}, \ldots, p_{2k'}), \ldots, (p_{d1}, \ldots, p_{dk'})\}$, for $2 \leq k' \leq k$, the construction associates two types of deterministic rules of P_3:

i. For c in $[1, d]$,
$([p_1\ c^1], [p_2\ c^2], \ldots, [p_{k'}\ c^{k'}], [q\ c]) \rightarrow ([p_{c1}\ c^1]), [p_{c2}\ c^2], \ldots, [p_{ck'}\ c^{k'}], [q, c])$
ii. For c in $[d + x \cdot k + 1, d + x \cdot k + k]$ and for any integer x, $0 \leq x < k'$,
$([p_1\ c^1], [p_2\ c^2], \ldots, [p'_k\ c^{k'}], [q\ c]) \rightarrow ([p_{c^x 1}\ c^1], [p_{c^x 2}\ c^2], \ldots, [p_{c^x k'}\ c^{k'}], [q\ c])$

Another transformation protocol to simulate the non-deterministic protocol P_1, denoted P_3', differs from P_3 by the value of m, the conditions on the inputs and by the transition function δ. Thus, for P_3', $m = d$. During any execution, each value in $[1, m]$ is the input value c of at least $k+1$ agents. To each rule of P_1, $(p_1, p_2, \ldots, p_{k'}) \rightarrow \{(p_{11}, p_{12}, \ldots, p_{1k'}), (p_{21}, \ldots, p_{2k'}), \ldots, (p_{d1}, \ldots, p_{dk'})\}$, for $2 \leq k' \leq k$, the construction associates the following deterministic rule of P_3': $([p_1\ c^1], [p_2\ c^2], \ldots, [p_{k'}\ c^{k'}], [q\ c]) \rightarrow ([p_{c1}\ c^1], [p_{c2}\ c^2], \ldots, [p_{ck'}\ c^{k'}], [q\ c])$. To see the correctness of the transformations, notice that Lem. 1 holds also for P_3 and P_3'. That is, given any configuration C_3 of the transformed protocol (P_3 or P_3') and its projection C_1 on P_1, for any C_1' such that $C_1 \rightarrow C_1'$, there exists a configuration C_3' such that $C_3 \rightarrow C_3'$, and $C_1' = \Pi_{P_1}(C_3')$. The rest of the correctness proof follows the proof of Theorem 2.[5] □

[5] The required memory for an agent in P_3 is larger than the one in P_3'. However, when $k \ll d$, P_3 may be more advantageous. In this case, the state space requirements for the two transformations differ only slightly, though the minimum number of agents required by P_3' may be much larger than the one of P_3.

3.3 Equality by Simulation with Empty Outputs

Theorem 1 states that there is no *Rabin and Scott*-like construction for population protocols, at least with the original definitions of [5,7]. We note that the negative property strongly depends on the definition of what an output value can be. We think this definition can be changed, without reappraisal of the basic model of population protocols. In the sequel, we investigate the way of modifying the definition of the output of a configuration, in order to get an equivalence result for the output languages. The idea we develop is to consider an *empty output* ϵ for a configuration, serving as an identity element in the monoid generated by output values of configurations. That is, we allow the empty output ϵ to be a possible output value for a configuration such that for any segment of an output word o, $(o, \epsilon) = (\epsilon, o) = o$.

Intuitively, this idea of introducing empty outputs in the model can be helpful in the following way. For instance, assume that agents, in the deterministic protocol (simulating the non-deterministic one), hold different integers used as a switch to indicate one of the possible non-deterministic choices. These switch values can be changed by the protocol. A problem arises when an agent u in a state p, holding the switch value c, interacts with an agent v in a state q, but the non-deterministic choice c' has to be simulated to obtain a specific output word (to obtain equality of output languages with the non-deterministic protocol). In this situation, one would like to perform some transitions (called *null-transitions*, in the sequel) to obtain a configuration where the switch value c' is in u and the rest of the states of u and v stays unchanged. However, the outputs of the intermediary configurations reached by these null-transitions are repetitions of the same value. This may result in an output word that is not in the output language of the corresponding non-deterministic protocol. With empty outputs, it is possible to remove such repetitions of the same output and obtain the equality of the output languages. Notice that a difficulty comes from the fact that the same configuration can be reached either by a null or a non-null-transition. In the first case, it is required to output the empty output, but not in the second.

Theorem 4. *In terms of generated output languages, the non-deterministic and the deterministic population protocols are equivalent in the model allowing empty outputs for configurations.*

To prove the theorem, we present a general technique to transform the rules of *any* non-deterministic population protocol P_1 into the deterministic rules of a population protocol P_3, in the model with empty outputs. Next, we prove that $L(P_1) = L(P_3)$ (see Theorem 5). The transformation we propose, denoted D, takes as an input a protocol P_1 and another deterministic transformation D'. It is required that D' applied to P_1, denoted $D'(P_1)$, results in a deterministic protocol P_2 satisfying conditions defined in Property 1 below. We write $P_2 = D'(P_1)$ and $P_3 = D(D'(P_1))$. In the sequel, we show that there exists such a transformation D', e.g., the transformation presented in [5] (see Lem. 2). Recall that this transformation (in [5]) only applies to some sub-class of protocols and

does not provide the equality of languages for non/deterministic protocols even for this sub-class.

We use the following definitions to state Property 1 and to define D. Let P and P' be two protocols with sets of states Q and $Q \times Q''$ respectively, for some set Q''. A transition t of P', $(p,q) \to (p',q')$, is called a *(P-)null-transition*, if $(p,q) \neq (p',q')$, but $\Pi_P(p,q) = \Pi_P(p',q')$. Two consecutive and different configurations C_1, C_2 in an execution of P' are called *(P-)similar*, if C_2 is obtained from C_1 by a P-null-transition (that is, $\Pi_P(C_1) = \Pi_P(C_2)$).

Property 1. Let P_1 be any non-deterministic population protocol with a set of states Q_1. Protocol P_2 is said to satisfy Property 1, if it satisfies the following conditions:

1. The protocol P_2 is a deterministic protocol with a set of states $Q_1 \times Q'$, for some set Q'. The projection of the rules of P_2 on P_1, is the set of rules of P_1.
2. The output of a configuration C in P_2 is the output of the configuration $\Pi_{P_1}(C)$ of P_1.
3. For every initial configuration C_0 of P_1, there is one initial configuration C'_0 of P_2 such that $\Pi_{P_1}(C'_0) = C_0$.
4. For every two configurations C, C' of P_2, if $C \to C'$, then $C \neq C'$.
5. Let C_2 be a configuration of P_2 such that $C_1 = \Pi_{P_1}(C_2)$. Let C'_1 be a configuration of P_1 such that $C_1 \to C'_1$. Then, there exists a configuration C'_2 such that $C_2 \stackrel{*}{\to} C'_2$ and $\Pi_{P_1}(C'_2) = C'_1$. In addition, C'_2 is reachable from C_2 using a finite number of null transitions of P_2, except for the last transition that results in C'_2.

Definition of the transformation D**.** The main idea of the transformation D is to simulate $P_2 = D'(P_1)$ (satisfying Property 1) while eliminating the effect of the P_1-null-transitions of P_2 in the output words. This is done by introducing empty outputs, for obtaining the equality of output languages. Now, we define the protocol $P_3 = D(P_2)$. Let Q_1 and $Q_2 = Q_1 \times Q'$ be the sets of states of P_1 and P_2, respectively. Starting from P_2, we build a deterministic population protocol P_3, which has a lot of similarities with P_2, but differs mainly in the definition of states (configurations) and configuration outputs. The set of states of P_3 is $Q_3 = Q_2 \times Q_2$. Then, a configuration of P_3 can be viewed as a pair (C^*, C) of configurations of P_2. For every transition (rule) of P_2, $(p,q) \to (p',q')$, D associates a transition $([p^*\ p],[q^*\ q]) \to ([p\ p'],[q\ q'])$ of P_3. Thus, iff $C \to C'$ in P_2, then $(C^*, C) \to (C, C')$ in P_3. In an execution of P_3, a component C^* of a configuration (C^*, C) can be viewed as the previous configuration in the corresponding execution of P_2, and C can be viewed as the actual configuration. The reason of doing that is to be able to locate P_1-similar configurations in an execution (resulting from the null-transitions in P_2) and "eliminate" their output from the output word. Thus, the output of a configuration (C^*, C) is defined to be the empty output ϵ, if C^* and C are P_1-similar. Otherwise, the output of (C^*, C) is the output of C in P_2 (which is the output of $\Pi_{P_1}(C)$ in P_1, by Property 1). For every initial configuration C_0 of P_2, (C_0, C_0) is the initial configuration of P_3.

Due to the lack of space, the formal proof of the following theorem is omitted and can be found in [6].

Theorem 5. *Consider a population protocol model allowing empty outputs for configurations. Let P_1 be any non-deterministic population protocol and a protocol $P_2 = D'(P_1)$ (satisfying Property 1). Let $P_3 = D(P_2)$. Then, $L(P_1) = L(P_3)$.*

Lemma 2. *Given any non-deterministic population protocol P_1 and the transformation D' presented in [5], $D'(P_1) = P_2$ is a deterministic protocol satisfying Property 1.*

Proof. All conditions of Property 1, except the last, trivially hold for P_2, by the construction of D' in [5]. Condition 5 holds by lemmas 3.1 and 3.2 in [5] and the fact that the statements of these lemmas are achieved by executing P_1-null-transitions only, as it is shown in their proofs. □

4 Inclusion with Interactions of More than Two Agents

In this section, we consider the weaker requirement of inclusion of output languages of non/deterministic protocols. A natural way to obtain that is, like in Sec. 3.2, to allow interactions with three (or more) agents. The idea is to use the secondary responder with a required switch value to be always able to execute deterministically any possible transition of the non-deterministic transition function. It appears that, when only inclusion is required (in contrast with Sec. 3.2), it is not necessary for the switch values to be initially distinct. Indeed, we provide a protocol (Protocol 1 below) that, starting from a symmetrical initial configuration, distributes the different switch values between the agents. The idea of Prot. 1 is to generalize a (circulating) leader election population protocol proposed in [1] to manage several (instead of one) leader marks (which we call here tokens). Note that Prot. 1 cannot be used to obtain equality.

Thus, we propose a deterministic protocol Prot. 1 that distributes tokens of m ($\mathbf{n} \geq m \geq 1$) different types (represented by integers in $[1, m]$) between $\mathbf{n}$ agents. By Lem. 3 proven in [6], eventually, there is exactly one token of each type and every agent holds at most one token.[6]

It is assumed that there are at least m agents and that initially, each agent holds one token of each type. Note that these initial states are uniform and the protocol works in any PP_k model, for any integer $k > 1$.

Lemma 3. *Eventually, in each configuration reached by an execution of Prot. 1, there is exactly one token of each type and every agent holds at most one token.*

[6] Note that the property given by the lemma does not state that eventually the same token stays with the same agent. On the contrary, the protocol ensures that the tokens are always exchanged between the agents (lines 2-3). This makes the protocol work for populations with interaction graph of *any* topology.

Protocol 1. To distribute m tokens of different types between n agents

Initialization:

Every agent x has a set $T_x = \{t_1, t_2, \ldots, t_m\}$ of $m \geq 1$ different tokens. For every pair of agents x, y, $T_x = T_y$.

1: **when** an initiator x interacts with a primary responder y **do**
2: **if** $(T_x \cap T_y) = \emptyset \bigwedge |T_x| = |T_y|$ **then**
3: $T' \leftarrow T_x, T_x \leftarrow T_y, T_y \leftarrow T'$ // *exchange the tokens*
4: // *distribute the tokens*
5: **if** $(T_x \cap T_y) \neq \emptyset \bigwedge (T' \leftarrow (T_x \cap T_y) = \{t'_1, t'_2, \ldots, t'_{|T'|}\})$ **then**
6: $T_y \leftarrow T_y \setminus \{t'_1, t'_2, \ldots, t'_{\lceil \frac{|T'|}{2} \rceil}\}$
7: **if** $|T' > 1|$ **then**
8: $T_x \leftarrow T_x \setminus \{t'_{\lceil \frac{|T'|}{2} \rceil + 1}, \ldots, t'_{|T'|}\}$
9: **if** $(T_x \cap T_y) = \emptyset \bigwedge (dif \leftarrow |T_x| - |T_y|) > 1 \wedge (T_x = \{t^x_1, t^x_2, \ldots, t^x_{|T_x|}\})$ **then**
10: $size_x \leftarrow |T_x|, size_y \leftarrow |T_y|$
11: $T_x \leftarrow \{t^x_1, \ldots, t^x_{\lceil \frac{dif}{2} \rceil + size_y}\}$
12: $T_y \leftarrow T_y \cup \{t^x_{\lceil \frac{dif}{2} \rceil + size_y + 1}, \ldots, t^x_{size_x}\}$

Now, given a non-deterministic population protocol P_1 in PP_2 (or in PP_k, for any integer $k > 1$), we build a deterministic version PDI of P_1 (given by Protocol 2 below) such that $L(PDI) \subseteq L(P_1)$ (see Theorem 6). That is, the deterministic protocol PDI solves the same problems as P_1. To build PDI, we combine Prot. 1 with protocol P_2 (for PP_3), or with its generalization P_3 (for PP_{k+1}), constructed in Sec. 3.2. In the following, either of these protocols is denoted by P^*. In Sec. 3.2, P^* is constructed in such a way that $L(P_1) = L(P^*)$. For that, in particular, non-uniform initial states are assumed by the transformations in Sec. 3.2. Here, to achieve only inclusion, we can drop this assumption with the help of Prot. 1.

Thus, the composition PDI is obtained by taking the Cartesian product of the state sets of Prot. 1 and P^*, and by updating the states for each protocol independently. The output of a configuration C in PDI is the output of the configuration $\Pi_{P^*}(C)$ in P^*.

Theorem 6. *Consider any population $\hat{\mathcal{A}}$ with $\mathbf{n} \geq d + k^2$ and with complete interaction graph. Let P_1 be a non-deterministic population protocol in PP_k (for any integer $k > 1$) on population $\hat{\mathcal{A}}$. Then, the protocol PDI, given by Protocol 2, is the deterministic version of P_1 on $\hat{\mathcal{A}}$ in the model of PP_{k+1} such that $L(PDI) \subseteq L(P_1)$. That is, the deterministic protocol PDI solves the same problems for $\hat{\mathcal{A}}$, in PP_{k+1}, as the non-deterministic protocol P_1, in PP_k.*

Proof. In the composition PDI, P^* reads the variables of Prot. 1 (the content of the set of tokens T), on each interaction (lines 5,7, Prot. 2). However, Prot. 1 neither reads, nor writes in the variables of P^*. Thus, for PDI, the conditions of a *fair composition* [8,10] holds, as well as Lem. 3. Thus and by line 8, eventually, the requirement of P^* on the switch values c is satisfied. That is, eventually, each value in $[1, m]$ is the value c of at least one agent. Thus, *eventually*,

Protocol 2. *PDI* - deterministic transformation in PP_{k+1} with uniform initial states

Initialization:

Initialize the variables of Prot. 1 and the switch variable c_x of P^* to 1 ($c_x \leftarrow 1$, for every agent $x \in \mathcal{A}$). Initialize the projection of states of PDI on P^* as in P^*.

```
1: when interaction occurs do
2:    ⟨execute transition of Prot. 1⟩
3:    ⟨execute transition of P*⟩
4:    for all agent x in the interaction do
5:       if |T_x| > 1 then
6:          c_x ← 1
7:       else if T_x = {t_i} then
8:          c_x ← i
```

Lem. 1 holds for the projection of PDI on P^*. Then, by Lem. 1, exactly as in the proof of Theorem 2, one proves that given any globally fair execution e_4 of PDI, $\Pi_{P_1}(e_4)$ is a globally fair execution of P_1. Then, the output word of e_4 is in the output language of P_1. □

Acknowledgments. The authors would like to thank the reviewers for their thoughtful comments and suggestions.

References

1. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably Computable Properties of Network Graphs. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 63–74. Springer, Heidelberg (2005)
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: PODC, pp. 290–299 (2004)
3. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. DC 18(4), 235–253 (2006)
4. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. DC 20(4), 279–304 (2007)
5. Angluin, D., Aspnes, J., Fischer, M.J., Jiang, H.: Self-stabilizing population protocols. TAAS 3(4) (2008)
6. Beauquier, J., Burman, J., Rosaz, L., Rozoy, B.: Non-deterministic population protocols (extended version). Technical Report hal-00736261, INRIA (2012)
7. Fischer, M., Jiang, H.: Self-stabilizing Leader Election in Networks of Finite-State Anonymous Agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 395–409. Springer, Heidelberg (2006)
8. Herman, T.: Adaptivity through Distributed Convergence. Ph.D. Thesis. University of Texas at Austin (1991)
9. Rabin, M.O., Scott, D.: Finite automata and their decision problems 3(2), 114 (1959)
10. Tel, G.: Introduction to Distributed Algorithms, 2nd edn. Cambridge University Press (2000)

Stochastic Modeling of Dynamic Distributed Systems with Crash Recovery and Its Application to Atomic Registers

Silvia Bonomi[1], Andreas Klappenecker[2], Hyunyoung Lee[2], and Jennifer L. Welch[2]

[1] Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy
[2] Texas A&M University, College Station, TX 77843–3112, USA

Abstract. In a dynamic distributed system, processes can join and leave the system. We consider such a system in which processes are subject to crash failures from which they may recover. Assuming a stochastic model for joining, leaving, crashing, and recovering of processes, we provide a probabilistic analysis of the long-term behavior of the system. As an example of the utility of our modeling, we provide a specification and implementation of an atomic register in such a system. The dynamic nature of the system can cause all active processes to leave or crash, leaving the system in a dormant state. We analyze the average time spent in dormant states that can give us some insight into the behavior of the register system.

Keywords: Stochastic Modeling, Dynamic Distributed System, Dynamic Atomic Register.

1 Introduction

Dynamic distributed systems are characterized by the evolution of a set of processes over time. Processes can join the system, participate in the computation, and subsequently leave. Examples of such systems include data centers, peer-to-peer systems, ad hoc networks, battery powered sensor networks, and many more. Due to the dynamic nature of the system, one cannot, in general, precisely predict the number of processes that will be in the system at a given time. However, it is often possible to give a stochastic model that describes the joining and leaving behavior of the processes.

The voluntary arrival and departure of processes is not the only source of dynamicity. Additionally, hardware and software errors may cause processes to crash. Obviously, one cannot precisely predict the occurrence of, say, a hardware failure. However, one can often give a fairly accurate stochastic model for such failures in this case as well. Since it is often possible to find a remedy for the crash (for instance, by replacing a faulty device by a new one or by rebooting a computer), we allow for recovery from crashes.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 76–90, 2012.

In this paper, we give a comprehensive stochastic model for joining, crashing, recovering, and leaving of processes using a two-dimensional continuous-time Markov model. Even though higher-dimensional Markov models often resist analysis, we succeed in this case by determining its stationary distribution. This allows us to determine the average time the system spends in dormant states without active processes.

Our model is a good fit for distributed shared storage systems. Distributed storage systems often have to deal with large fluctuations in load; for example, the number of replicas can be increased to perform local reads efficiently and can be decreased in periods of low load to reallocate resources. Furthermore, if the processes are prone to crash, it makes sense to increase the number of replicas in an effort to preserve the system state. As an example of the utility of our model, we study a simple distributed shared storage system, i.e., a multi-writer multi-reader atomic register. We propose a definition of atomicity that is tailored for dynamic systems, provide the algorithms for joining, reading, writing, and leaving, and give their correctness proofs. We also determine the average time that the system spends in a dormant state during which a register might lose its state.

2 Related Work

Churn Models. The study of the churn phenomenon has received much attention in recent years, especially in the context of peer-to-peer networks. Inspired by real traces showing high churn rates and short session times in file-sharing peer-to-peer systems (e.g. [6] and [12]), some probabilistic models of the session time have been presented [12] trying to mathematically represent the evolution of a peer-to-peer network. Many studies have been carried out to model and evaluate the resiliency of an overlay network with respect to the continuous arrival and departure of processes. In particular, Poisson-based models for arrival and departure have been considered [19], [16] as well as non-exponential distribution for the lifetime [18].

All these models provide a probabilistic distribution on the number of processes in the network but they do not consider the possibility of a recovery after a failure, i.e., leaves are passive and considered equivalent to failures and a possible recovery is addressed as a new join. This modeling choice is motivated by the fact that in an overlay maintenance protocol, the state of the computation depends on the current composition of the network and it evolves over time as a consequence of the churn; thus, there is no need to consider explicitly a recovery procedure as it is equivalent to a new join. Our model is instead suitable for applications that maintain a global state independent of the membership of the system (e.g., the value of a register does not depend on the replicas that implement the memory) and that can benefit from having different procedures for crash recovery.

In [7], the authors studied how to reduce the effect of the churn on a distributed computation by intelligently selecting nodes running the application.

In [1,21] *infinite arrival models* are presented, which capture the evolution of the network removing the constraint of having a predefined and constant size n. However such models do not give any indication how the joins or the leaves happen. More recently, other models have been proposed to take into account the process behavior by considering deterministic distributions [15], [5] on the join and leave of nodes to make the analysis more tractable.

Registers in Churn Prone Environments. Several recent works (e.g., [2], [9], [10], [11], [20]) address the implementation of registers in a dynamic distributed system characterized by quiescent[1] churn. In [20], a Reconfigurable Atomic Memory for Basic Object (RAMBO) is presented. RAMBO works on top of a distributed system where processes can join and fail by crashing. In RAMBO, the notion of churn is abstracted by a sequence of configurations.

In [2] Aguilera et al. showed that a crash-resilient atomic register can be emulated without consensus and thus in a fully asynchronous distributed system, provided that the number of reconfigurations is finite, and hence the churn is quiescent. In this work, churn is assumed to be confined in specific time intervals (reconfigurations) under the assumption that a minority of process departures occur in such intervals.

In [4], the problem of building a regular register in a distributed system prone to non-quiescent churn is considered. In particular, [4] shows that it is not possible to implement a regular register in a fully asynchronous system if the churn is non-quiescent. Moreover, it provides two algorithms that solve the problem both in a synchronous and in a partially synchronous system as soon as the churn rate satisfies specific constraints.

In [14], the notion of a dynamic regular register, i.e. a regular register that can lose its current state for a while due the to effect of churn, is introduced. In more detail, a dynamic regular register behaves as a regular register as long as at least one active process is in the system with the current state of the register; if all the active processes leave, then the state of the register is lost until processes again join and a write occurs. In this paper, we extend the probabilistic model considered in [14] by explicitly taking into account failures and possible recoveries.

In [3], Attiya et al. presented sharing memory atomically and robustly in failure-prone asynchronous distributed systems, but without taking the churn into consideration.

3 System Model

We consider a crash-prone dynamic distributed system in which processes continually enter and leave and are subject to repeated crashes from which they can recover. Processes communicate with each other through a broadcast service. We assume that processes have access to perfectly synchronized clocks and

[1] Churn is said to be quiescent if there exists a time after which arrivals and departures from the system stop long enough to allow the progress of the computation.

that there is a bound, known to the processes, on the message delays. If process p sends message m at time t using the *broadcast* service and if p stays in the system throughout the time interval $[t, t+\delta]$, then for every process q that is also in the system throughout the time interval $[t, t+\delta]$, q receives m at exactly time $t+\delta$. In addition, if any process q that stays in the system throughout $[t, t+\delta]$ receives a broadcast message m at time $t+\delta$, then every process r that stays in the system throughout $[t, t+\delta]$ receives m at time $t+\delta$, regardless of the behavior of the sending process p [2].

When a process enters the system, it executes a *join* protocol, after which it is considered *active*. An active process may decide to leave the system, at which time it executes a *leave* protocol. An active (or joining) process might crash at some point, and a crashed process might recover; in the latter case, the process executes a *recovery* protocol in order to become active again.

We divide the computation into eras with active processes and dormant periods without active processes. Specifically, let $A(t)$ denote the number of active processes at time t. We assume that at time 0 there are no active processes, $A(0) = 0$. If $t > 0$ is a time with some active processes, $A(t) > 0$, then the *era* containing t is the time interval $[t_b, t_e)$ with

$$t_b := \sup\{s | s < t \text{ and } A(s) = 0\} \quad \text{and} \quad t_e := \inf\{s | s > t \text{ and } A(s) = 0\}.$$

Between two eras is a *dormant* period without active processes.

4 The Dynamic Model

Our goal is to model the number of active and crashed processes at any given time in the system. We will use a two-dimensional Markov process for this purpose. In general, it can be extremely challenging to analyze a higher-dimensional Markov process, often leading to mathematical problems whose solution seems to elude anyone.

Two-dimensional Markov Process. Let $\mathbf{N}_0$ denote the set of nonnegative integers and $\mathbf{P} := \mathbf{N}_0 \times \mathbf{N}_0$ the set of pairs of nonnegative integers.

The dynamics of the active and crashed processes can be modeled by a two-dimensional Markov process. Indeed, let $X_A(t)$ and $X_C(t)$ respectively denote the number of active and crashed processes at time t. Then

$$\{(X_A(t), X_C(t)) : t \geq 0\}$$

is a continuous-time Markov process with states in $\mathbf{P}$.

We assume that processes arrive following a Poisson process with rate $\lambda > 0$. A joining process will either become active with probability p_{JA} or immediately

[2] A message service that provides only an upper bound δ on the delivery time can easily be transformed into one with a precise delivery time δ because of the perfectly synchronized clocks: the sender tags each message with the sending clock time t and the recipient delays processing of the message until its clock reads $t+\delta$.

crash with probability p_{JC}, independent of other processes. Therefore, the external arrival of active processes is governed by a Poisson process with rate λp_{JA}, and the external arrival of crashed processes is governed by a Poisson process with rate λp_{JC}.

We assume that the time T that an active process stays in the system until it either leaves or crashes is exponentially distributed with parameter μ_A, written shortly as $T \sim \exp(\mu_A)$. An active process will leave with probability p_{AL} or crash with probability p_{AC}, independently of other processes. Applying Bernoulli sampling to the time of the active process yields that the time T_L until the process leaves satisfies $T_L \sim \exp(\mu_A p_{AL})$, and the time T_C until the process crashes satisfies $T_C \sim \exp(\mu_A p_{AC})$. Thus, the time T until an active process either leaves or crashes is indeed given by $T = \min\{T_L, T_C\}$, as $T \sim \exp(\mu_A) = \exp(\mu_A p_{AL} + \mu_A p_{AC})$.

Similarly, we assume that the time T' that a crashed process stays in the system is exponentially distributed with parameter μ_C. A crashed process will either recover with probability p_{CA} or leave with probability p_{CL}. Thus, the time T'_R until a crashed process recovers is exponentially distributed with parameter $\mu_C p_{CA}$, and the time T'_L until a crashed process passively leaves is exponentially distributed with parameter $\mu_C p_{CL}$.

Given these assumptions, we can now summarize the entire Markov process. A state of this Markov process is given by a pair $x = (a, c)$ of nonnegative integers, where a denotes the number of active processes and c denotes the number of crashed processes. We assume that the transition rates from one state to the next are exponentially distributed with the following transition rates:

$$\begin{aligned}
q((a,c),(a+1,c)) &= \lambda p_{JA} =: \iota,\\
q((a,c),(a,c+1)) &= \lambda p_{JC} =: \kappa,\\
q((a,c),(a-1,c)) &= a\mu_A p_{AL} =: a\,\mu,\\
q((a,c),(a,c-1)) &= c\mu_C p_{CL} =: c\,\nu,\\
q((a,c),(a-1,c+1)) &= a\mu_A p_{AC} =: a\,\alpha,\\
q((a,c),(a+1,c-1)) &= c\mu_C p_{CA} =: c\,\beta,
\end{aligned}$$

$q((a,c),(a,c)) = -\sum_{y\in\mathbf{P}} q((a,c),y)$, and 0 otherwise. We will use the index-free abbreviations $\alpha = \mu_A p_{AC}$ for the crashing rate, $\beta = \mu_C p_{CA}$ for the recovery rate, $\mu = \mu_A p_{AL}$ for the active leaving rate parameter, $\nu = \mu_C p_{CL}$ for the passive leaving rate parameter, $\iota = \lambda p_{JA}$ for the joining rate of active processes, and $\kappa = \lambda p_{JC}$ for the joining rate of crashing processes. The state transitions are illustrated in Fig. 1.

Ergodicity. A continuous-time Markov chain on the countable state space $\mathbf{P} = \mathbf{N}_0 \times \mathbf{N}_0$ is called ergodic if and only if there exists a probability measure π sastisfying the balance equations

$$\pi(x) \sum_{y\in\mathbf{P}} q(x,y) = \sum_{y\in\mathbf{P}} \pi(y) q(y,x)$$

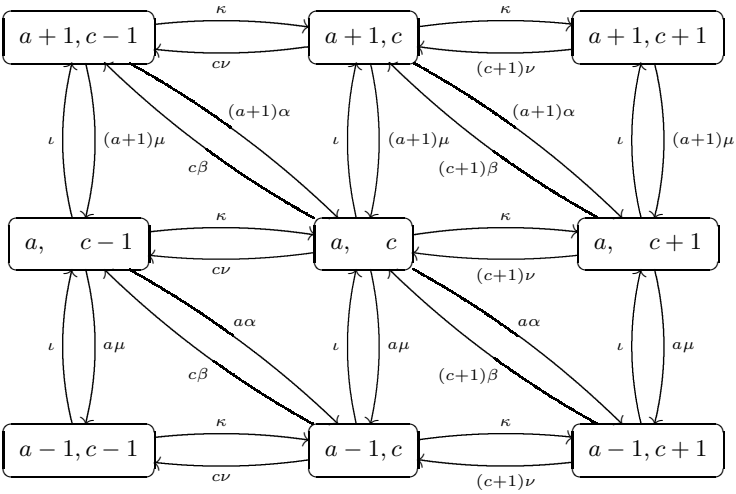


Fig. 1. The state transitions of the two-dimensional Markov chain describing the dynamics of the number of active and crashed processes. All transitions concerning the state (a, c) are shown.

for all x in $\mathbf{P}$. A probability distribution π satisfying the balance equations is called the stationary distribution. Loosely speaking, the balance equations equate the probabilities that "flow out" of the state x with the probabilities that "flow in". In our case, the balance equations are given by

$$\begin{aligned}\pi(a,c)\big(\iota+\kappa+a\alpha+a\mu+c\nu+c\beta\big)\\ =\pi(a-1,c)\iota+\pi(a,c-1)\kappa\\ +\pi(a+1,c-1)(a+1)\alpha+\pi(a+1,c)(a+1)\mu\\ +\pi(a,c+1)(c+1)\nu+\pi(a-1,c+1)(c+1)\beta\end{aligned} \tag{1}$$

as one can verify by inspecting Fig. 1. The border cases $\pi(0, c)$ and $\pi(a, 0)$ can be dealt with by setting $\pi(-1, b) = 0$ and $\pi(b, -1) = 0$ for all b.

The stationary distribution π of a Markov chain determines the long-term behavior of the chain,

$$\pi(a,c)=\lim_{t\to\infty}\Pr[(X_A(t),X_C(t))=(a,c)].$$

Therefore, it is of considerable interest to determine the stationary distribution of our Markov chain.

Stationary Distribution. Our next goal is to show that our Markov process is ergodic by explicitly calculating its stationary distribution. It will be convenient to first determine the total arrival rate λ_A of active processes and the total arrival rate λ_C of crashed processes.

The total arrival rate λ_A of active processes is given by the sum of the rate of newly joining processes that become active and the rate of active processes that recover from crashes,

$$\lambda_A = \iota + \lambda_C p_{CA}. \tag{2}$$

Similarly, the total arrival rate λ_C of crashed processes is given by the sum of the rate of newly joining processes that immediately crash and the rate of already active processes that crash,

$$\lambda_C = \kappa + \lambda_A p_{AC}. \tag{3}$$

Solving the system of linear equations resulting from (2) and (3) yields

$$\lambda_A = \frac{\iota + \kappa p_{CA}}{1 - p_{AC}p_{CA}} \quad \text{and} \quad \lambda_C = \frac{\iota p_{AC} + \kappa}{1 - p_{AC}p_{CA}}. \tag{4}$$

Theorem 1. *The stationary distribution of our Markov process is given by*

$$\pi(a,c) = \exp(-\lambda_A/\mu_A)\frac{1}{a!}\left(\frac{\lambda_A}{\mu_A}\right)^a \exp(-\lambda_C/\mu_C)\frac{1}{c!}\left(\frac{\lambda_C}{\mu_C}\right)^c, \tag{5}$$

where a and c are nonnegative integers.

Proof. Since π is the product measure of a Poisson measure with mean λ_A/μ_A and a Poisson measure with mean λ_C/μ_C, it is in particular a probability measure. It remains to show that π satisfies the balance equations.

Let us first assume that $a > 0$ and $c > 0$. The left hand side of (1) can be simplified as follows:

$$\pi(a,c)\big(\iota + \kappa + a\alpha + a\mu + c\nu + c\beta\big) = \pi(a,c)(\lambda + \mu_A a + \mu_C c).$$

We will show next that the right hand side of (1) can be simplified to the same form when π is of the form (5). The definition of π implies the following equalities:

$$\pi(a+1,c) = \pi(a,c)\frac{\lambda_A}{\mu_A(a+1)}, \qquad \pi(a,c+1) = \pi(a,c)\frac{\lambda_C}{\mu_C(c+1)},$$
$$\pi(a-1,c) = \pi(a,c)\frac{\mu_A a}{\lambda_A}, \qquad \pi(a,c-1) = \pi(a,c)\frac{\mu_C c}{\lambda_C},$$
$$\pi(a+1,c-1) = \pi(a,c)\frac{\lambda_A \mu_C c}{\mu_A(a+1)\lambda_C}, \quad \pi(a-1,c+1) = \pi(a,c)\frac{\lambda_C \mu_A a}{\mu_C(c+1)\lambda_A}.$$

Substituting these equations in the right hand side of (1) yields

$$\begin{aligned}
&\pi(a-1,c)\iota + \pi(a,c-1)\kappa + \pi(a+1,c-1)(a+1)\alpha \\
&\quad + \pi(a+1,c)(a+1)\mu + \pi(a,c+1)(c+1)\nu + \pi(a-1,c+1)(c+1)\beta \\
&\quad = \pi(a,c)\left(\frac{\mu_A a}{\lambda_A}\iota + \frac{\mu_C c}{\lambda_C}\kappa + \frac{\lambda_A \mu_C c}{\mu_A(a+1)\lambda_C}(a+1)\alpha\right. \\
&\qquad \left. + \frac{\lambda_A}{\mu_A(a+1)}(a+1)\mu + \frac{\lambda_C}{\mu_C(c+1)}(c+1)\nu + \frac{\lambda_C \mu_A a}{\mu_C(c+1)\lambda_A}(c+1)\beta\right) \\
&\quad = \pi(a,c)\left(\frac{\mu_A a}{\lambda_A}\iota + \frac{\mu_C c}{\lambda_C}\kappa + \frac{\lambda_A \mu_C c}{\lambda_C}p_{AC} + \lambda_A p_{AL} + \lambda_C p_{CL} + \frac{\lambda_C \mu_A a}{\mu_C \lambda_A}\mu_C p_{CA}\right)
\end{aligned}$$

where the definitions and canceling common factors were used in the latter equality. It follows from (2) and (3) that $\lambda_C p_{AC} = \lambda_A - \iota$ and $\lambda_A p_{AC} = \lambda_C - \kappa$ hold. Applying these equations to the third and last terms in the previous displayed equation and removing canceling terms, we get

$$\pi(a,c)\left(\mu_C c + \mu_A a + \lambda_A p_{AL} + \lambda_C p_{AL}\right).$$

Using (4) and simplification shows that $\lambda_A p_{AL} + \lambda_C p_{AL} = \lambda$. Therefore, we can conclude that

$$\pi(a,c)\left(\mu_C c + \mu_A a + \lambda_A p_{AL} + \lambda_C p_{AL}\right) = \pi(a,c)\left(\lambda + \mu_A a + \mu_C c\right).$$

This shows that for our choice of π, the balance equations are satisfied when $a > 0$ and $c > 0$.

A similar calculation shows that the balance equations also hold in the border cases (when $a = 0$ or $c = 0$).

Even though space constraints do not allow us to elaborate, we note that the curious product form of the stationary distribution can be explained using the theory of queuing networks [8].

Corollary 2. *The average time spent in a dormant period converges almost surely to* $\exp(-\lambda_A/\mu_A)$.

Proof. The system is dormant when there are no active processes. Let $\Delta\colon \mathbf{P} \to \mathbf{R}$ denote the characteristic function of dormant states, that is, $\Delta(0,c) = 1$ for all c, and $\Delta(a,c) = 0$ for all $a > 0$. Then

$$\begin{aligned}\sum_{(a,c)\in\mathbf{P}} \pi(a,c)\Delta(a,c) &= \exp(-\lambda_A/\mu_A) \sum_{c\in\mathbf{N}_0} \exp(-\lambda_C/\mu_C)\frac{1}{c!}\left(\frac{\lambda_C}{\mu_C}\right)^c \\ &= \exp(-\lambda_A/\mu_A).\end{aligned}$$

It follows from the ergodic theorem [22, Sect. 5.5] that the average time spent in the dormant period satisfies

$$\Pr\left[\lim_{t\to\infty}\frac{1}{t}\int_0^t \Delta(X_A(s), X_C(s))ds = \exp(-\lambda_A/\mu_A)\right] = 1,$$

which proves our claim.

The preceding corollary indicates that the fraction of time during which the system has no active processes is significant, with the exact value depending on the arriving, recovering, and leaving rate of processes.

5 Dynamic Atomic Register

As an example to show the utility of the modeling and analysis in the previous section, we now consider the problem of providing a shared register in a dynamic system with crashes and recoveries. We first give a specification of such a register and then present the algorithm together with its analysis.

5.1 Specification

Processes work together to implement a shared read-write register by replicating the state across the processes. An active process can invoke *read* and *write* operations on the simulated register. We use $\perp$ to indicate a default value of the register. The desired consistency condition is *atomicity* [17], also known as *linearizability* [13]. In our model atomicity means that each execution must satisfy the following properties:

[A1] There exists a bound B such that if an operation is invoked at time t by process p and p is active throughout the time interval $[t, t + B]$, then the operation completes.

[A2] Let CW be the set of completed write operations and VR be the set of read operations that return a non-default value. There exists a subset IW of the incomplete write operations and a total order on $IW \cup CW \cup VR$ such that (i) each read in the total order returns the value of the latest preceding write in the total order, and (ii) the total order respects the real-time order in the execution of non-overlapping operations.

[A3] A read invoked at time t in an era $[t_b, t_e)$ can return the value $\perp$ only if there is no completed write in the time interval $[t_b, t)$ and the first process to become active in this era did not receive any non-default values upon joining or recovering.

We call such a register a *dynamic atomic register*. When restricted to a single era after the first write, the definition of an atomic register in a dynamic distributed system coincides with the familiar definition of an atomic register in a static distributed system. However, the value of the register can be lost at the end of an era, and be replaced by the default value $\perp$.

5.2 Implementation

Each process p_i keeps a local variable val_i that stores (a replica for) the current value of the simulated register. Associated with the register value is a timestamp ts_i, which consists of an ordered pair of a clock time and the process id. The process also has a boolean variable $active_i$ indicating whether it is active.

When a process enters the system, it executes a *join* protocol: First it broadcasts an INQUIRY message and waits 2δ time, during which, after each δ time elapses, it broadcasts an UPDATE message with its value and timestamp. Whenever a process receives an INQUIRY message, it broadcasts an UPDATE message with its value and timestamp. Whenever the joining process receives an UPDATE message with a larger timestamp than its own during its waiting interval, it updates its own value and timestamp to those in the message. (Timestamps are compared lexicographically.) At the end of the 2δ waiting interval, the process becomes active. (A simple optimization is for UPDATE messages to be broadcast only if the sender's *val* variable is not $\perp$.)

Once a process is active, it can handle read and write requests on the simulated register. A read is done locally, by simply returning the value in the process' local

register variable. A write is done by setting the first component of the timestamp to the current time, broadcasting the value to be written together with the new timestamp in an UPDATE message, waiting δ time before updating its register variable to the written value, and completing the write operation. Whenever a process receives an UPDATE message with a larger timestamp than its own, it updates its own value and timestamp to those in the message.

Processes are subject to unannounced crashes and it is possible for a process to recover from a crash. When a process recovers from a crash, it executes a *recovery* protocol, which is exactly the same as the *join* protocol. Processes, whether active or crashed, leave the system silently, without notifying other processes.

See Algorithm 1 for the pseudocode; none is given for reacting to a *crash* event, as the process simply becomes nonresponsive and loses all its state. If a process receives multiple messages at the same time, it handles them in increasing order of the sender's id. We assume that local processing time is negligible compared to δ.

Algorithm 1. Dynamic Atomic Register Algorithm for Process p_i

```
 1: when join or recover is invoked at p_i:
 2: val_i := ⊥, ts_i := (−1, i), active_i := false,
 3: broadcast(INQUIRY); wait(δ)
 4: broadcast(UPDATE ⟨val_i, ts_i⟩); wait(δ)
 5: broadcast(UPDATE ⟨val_i, ts_i⟩); active_i := true
 6: return DONE
 7:
 8: when (INQUIRY) is received from p_j: broadcast(UPDATE ⟨val_i, ts_i⟩)
 9:
10: when (UPDATE ⟨v, ts⟩) is received from p_j:
11: if ts_i < ts then (val_i, ts_i) := (v, ts) end if
12:
13: when read() is invoked: return val_i
14:
15: when write(v) is invoked:
16: ts_i := (clock time, i)
17: broadcast(UPDATE ⟨v, ts_i⟩); wait(δ)
18: val_i := v
19: return ACK
```

5.3 Analysis of Register Algorithm

In this section we argue that our algorithm implements a dynamic atomic register. First, notice that each read occurs locally, without any waiting time, and each write takes exactly δ time. Thus setting $B = \delta$ satisfies condition [A1] of the definition of dynamic atomicity.

We now consider condition [A2]. Fix an execution of the algorithm. Let VR be the set of reads that return non-$\perp$ in the execution and CW be the set of

completed writes in the execution. Consider any read r in VR. Let v be the value returned by r, let ts be the timestamp associated with v, and let $\omega(r)$ be the write that created the pair (v, ts). Let IW be $\{\omega(r) : r \in VR\} \setminus CW$; i.e., we consider exactly those incomplete writes that wrote a value that was returned by some read in the execution.

Order the operations in the set $IW \cup CW \cup VR$ by their associated timestamps, breaking ties by putting a write with timestamp ts before all reads with timestamp ts and ordering all reads with the same timestamp in the same order that they occur in the execution (since reads are instantaneous, they do not overlap each other). We show this total order satisfies the conditions [A2.i] and [A2.ii].

Condition [A2.i] holds by the construction of the total order: each write has a unique timestamp, each read is associated with the timestamp of exactly one write, and each read with a given timestamp occurs after the associated write occurs and is separated from it only by other reads.

To show condition [A2.ii], consider two operations op_1 and op_2 such that op_1 ends before op_2 begins in the execution. We do a case analysis to show that op_1 appears in the total order before op_2.

Case 1: Suppose op_1 and op_2 are both reads. If they have the same timestamp, then the construction of the total order ensures they appear in the correct order. Suppose they have different timestamps. The nature of the broadcast primitive together with the code that compares timestamps before updating ensures that every process updates its replica in the same order and at the same times. Thus it must be that op_1's timestamp is less than that of op_2. Thus the corresponding write for op_1 occurs in the total order before the corresponding write for op_2, and op_1 appears in the total order before op_2.

Case 2: Suppose op_1 and op_2 are both writes. Their timestamps are the clock times when they are invoked. Since clocks are perfectly synchronized, the timestamp for op_1 is less than the timestamp for op_2, and thus op_1 appears in the total order before op_2.

Case 3: Suppose op_1 is a read and op_2 is a write. Since the timestamp for op_2 is the time when it is invoked, this timestamp must be larger than that of op_1. Thus in the total order op_1 occurs before op_2.

Case 4: Suppose op_1 is a write and op_2 is a read. Since op_1 lasts δ time and op_2 starts after op_1 completes, the process executing op_2 has received the UPDATE message from op_1 before op_2 begins. Thus the timestamp for op_2 is at least that of op_1, and op_1 appears before op_2 in the total order.

Finally, we consider condition [A3], which limits the situations in which a read can return $\perp$. We need the following lemma about the efficacy of the join/recover operation in keeping the register value alive during an era.

Lemma 3. *Suppose at time t in era $[t_b, t_e)$ an active process p_i has $val_i \neq \perp$. Then for all times $t' \in [t, t_e)$, every process p_j that is active at time t' has $val_j \neq \perp$.*

Proof. Suppose in contradiction there is a time when Lemma 3 is violated and let t' be the earliest such time. In other words, at t', there is some active process p_j that has $val_j = \perp$ but there is an earlier time $t < t'$ in the same era at which some process p_i has $val_i = v$, where $v \neq \perp$. The only timestamp associated with the value $\perp$ is -1, and -1 is only associated with $\perp$ (as we assume that clock values are nonnegative). Thus during an interval of time in which a process is active, it never changes its *val* variable from non-$\perp$ to $\perp$. Therefore t' must be the time at the end of an execution of join/recover for p_j when it sets $active_j$ to true, and p_j entered the system at time $t' - 2\delta$. It follows that p_j never receives an UPDATE message with a non-$\perp$ value during this join/recover.

We organize the remaining proof by a case analysis. It might be helpful to consult Fig. 2 for an illustration of the various cases.

Case 1: When p_i receives p_j's INQUIRY message, at time $t' - \delta$, p_i is active. Then p_i broadcasts its value-timestamp pair in an UPDATE message which is received by p_j at time t'. Since we have argued that p_j never receives a non-$\perp$ value, p_i must not yet have the value v. So p_i receives the value v between $t' - \delta$ and t. The only way that p_i receives a value is via an UPDATE message. Since update messages are broadcast and since both p_i and p_j are in the system for the δ interval of time preceding the arrival of this UPDATE(v) at p_i, p_j must also receive the UPDATE(v) message. Contradiction.

Case 2: When p_i receives p_j's INQUIRY message, at time $t' - \delta$, p_i is not active. Since p_i is active at time $t < t'$, though, p_i must be doing a join/recovery at time $t' - \delta$. Let s be such that p_i is joining throughout $[s - 2\delta, s]$, where $t' - \delta < s \leq t$.

Case 2.1: p_i receives an UPDATE(v) message in the interval $[s-2\delta, s-\delta)$. Then p_i broadcasts UPDATE(v) at time $s-\delta$, with $s-\delta > t'-2\delta$. Since both p_i and p_j are in the system throughout the time interval $[s - \delta, s]$, p_j receives the UPDATE(v) message from p_i at time s, with $t' - \delta < s < t'$, contradiction.

Case 2.2: p_i receives an UPDATE(v) message at some time in the interval $[s - \delta, t' - \delta)$. Then when p_i receives p_j's INQUIRY message at $t' - \delta$, p_i responds by broadcasting UPDATE(v), which is received by p_j at time t', contradiction.

Case 2.3: p_i receives an UPDATE(v) message at some time s' in the interval $[t'-\delta, t]$. Since p_i and p_j are both in the system throughout the interval $[s'-\delta, s']$, p_j also receives the UPDATE(v) message at s', with $s' \leq t < t'$, contradiction.

Suppose a read r, which occurs at time t inside era $[t_b, t_e)$, returns $\perp$. Let p_i be the process executing r. By Lemma 3, since only active processes execute reads and p_i returns val_i, it must be that at all previous times in the era, every active process has its *val* variable equal to $\perp$. Thus no write completes during $[t_b, t)$ and the first process to become active in this era never receives a non-$\perp$ value during its join/recovery. We conclude that [A3] holds.

The discussion above proves:

Theorem 4. *Algorithm 1 implements a dynamic atomic register.*

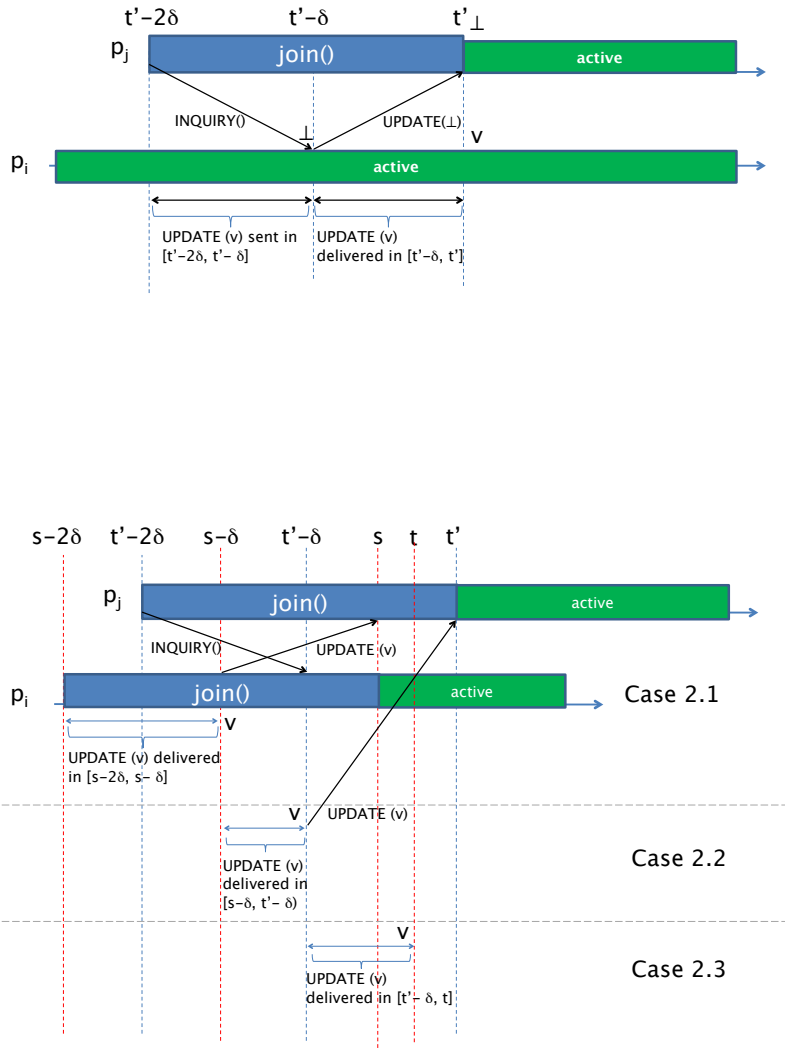


Fig. 2. The top figure illustrates Case 1: p_i is active when it receives p_j's INQUIRY. The bottom figure illustrates Case 2: p_i is not active when it receives p_j's INQUIRY. Case 2.1 represents that p_i is in the first δ time of its join when it receives UPDATE(v) and it broadcasts UPDATE(v) by line 4 of Algorithm 1. Case 2.2 is that p_i is after the first δ time of its join but before it gets p_j's INQUIRY when it receives UPDATE(v) and it broadcasts UPDATE(v) by line 8 of Algorithm 1 upon receipt of p_j's INQUIRY. Case 2.3 is that p_i gets UPDATE(v) after it has received p_j's INQUIRY.

We can augment the analysis of the algorithm in the preceding theorem by recalling the result in Corollary 2, which quantifies how long the system will have no active processes. For the atomic register application, having no active processes means that the simulated register has no state. Thus any application using the register must be able to handle such a situation.

6 Conclusions

We introduced a stochastic model for dynamic distributed systems that comprises join and leave as well as crash and recovery. Although modeling the crashing processes separately from processes leaving the system significantly complicates the stochastic model, the distinction is quite useful, as the crash rates and the leaving rates of processes usually differ significantly.

In general, it is often undesirable for a dynamic distributed system to lose its state. Our stochastic model might be helpful when making predictions about the probability of the start of a dormant period within the next t seconds. Therefore, the stochastic model can enable one to save the state of the system to a non-volatile medium before such state loss happens.

Even though dynamic distributed systems have been in widespread practical use, the development of the underlying theory of such systems has begun rather recently. We defined a notion of an atomic register that is compatible with a churn-prone environment. We gave algorithms that implement such a dynamic register and proved their correctness.

It would be desirable to extend the stochastic analysis to understand other aspects of the behavior of systems with churn. For instance, what is the probability of having fewer active processes than crashed processes? The general framework of our stochastic analysis can be extended to handle other distributions of joining, leaving, crashing, and recovering. Future work includes the best way to implement the dynamic atomic register when relaxing the assumptions about the synchronized clocks and the broadcast service.

Acknowledgements. This research was supported in part by NSF grant CCF 1018500 and NSF grant 0964696. We would like to thank the anonymous referees for helpful comments.

References

1. Aguilera, M.: A pleasant stroll through the land of infinitely many creatures. SIGACT News 35(2), 36–59 (2004)
2. Aguilera, M., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM 58(2), 7:1–7:32 (2011)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM 42(1), 124–142 (1995)
4. Baldoni, R., Bonomi, S., Kermarrec, A., Raynal, M.: Implementing a Register in a Dynamic Distributed System. In: 29th International Conference on Distributed Computing Systems, ICDCS 2009 (2009)

5. Baldoni, R., Bonomi, S., Raynal, M.: Implementing a Regular Register in an Eventually Synchronous Distributed System Prone to Continuous Churn. IEEE Transaction on Parallel Distributed Systems 23(1), 102–109 (2012)
6. Bhagwan, R., Savage, S., Voelker, G.M.: Understanding Availability. In: Kaashoek, M., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 256–267. Springer, Heidelberg (2003)
7. Brighten, G., Shenker, S., Stoica, I.: Minimizing churn in distributed systems. In: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM 2006, pp. 147–158. ACM, New York (2006)
8. Chen, H., Yao, D.: Fundamentals of Queueing Networks – Performance, Asymptotics, and Optimization. Springer, New York (2001)
9. Chockler, G., Gilbert, S., Gramoli, V., Musial, P., Shvartsman, A.: Reconfigurable distributed storage for dynamic networks. Journal Parallel Distributed Computing 69(1), 100–116 (2009)
10. Gilbert, S., Lynch, N., Shvartsman, A.: Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. In: International Conference on Dependable Systems and Networks, DSN 2003, p. 259. IEEE Computer Society, Los Alamitos (2003)
11. Gilbert, S., Lynch, N., Shvartsman, A.: Rambo: a robust, reconfigurable atomic memory service for dynamic networks. Distributed Computing 23, 225–272 (2010)
12. Gummadi, K., Dunn, R., Saroiu, S., Gribble, S., Levy, H., Zahorjan, J.: Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 314–329. ACM, New York (2003)
13. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 463–492 (1990)
14. Klappenecker, A., Lee, H., Welch, J.L.: Dynamic Regular Registers in Systems with Churn. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 296–310. Springer, Heidelberg (2011)
15. Ko, S., Hoque, I., Gupta, I.: Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In: Proceedings of the 2008 Symposium on Reliable Distributed Systems, SRDS 2008, pp. 259–268. IEEE Computer Society, Washington, DC (2008)
16. Krishnamurthy, S., El-Ansary, S., Aurell, E., Haridi, S.: A Statistical Theory of Chord Under Churn. In: van Renesse, R. (ed.) IPTPS 2005. LNCS, vol. 3640, pp. 93–103. Springer, Heidelberg (2005)
17. Lamport, L.: On interprocess communication, Part I: Models, Part II: Algorithms. Distributed Computing 1(2), 77–101 (1986)
18. Leonard, D., Yao, Z., Rai, V., Loguinov, D.: On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. IEEE/ACM Transaction on Networking 15(3), 644–656 (2007)
19. Liben-Nowell, D., Balakrishnan, H., Karger, D.: Analysis of the evolution of peer-to-peer systems. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, PODC 2002, pp. 233–242. ACM, New York (2002)
20. Lynch, N., Shvartsman, A.A.: RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 173–190. Springer, Heidelberg (2002)
21. Merritt, M., Taubenfeld, G.: Computing with Infinitely Many Processes. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 164–178. Springer, Heidelberg (2000)
22. Resnick, S.: Adventures in Stochastic Processes. Birkhäuser, Boston (1992)

When and How Process Groups Can Be Used to Reduce the Renaming Space

Armando Castañeda[1,⋆,⋆⋆], Michel Raynal[2,3], and Julien Stainer[2]

[1] Department of Computer Science, Technion, Haifa 32000, Israel
[2] Institut Universitaire de France
[3] IRISA, Université de Rennes, 35042 Rennes Cedex, France
armando@cs.technion.ac.il, {raynal,julien.stainer}@irisa.fr

Abstract. Considering the M-renaming problem and process groups, this paper investigates the following question: Is there a relation between the number of groups and the size of the new name space M? This question can be rephrased as follows: Can the initial partitioning of the processes into m groups allows the size of the renaming space M to be reduced, and if yes, how much?

This paper answers the previous questions. Let n denote the number of processes. Assuming that the processes are initially partitioned into $m = n - \ell$ non-empty groups, such that each process knows only its identity and its group number, the paper first presents a wait-free M-renaming algorithm whose size of the new name space is $M = n + 2\ell - 1$. For $\frac{n}{2} < m \leq n - 1$ (i.e. $1 \leq \ell < \frac{n}{2}$), we have $M < 2n - 1$, which shows that, when the number of groups is greater than $\frac{n}{2}$, groups allow to circumvent the renaming lower bound in read/write systems. Then, on the lower bound size, the paper shows that there are pairs of values (n, m) such that there is no read/write wait-free M-renaming algorithm for which $M \leq 2n - 2$. This impossibility result breaks our hope to have a renaming algorithm providing a new name space whose size would decrease "regularly" as the number of groups increases from 1 to n. Finally, the paper considers the case where each group includes at least s processes. This algorithm shows that, when m is such that $\frac{n}{s+1} < m < \frac{n}{s}$, there is an M-renaming algorithm where $M = 3n - (s+1)m - 1 = n(2-s) + (s+1)\ell - 1$. Hence, the paper leaves open the following question: For any n and $s = 1$, does the predicate $m > \frac{n}{2}$ define a threshold on the number of groups which allows the $2n - 2$ lower bound on the renaming space size to be bypassed?

Keywords: Asynchronous read/write model, Crash failure, Distributed computability, Process group, Renaming problem, Snapshot object, Wait-freedom.

1 Introduction

The renaming problem In the *M-renaming* problem, n asynchronous processes with distinct initial names belonging to a large name space $[1..N]$, $n << N$, have

⋆ This work was partially done while the first author was at IRISA-INRIA in Rennes.
⋆⋆ Supported in part at the Technion by an Aly Kaufman Fellowship.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 91–105, 2012.

to cooperate in order to choose distinct new names in a smaller new name space $[1..M]$, $M < N$. This problem has initially been introduced to investigate a non-trivial problem that (differently from consensus) can be solved in asynchronous read/write memory systems prone to any number of process crashes [4].

Since its introduction, lots of papers have been devoted to this problem (see [12] for an introductory survey). The most important results associated with the renaming problem concern the smallest value of M that can be attained. More precisely, it is shown in [7,10] that for an infinite number of values of n, $M = 2n - 1$ is the lower bound on the size of the new name space. For the other values of n, there exists a wait-free algorithm that solves M-renaming with $M = 2n - 2$ [11]. Here we are interested in bypassing the lower bound $M = 2n - 1$ for those values of n for which there is no M-renaming wait-free algorithm with $M < 2n - 1$.

Bypassing the limits. Several approaches have been investigated to circumvent the previous lower bound on the value of M. One consists in considering a computation model reacher than the base read/write model. As an example, it is possible to rename in an optimal new name space $M = n$, as soon as the shared memory provides processes with a test&set operation [12].

In the *k-set agreement* problem, $1 \leq k \leq n - 1$, each process proposes a value and has to decide a value such that at most k distinct values are decided and a decided value is a proposed value. Relations between the k-set agreement problem and the renaming problem have been investigated in [14,15,17] where it is shown that processes can rename in a new name space whose size is $M = n+k-1$ if they have access to k-set agreement objects. It is shown in [16] that, for any value of n, $(n-1)$-set agreement can be used to implement $(2n-2)$-renaming while the opposite is impossible when n is odd.

Another approach to circumvent the $M = 2n - 1$ lower bound consists in weakening the termination property of the renaming problem. It is shown in [21] that $(n + k - 1)$-renaming can be solved in all runs such that, after some finite time, at most k processes execute concurrently (this behavioral assumption is called k-obstruction-freedom).

In the *(g, M)-group renaming* [13] problem the processes are partitioned into g groups and it is required that processes in distinct group decide distinct new names, however it is allowed that processes in the same group decide (not necessarily) the same name. Upper and lower bounds of the (g, M)-group renaming problem are shown in [2,3].

Content of the paper. This paper considers the case where processes are initially given some additional information, but not so much (providing each process with its new name would make the problem trivial!). More precisely, each process belongs to a group, and it knows only the name of its group and the number of groups. This type of additional information has already been indirectly investigated for two specific cases. More specifically, it is shown in [16], that if the processes are initially split into two non-empty groups, then, for any value of n, $(2n - 2)$-renaming is solvable. At the other extreme, it is shown in [20] that if

the processes are arbitrarily split into $(n-1)$ non-empty groups, then, for any value of n, $(n+1)$-renaming is solvable.

This paper addresses the case where the processes are initially partitioned into several non-empty groups. Assuming m groups, one may conjecture (in the light of [16,20]) that processes in the group i, $1 \leq i \leq m$, could rename in $2q_i - 1$ new names, where q_i is the number of processes in the group i, giving rise to a new name space whose size would be $\sum_{i=1}^{m}(2q_i - 1) = 2n - m$ (since $\sum_{i=1}^{m} q_i = n$). Unfortunately, as shown in the paper, this intuition is incorrect, when considering the large class of comparison-based algorithms, i.e., algorithms in which processes can only compare their initial identities (to the best of our knowledge, all known renaming algorithms in the literature are comparison-based). This is mainly due to the fact that the size of the groups remain unknown to the processes (processes know only that no group is empty).

More generally, the paper addresses the case where the processes are split into $m = n - \ell$ non-empty groups, where $1 \leq m \leq n$, hence $0 \leq \ell \leq n-1$. To facilitate the presentation, we sometimes use $n - \ell$ (with $0 \leq \ell \leq n-1$) to denote the number m of groups. It has the following contributions.

- Assuming that the processes are arbitrarily split into $m = n - \ell$ groups, where $1 \leq \ell < \frac{n}{2}$, and each process knows the value m, the paper first presents a read/write wait-free $(n + 2\ell - 1)$-renaming algorithm.
  Let us observe that, as $\left(1 \leq \ell < \frac{n}{2}\right) \Rightarrow (n+1 \leq n+2\ell-1 \leq 2n-2)$, it follows that this algorithm allows for a renaming space smaller or equal to $2n-2$ when $\ell < \frac{n}{2}$, i.e., when the number of groups $m > \frac{n}{2}$.
- The paper then focuses on lower bounds results. It shows that there are pairs of values (n, m) for which there is no read/write wait-free comparison-based $(2n-2)$-renaming algorithm. This shows that there are cases in which groups are useless when one wants to bypass the $M = 2n-1$ lower bound.
- Then, the paper considers the case where the minimal number of processes in each group (denoted $s \geq 1$) is known by each process (the groups can be of distinct sizes). It shows that, when m is such that $\frac{n}{s+1} < m \leq \frac{n}{s}$, there is an algorithm solving $(3n - (s+1)m - 1)$-renaming if $n \neq ms$ and $((2s-1)m)$-renaming if $n = ms$.
  As $(n < (s+1)m) \Rightarrow (3n - (s+1)m - 1 < 2n-1)$ and $\big((s > 1) \wedge (n = ms)\big) \Rightarrow \big((2s-1)m = 2n - m < 2n-1\big)$, it follows that, under the constraint $n < (s+1)m \leq n+m$, the knowledge of s by the processes allows the size of the new name space to still be reduced even if $m \leq \frac{n}{2}$.

We use algebraic topology techniques for proving the lower bound results. The upper bounds are complemented with numerical experiments. It is important to stress that the research in this paper differs from [2,3,13] in the requirement that processes in the same group must decide distinct new names, contrary to [2,3,13] in which processes in the same group may decide the same new name.

Roadmap. The paper is made up of 6 sections. Section 2 presents the basics of the computation model. Section 3 presents and proves correct an $(n + 2\ell - 1)$-renaming algorithm, that assumes the processes are initially partitioned into

$m = (n - \ell)$ groups. Section 4 proves the impossibility of $(2n - 2)$-renaming for some (n, m) pairs of values. Section 5 extends the algorithm presented in Section 3 to the more general case where the minimal group size s can be greater than one, and this value is known by each process. Finally, Section 6 concludes the paper.

2 Computation Model

Due to space limitations and the fact that this model is widely used in the literature, we do not explain it in detail. We restate only its aspects that are important for the paper.

Read/write wait-free system model. This paper considers the usual asynchronous, wait-free shared memory system where at most $n - 1$ out of $n \geq 2$ processes $p_1, \ldots, p_n$ can fail by crashing. Processes communicate by accessing single-writer/ multi-reader (SWMR) atomic registers. The subscript i is called the *index* of p_i. Each process p_i has a private input, denoted id_i which is its initial name. It is known by the processes that no two of them have the same initial name.

A *participating* process is a process that takes at least one step in the considered run. Those that take a finite number of steps are *faulty* (sometimes called *crashed*), the others are *correct* (or *non-faulty*). A non-participating process is a faulty process. The algorithms designed for this computation model have to work despite up to $n - 1$ faulty processes (wait-free algorithms [18]).

Index-independent algorithm. Generally speaking, in an index-independent algorithm, indexes are used only for addressing purposes, namely, when a process p_i writes a value to an array of SWMR registers A, its index is used to deposit the value in $A[i]$, and when p_i reads A, it gets back a vector of n values, where the jth entry of the vector is associated with p_j. Moreover, such a read of A appears as if it has been executed atomically (an atomic snapshot object can be built from read/write registers [1]). The processes cannot use indexes for computation.

Formally, an algorithm $\mathcal{A}$ is *index-independent* if the following holds for every run r and every permutation $\pi()$ of the process indexes. Let r_π be the run obtained from r by permuting the input values according to $\pi()$ and, for each step, the index i of the process that executes the step is replaced by $\pi(i)$. Then r_π is a run of $\mathcal{A}$. Consider a permutation $\pi()$ such that $\pi(i) = j$. The index-independence ensures that p_j behaves in r_π exactly as p_i behaves in r: it decides the same thing in the same step. In an index-independent algorithm, if the output of p_i in r is v, then the output of $p_{\pi(i)}$ in r_π is v, i.e., the output of a process does not depend on indexes, it depends only on the inputs (ids), and on the interleaving.

Comparison-based algorithm. Intuitively, an algorithm $\mathcal{A}$ is *comparison-based* if processes use only comparisons $(<, =, >)$ on their inputs (here, initial names). More formally, let us consider the ordered inputs $i_1 < i_2 < \cdots < i_n$ of a run

r of $\mathcal{A}$ and any other ordered inputs $j_1 < j_2 < \cdots < j_n$. The algorithm $\mathcal{A}$ is comparison-based if the run r' obtained by replacing in r each i_ℓ by j_ℓ, $1 \leq \ell \leq n$ (in the corresponding process), is a run of $\mathcal{A}$. Notice that each process decides the same output in both runs, and at the same step. In other words, the decisions in r and r' are the same because the relative order of the inputs is the same.

Note that a comparison-based algorithm is not necessarily index-independent and an index-independent algorithm is not necessarily comparison-based.

The M-renaming task. In the M-renaming task on n processes [4], each process p_i starts with a distinct identity id_i from a set $\{1, \ldots, N\}$, $N \geq 2n-1$, and has to decide a value in such a way that the following properties are satisfied. (a) Termination: each correct process decides a value; (b) Validity: a decided value belongs to $\{1, \ldots, M\}$; (c) Uniqueness: No two processes decide the same value.

Here we consider a version of the renaming task in which processes are given extra information: the processes are initially partitioned into m groups, $1 \leq m \leq n-1$. More formally, each process p_i starts with a pair (id_i, gid_i) such that (1) $id_i \in \{1, \ldots, N\}$; (2) $gid_i \in \{1, \ldots, m\}$; (3) for every pair (i, j): $i \neq j \Rightarrow id_i \neq id_j$; (4) $|\{gid_i : 1 \leq i \leq n\}| = m$ and (5) $\forall gid \in \{1, \ldots, m\} : \exists i \in \{1, \ldots, n\} : gid_i = gid$.

3 From $(n - \ell)$ Non-empty Groups to $(n + 2\ell - 1)$-Renaming

Principles of the algorithm. The idea that underlies the algorithm is simple: if a group has a single process, then this process inherits the name of its group. Otherwise, the processes in non-singleton groups compete to obtain new names from a common space name.

Global and local variables. The processes communicate with a snapshot object denoted $STATE[1..n]$. For any i, $STATE[i]$ is made up of three fields (each initialized to $\perp$): $STATE[i].prop$ is the new name value currently proposed by p_i; $STATE[i].id$ is its initial name, and $STATE[i].gid$ its group name.

Each process p_i manages two main local variables: $prop_i$ which contains its current new name proposal, and $snap_i$ which is a local array where p_i saves the last value it has obtained from the snapshot object $STATE$.

Algorithm. The algorithm is described in Figure 1. It is inspired from the read/write wait-free $(2n-1)$-renaming algorithm described in [6]. Its main difference lies in the way is defined the sequence of integers from which a process extracts its next new name proposal (this is the sequence denoted $free_i$ in Figure 1).

When a process invokes new_name(id_i, gd_i), it first considers its group name as its new name proposal (line 1), and informs the other processes (line 3). It then takes a snapshot of $STATE[1..n]$ (line 4), and decides its current name proposal if it is the only process with this proposal (lines 5-6). Let us observe that, as $m + \ell = n$, no process in a singleton group proposes a new name $\geq n - \ell + 1$.

If p_i discovers that it is not in a singleton group, it has to define a new proposal. To that end, p_i computes (from its point of view as defined by $snap_i$) first the set

```
operation new_name(id_i, gid_i) is
(1)   prop_i ← gid_i;
(2)   while true do
(3)     STATE[i] ← ⟨prop_i, id_i, gid_i⟩;
(4)     snap_i ← STATE.snapshot();
(5)     if (∀ j ≠ i : snap_i[j].prop ≠ prop_i)
(6)       then return (prop_i)
(7)       else let function gcard_i(g) = |{j : snap_i[j].gid = g}|;
(8)            namespace_i ← {g : gcard(g) > 1} ∪ {n − l + 1, ..., n + 2l − 1};
(9)            let props_i = {snap_i[j].prop : snap_i[j].prop ∈ namespace_i};
(10)           let free_i  = namespace_i \ props_i;
(11)           let comp_i  = {snap_i[j].id : snap_i[j].prop ∈ namespace_i};
(12)           let r_i     = rank of id_i in comp_i;
(13)           prop_i ← the r_ith integer in the increasing sequence free_i
(14)    end if
(15)  end while.
```

Fig. 1. From $m = (n - \ell)$ non-empty groups to $(n + 2\ell - 1)$-renaming $(1 \leq \ell < \frac{n}{2})$

$namespace_i$ containing the identifiers of the groups with more than one process and the new names that are not group identifiers ($\{n - l + 1, \ldots, n + 2l - 1\}$) (line 8) from which p_i will extract its new name proposal. Then, p_i gathers the proposals of names that belong to $namespace_i$ (line 9) in order to obtain the set $free_i$ of free names from $namespace_i$ (line 10). It extracts $comp_i$, the set of identifiers of processes proposing a name in $namespace_i$ (let us notice that, $comp_i$ does not contain processes that belongs to a singleton group). Finally, p_i computes its new name proposal $prop_i$ according to the rank of its initial name in the set of competing processes $comp_i$ (lines 12-13). Then, p_i restarts competing at line 3.

Theorem 1. *Assuming that the processes are initially partitioned into $m = n-\ell$ groups ($1 \leq m \leq n-1$), and each process knows initially only its identity and its group number, the algorithm of Figure 1 is a read/write wait-free $(n + 2\ell - 1)$-renaming algorithm.*

Due to lack of space, the correctness proof of the algorithm is not presented. Let us observe that, while the algorithm works for any $\ell \in \{1, ..., n - 1\}$, it is not interesting when $\frac{n}{2} < \ell \leq n - 1$, i.e., when the number of groups m is such that $1 \leq m = n - \ell < \frac{n}{2}$. This is because, in these cases, the algorithm does not provide a renaming such that $M < 2n - 1$.

4 Is It Possible to Do Better?

As explained in the Introduction, we could hope that the initial partitioning of n processes into m groups, allows the processes in group i to independently rename into a space of size $2q_i - 1$ (q_i is the number of processes in the group),

in order to obtain a new name space of size $\sum_{i=1}^{m}(2q_i - 1) = 2n - m$. Theorem 2 shows this is not true, for the large class of comparison-based based algorithms. More formally, Theorem 2 shows that there are pair of values (n, m) such that it is not even possible to bypass the lower bound $M = 2n - 1$.

This section then focus on the case in which the number of groups has the form $m = n - \ell$, where $1 \leq \ell < \frac{n}{2}$. Theorem 3 contains a lower bound that complements that $n + 2\ell - 1$ upper bound proved by the algorithm presented in the previous section. .

The proof of of Theorem 3 is based on the known topological approach to distributed computing [8,19,22]. We assume the reader is familiar with this approach.

Theorem 2. *Let $x \geq 2$, $n = p^x$ a prime power, and $1 \leq y < x$. Let us assume that the n processes are initially partitioned into p^{x-y} groups. Then, there is no read/write, wait-free, comparison-based $(2n-2)$-renaming algorithm.*

Proof. Suppose there exists such an algorithm $\mathcal{A}$. We use the known topological approach to distributed computing for proving $\mathcal{A}$ cannot exist. In particular, we use a similar idea to the one in [10], where it is proved that the weak symmetry-breaking (WSB) problem on n' processes is not read/write, wait-free solvable if n' is a prime power.

Let $\mathcal{I}^{n-1}$ and $\mathcal{O}^{n-1}$ be the input and output complexes of the $(2n-2)$-renaming task on n processes. Consider the protocol complex $\mathcal{A}^{n-1}$ of $\mathcal{A}$. It is known $\mathcal{A}^{n-1}$ is a chromatic, connected and orientable $(n-1)$-pseudomanifold [5,8,19,22]. Since $\mathcal{A}$ solves $(2n-2)$-renaming, it induces a color-preserving, simplicial map $\delta : \mathcal{A}^{n-1} \to \mathcal{O}^{n-1}$ such that for every input simplex $\sigma \in \mathcal{I}^{n-1}$, $\delta(\mathcal{A}^{n-1}(\sigma)) \subseteq \triangle(\sigma)$, where $\triangle$ is the recursive map relating $\mathcal{I}^{n-1}$ and $\mathcal{O}^{n-1}$. Such a map represents the decisions of the processes in $\mathcal{A}$: $\delta(v)$ is the decision of the process corresponding v.

Using δ we define a binary coloring b over $\mathcal{A}^{n-1}$: for every vertex v of $\mathcal{A}^{n-1}$, $b(v) = \delta(v) \mod 2$. $\mathcal{A}^{n-1}$ cannot have monochromatic $(n-1)$-simplexes under b because (1) the space $[1, \ldots, 2n-2]$ has exactly $n-1$ odd names and exactly $n-1$ even names, and (2) δ is simplicial and color-preserving, hence every $(n-1)$-simplex of $\mathcal{A}^{n-1}$ is mapped to an $(n-1)$-simplex of $\mathcal{O}^{n-1}$, which has distinct output names at its vertexes.

Now recall that the $n = p^x$ processes are initially split into $\frac{n}{p^y} = p^{x-y}$ groups. Without loss of generality, let $0, \ldots, p^{x-y} - 1$ be those groups. Consider the input simplex $\sigma^{n-1} \in \mathcal{I}^{n-1}$ defined as follows. In σ^{n-1} each group has exactly p^y processes: for every process $p_i \in \Pi = \{p_0, \ldots, p_{n-1}\}$, $gid_i = \left\lfloor \frac{i}{p^y} \right\rfloor$ and $id_i = p^{x-y} + \left\lfloor \frac{i}{p^y} \right\rfloor + (i \mod p^y)$. Note that for each p_i, $id_i > p^{x-y} - 1 \geq gid_i$; moreover, for any two processes p_i and p_j

$$gid_i < gid_j \Rightarrow id_i < id_j. \tag{1}$$

For the rest of the proof, let us fix σ^{n-1} and the subcomplex $\mathcal{A}^{n-1}(\sigma^{n-1})$ of $\mathcal{A}^{n-1}$ (the subcomplex containing all reachable executions starting from σ^{n-1}). For simplicity, let $\widehat{\mathcal{A}}^{n-1}$ denote $\mathcal{A}^{n-1}(\sigma^{n-1})$. As already explained, $\widehat{\mathcal{A}}^{n-1}$ cannot have monochromatic $(n-1)$-simplexes. We will show that the fact $\mathcal{A}$ is comparison-based induces symmetry properties on the binary coloring b of $\widehat{\mathcal{A}}^{n-1}$. As we shall see, those properties imply $\widehat{\mathcal{A}}^{n-1}$ has at least one monochromatic $(n-1)$-simplex under b, hence such a symmetric binary coloring b cannot exist, from which follows $\mathcal{A}$ does not exist.

Consider now a connected, orientable and chromatic $(n-1)$-pseudomanifold $\mathcal{B}^{n-1}$, with a binary coloring. In [10] it is proved that the number of monochromatic $(n-1)$-simplexes, $\#R$, of $\mathcal{B}^{n-1}$ totally depends on its boundary, $bd(\mathcal{B}^{n-1})$ (Lemmas 33 and 35). Let us suppose $\mathcal{B}^{n-1}$ corresponds to the subcomplex containing all reachable executions of a read/write, wait-free algorithm $\mathcal{B}$ starting from an input $(n-1)$-simplex τ^{n-1}. What is also showed in [10] is that for each proper face ρ of τ^{n-1}, i.e., $\rho \in bd(\tau^{n-1})$, there is an integer r_ρ such that

$$\#R = 1 + \sum_{\rho \in bd(\tau^{n-1})} r_\rho.$$

Roughly, $\#R$ is on function of the boundary of $\mathcal{B}^{n-1}$, which is $bd(\mathcal{B}^{n-1}) = \cup_{\rho \in bd(\tau^{n-1})} \mathcal{B}^{n-1}(\rho)$. Thus each $\rho \in bd(\tau^{n-1})$ "adds something" (namely, r_ρ) to the value $\#R$. (we do not discuss where the value 1 in the equation comes from; this is not relevant for our purposes and demands to get into details.)

Now, if the algorithm $\mathcal{B}$ is comparison-based, then, for any two proper i-faces ρ and ρ' of τ^{n-1}, if the inputs of the processes follow the same relative order in ρ and ρ', respectively, then $r_\rho = r_{\rho'}$. Intuitively, since the inputs follow the same relative order, the decisions of the processes in $\mathcal{B}^{n-1}(\rho)$ and $\mathcal{B}^{n-1}(\rho')$ must be the same (since processes only use comparison operations), which implies $r_\rho = r_{\rho'}$ (this value is denoted $r_{[\rho]}$). Thus, for each dimension i, $0 \leq i \leq n-2$, the i-faces of τ^{n-1} are split into equivalence classes: each equivalence class contains i-faces in which the inputs follow the same relative order. In this way, $\#R$ has the following form, where $size([\rho])$ denotes the size of class $[\rho]$:[1]

$$\#R = 1 + \sum_{i=0}^{n-2} \sum_{\text{for each class } [\rho] \text{ in dim } i} r_{[\rho]} \times size([\rho]). \tag{2}$$

Consider again the complex $\widehat{\mathcal{A}}^{n-1}$ and the input simplex σ^{n-1}. So, we have the number of monochromatic $(n-1)$-simplexes of $\widehat{\mathcal{A}}^{n-1}$, $\#R$, has the form in Equation (2). In what follows we prove that for each equivalence class $[\rho]$, $size([\rho])$ is divisible by p (recall that $n = p^x$), from which follows that $\#R = 1 + p \times \lambda$, for some integer λ. Therefore, $\#R \neq 0$ because there is no integer λ

[1] For the WSB task studied in [10], there is only one equivalence class for each dimension i, hence $\#R = 1 + \sum_{i=0}^{n-2} \binom{n}{i+1} r_i$, since τ^{n-1} has $\binom{n}{i+1}$ faces of dimension i.

such that $p \times \lambda = -1$, and hence $\widehat{\mathcal{A}}^{n-1}$ must have at least one monochromatic $(n-1)$-simplex, which is a contradiction.

As already mentioned, two i-faces of σ^{n-1} belong to the same equivalence class if and only if the inputs follow the same relative order. We formalize this as follows.

Consider an i-face ρ of σ^{n-1}. Each process p_i in ρ has a tuple (id_i, gid_i) as input. Given two tuples (id_i, gid_i) and (id_j, gid_j), we say $(id_i, gid_i) < (id_j, gid_j)$ if and only if $id_i < id_j$ and $gid_i < gid_j$; similarly, $(id_i, gid_i) \leq (id_j, gid_j)$ if and only if $id_i < id_j$ and $gid_i = gid_j$.

Let $v_0, \ldots, v_i$ the inputs of the processes in ρ (so each v_j is a pair (id_j, gid_j)). Due to Equation (1), $v_0, \ldots, v_i$ can be reordered $\widehat{v}_0, \ldots, \widehat{v}_i$ such that there exist $0 \leq s_1 < \ldots < s_g \leq i$ such that

$$\widehat{v}_0 \leq \cdots \leq \widehat{v}_{s_1} \quad < \quad \widehat{v}_{s_1+1} \leq \cdots \leq \widehat{v}_{s_2} \quad < \quad \cdots \quad < \quad \widehat{v}_{s_g+1} \leq \cdots \leq \widehat{v}_i.$$

This is a *group-order* of the inputs $v_0, \ldots, v_i$. Note that $g+1$ corresponds to the number of distinct groups in ρ, i.e., $g + 1 = |\{v_j.gid_j : v_j \in \rho\}|$.

Let us consider now two i-faces ρ and ρ' of σ^{n-1} and let $v_0, \ldots, v_i$ and $u_0, \ldots, u_i$ be the inputs of the processes in ρ and ρ', respectively. Faces ρ and ρ' belong to the same equivalence class if and only if there exist group-orders $\widehat{v}_0, \ldots, \widehat{v}_i$ and $\widehat{u}_0, \ldots, \widehat{u}_i$ such that for every $0 \leq k < \ell \leq i$, $\widehat{v}_k < \widehat{v}_\ell$ if and only if $\widehat{u}_k < \widehat{u}_\ell$.

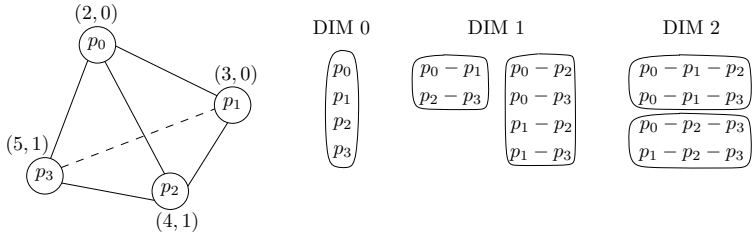


Fig. 2. Input simplex of dimension 3 (p_0 and p_1 belong to group 0, while p_2 and p_3 belong to group 1)

To conclude the proof of the theorem, we prove that $size([\rho])$ is divisible by p, for every proper face ρ of σ^{n-1}. Figure 2 depicts an example of this for $n = 2^2$ processes and 2 groups. Processes p_0 and p_1 belong to group 0 while p_2 and p_3 belong to group 1. Figure 2 shows the equivalence classes for each dimension.

Consider any proper i-face ρ of σ^{n-1}. Let $g_1, \ldots, g_s$ be the groups that appear in ρ (hence s distinct groups appear in ρ). Let $\widehat{g}_1, \ldots, \widehat{g}_s$ be a reorder of $g_1, \ldots, g_s$ such that $\widehat{g}_1 < \cdots < \widehat{g}_s$. Let $\#\widehat{g}_j$ denote the number of processes in ρ that belong to group $\widehat{g}_j$. Thus, $[\rho]$ contains every i-face of σ^{n-1} (and only those faces) in which s groups appear and, for all $j, 1 \leq j \leq s$, the j-th group contains $\#\widehat{g}_j$ processes.

We identify two cases:

1. $s = p^{x-y}$, namely all groups appear in ρ. Due to the fact that in σ^{n-1} every group has exactly p^y processes, it follows $[\rho]$ contains
$$\binom{\#\widehat{g}_1}{p^y}\binom{\#\widehat{g}_2}{p^y}\cdots\binom{\#\widehat{g}_{p^{x-y}}}{p^y}$$
i-faces. Since ρ is a proper face of σ^{n-1}, there must be at least one group $\widehat{g}_j$ such that $\#\widehat{g}_j < p^y$. It is not hard to see that p is factor of $\binom{\#\widehat{g}_j}{p^y}$, hence $size([\rho])$ is divisible by p.
2. $s < p^{x-y}$, namely, not all groups appear in ρ. In this case $[\rho]$ contains
$$\binom{s}{p^{x-y}}\binom{\#\widehat{g}_1}{p^y}\binom{\#\widehat{g}_2}{p^y}\cdots\binom{\#\widehat{g}_{p^{x-y}}}{p^y}$$
i-faces. It is easy to check that p is factor of$\binom{s}{p^{x-y}}$, thus $size([\rho])$ is divisible by p. This concludes the proof of the theorem.

Theorem 3. *Let n, ℓ be integers such that $\ell = 2^x$ and $\ell \leq \lfloor \frac{n}{2} \rfloor - 1$. Let the n processes be partitioned into $n-\ell$ groups. Then, there is no read/write, wait-free, comparison-based $(n + 2\ell - 2)$-renaming algorithm.*

Proof. The proof consists in showing that, if there exists an algorithm as the one the theorem considers, then one can derive a read/write, wait-free, comparison-based algorithm that solves the $(2 \times 2\ell - 2)$-renaming on 2ℓ processes, where the processes are initially split into ℓ groups, which contradicts Theorem 2.

Suppose there is a read/write, wait-free and comparison-based algorithm $\mathcal{A}$ that solves $(n+2\ell-2)$-renaming, where the n processes are initially split into $n-\ell$ groups. Consider an execution E in which only the $n-2\ell$ processes $p_{2\ell+1}, \ldots, p_n$ participate and decide; the processes start with the last ids in the input space $[1, \ldots, N]$, and each process starts in a distinct group taken from the last $n-2\ell$ groups in the space $[1, \ldots, n-\ell]$, namely $[\ell+1, \ldots, n-\ell]$.

At the end of E, processes $p_{2\ell+1}, \ldots, p_n$ decide $n-2\ell$ distinct names in the space $[1, \ldots, n+2\ell-2]$, hence there is a subspace Z of $[1, \ldots, n+2\ell-2]$ with $2\times 2\ell - 2$ names that no process decided. Consider an extension E' of E in which the processes $p_1, \ldots, p_{2\ell}$ are initially split into the ℓ groups in $[1, \ldots, \ell]$. In E', every deciding process decides a distinct name in Z. Therefore, using $\mathcal{A}$ and E we can derive a read/write and wait-free algorithm $\mathcal{B}$ that solves $(2 \times 2\ell - 2)$-renaming on 2ℓ processes, where the processes are arbitrarily split into ℓ groups. Moreover, $\mathcal{B}$ is comparison-based due to the fact that $\mathcal{A}$ is comparison-based. This is a contradiction to Theorem 2.

5 When the Groups Have a Known Minimal Size $s \geq 1$

The previous sections assume only that the groups are not empty (they constitute a partitioning of the processes). What does happen if each group includes at least $s \geq 1$ processes, and the value s is initially known by the processes? Do this additional information allows for a better renaming algorithm? Is $m > \frac{n}{2}$ still a threshold to have an M-renaming algorithm with $M < 2n-2$? These are the questions addressed in this section.

5.1 An Algorithm for Groups of Size at Least s

An algorithm answering the previous question is presented in Figure 3. According to the values of n, m, and s, the size M of the new name space is the following: $M = m(2s-1)$ if $n = ms$ and $M = 3n - (s+1)m - 1$ if $ms < n < (m+1)s$. Finally, when $n \geq m(s+1)$, we have $M \geq 2n-1$, leading to a suboptimal algorithm ($(2n-2)$-renaming is wait-free solvable as soon as $m \geq 2$). Hence, the algorithm is interesting for $n < m(s+1)$.

Principle of the algorithm. The algorithm is a generalization of the one presented in Figure 1. Its principle is the following one.

Initially, the new name space $[(g-1)(2s-1)+1..g(2s-1)]$ is statically assigned to the processes of each group g, $1 \leq g \leq m$. A process executes one or two stages. The first stage lasts until a process decides a new name, or its invocation of $STATE$.snapshot() returns it an array with more than s proposals from processes of its own group.

When a process proceeds to the second stage (if it ever does), it computes, from the value $snap_i$ returned by its last invocation of $STATE$.snapshot(), an extended name space, which is composed of

- The name spaces $[(g'-1)(2s-1)+1..g'(2s-1)]$ initially attributed to the groups g' that contain at least $s+1$ processes, plus
- A common space starting at $m(2s-1)+1$.

Description of the algorithm. The lines with the same number in both algorithms are the same. Line 1 which is suffixed by M has been slightly modified. The lines numbered New8.1, until New8.7 are new lines replacing line 8 of the base algorithm.

The main structure of the algorithm remains the same: a process p_i repeatedly writes in its dedicated register $STATE[i]$ a triple containing $prop_i$, the new name it proposes, plus its process and group identifiers (line 3). It then takes a snapshot of the entire array (line 4) and decides if no other process proposes the same name (lines 5-6).

If another process proposed the same new name as p_i (they conflict), p_i computes a new proposal in a way that extends the one used in the base algorithm (Figure 1). The function $gcard_i$ defined at line New8.1 computes, for a given group identifier g, the number of processes that belong to that group and whose writes appear in p_i's snapshot $snap_i$. It is the current cardinal of group g from p_i's point of view.

At line New8.1, p_i checks if its group has more than s processes. If it is not the case, it executes line New8.2, setting the variable $namespace_i$ (in which it will choose its next proposal) to the set $\{(gid_i-1)(2s-1)+1, \ldots, gid_i(2s-1)\}$, which is initially reserved for the group gid_i. In the other case (at least $s+1$ processes of p_i's group appear in its last snapshot value), p_i sets $namespace_i$ to the union of the name spaces initially reserved for groups that now have at least $s+1$ processes (line New8.3), and the (non-reserved) remaining names (if any) are added to this merged name space (lines New8.4-New8.5).

Lines 9-13 are similar to the previous algorithm. p_i first computes $props_i$, the set of the names currently proposed in the previously chosen name space. It then retrieves the complementary set $free_i$ of non-proposed names and the set $comp_i$ of identities of processes competing in the same new name space. Finally p_i determines its rank r_i among them, and chooses the r_ith value of $free_i$, the ordered sequence of available names, as its new proposal.

```
operation new_name(id_i, gid_i) is
(1 M) prop_i ← (gid_i − 1)(2s − 1) + 1;
(2)   while true do
(3)     STATE[i] ← ⟨prop_i, id_i, gid_i⟩;
(4)     snap_i ← STATE.snapshot();
(5)     if (∀ j ≠ i : snap_i[j].prop ≠ prop_i)
(6)       then return (prop_i)
(7)       else let function gcard_i(g) = |{j : snap_i[j].gid = g}|;
(New8.1)        if (gcard_i(gid_i) ≤ s)
(New8.2)          then namespace_i ← {(gid_i − 1)(2s − 1) + 1, ..., gid_i(2s − 1)}
(New8.3)          else namespace_i ← ⋃_{y∈{g:gcard_i(g)>s}} {(y − 1)(2s − 1) + 1, ..., y(2s − 1)};
(New8.4)               if (n ≠ ms)
(New8.5)                 then namespace_i ← namespace_i ∪
                                  {m(2s − 1) + 1, ..., 3n − (s + 1)m − 1};
(New8.6)               end if
(New8.7)        end if;
(9)             let props_i = {snap_i[j].prop : snap_i[j].prop ∈ namespace_i};
(10)            let free_i  = namespace_i \ props_i;
(11)            let comp_i  = {snap_i[j].id : snap_i[j].prop ∈ namespace_i};
(12)            let r_i     = rank of id_i in comp_i;
(13)            prop_i ← the r_ith integer in the increasing sequence free_i
(14)    end if
(15)  end while.
```

Fig. 3. From $m = (n - \ell)$ groups containing at least s processes to M-renaming where $M = m(2s-1)$ for $n = ms$, and $M = \big(3n - (s+1)m - 1\big)$ for $ms < n < m(s+1)$

5.2 The Size of the New Name Space

Size of the new name space Let us consider several cases according to the values of n, m, and s.

- $n = ms$. We have then $M = m(2s-1) = 2n - m$, and no process proceeds to the second phase.
- $n > ms$. Let us consider two sub-cases.
  - $n < (s+1)m$. We have then $M = 3n-(s+1)m-1$ (see below). Moreover, as $(n < (s+1)m) \Rightarrow \big(3n - (s+1)m - 1 \leq 2n - 2\big)$, there is a benefit in knowing the value s.
  - $n \geq (s+1)m$. In this case, it is possible that all the processes enter the second phase. When this occurs we have $M \geq 2n - 1$, and the algorithm has no added value.

Assuming $ms < n < m(s+1)$, let G_s be the number of groups that include exactly s processes. It follows that the size of the new name space statically reserved for these sG_s processes is $M_1 = (2s-1)G_s$. Moreover, the size of the new

name space for the processes that enter the second phase is $M_2 = 2(n - sG_s) - 1$. Hence, the total size of the new name space (composed of consecutive integers) is $M = M_1 + M_2 = (2s-1)G_s + 2(n - sG_s) - 1 = 2n - G_s - 1$. By assumption, we have $G_s \geq m - (n \bmod m) = m - (n - ms) = (s+1)m - n$. It follows that $M \leq 3n - (s+1)m - 1$.

Consequently, the knowledge of s allows to solve (a) $(n, (2s-1)m)$-renaming when the n processes are partitioned into m groups of equal size s; (b) $(n, 3n - (s+1)m - 1)$-renaming when the n processes are partitioned into m groups of minimal size s, and $ms < n < m(s+1)$.

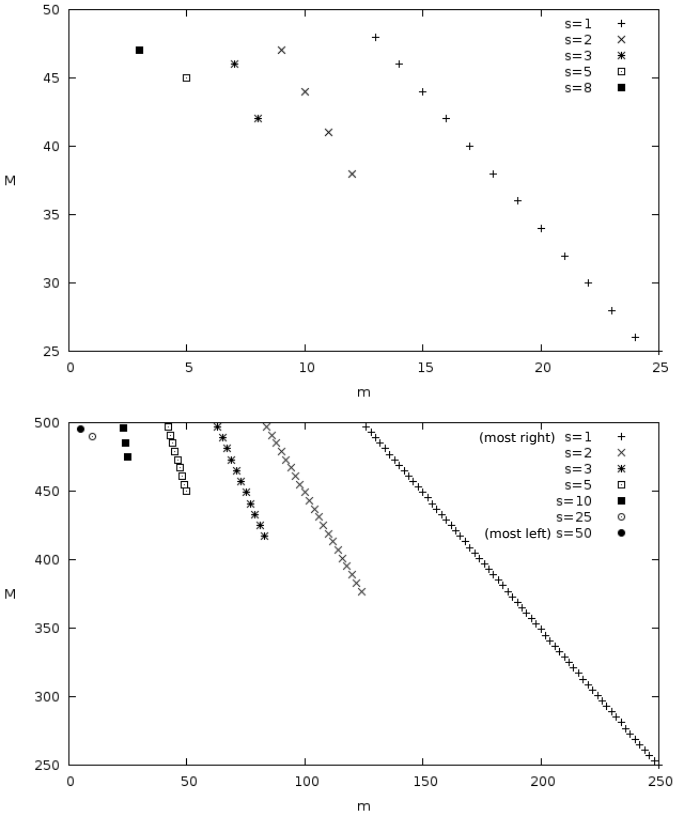


Fig. 4. Size of the renaming space according to m and s, for $n = 25$ (up) and $n = 250$ (bottom)

Numerical results. Curves associated with the value $M = 3n - (s+1)m - 1$ provided by the previous algorithm are depicted in Figure 4. The x-axis is associated with the number of groups m, while the y-axis is associated withe size of the new name space M.

The figure at the left considers $n = 25$ processes, $1 \leq m \leq 25$, and $25 \leq M \leq 49$. Each line corresponds to a value of $s \in \{1, 2, 3, 5, 8\}$. As an example, there are two pairs of values (s, m) that provide us with $M = 38$ (i.e., $M \simeq 120\%n$). They are the pairs of integers $(s = 1, m = 18)$ and $(s = 2, m = 12)$. This illustrates a tradeoff relating the number of groups m and their minimal size s, when a maximal value is a priori imposed on M. The figure also shows in which proportion increasing the minimal size of groups entails an increase in the size M.

The figure at the right complements the figure at the left. It considers 10 times more processes, i.e., $n = 250$. The new point illustrated by this figure is that more processes allow more values of s to be meaningful.

6 Conclusion

Considering the renaming problem, the aim of this paper was to investigate the impact of process groups (and their minimal size) on the size of the renaming space. Two main results have been presented. The first is an $(n+2\ell-1)$-renaming algorithm, where the number of groups is $m = n - \ell$, $1 \leq m \leq n - 1$. The important observation is that, when $n-1 \geq m > \frac{n}{2}$ (i.e., $1 \leq \ell < \frac{n}{2}$), $n+2\ell-1 < 2n - 2$. Hence, groups allow to circumvent the lower bound on the new name space in read/write systems. The second result is an impossibility proof showing that there are pairs (n, m) for which groups do not allow to bypass the $2n - 1$ lower bound. An additional result is the extension of the algorithm to groups of possibly different sizes, each containing at least s processes.

The paper leaves consequently open the following question: When considering the additional computational power given to processes by an m-group partitioning, does the predicate $m > \frac{n}{2}$ defines a tight lower bound on the number of groups to bypass the $2n - 2$ lower bound on the size of the new name space for all renaming algorithms (including those which are not comparison-based)?

Acknowledgments. This work has been partially supported by the French ANR project DISPLEXITY devoted to the computability and complexity in distributed computing.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. Journal of the ACM 40(4), 873–890 (1993)
2. Afek, Y., Gafni, E., Lieber, O.: Tight Group Renaming on Groups of Size g Is Equivalent to g-Consensus. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 111–126. Springer, Heidelberg (2009)
3. Afek, Y., Gamzu, I., Levy, I., Merritt, M., Taubenfeld, G.: Group Renaming. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 58–72. Springer, Heidelberg (2008)
4. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an Asynchronous Environment. Journal of the ACM 37(3), 524–548 (1990)

5. Attiya, H., Rajsbaum, S.: The Combinatorial Structure of Wait-Free Solvable Tasks. SIAM Journal on Computing 31(4), 1286–1313 (2002)
6. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn., 414 pages. Wiley-Interscience (2004)
7. Attiya, H., Paz, A.: Counting-Based Impossibility Proofs for Renaming and Set Agreement. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 356–370. Springer, Heidelberg (2012)
8. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on Theory of Computing, STOC 1993, pp. 91–100. ACM Press (1993)
9. Castañeda, A., Imbs, D., Rajsbaum, S., Raynal, M.: Renaming Is Weaker Than Set Agreement But for Perfect Renaming: A Map of Sub-consensus Tasks. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 145–156. Springer, Heidelberg (2012)
10. Castañeda, A., Rajsbaum, S.: New Combinatorial Topology Upper and Lower Bounds for Renaming: The Lower Bound. Distributed Computing 22(5-6), 287–301 (2010)
11. Castañeda, A., Rajsbaum, S.: New Combinatorial Topology Upper and Lower Bounds for Renaming: The Upper Bound. Journal of the ACM 59(1), 3 (2012)
12. Castañeda, A., Rajsbaum, S., Raynal, M.: The renaming problem in shared memory systems: an introduction. Elsevier Computer Science Review 5, 229–251 (2011)
13. Gafni, E.: Group-Solvability. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 30–40. Springer, Heidelberg (2004)
14. Gafni, E.: Renaming with k-Set-Consensus: An Optimal Algorithm into $n + k - 1$ Slots. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 36–44. Springer, Heidelberg (2006)
15. Gafni, E., Mostéfaoui, A., Raynal, M., Travers, C.: From Adaptive Renaming to Set Agreement. Theoretical Computer Science 410, 1328–1335 (2009)
16. Gafni, E., Rajsbaum, S., Herlihy, M.: Subconsensus Tasks: Renaming Is Weaker Than Set Agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 329–338. Springer, Heidelberg (2006)
17. Gafni, E., Raynal, M., Travers, C.: Test&set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. In: 26th IEEE Symposium on Reliable Distributed Systems, SRDS 2007, pp. 93–102. IEEE Computer Society Press (2007)
18. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
19. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
20. Imbs, D., Rajsbaum, S., Raynal, M.: The Universe of Symmetry Breaking Tasks. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 66–77. Springer, Heidelberg (2011)
21. Imbs, D., Raynal, M.: On Adaptive Renaming under Eventually Limited Contention. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 377–387. Springer, Heidelberg (2010)
22. Saks, M., Zaharoglou, F.: Wait-Free k-Set Agreement Is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)

Electing a Leader in Multi-hop Radio Networks*

Bogdan S. Chlebus[1], Dariusz R. Kowalski[2], and Andrzej Pelc[3]

[1] Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO 80217, USA
[2] Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
[3] Département d'informatique, Université du Québec en Outaouais, Gatineau, Québec J8X 3X7, Canada

Abstract. We consider the task of electing a leader in a distributed manner in ad hoc multi-hop radio networks. Radio networks represent the class of wireless networks in which one frequency is used for transmissions, network's topology can be represented by a simple undirected graph with some n nodes, and there is no collision detection. We give a randomized algorithm electing a leader in $\mathcal{O}(n)$ expected time and prove that this time bound is optimal. We give a deterministic algorithm electing a leader in $\mathcal{O}(n \log^{3/2} n \sqrt{\log \log n})$ time. By way of application, we show how to perform gossiping with combined messages in $\mathcal{O}(n \log^{3/2} n \sqrt{\log \log n})$ time by a deterministic algorithm, and in $\mathcal{O}(n)$ expected time by a randomized algorithm.

Keywords: radio network, leader election, distributed algorithm, gossiping, randomization, lower bound.

1 Introduction

Wireless networking involves diverse applications, standards and categorizations of networks, which is reflected in the variety of algorithmic models of such networks. We consider an abstract model of synchronous wireless networks in which one frequency is used for transmissions and any simple undirected connected graph can represent a network topology. It is called the "graph model" of wireless networks or simply the "radio networks model." This paper addresses the task of electing a leader in a distributed manner in such radio networks that do not have collision detection available and have arbitrary connected topologies.

Electing a leader is among the fundamental tasks in distributed computing [2]. The goal is to obtain the status of leader for precisely one node while every other node is not a leader but gets to know the leader. We seek algorithms electing a leader in which the above specification of the task holds in every execution. We

* The work of the first author was supported by the NSF Grant 1016847. The work of the second author was Supported by the Engineering and Physical Sciences Research Council [grant number EP/G023018/1]. The work of the third author was Supported by a NSERC discovery grant and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 106–120, 2012.

also consider the communication problem of gossiping in which each node is initialized with an individual input rumor and the task for every node is to learn all the rumors.

Each of our distributed algorithms can be executed in an arbitrary radio network with the topology represented as a simple undirected connected graph. We assume that a bound on the range of names of nodes and a bound on the size of the network can be part of code but the nodes are not assumed to know the network's topology or even their neighbors; such protocols are usually referred to as being for "ad hoc" networks. We assume that networks are synchronous. This includes the assumption that the nodes start an execution of an algorithm simultaneously. Next, synchrony means that any execution is structured into rounds, so that a transmission by a node in a round reaches every neighbor of the node in this round. There is no collision detection available, which means that a node obtains the same feedback by its hardware from the network in any round when no neighbors transmit as in any round when at least two of the neighbors do.

Our results. We give a randomized algorithm to elect a leader in $\mathcal{O}(n)$ expected time, and also prove that this time bound is optimal. The fastest previously known randomized leader election algorithm for radio networks without collision detection worked in $\mathcal{O}(n \log n)$ expected time; it was given by Czumaj and Rytter [9]. The $\mathcal{O}(n)$ bound on the expected time of randomized leader election demonstrates that randomization allows to improve time performance of leader election in radio networks over deterministic algorithms; this is because there is the $\Omega(n \log n)$ lower bound on deterministic leader election in single-hop radio networks without collision detection, given by Kowalski and Pelc [17].

We develop a deterministic algorithm for radio networks without collision detection which elects a leader in $\mathcal{O}(n \log^{3/2} n \sqrt{\log \log n})$ time. This improves upon the fastest previously known $\mathcal{O}(n \log^2 n \log \log n)$ time performance of the algorithm given by Vaya [24].

As an application, we show how to perform gossiping in ad hoc radio networks with combined messages, that is, when a message can carry any number of rumors. We give a deterministic gossiping algorithm of $\mathcal{O}(n \log^{3/2} n \sqrt{\log \log n})$ time performance. This improves upon the fastest previously known algorithm working in $\mathcal{O}(n \log^2 n \log \log n)$ time given by Vaya [24]. We give a randomized gossiping algorithm of $\mathcal{O}(n)$ expected time. This demonstrates that randomization may speed up gossiping, with respect to deterministic protocols in ad hoc radio networks without collision detection, as there is the $\Omega(n \log n)$ lower bound on gossiping by deterministic algorithms in single-hop radio networks, given by Kowalski and Pelc [17].

Due to space limit, the proofs are deferred to the full version of the paper.

Related work. A distributed algorithm for electing a leader in general wired networks, with the optimum message complexity, was given by Gallager et al. [11], who built on the connection between message-optimal leader election and finding a spanning tree in a network. Time optimality of finding a leader, and a spanning tree, in a distributed manner in general wired networks was subsequently

investigated by Awerbuch [3] and Garay et al. [12]. Electing a leader has been also studied in special classes of wired networks. The case of complete networks was considered by Afek and Gafni [1]. For the related work on electing a leader in rings, see an exposition of this topic in the book by Attiya and Welch [2] and the references therein. Leader election in general dynamic networks was considered by Kuhn et al. [19].

The problem of electing a leader in multi-hop radio networks has been considered in the literature in four different settings, determined by whether collision detection is available or not and whether algorithms are to be deterministic or randomized. If collision detection is available, Kowalski and Pelc [17] showed that a leader can be elected deterministically in $\mathcal{O}(n)$ worst-case time, which is optimal when time performance is expressed as a function of n. Next we consider the model without collision detection. There is the $\Omega(n \log n)$ lower bound for deterministic algorithms for this model [17], even for single-hop networks, showed by applying the combinatorial bounds given by Clementi et al. [8]. A deterministic algorithm for electing a leader can be structured to resort to an algorithm for broadcasting. Vaya [24] proposed to use the algorithm for broadcasting in radio networks developed by De Marco [10], which operates in $\mathcal{O}(n \log n \log \log n)$ time, obtaining an algorithm to elect a leader in $\mathcal{O}(n \log^2 n \log \log n)$ time.

A randomized leader-election algorithm of $\mathcal{O}(n \log n)$ expected time can be obtained by combining a fast randomized broadcast with the paradigm of binary search in a polynomial range of names. To this end, one can use randomized algorithms of $\mathcal{O}(D \log(n/D) + \log^2 n)$ expected time in networks of diameter D, given by Czumaj and Rytter [9] and by Kowalski and Pelc [16].

Single-hop radio networks, also known as multiple access channels, have been investigated in two variants, with collision detection and without it. For the model without collision detection, the analysis given by Clementi et al. [8] implies that deterministic leader election can be accomplished in $\mathcal{O}(n \log n)$ worst-case time, which is optimal by the combinatorial bounds given there, see [17]. Regarding the expected-time complexity of randomized algorithms without collision detection, Bar-Yehuda et al. [4] showed the $\mathcal{O}(\log n)$ upper bound and Kushilevitz and Mansour [20] showed the $\Omega(\log n)$ lower bound. Leader election for channels with collision detection was investigated by Willard [25], who gave the $\mathcal{O}(\log \log n)$ expected-time randomized protocol and proved its optimality in a restricted class of protocols; other protocols were given by Nakano and Olariu [22]. Jurdziński et al. [15] studied energy consumption needed to elect a leader in multiple access channels without collision detection.

Next we briefly survey the literature on distributed gossiping in radio networks. Vaya [24] gave a deterministic gossiping protocol with combined messages, working in $\mathcal{O}(n \log^2 \log \log n)$ time, which improved upon the fastest previously known algorithm by Gąsieniec et al. [14] that worked in $\mathcal{O}(n \log^4 n)$ time. Czumaj and Rytter [9] gave a randomized Las Vegas gossiping protocol of $\mathcal{O}(n \log^2 n)$ expected-time performance for the model of general directed networks with combined messages. Chlebus et al. [6] studied generalizations of multi-broadcast and rumor gathering, when the number of rumors is arbitrary but messages are

bounded. Chlebus et al. [7] gave a randomized algorithm and lower bounds for the problem when a set of nodes needs to perform an instance of many-to-many communication, with an upper bound on the distance of a pair of participants as a parameter in performance bounds, for the model of combined messages. A comprehensive survey of gossiping in radio networks was given by Gąsieniec [13].

2 Preliminaries

There are n nodes in a network. Each node has a unique integer *name* assigned to it. The nodes' names are in the interval $[0, N-1]$, for some positive integer N. We call N the *range of names*. It is assumed that $N = \mathcal{O}(n^\gamma)$, for a constant $\gamma > 1$, which means that N is polynomial in n. We denote by η an upper bound on the number of nodes n, that is, $\eta \geq n$. It is assumed that $\eta = \Theta(n)$, which means that η is an upper bound on the size of the network that is linear in n.

Some information is said to be *known* by a node when it can occur in a code executed by the node. We assume that each node knows its own name. We seek distributed algorithms for networks that are *unknown* or *ad hoc*, which means that algorithms are expected to work for any connected graph topology, with a range of names N and an estimate on the network's size η being the only global parameters of the network known to the nodes.

Communication proceeds in synchronous rounds. All the nodes begin executing an algorithm simultaneously. In each round, a node either listens ready to receive transmissions from the neighbors or transmits itself.

For the task to elect a leader, we assume that a message can carry $\mathcal{O}(1)$ nodes' names, which means carrying $\mathcal{O}(\log n)$ bits. Regarding gossiping, there are two models popular in the literature. One stipulates that a message can carry at most one input rumor along with $\mathcal{O}(\log n)$ auxiliary control bits; the model is called of *separate* or *bounded* messages. The model of *combined* or *unbounded* messages allows a message to carry up to n input rumors along with $\mathcal{O}(\log n)$ auxiliary control bits. We assume the model of combined messages for gossiping.

For randomized algorithms, we use the phrase *with high probability (whp)* to mean with probability at least $1 - n^{-c}$, for some constant $c > 0$.

Radio networks. We consider radio networks as a class of wireless networks in which topologies are represented by undirected graphs and one radio frequency is used for transmissions [5,18,23]. The graph representing a radio network's topology is called the *reachability graph* of the network; an edge represents the ability of direct transmissions between its endpoints. A message transmitted by a node reaches all its neighbors in the round of the transmission. A message is said to be *heard* by a node when it is received successfully. A node hears a message in a round precisely when the node acts as a receiver and exactly one of its neighbors transmits in this round. If at least two neighbors of a node u transmit simultaneously in a given round, then none of the messages is heard by u and we say that a *collision* occurred at u. A node perceives a round as "silence" when no neighbor transmits in this round. We work with the model of radio networks *without collision detection*, in which a node cannot distinguish a round of collision from one of silence solely by the feedback from the channel.

We use simple connected graphs to represent topologies of radio networks. Given such a graph, the notation $\Gamma(v)$ denotes the set of neighbors of a node v. We let $\Gamma(A)$ stand for the union of all the sets $\Gamma(v)$, for $v \in A$, where A is any set of nodes of the network. The degree of a node v is denoted by $d(v)$. For any two nodes v and u, we define the *radio distance between v and u* as the minimum sum of the degrees of the nodes on a shortest path between v and u.

Selective families. A sequence $\mathcal{F}(m,k) = (F_0, \ldots, F_{r-1})$, of sets of integers $F_i \subseteq [0, m-1]$, for $0 \leq i < r$, is an (m,k)-*selective family* if, for each set of integers $S \subseteq [0, m-1]$ of at most k elements, there exists $F_j \in \mathcal{F}(m,k)$ such that $|F_j \cap S| = 1$. Such a selective family is said to be *of length* r. A selective family determines a transmission schedule by using the sets in the selective family in a round robin manner. We say that an (N,k)-selective family $\mathcal{F} = (F_0, \ldots, F_{r-1})$ is *executed* in a sequence of consecutive rounds, if the nodes transmitting in round i are precisely those whose names are in F_k, where k and i are congruent modulo r.

We will use the fact, proved by Clementi et al. [8], that, for any positive integers $m \geq k$, there exists an (m,k)-selective family of $\mathcal{O}(k \log \frac{m}{k})$ length. Let α be a constant such that there exists an (N,k)-selective family of length at most $\alpha \cdot k \log N$, for any $1 \leq k \leq N$. Here and elsewhere, the notation $\log x$ denotes the binary logarithm $\log_2 x$.

Designated broadcast algorithm. Whenever we need to perform broadcasting from a single source node, we use the algorithm developed by De Marco [10], to which we refer as the *designated radio broadcast* in such a context. This algorithm completes broadcasting in $T(n,N) = \mathcal{O}(n \log N \log\log N)$ time, after initiation by a source node, in any network with a connected reachability graph of at most n nodes and with N as their range of names. We briefly describe how the algorithm is structured. The source transmits only in the first round of an execution. When a node receives the broadcast message, along with control bits representing the number of rounds that have passed since the start of the broadcast execution, then it becomes *activated*. An activated node immediately begins performing transmissions according to a schedule that is determined only by the following: this node's name, the range of names N, the linear upper bound η on the size of the network, which is used instead of n, and the round of activation of the node as counting from the start of the execution.

Depth-first traversal. We use a protocol to traverse a radio network by a token in a depth-first manner, which was developed by Kowalski and Pelc [16]. That protocol, when initiated by a node, performs a depth-first traversal of the network by a token in $\mathcal{O}(n \log n)$ rounds, producing a depth-first spanning tree rooted at the initiating node. When the token is to be forwarded from a node v to one of its unvisited neighbors, then v and its neighbors need to identify one of such unvisited nodes. There are two nodes that participate in this operation: one is v and another is a neighbor of v which is referred to as the *witness for* v. The witness for the initiating node needs to be selected first from among its neighbors, while for any other node v, it is the node from which the node v got the token that conveniently serves as the witness for v. To select an unvisited

neighbor of a node v, when this is the initiating node, v invokes a binary search among the names in $[0, N-1]$. To implement such a binary search, v uses a procedure $\mathsf{Echo}_v(w, A)$, where A is a set of neighbors of v and the witness w does not belong to A. Executing $\mathsf{Echo}_v(w, A)$ takes two rounds and allows the node v to distinguish between the following three cases: $A = \emptyset$, $|A| = 1$, and $|A| > 1$. In a search for a neighbor, the set A initially consists of all the unvisited neighbors and its size is halved by each subsequent call of Echo.

Revealing neighbors. For two neighbors v and w in a radio network, we say that w is *revealed to* v when v knows w's name. Node w gets revealed to v when v hears the first message from w. Such a message is understood to bring the sender's name. We use a randomized procedure to reveal all the neighbors of a node v. This procedure can be designed for a multiple access channel, as v with its neighbors can simulate a channel for the v's neighbors to transmit on it, with a constant slowdown per transmission. The node v transmits a message to notify its neighbors that revealing has started. Only the neighbors that v does not know of participate in this revealing; let this set be denoted as A. We assume that v knows at least one neighbor w which serves as witness. The node v invokes $\mathsf{Echo}_v(w, A)$. If $A = \emptyset$ then v transmits a signal to terminate revealing. If A is a singleton then v has already heard from v in a round of $\mathsf{Echo}_v(w, A)$ so v also terminates revealing. If $|A| > 1$ then v continues with a simulation of the algorithm for the multiple access channel developed by Martel [21], which makes every member of A eventually heard by v; we refer to this algorithm as *the Martel's protocol*. Let k be the number of such unrevealed neighbors. The Martel's protocol takes at most $c \cdot (k + \log n)$ rounds with high probability [21], for some sufficiently large constant c.

3 Auxiliary Procedures

In this section we present four auxiliary procedures, three of them are deterministic and one is randomized, that will serve as building blocks for our leader election algorithms for radio networks.

3.1 Partial Multi Broadcast

Suppose that some generic information μ is known to every node from a set S. We want the information μ to be disseminated across the network, with all the nodes in S starting the process of dissemination simultaneously. We assume that initially each node v of the network knows whether v itself belongs to S or not, without necessarily knowing other elements of S. There is a parameter z, for $0 < z \leq 1$, known to all the nodes in S. The goal is for at least a fraction z of all the nodes of the network to get to know μ by the end of the process. In applications, this number z will be a function $z = z(\eta)$ of η, converging to 0 with n growing to infinity.

The following procedure $\mathsf{Partial\_Multi\_Broadcast}(S, z)$ achieves this goal; the numbers η and N known by the nodes are not listed among the parameters of this

procedure for brevity. In an execution, every node is either active or passive, this status being subject to changes. All the nodes in S start as active. An execution is structured to consist of two parts. In the *first part*, we use the designated radio broadcast algorithm as follows. In the beginning, the nodes in S set the source message to μ and the global execution time counter to 2. The intuition is that this process will mimic a scenario when there is a conceptual source connected only to the nodes in S that has just sent message μ to the nodes in S in the conceptual (preceding) round 1. Starting from this point, all the nodes execute the designated radio broadcast, as specified in Section 2. The first part ends in round $T(z\eta, N)$, where $T(k, N)$ is the upper bound on time performance of the designated broadcast in networks of *at most* k nodes, for $k \leq \eta$. This is followed by the *second part*, in which each node in S executes three times, one by one, an (N, η)-selective family of length $\alpha\eta \log N = \Theta(n \log n)$. For an execution of this procedure, the set of nodes that receive a message from some node in S is denoted W_S.

Lemma 1. *Consider* Partial_Multi_Broadcast(S, z) *executed with* z *such that* $0 < z \leq 1$ *and* $z = o(1)$. *Then the execution has these properties:*
(i) $|S \cup W_S| \geq zn$,
(ii) all the nodes of distance at most 3 *from the set* S *end up in* W_S,
(iii) termination occurs in $\mathcal{O}(zn \log n \log\log n)$ *rounds.*

3.2 Ultra-Selectors and Combined-Ultra-Selectors

Following [6], we define an (N, a, ε)-*ultra-selector* to be a sequence of sets such that for any set $A \subseteq [0, N-1]$ of a size that is at most a but greater than $a/2$ there is at least an ε fraction of the sets in the sequence that share precisely one element with A, for a given $1 \leq a \leq N$ and $0 < \varepsilon < 1$. The number of sets in the sequence is the *length* of the ultra-selector. A sequence obtained by concatenating $(N, a/2^i, \varepsilon)$-ultra-selectors, for $i = 0, 1, \ldots, \log a$, is an (N, a, ε)-*combined-ultra-selector*.

Lemma 2. *There exists an* (N, a, ε)*-combined-ultra-selector of length at most* $4\beta a \log(2N/a)$, *for some* $\beta \geq 2$ *depending on* ε.

Lemma 3. *For any numbers* a *and* ε *such that* $1 \leq a < N$ *and* $0 < \varepsilon \leq 1/32$, *if* $\mathcal{S}$ *is an* (N, a, ε)*-combined-ultra-selector then, for any set* $A \subseteq [0, N-1]$ *with at most* a *elements, at least* $\varepsilon \log(2N/a)$ *sets in* $\mathcal{S}$ *share a single element with* A.

3.3 Multiple DFS Traversals

We introduce a method to traverse a radio network, which is an extension of the DFS traversal. The idea is to have many nodes simultaneously launch tokens to perform independent DFS traversals. A token carries with it a list of the visited nodes. Each node maintains its record of the names already visited by

passing tokens, and the list of its known neighbors. We limit the time available for such traversals in advance, so that some executions may fail to have all the nodes visited, even if there is only one token in the network at a time. Our ultimate goal is to guarantee that after a logarithmic number of executions of this procedure with a single token, some of them will accomplish a DFS traversal, and, moreover, all nodes can recognize such an execution.

Procedure Multi_DFS_Traversal(N, b, R) operates as follows. Let R be a subset of nodes, each with a unique neighbor designated as witness. Each node knows whether it belongs to R or not, without knowing the other elements of R. Each node v in R knows its witness and the witness of v knows v. Each node in R initiates a DFS traversal by launching a token. A token's traversal terminates at the latest when it returns back to its originator and there are no more of its neighbors to visit, but it may terminate earlier when the token gets destroyed by a visited node. An execution of Multi_DFS_Traversal(N, b, R) takes precisely b rounds. The details are as follows.

There are two parallel threads. A node participates in executing one thread at a time. Additionally, a token-dropping mechanism is applied on top of the two threads.

The first thread. This thread takes care of sending messages with tokens. Each token carries the following information: the name of the initiator, the number of token's hops from node to node, the number of visited nodes without multiplicities, and the maximum name of a visited node. Every node visited by a token stores a record of the information carried by the token.

A node may know only some of its neighbors in a round of an execution, but it keeps discovering them by hearing the tokens they pass from one to another, and also learning which neighbors have already been visited. This mechanism allows the token to carry only a logarithmic number of control bits, instead of the whole list of the visited nodes. A node's neighbor is *unvisited* when the node has not heard yet that this neighbor has received the token. If a node holding a token knows some of its still unvisited neighbors, then it forwards the token to the unvisited neighbor with the smallest name, with an acknowledgment arriving back in the following round. Upon getting an acknowledgment, the sender erases the token from its local memory. The recipient becomes the token's holder after acknowledging receipt. If a node holding a token does not know of any unvisited neighbor, then the node passes control to the second thread and expects that that thread will compute the next destination of the token.

The second thread. This thread takes care of checking whether there is a neighbor not visited yet by this token. Such a neighbor cannot be known by the node holding the token. Checking for being visited is accomplished by executing the procedure of searching for a neighbor (see Section 2, Depth-first traversal). If the search returns that there is no such a neighbor then the token is forwarded to the parent. If the search for a neighbor finds an unvisited node, then the thread returns this neighbor. After this control goes back to the first thread which continues forwarding the token.

Rules to drop tokens. The above two threads guarantee that the token visits all nodes and returns to its source in case when there is just single token, that is, when $|R| = 1$, and an execution is sufficiently long, say, $b = \Theta(\eta \log N) = \Theta(n \log n)$; see [16]. When there is no leader, it is not immediately clear how to choose a singleton set R in a distributed way. Our approach is to allow for $|R| > 1$ and consider a logarithmic number of execution of the procedure, when sets R are selected according to some combined-ultra-selector. A motivation for this is to increase a chance that in some of these executions the sets R are singletons, so a DFS traversal will be completed. This is combined with having executions of the procedure take only $b = \Theta(\eta) = \Theta(n)$ rounds.

In order to deal with the case $|R| > 1$, an additional mechanism to drop tokens is used along the two threads described above. Its purpose is to provide safety when multiple tokens exist simultaneously. As we will show, the property of eventual DFS traversal can be achieved by considering only runs when the sets R are singletons. Every time when either another token is discovered in a newly visited node or there is some inconsistency with the feedback received from overheard messages, as compared to what is expected if there were just a single token, then the token is abandoned. This means that the node holding the token changes its status to one not holding this token while this fact is not communicated to the neighbors nor the token forwarded to them.

Next we summarize the four ways in which a node can dispose of a token:

1) The node passes the token to a neighbor through the first thread.

2) The node has just received the token through the first thread and it has a record of some other token visiting before.

3) A state inconsistent with just one token being around occurs in the searching procedure of the second thread, that is, either no witness node is heard or a node recorded as visited or outside of the current range of search is heard.

4) The node hears two silences in the first step of a sub-routine used to search for an unvisited neighbor in the second thread, when executing the echo procedure, which indicates that there are at least two such neighbors, while no neighbor is returned in this execution of the subroutine.

A node v in R completes Multi_DFS_Traversal(N, b, R) successfully if its token returns to v by round b since its launching and, upon such a return, the neighbors of v have been already visited. In applications, an execution of Multi_DFS_Traversal(N, b, R) may not be completed successfully by any node in R. We perform Multi_DFS_Traversal(N, b, R) multiple times leveraging the property that each execution increases the nodes' knowledge about their neighbors.

Lemma 4. *Let $R(i)$ be singleton sets, for $i \leq \frac{1}{64} \log N$. There is a constant $g > 8$ such that in the executions of procedure Multi_DFS_Traversal$(N, g\eta, R(i))$, carried out consecutively for $i \leq \frac{1}{64} \log N$, the unique element in $R(\frac{1}{64} \log N)$ completes its execution successfully after having visited all the nodes.*

We remark that Lemma 4 holds even if the executions of procedure Multi_DFS_Traversal$(N, g\eta, R(i))$, for $i \leq \frac{1}{64} \log N$, are separated by some other runs with sets R such that $|R| > 1$. Intuitively, this is because the knowledge about

neighbors does not decrease in the executions with $|R| > 1$, and it suffices to have an inductive-type of argument work.

3.4 Randomized Multiple DFS Traversals

We will use a randomized modification of procedure Multi_DFS_Traversal. It will be also executed multiple times, to have the nodes accumulate knowledge of their neighbors. This procedure is called Rand_Multi_Tree_Search(η, R).

Procedure Rand_Multi_Tree_Search(η, R) is executed as follows:

Each node is *equipped* with a subset of its neighbors, by which we mean that it knows the names of neighbors in this subset. All the nodes in R have the status visited and revealed. In the beginning, a witness for each node in R needs to be identified. This is accomplished by each node in R notifying its neighbors, which triggers executing a fixed (N, η)-selective family of length $c\eta \log N = \mathcal{O}(n \log n)$, for some constant $c > 0$; see [8].

We want the nodes in R launch tokens that will then explore the network, but not necessarily in a DFS manner. Instead of applying an $\mathcal{O}(\log N)$ procedure of selecting the next node, or checking that there is no revealed neighbor, as we did in the deterministic setting, we proceed as follows.

A node in R treats itself as a root, in that it has no parent, while every other node sets the node from which it received the token for the first time as its parent. When a node v is visited for the first time, or if it is a node in R in the beginning of the execution, it reveals all its known neighbors to the other neighbors, if any, by transmitting the known neighbors' names one by one.

Each node keeps a record of all its revealed neighbors and of the visited ones. A node v marks its neighbor w as visited if v receives a message with the token sent by w; in such a situation, the node v does not have to be the intended recipient of the message and the token.

Each time we need to verify if there is an unvisited neighbor of a node then procedure Echo is used.

Rand_Multi_Tree_Search(η, R) is executed repeatedly so that a node may receive a token multiple times. All these executions keep up performing the Martel's protocol. If a node v receives the token and all of its neighbors have already been revealed, for instance, when the node receives the token either again after releasing it before or during the first token's visit but after completing the execution of the Martel's protocol, then v sends the token to the unvisited neighbor of the smallest name. If there is no unvisited neighbor then v passes the token to its parent, in case it has a parent, otherwise it is considered as having completed the first part, in case the node v is in R. The root waits a suitable $\Theta(\eta) = \Theta(n)$ number of rounds, so that all the other tokens disappear, and then sends a token with its name to follow the route of its token from the first part. This second part also takes $\Theta(\eta) = \Theta(n)$ rounds.

When $|R| > 1$ then a node handling a token eliminates it each time a situation impossible for a single token occurs at a node handling the token, similarly as in the deterministic algorithm Multi_DFS_Traversal, so that at most one token survives. The resulting tree may not be a DFS tree but it still spans the graph.

4 Randomized Election

In this section we show how a leader can be elected by a randomized distributed algorithm in the optimal expected $\Theta(n)$ time. An execution of the algorithm is divided into epochs. A node iterates such epochs until it elects a leader.

Algorithm Randomized_Leader_Election

Repeat the following epochs, from (a) through (d), until a leader is elected:

Sub-epoch (a): Each node chooses to be a candidate in the current epoch with probability $1/\eta$ independently from the other nodes.

Sub-epoch (b): Each node selects a random number r from $[0, 4\eta - 1]$ and transmits its name in the r-th round of the sub-epoch. In the remaining rounds, the node just listens. At the end, each node has a set, possibly empty, of these names of its neighbors that were heard in this sub-epoch.

Sub-epoch (c): Procedure Rand_Multi_Tree_Search(η, R) is executed for the set R of candidates. Each node is equipped with a subset of its neighbors computed in Sub-epoch (b).

Sub-epoch (d): If a candidate receives its token back, the it launches the token again to perform the same traversal as in the previous sub-epoch (c). The token carries the candidate's name. At the end of this sub-epoch, each node that received the token, including the node that originated the token, elect the node whose name is carried by the token as the leader.

The sub-epochs have the following precisely determined duration: (a) takes one round; (b) takes 4η rounds; (c) takes takes $2\eta + 2c\eta = \mathcal{O}(n)$ rounds, where $ck + \log \eta$ is the complexity of the Martel's protocol on the multiple access channel with k active nodes to be revealed; (d) takes 2η rounds.

Lemma 5. *During Sub-epoch (b), the following holds whp for any node v:*
(i) if v has at most $\log n$ neighbors in the graph, then all these neighbors are heard by v;
(ii) if v has more than $\log n$ neighbors then at least $\log n + 1$ of them are heard by v.

Lemma 6. *When procedure Rand_Multi_Tree_Search(η, R) is executed in Sub-epoch (c) then the following gets accomplished whp:*
(i) at most one node from R gets elected and a DFS spanning tree rooted at this node is built;
(ii) exactly one node in R is selected if R is a singleton;
(iii) termination occurs in $\mathcal{O}(n)$ rounds.

Theorem 1. *Algorithm Randomized_Leader_Election elects a leader in $\mathcal{O}(n)$ expected time. The algorithm builds a tree rooted at the leader as a byproduct.*

We show that the expected time of algorithm Randomized_Leader_Election is optimal.

Theorem 2. *There is no randomized algorithm electing a leader in $o(n)$ expected number of rounds in all n-node networks.*

5 Deterministic Election

We develop a deterministic algorithm electing a leader in $\mathcal{O}(n \log^{3/2} n \sqrt{\log \log n})$ time.

We begin with an auxiliary algorithm that works in two stages. First, a set of representatives is selected locally, each being the node with the largest name in a suitable subgraph. The next stage is to elect one leader among the representatives.

An execution of the algorithm is considered as having terminated when all the nodes had become aware that the task of electing a leader is completed. This strong termination property is what defines the time performance.

Our algorithm is executed with a parameter $0 < z \leq 1$ to be specified later in order to optimize the time complexity.

The following is the underlying idea on which the algorithm is based:

First, procedure Partial_Multi_Broadcast is used to select a set R of at most $\frac{1}{z}$ nodes called *representatives*. Each representative selects one neighbor, to serve as a witness, in order to start algorithm Multi_DFS_Traversal$(N, g\eta, \cdot)$ from this representative, where $g > 8$ is the constant from Lemma 4. Next, we combine algorithm Multi_DFS_Traversal$(N, g\eta, \cdot)$ with a specific transmission schedule, called ultra-selector, in order to perform consecutive executions of procedure Multi_DFS_Traversal$(N, g\eta, \cdot)$ starting from different subsets of the set of representatives. More precisely, the ith time segment is reserved to execute algorithm Multi_DFS_Traversal$(N, g\eta, F_i)$ starting from the representatives in F_i, for consecutive sets F_i in the ultra-selector. Due to the suitable properties of ultra-selectors, at least 1 in 32 of these executions will result in the subsets of representatives being singletons. This will enable algorithm Multi_DFS_Traversal$(N, g\eta, F_i)$ to be completed correctly in all these executions, and moreover, one of them completes a DFS tour of the network and elects a leader.

The novelty of our approach is in selecting each representative locally as a node with the largest name in some subgraph, without performing full communication in the network. After this first stage has been accomplished, we still need to elect one leader among the representatives in the second stage.

Algorithm Deterministic_Leader_Election(z)

Stage 1: Electing representatives.
This stage proceeds through $\log N + 1$ epochs of equal length. Each node starts as active. Epoch i begins by executing Partial_Multi_Broadcast(S_i, z), where S_i is the set of those nodes active in the beginning of epoch i that have bit 1 in the ith position of the binary representation of their name. Epoch i continues by executing Partial_Multi_Broadcast(S'_i, z), where S'_i is defined as the set of nodes that received a message in the first part of epoch i, that is, during the execution of Partial_Multi_Broadcast(S_i, z). In both parts, the same default message is used by all nodes. At the end of epoch i, if the bit 0 is in the ith position of the binary representation of its name, then the node that received a message during epoch i becomes inactive. Active nodes become *representatives* at the end of this stage.

Stage 2: Electing a leader by representatives.
We proceed in two sub-stages:
Sub-stage 2.a: Each representative chooses a witness among its neighbors.

> An (N, η)-selective family of $\alpha\eta \log N = \mathcal{O}(n \log n)$ length is executed, in which every node participates and transmits just its name. After the family's execution is over, each representative transmits its name and the name of its witness.

Sub-stage 2.b: The representatives combine executing an $(N, \frac{1}{z}, 1/32)$-combined-ultra-selector with procedure Multi_DFS_Traversal so as to select as leader the node with the largest name.

> This sub-stage proceeds through a epochs, where a is the length of a given $(N, \frac{1}{z}, 1/32)$-combined-ultra-selector $\mathcal{F} = \{F_1, \ldots, F_a\}$. Epochs have the same length, equal to twice the upper bound on the length of procedure Multi_DFS_Traversal. In epoch i, procedure Multi_DFS_Traversal$(N, g\eta, R_i)$ is run, where R_i is the set of representatives that are in F_i and the constant $g > 8$ is from Lemma 4. Every token that succeeds in a run repeats its run in the consecutive $g\eta$ rounds, called *confirming rounds*, in order to announce that it succeeded. If there are no such tokens, then the nodes stay idle during these $g\eta$ rounds. A node's name heard last during the confirming rounds is understood as the leader's name by a node.

Regarding Sub-stage 2.a: The definition of an (N, η)-selective family guarantees that each representative hears a single neighbor in some round of executing a selective family, and the first such a neighbor becomes the witness of this representative. A transmission by a representative informs its witness that it has been chosen. As the representatives are sufficiently far away from each other, by the properties of Partial_Multi_Broadcast, all their transmissions are heard.

Regarding Sub-stage 2.b: The number a is $\mathcal{O}(\frac{1}{z} \log N)$, by Lemma 2. Note that confirming rounds may not suffice for a token to cover the whole network.

Lemma 7. *Consider an execution of algorithm Deterministic_Leader_Election(z). Upon completion of Stage 1 the number of representatives is between 1 and $\frac{1}{z}$. The distance between any two representatives in the network is at least 4.*

Algorithm Deterministic_Leader_Election is defined as Deterministic_Leader_Election(z) in which we substitute $\frac{1}{\sqrt{\log \eta \log\log \eta}}$ as the value of the parameter z.

Theorem 3. *Algorithm Deterministic_Leader_Election produces a leader within $\mathcal{O}(n \log^{3/2} n \sqrt{\log\log n})$ time.*

6 Gossiping

We now show an application of our leader election algorithms to obtain fast gossiping algorithms in the model of combined messages. Initially, each node of the network has a rumor and the goal is to make all rumors known to all nodes.

A message can contain an arbitrary number of rumors and additionally $\mathcal{O}(\log n)$ control bits. The algorithm resorts to a subroutine to elect a leader and produce a spanning tree rooted at this leader. The bird's-eye view of the algorithm:

Algorithm Gossip

Part 1 : Elect a leader and produce a spanning tree rooted at the leader.
Part 2 : Send a token along the spanning tree, to gather all the rumors at the visited nodes and bring them to the leader.
Part 3 : Send a token again along the spanning tree, to disseminate the rumors among the nodes.

In this algorithm, we use either the deterministic or the randomized algorithms for leader election. In the randomized case, the leader attempts to build a spanning tree. The leader applies Rand_Multi_Tree_Search, with just one source. We want to be certain that a tree has been constructed. To this end, the leader launches a token which traverses the tree and verifies in each visited node if there is a neighbor omitted from the tree, by using procedure Echo. When such an omitted node is located then it is attached as a neighbor in the tree to the node hosting the token at the moment.

Theorem 4. *The instantiation of algorithm Gossip with deterministic leader election terminates in* $\mathcal{O}(n \log^{3/2} n \sqrt{\log \log n})$ *time. The instantiation of algorithm Gossip with randomized leader election terminates in* $\mathcal{O}(n)$ *expected time.*

7 Conclusion

We presented fast distributed algorithms for electing a leader and for gossiping with combined messages. The bounds for the randomized case are tight. Regarding deterministic solutions for these problems, we give algorithms of $\mathcal{O}(n \log^{3/2} n \sqrt{\log \log n})$ time performance, while $\Omega(n \log n)$ is the greatest known lower bound. How to close this gap is a natural open question.

References

1. Afek, Y., Gafni, E.: Time and message bounds for election in synchronous and asynchronous complete networks. SIAM J. on Computing 20(2), 376–394 (1991)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. John Wiley (2004)
3. Awerbuch, B.: Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In: Proceedings of the 19th ACM Symposium on Theory of Computing, STOC, pp. 230–240 (1987)
4. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. Journal of Computer and System Sciences 45(1), 104–126 (1992)
5. Censor-Hillel, K., Gilbert, S., Kuhn, F., Lynch, N.A., Newport, C.C.: Structuring unreliable radio networks. In: Proceedings of the 30th ACM Symposium on Principles of Distributed Computing, PODC, pp. 79–88 (2011)

6. Chlebus, B.S., Kowalski, D.R., Pelc, A., Rokicki, M.A.: Efficient Distributed Communication in Ad-Hoc Radio Networks. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 613–624. Springer, Heidelberg (2011)
7. Chlebus, B.S., Kowalski, D.R., Radzik, T.: Many-to-many communication in radio networks. Algorithmica 54(1), 118–139 (2009)
8. Clementi, A.E.F., Monti, A., Silvestri, R.: Distributed broadcast in radio networks of unknown topology. Theoretical Computer Sciences 302(1-3), 337–364 (2003)
9. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. Journal of Algorithms 60(2), 115–143 (2006)
10. De Marco, G.: Distributed broadcast in unknown radio networks. SIAM Journal on Computing 39(6), 2162–2175 (2010)
11. Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Transactions on Programming Languages and Systems 5(1), 66–77 (1983)
12. Garay, J.A., Kutten, S., Peleg, D.: A sublinear time distributed algorithm for minimum-weight spanning trees. SIAM J. on Computing 27(1), 302–316 (1998)
13. Gąsieniec, L.: On Efficient Gossiping in Radio Networks. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 2–14. Springer, Heidelberg (2010)
14. Gąsieniec, L., Pagourtzis, A., Potapov, I., Radzik, T.: Deterministic communication in radio networks with large labels. Algorithmica 47(1), 97–117 (2007)
15. Jurdziński, T., Kutyłowski, M., Zatopiański, J.: Efficient algorithms for leader election in radio networks. In: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, PODC, pp. 51–57 (2002)
16. Kowalski, D.R., Pelc, A.: Broadcasting in undirected ad hoc radio networks. Distributed Computing 18(1), 43–57 (2005)
17. Kowalski, D.R., Pelc, A.: Leader Election in Ad Hoc Radio Networks: A Keen Ear Helps. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 521–533. Springer, Heidelberg (2009)
18. Kuhn, F., Lynch, N.A., Newport, C.C., Oshman, R., Richa, A.W.: Broadcasting in unreliable radio networks. In: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing, PODC, pp. 336–345 (2010)
19. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC, pp. 513–522 (2010)
20. Kushilevitz, E., Mansour, Y.: An $\Omega(D \log(N/D))$ lower bound for broadcast in radio networks. SIAM Journal on Computing 27(3), 702–712 (1998)
21. Martel, C.U.: Maximum finding on a multiple access broadcast network. Information Processing Letters 52(1), 7–15 (1994)
22. Nakano, K., Olariu, S.: A survey on leader election protocols for radio networks. In: Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks, I-SPAN, pp. 63–68 (2002)
23. Newport, C.C., Lynch, N.A.: Modeling radio networks. Distributed Computing 24(2), 101–118 (2011)
24. Vaya, S.: Faster gossiping in bidirectional radio networks with large labels. CoRR abs/1105.0479 (2011)
25. Willard, D.E.: Log-logarithmic selection resolution protocols in a multiple access channel. SIAM Journal on Computing 15(2), 468–477 (1986)

Tree Exploration by a Swarm of Mobile Agents

Jurek Czyzowicz*, Andrzej Pelc**, and Mélanie Roy

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

Abstract. A swarm of mobile agents starting at the root of a tree has to explore it: every node of the tree has to be visited by at least one agent. In every round, each agent can remain idle or move to an adjacent node. In any round all agents have to be at distance at most d, where d is a parameter called the *range* of the swarm. The goal is to explore the tree as fast as possible.

If the topology of the tree is known to the agents, we establish optimal exploration time for any range d and give an optimal exploration algorithm. The formula for the optimal exploration time of a tree by a swarm of agents depends on the range of the swarm and on the characteristics of the tree. If the tree is unknown, the quality of an exploration algorithm $\mathcal{A}$ is measured by comparing its time to that of the optimal algorithm having full knowledge of the tree. The ratio between these times, maximized over all starting nodes and over all trees, is called the *overhead* of algorithm $\mathcal{A}$. Overhead 2 is achieved when the swarm executes a DFS, remaining together all the time. We show that this overhead cannot be improved, for any range d.

Keywords: algorithm, exploration, swarm of mobile agents, tree.

1 Introduction

1.1 The Model and the Problem

A swarm of mobile agents starting at some node of the tree, called its root, has to explore the whole tree: every node of the tree has to be visited by at least one agent. Agents move in synchronous rounds: in every round, each agent can remain idle or move to an adjacent node. It is required that in any round every pair of agents be at distance at most d, where $d \geq 0$ is an integer parameter called the *range* of the swarm. This is motivated by a possible need of the agents for maintaining fast communication at all stages of the exploration. The time of an exploration algorithm is the number of rounds until the last node of the tree is visited by some agent. The goal is to explore the tree as fast as possible.

We consider two scenarios. In the first one, all agents have complete knowledge of the tree, i.e., they have a labeled isomorphic copy of it. In this case the

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

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 121–134, 2012.

algorithm is centralized and the goal is to achieve the shortest time for this particular tree. In the second scenario the topology of the tree is unknown. We assume that all nodes have distinct labels, and all ports at a node v are numbered 1,..., $\deg(v)$. Hence an agent can recognize nodes that it has already visited and edges that it has already traversed. However, it cannot tell the difference between edges incident to its current position that it has not yet explored, i.e., it does not know the other ends of such edges. If the agent decides to use such an unexplored edge, the actual choice of the edge belongs to the adversary, as we are interested in worst-case performance.

Fix a range d of the swarm. For a given exploration algorithm $\mathcal{A}$ not knowing the topology of the tree, the *time* $\tau(\mathcal{A}, T, v)$ of this algorithm run on a tree T from a starting node v is the worst-case number of rounds until the visit of the last node, taken over all of the above choices of the adversary. On the other hand, $opt(T, v)$ is the time of an optimal algorithm having complete knowledge of the tree. For a given tree T and a given starting node v, a natural measure of quality of an exploration algorithm $\mathcal{A}$ not knowing the topology is the ratio $\tau(\mathcal{A}, T, v)/opt(T, v)$ of its time to that of the optimal algorithm having complete knowledge of the tree, cf. [9]. This ratio represents the relative penalty payed by the algorithm for the lack of knowledge of the environment. The value

$$\mathcal{O}(\mathcal{A}) = \sup_{T \in \mathcal{T}} \max_{v \in T} \frac{\tau(\mathcal{A}, T, v)}{opt(T, v)},$$

where $\mathcal{T}$ is the class of all trees, is called the *overhead* of algorithm $\mathcal{A}$. It is the maximum relative penalty described above, over all starting nodes in all trees. The lower the overhead of an exploration algorithm, the closer is its performance (in the worst case) to that of the optimal algorithm having full knowledge of the environment.

Let DFS be any depth-first-search algorithm in the tree, executed by all agents staying together during the whole exploration. Since, for any tree T with e edges and any starting node v , we have $\tau(DFS, T, v) \leq 2e$ (depth-first search traverses each edge at most twice), and $opt(T, v) \geq e$ (every edge has to be traversed at least once), it follows that the overhead of DFS is at most 2.

1.2 Our Results

If the topology of the tree is known to the agents, we establish optimal exploration time for any range d and any tree, and give an optimal exploration algorithm. The formula for the optimal exploration time of a tree by a swarm of agents depends on the range of the swarm and on the characteristics of the tree. If the tree is unknown, we show that there is no exploration algorithm with overhead smaller than 2, for any range d. Hence DFS has always optimal overhead. We also observe that for known topology, the size of the swarm sufficient to execute the optimal algorithm is equal to the number of leaves in the tree. On the other hand, for unknown topology, the best overhead can be achieved by a single agent regardless of the range d.

Due to space constraints, proofs of several lemmas are omitted.

1.3 Related Work

Algorithms for graph exploration by mobile agents (often called robots) have been intensely studied in recent literature. A lot of research is concerned with the case of a single robot exploring the graph. In [1,3,4,8,14] the robot explores strongly connected directed graphs and it can move only in the direction from head to tail of an edge, not vice-versa. In particular, [8] investigates the minimum time of exploration of directed graphs, and [1,14] give improved algorithms for this problem in terms of the deficiency of the graph (i.e., the minimum number of edges to be added to make the graph Eulerian). Many papers, e.g., [9,12,13,20] study the scenario where the explored graph is undirected and the robot can traverse edges in both directions. In [20] it is shown that a graph with n nodes and e edges can be explored in time $e + O(n)$. In some papers, additional restrictions on the moves of the robot are imposed. It is assumed that the robot has either a restricted tank [5], forcing it to periodically return to the base for refueling, or that it is tethered, i.e., attached to the base by a rope or cable of restricted length [13]. In [9] the authors investigate the problem of how the availability of a map influences the efficiency of exploration. When the map is not available, the quality of the algorithm is measured by its overhead, as we do in the present paper.

In all the above papers, except [4] which deals with randomized algorithms, exploration is performed by a single robot. Deterministic exploration by many robots has been investigated mostly in the context when moves of the robots are centrally coordinated. In [18], approximation algorithms are given for the collective exploration problem in arbitrary graphs. In [2] the authors construct approximation algorithms for the collective exploration problem in weighted trees. On the other hand, in [17] the authors study the problem of distributed collective exploration of trees of unknown topology. The robots performing exploration start in the same node and can directly communicate with each other. As opposed to our scenario, no bound on the distances between the robots is assumed. Exploration of arbitrary anonymous graphs by a team of robots communicating through *whiteboards* has been studied in [7].

Another stream of research [6,15,16,19] concerned deterministic graph exploration by very weak robots that cannot communicate directly with each other and cannot leave any marks on nodes but have very strong perception capabilities: they can perceive the entire environment, i.e., see the whole graph with the locations of other robots in it. Probabilistic exploration in the ring in the above model has been the object of investigation in [10,11].

2 Exploration of a Known Tree

In this section we present an optimal algorithm for exploration of a known tree T by a swarm of mobile agents with range $d \geq 0$. All agents start at the root r of the tree. A leaf of a tree is any node of degree 1 other than the root. We denote by n the number of nodes in the tree and by h its height, i.e., the largest distance

from the root to a leaf. For any node v of the tree we denote by T_v the subtree of T rooted at v and consisting of all descendants of v in T. Let $dist(u, v)$ denote the distance between nodes u and v in the tree. For any node v of the tree and for any integer $x \geq 0$, we denote by $\Gamma_x(v)$ the set of nodes at distance at most x from v. We use the notation $\Gamma(v)$ for $\Gamma_1(v)$. Since the description and analysis of the algorithm depend on the parity of the range d, we divide the presentation into two subsections, corresponding to d even and d odd. We first present the algorithms assuming that the number of agents is unlimited. In particular, we use the instruction $expand(v)$, for a node v currently occupied by agents to mean that the set of agents occupying node v in some round is partitioned in such a way that subsets of this set occupy all nodes of $\Gamma(v)$ in the next round. For this to be possible, there must be sufficiently many agents in each considered set during the execution of the algorithm. We will later show that our algorithms can be modified to work in the same time using only m agents, where m is the number of leaves in the tree.

2.1 Even Range

In this section we assume that the range d is even: $d = 2\rho$ for some non-negative integer ρ. We start with the following well-known observation concerning the case $d = 0$, which corresponds to the exploration of the tree by a single agent. In this case the optimal algorithm is any depth-first-search traversal of the tree for which the last-visited leaf is a deepest one. This algorithm takes time $2(n-1)-h$. To see that this is optimal, notice that in any exploration algorithm the agent must traverse at least twice every edge except those on the branch to the last visited leaf, and all edges of this branch must be traversed at least once. We call this algorithm SA (for Single Agent). (Strictly speaking, there may be many such algorithms, depending on the order of exploration of children of each node: we choose some canonical order.)

Consider any $d = 2\rho$, for $\rho > 0$. We say that an exploration algorithm by a swarm with range d is a *kernel algorithm*, if in any round $k \leq \rho$ the set of nodes occupied by agents in round k is $\Gamma_k(r)$, and for any round $k > \rho$ there exists a node c_k, called the kernel in round k, such that the set of nodes occupied by agents in round k is $\Gamma_\rho(c_k)$. Intuitively, in a kernel algorithm, the set of occupied nodes first grows around the root to the maximum size allowed by the range. In the following rounds the kernel moves in the tree. In each round all nodes that can be occupied by agents (without violating the range) are occupied by them.

In order to describe our algorithm, we define the following subtree T' of the tree T.

Definition 1. *Fix a positive integer ρ. Let L be the set of nodes v for which the subtree T_v has height ρ. Let T' be the subtree of T which results from deleting all edges of trees T_v, for $v \in L$, and all nodes of these trees except the root $v \in L$ of each of them.*

We are now ready to describe the algorithm.

Algorithm Even-Range(d)
Let $\rho = d/2$. The algorithm consists of two phases: forming the swarm and exploration. If $\rho \geq h$, only the first phase is executed. During rounds $1, \ldots, h$, for all occupied nodes v the procedure $expand(v)$ is executed. Upon completion of this phase all nodes are explored. If $\rho < h$, the first phase lasts ρ rounds: during rounds $1, \ldots, \rho$, for all occupied nodes v the procedure $expand(v)$ is executed. At this point the swarm is formed, all nodes in $\Gamma_\rho(r)$ are occupied and the exploration phase starts. Let $c_\rho = r$ be the kernel in round ρ. In the exploration phase the kernel of the swarm executes algorithm SA in the tree T'. More precisely, in round $\rho + i$ the kernel of the swarm is the node which the single agent occupies in round i of the algorithm SA executed for the tree T'. It remains to describe the behavior of the agents. Suppose that in round k the kernel is the node $c_k = v$ and in round $k+1$ the kernel is the node $c_{k+1} = w$. Let F be the set of nodes u such that $dist(v, u) = \rho$ and $dist(w, u) = \rho - 1$. Let R be the set of nodes u such that $dist(v, u) = \rho$ and $dist(w, u) = \rho + 1$. In round $k+1$ agents in a node $u \in R$ move to its unique neighbor u' such that $dist(w, u') = \rho$, and for all nodes $u \in F$ the procedure $expand(u)$ is executed. Agents in all other nodes remain idle. This proceeds until algorithm SA is completed in the tree T' by the kernel.

The following lemma establishes correctness of Algorithm Even-Range(d) and its execution time.

Lemma 1. *Let d be a non-negative even integer and let T be a tree of height h with n nodes. Let X be the number of nodes v for which the subtree T_v rooted at v has height smaller than $d/2$. Then upon completion of Algorithm Even-Range(d) all nodes of T are explored. If $h \leq d/2$, the execution time is h. If $h > d/2$, the execution time is $2(n - X - 1) - h + d$.*

Proof. If $h \leq d/2$, then after h rounds of the first phase each node of the tree is visited by some agent of the swarm. Hence we may assume $h > d/2$. Since the kernel of the swarm visits the entire tree T' (it executes algorithm SA for this tree) and every node of T is at distance at most $d/2$ from some node in T', we conclude that all nodes of T are visited upon completion of Algorithm Even-Range(d). The time of execution is equal to the sum of the time of forming the swarm and of executing algorithm SA on the tree T'. The first is $d/2$ and the second is $2(n' - 1) - h'$, where n' is the number of nodes of T' and h' is the height of T'. We have $n' = n - X$ and $h' = h - d/2$. Hence the total execution time of Algorithm Even-Range(d) for $h > d/2$ is $d/2 + 2(n - X - 1) - (h - d/2) = 2(n - X - 1) - h + d$.

We now turn attention to the optimality of Algorithm Even-Range(d). Optimality is straightforward for $h \leq d/2$, hence assume $h > d/2$. We first show that for any exploration algorithm $\mathcal{A}$ by a swarm with even range d there exists a kernel algorithm $\mathcal{A}'$ with the same range whose execution time is equal to that of $\mathcal{A}$. For round $k \geq 1$, let N_k be the set of nodes occupied in round k by the agents

executing algorithm $\mathcal{A}$ and let N'_k be the set of nodes occupied in round k by the agents executing algorithm $\mathcal{A}'$. Given $\mathcal{A}$, the algorithm $\mathcal{A}'$ is described as follows.

Let $\rho = d/2$. In rounds $k = 1, \ldots, \rho$, for all occupied nodes u the procedure $expand(u)$ is executed. We have $N'_\rho = \Gamma_\rho(r)$. In all rounds $k \leq \rho$ the kernel is r. Consider a round $k \geq \rho$ and suppose that this is not the last round of $\mathcal{A}$. Let $c_k = v$ be the kernel of $\mathcal{A}'$ after round k. We define the kernel c_{k+1} after round $k+1$. If $N_{k+1} \subseteq N'_k$ then $c_{k+1} = c_k$. Otherwise, let $Z = N_{k+1} \setminus N'_k$ and let Y be the set of nodes in N_k adjacent to some $z \in Z$. Let w be the unique neighbor of c_k for which there exists a node $y \in Y$, such that $dist(c_k, y) = dist(w, y) + 1$. We put $c_{k+1} = w$. This completes the definition of the kernel. The moves of agents are defined as in Algorithm Even-Range(d). More specifically, let F be the set of nodes u such that $dist(v, u) = \rho$ and $dist(w, u) = \rho - 1$. Let R be the set of nodes u such that $dist(v, u) = \rho$ and $dist(w, u) = \rho + 1$. In round $k+1$ agents in a node $u \in R$ move to its unique neighbor u' such that $dist(w, u') = \rho$, and for all nodes $u \in F$ the procedure $expand(u)$ is executed. Agents in all other nodes remain idle.

The above described algorithm $\mathcal{A}'$ is a kernel algorithm. By definition of $\mathcal{A}'$, its execution time is equal to that of $\mathcal{A}$. In order to prove its correctness, it is enough to observe that $N_k \subseteq N'_k$, for any round k. This follows by induction, since $N'_k = \Gamma_\rho(c_k)$, for $k \geq \rho$.

In view of the above, in the proof of optimality of Algorithm Even-Range(d) it is enough to consider kernel algorithms. Hence the following lemma establishes optimality of Algorithm Even-Range(d).

Lemma 2. *Let d be a non-negative even integer and let T be a tree of height h with n nodes. Let X be the number of nodes v for which the subtree T_v rooted at v has height smaller than $d/2$.*
1. If $h \leq d/2$, then every kernel exploration algorithm by a swarm with range d has execution time at least h.
2. If $h > d/2$, then every kernel exploration algorithm by a swarm with range d has execution time at least $2(n - X - 1) - h + d$.

Lemmas 1 and 2 imply the following theorem:

Theorem 1. *Algorithm Even-Range(d) is a correct and optimal algorithm for exploring a given tree by a swarm of mobile agents with even range d.*

2.2 Odd Range

In this section we assume that the range d is odd: $d = 2\rho + 1$ for some non-negative integer ρ. We first give the algorithm and its analysis for the special case of $d = 1$ and then show how this can be generalized for an arbitrary odd range d.

Range $d = 1$. Consider the algorithm SA for the tree T, starting at the root r. Let $(v_0, v_1, \ldots, v_k)$, where $v_0 = r$, be the sequence of nodes of T, such that v_i

is the node visited in round i by the agent executing SA. Since every leaf of the tree is visited exactly once, this sequence induces an order of the leaves of T. Let $(f_1, \ldots, f_m)$ be the subsequence of $(v_0, v_1, \ldots, v_k)$ consisting of leaves. A leaf f_i will be called odd (even), if i is odd (even). Consider the sequence $(i_1, i_2, \ldots, i_m)$, such that $f_j = v_{i_j}$. We partition the sequence $(0, 1, \ldots, k)$ of indices into disjoint segments $[0, \ldots, i_1]$, $[i_1 + 1, \ldots i_2]$, ..., $[i_{m-1} + 1, \ldots, i_m]$. We call a segment odd if it finishes with the index of an odd leaf and even if it finishes with the index of an even leaf.

Algorithm Range-One

In round 1 all agents are partitioned into two non-empty subsets, called *head* and *back*. The head goes to node v_1 and the back remains at the root v_0. In every round i, for $1 \leq i < k$, the head is in node $v_{H(i)}$ and the back is in node $v_{B(i)}$, where indices $H(i)$ and $B(i)$ are defined as follows:

$$H(i+1) = \begin{cases} 1 & \text{if } i = 0 \\ H(i) + 1 & \text{if } v_{H(i)+1} \text{ is not an even leaf} \\ H(i) + 3 & \text{if } v_{H(i)+1} \text{ is an even leaf} \end{cases}$$

$$B(i+1) = \begin{cases} 0 & \text{if } i = 0 \\ B(i) + 1 & \text{if } v_{B(i)+1} \text{ is not an odd leaf} \\ B(i) + 3 & \text{if } v_{B(i)+1} \text{ is an odd leaf} \end{cases}$$

In order to establish the correctness of Algorithm Range-One we will need the following technical lemma.

Lemma 3. *For any round $i \geq 1$ the following properties are satisfied.*

1. *If $H(i)$ is in an odd segment, then $B(i) = H(i) - 1$.*
2. *If $H(i)$ is in an even segment, then $B(i) = H(i) + 1$.*
3. *$H(i)$ is not an index of an even leaf, and $B(i)$ is not an index of an odd leaf.*
4. *$H(i)$ and $B(i)$ belong to the same segment.*

We are now ready to prove the correctness and optimality of Algorithm Range-One.

Theorem 2. *Algorithm Range-One is correct.*

Proof. The proof is split into four claims. The first three claims show that all moves of agents are legal and the fourth claim shows that all nodes of the tree are visited.

Claim 1. $v_{B(i+1)}$ is adjacent to $v_{B(i)}$.

By Lemma 3 $B(i)$ is not an index of an odd leaf. We have two cases.

Case 1. $B(i)$ is not an index of an even leaf

If $B(i) + 1$ is not an index of an odd leaf, we have $B(i + 1) = B(i) + 1$, hence $v_{B(i+1)}$ is adjacent to $v_{B(i)}$. If $B(i) + 1$ is an index of an odd leaf we have $v_{B(i)} = v_{B(i)+2}$. By definition, $v_{B(i)+3}$ is adjacent to $v_{B(i)+2}$. Hence $v_{B(i)+3}$

is adjacent to $v_{B(i)}$. By the algorithm, $B(i+1) = B(i) + 3$. Hence $v_{B(i+1)}$ is adjacent to $v_{B(i)}$.

Case 2. $B(i)$ is an index of an even leaf

Hence $B(i)+1$ is not an index of a leaf. By the algorithm, $B(i+1) = B(i)+1$. Hence $v_{B(i+1)}$ is adjacent to $v_{B(i)}$. This completes the proof of Claim 1.

Claim 2. $v_{H(i+1)}$ is adjacent to $v_{H(i)}$.

The proof is analogous to that of Claim 1.

Claim 3. $v_{H(i)}$ and $v_{B(i)}$ are adjacent, for any round $i \geq 1$.

By Lemma 3, if $H(i)$ is in an odd segment, then $B(i) = H(i) - 1$ and if $H(i)$ is in an even segment, then $B(i) = H(i) + 1$. Hence in both cases nodes $v_{H(i)}$ and $v_{B(i)}$ are adjacent.

Claim 4. Every node of the tree T is visited by some agent.

By the algorithm, the head visits all odd leaves and the back visits all even leaves, hence all leaves are visited. Exploration of all leaves implies exploration of all nodes.

In order to prove the optimality of Algorithm Range-One, we first establish its execution time.

Lemma 4. *The execution time of Algorithm Range-One is* $2n-m-h-1$*, where* n *is the number of nodes of the tree,* m *is the number of leaves, and* h *is the height of the tree.*

Proof. Let $\overline{T}$ be the tree resulting from the original tree T by removing even leaves $f_2, f_4, \ldots$, as well as the incident edges. The execution time of Algorithm Range-One in the tree T is equal to the execution time of algorithm SA in the tree $\overline{T}$, where the head simulates actions of the single agent in SA.

Case 1. The number m of leaves of the tree T is even; $m = 2b$.

The last visited leaf f_m is even, hence it is visited by the back. At the end, the head is at distance $h-1$ from the root. The execution time is $2(n'-1)-(h-1)$, where $n' = n - b$ is the number of nodes of tree $\overline{T}$. Hence the execution time is $2(n'-1)-(h-1) = 2(n-b-1)-(h-1) = 2n-m-h-1$.

Case 2. The number m of leaves of the tree T is odd; $m = 2b+1$.

The last visited leaf f_m is odd, hence it is visited by the head. At the end, the head is at distance h from the root. The execution time is $2(n'-1)-h$, where $n' = n - b$ is the number of nodes of tree $\overline{T}$. Hence the execution time is $2(n'-1)-h = 2(n-b-1)-h = 2n-m-h-1$.

In the optimality proof we will also use the following lemma.

Lemma 5. *In an optimal algorithm with range* $d = 1$ *two consecutive leaves cannot be visited by the same agent.*

The following lower bound, together with Lemma 4, shows that Algorithm Range-One is optimal.

Lemma 6. *Any algorithm to visit a tree* T *by a swarm of agents with range 1 must use time at least* $2n-m-h-1$*, where* n *is the number of nodes of the tree,* m *is the number of leaves, and* h *is the height of the tree.*

Proof. Consider an optimal algorithm $\mathcal{A}$. Order all leaves by their first visit according to $\mathcal{A}$. Let $g_1, \ldots, g_m$ be this order. By Lemma 5, the same agent cannot visit consecutive leaves. Hence, without loss of generality, some agent a visits odd leaves and some other agent b visits even leaves (with respect to this order). Let T^* be the tree resulting from removing from T all even leaves g_i with their incident edges. Hence a visits the entire tree T^*. If m is odd, $m = 2b + 1$, the number of nodes in T^* is $n' = n - b$ and hence visiting the entire tree T^* by agent a must take time at least $2(n' - 1) - h = 2n - m - h - 1$. If m is even, $m = 2b$, the last visited leaf is visited by agent b. At this time agent a is at distance at least $h - 1$ from the root. The number of nodes in T^* is $n' = n - b$ and hence visiting the entire tree T^* by agent a must take time at least $2(n' - 1) - (h - 1) = 2n - m - h - 1$.

Theorem 2, Lemma 4 and Lemma 6 imply:

Theorem 3. *Algorithm Range-One is a correct and optimal algorithm for exploring a tree by a swarm of mobile agents with range* 1.

Odd Range $d > 1$. We now show how Algorithm Range-One can be transformed to work for any odd range $d > 1$. In essence, the relation between the general case of the odd range and the case of range 1 is similar to the relation between the general case of the even range and the case of a single agent, the latter being equivalent to range 0.

Consider any $d = 2\rho + 1$, for $\rho > 0$. We say that an exploration algorithm by a swarm with range d is a *kernel algorithm*, if in round 1 the agents occupy the root r and an adjacent node s, in any round $1 + k$, for $k = 1, 2, \ldots, \rho$, the set of nodes occupied by agents is $\Gamma_k(r) \cup \Gamma_k(s)$, and for any round $k > \rho+1$ there exist adjacent nodes u_k and v_k, where the set $\{u_k, v_k\}$ is called the kernel in round k, such that the set of nodes occupied by agents in round k is $\Gamma_\rho(u_k) \cup \Gamma_\rho(v_k)$. Intuitively, in a kernel algorithm for an odd range, the set of occupied nodes first grows to two nodes, the root and some adjacent node, then grows around these two nodes to the maximum size allowed by the range, and then in the following rounds the kernel moves in the tree and in each round all nodes that can be occupied by agents (without violating the range) are occupied by them.

In order to describe our algorithm, we use the subtree T' of the tree T described in Definition 1. We are now ready to describe the algorithm.

Algorithm Odd-Range(d)
Let $\rho = \lfloor d/2 \rfloor$ and let D be the diameter of the tree T. If $D \leq d$ then during rounds $1, \ldots, h$, for all occupied nodes v the procedure *expand*(v) is executed. After round h all nodes are explored. If $D > d$, the algorithm consists of two phases: forming the swarm and exploration. The first phase lasts $\rho+1$ rounds. Let s be the first node other than the root r, visited in the execution of Algorithm Range-One on the tree T'. In the first round of the first phase all nodes are partitioned into two subsets and one subset remains at r, while the other occupies s. In rounds $2, 3, \ldots, \rho + 1$, for all occupied nodes v the procedure *expand*(v)

is executed. At this point the swarm is formed, all nodes in $\Gamma_\rho(r) \cup \Gamma_\rho(s)$ are occupied and the exploration phase starts.

Let $\{u_{\rho+1}, v_{\rho+1}\}$, where $u_{\rho+1} = r$ and $v_{\rho+1} = s$ be the kernel in round $\rho + 1$. In the exploration phase the kernel of the swarm executes Algorithm Range-One in the tree T'. More precisely, in round $\rho+i$ the kernel of the swarm is composed of two adjacent nodes which are occupied by the agents in round i of Algorithm Range-One executed for the tree T'. It remains to describe the behavior of the agents. Suppose that in round k the kernel is the set $\{u, v\}$ and in round $k + 1$ the kernel is the set $\{v, w\}$. (Notice that in Algorithm Range-One the sets of nodes occupied by the agents in consecutive rounds have intersection of size 1.) Let F be the set of nodes z such that $dist(z, v) = \rho$ and $dist(z, w) = \rho - 1$. Let R be the set of nodes z such that $dist(z, u) = \rho$ and $dist(z, v) = \rho + 1$. In round $k + 1$ agents in a node $z \in R$ move to its unique neighbor z' such that $dist(z', v) = \rho$, and for all nodes $z \in F$ the procedure $expand(z)$ is executed. Agents in all other nodes remain idle. This proceeds until Algorithm Range-One is completed in the tree T' by the kernel.

The following lemma establishes correctness of Algorithm Odd-Range(d) and its execution time.

Lemma 7. *Let d be a positive odd integer, $d = 2\rho + 1$, and let T be a tree of height h and diameter D. Let T' be the subtree of T described in Definition 1. Let n' be the number of nodes of T', m' the number of leaves of T' and $h' = h - \rho$ the height of T'.*

Upon completion of Algorithm Odd-Range(d) all nodes of T are explored. If $D \leq d$, then the execution time is h. If $D > d$, then the execution time is $\rho + 2n' - m' - h' - 1$.

The proof of optimality of Algorithm Odd-Range(d) is similar to that for Algorithm Even-Range(d). For $D \leq d$ optimality is straightforward, hence we may assume $D > d$. Similarly as before we show that for any algorithm $\mathcal{A}$ with odd range d there exists a kernel algorithm $\mathcal{A}'$ with the same range, whose execution time is equal to that of $\mathcal{A}$. The construction of $\mathcal{A}'$ from $\mathcal{A}$ is analogous to the case of even range. Then using Lemma 2 we show that every kernel algorithm must use time at least $\rho + 2n' - m' - h' - 1$, where n', m' and h' are as in Lemma 7. The proof of this part is similar to that of Lemma 2. This implies optimality of Algorithm Odd-Range(d). Hence we get:

Theorem 4. *Algorithm Odd-Range(d) is a correct and optimal algorithm for exploring a given tree by a swarm of mobile agents with odd range $d > 1$.*

Algorithms Even-Range(d), Range-One, and Odd-Range(d) for $d > 1$ cover all cases of exploration of a known tree by a swarm of agents. We showed that they are all correct and optimal.

2.3 The Number of Agents

We finally consider the problem of the number of mobile agents sufficient to perform exploration by a swarm of agents in optimal time. In the description of

our algorithms we used procedure *expand*(v), that required partitioning the set of agents located at the node v in a given round and dispatching the subsets of agents to the neighboring nodes in the next round. Implemented naively, this approach would require an exponential number of agents. However, we now show how our optimal algorithms can be modified to be executed by a swarm of agents whose size is equal to the number of leaves.

Let $\{f_1, \ldots, f_m\}$ be the set of leaves of the given tree T, and suppose that there are m agents in the swarm. Let $\{a_1, \ldots, a_m\}$ be the set of agents. Consider any optimal algorithm $\mathcal{A}$ of exploring the tree T with a given range d. For any round i of the execution of the algorithm, let S_i denote the set of nodes occupied by agents in round i. We construct the algorithm $\mathcal{A}'$ with the same range d as follows. In round i of $\mathcal{A}'$ agent a_j is located in the unique node $v(i,j)$ of S_i closest to leaf f_j. Since for any i and j nodes $v(i,j)$ and $v(i+1,j)$ are either equal or adjacent, the moves of agents satisfying the above condition are possible to execute. Since by definition, for any i the set S_i has diameter at most d, algorithm $\mathcal{A}'$ respects the range d of the swarm. Since algorithm $\mathcal{A}$ explores the entire tree T, so does $\mathcal{A}'$. In fact the leaf f_j is visited by agent a_j in this algorithm. Since the execution time of $\mathcal{A}$ and $\mathcal{A}'$ is the same, algorithm $\mathcal{A}'$ is optimal by optimality of $\mathcal{A}$.

3 Exploration of an Unknown Tree

In this section we consider exploration of an unknown tree by a swarm of mobile agents. As mentioned in the introduction, in this scenario a natural measure of quality of an exploration algorithm is its overhead. The main result of this section shows that, for any range d, there is no exploration algorithm by a swarm with range d that has overhead smaller than 2. Hence depth-first-search, in which all agents move together, has optimal overhead.

Theorem 5. *Every exploration algorithm for trees by a swarm of agents with range $d \geq 0$ has overhead at least 2.*

Proof. Consider any exploration algorithm $\mathcal{A}$ by a swarm with range d. We will show that it must have overhead at least 2, even when restricted to the class of lines, i.e., trees with two leaves. Call one of the directions of the line with respect to the starting node the right direction and the other the left direction. Assume that the starting node is at 0 and positive numbers are right of 0 while negative numbers are left of 0. There are three possible cases for the initial part of the run of $\mathcal{A}$ before an endpoint is reached for the first time, corresponding to three types of algorithms (cf. [9]):

Type 1
There exist two infinite strictly increasing sequences $(a_1, a_2, ...)$ and $(b_1, b_2, ...)$ of natural numbers, such that, first some agent reaches a_1, then some agent reaches $-b_1$, then some agent reaches a_2, then some agent reaches $-b_2$, etc. until an endpoint is reached for the first time.

Type 2
There exist two strictly increasing sequences $(a_1, a_2, ..., a_i)$ and $(b_1, b_2, ..., b_{i-1})$ of natural numbers, such that, first some agent reaches a_1, then some agent reaches $-b_1$, then some agent reaches a_2, then some agent reaches $-b_2$, etc. then some agent reaches a_i, and then some agent goes left till the endpoint.
Type 3
There exist two strictly increasing sequences $(a_1, a_2, ..., a_i)$ and $(b_1, b_2, ..., b_i)$ of natural numbers, such that, first some agent reaches a_1, then some agent reaches $-b_1$, then some agent reaches a_2, then some agent reaches $-b_2$, etc. then some agent reaches a_i, then some agent reaches $-b_i$, and then some agent goes right till the endpoint.

We will show that each of the above three types of exploration algorithms has overhead at least 2. We may assume that $d \geq 1$, as for $d = 0$ exploration by a swarm is equivalent to exploration by a single agent and in this case the result follows from [9].

Algorithms of type 1.
Let $\alpha = 4d - 3$. Let k be such that $a_{k+1}, b_k \geq \alpha$. Let $a = a_{k+1}$, $b = b_k$ and let the line L_n be $[-b-1, -b, ..., 0, ..., a, a+1]$. The line has length $n = a+b+2$. We have $\tau(\mathcal{A}, L_n, 0) \geq 4b + 3a - \alpha$. Indeed, by the time the rightmost agent reaches point a, it already made at least $2b + a - 2d$ steps. Then the leftmost agent makes at least $a + b - d$ additional steps, by regularity. Now the leftmost agent is at distance 1 from the left endpoint, and the right endpoint is yet unexplored. Hence at least $n + 1 - d = a + b - d + 3$ additional steps are needed, for a total of at least $4b + 3a - 4d + 3 = 4b + 3a - \alpha$ steps.

On the other hand, $opt(L_n, 0) = 2a + b + 2 - 2d$ if $b \geq a$, and $opt(L_n, 0) = 2b+a+2-2d$ if $a \geq b$. Hence $opt(L_n, 0) \leq 2a+b$ if $b \geq a$, and $opt(L_n, 0) \leq 2b+a$ if $a \geq b$.

Case 1. $a \geq b$.
In this case we have: $a \geq \alpha$, hence $4b + 3a - \alpha \geq 4b + 2a = 2(2b + a)$, which implies

$$\frac{\tau(\mathcal{A}, L_n, 0)}{opt(L_n, 0)} \geq \frac{4b + 3a - \alpha}{2b + a} \geq 2.$$

Case 2. $b \geq a$.
In this case we have: $b \geq \alpha$, hence $2b - \alpha \geq a$, hence $4b + 3a - \alpha \geq 2b + 4a = 2(b + 2a)$, which implies

$$\frac{\tau(\mathcal{A}, L_n, 0)}{opt(L_n, 0)} \geq \frac{4b + 3a - \alpha}{b + 2a} \geq 2.$$

This proves $\mathcal{O}(\mathcal{A}) \geq 2$ for algorithms of type 1.

Algorithms of type 2.
It is enough to show that, for any $\epsilon > 0$, there exists a line L_n, and a position of the starting node v in it, such that

$$\frac{\mathcal{C}(\mathcal{A}, L_n, v)}{opt(L_n, v)} \geq 2 - \epsilon.$$

Fix an $\epsilon > 0$, and the index i given by the algorithm (the index of the last turn left before going indefinitely left, until the endpoint is reached). Let $a = a_i$. Let n be such that

$$\frac{2n + a - 1 - 3d}{n + a - 1 - 2d} \geq 2 - \epsilon.$$

(Such an integer n exists, since, for any fixed a and d, this fraction converges to 2 as n grows.) Let $k = n - a - 1$. Let the line L_n be $[-k, -k+1, ..., 0, ..., a, a+1]$. The line has length n. We have $\tau(\mathcal{A}, L_n, 0) \geq 2n + a - 1 - 3d$. Indeed, by the last turn left of the leftmost agent before going indefinitely left, this agent makes at least $a - d$ steps. Then it makes at least $n - 1 - d$ steps to reach the left endpoint, and still at least $n - d$ steps have to be made by the rightmost agent to reach the right endpoint, yet unexplored.

On the other hand, $opt(L_n, 0) \leq n + a + 1 - 2d$. Hence we have

$$\frac{\tau(\mathcal{A}, L_n, 0)}{opt(L_n, 0)} \geq \frac{2n + a - 1 - 3d}{n + a + 1 - 2d} \geq 2 - \epsilon.$$

This proves $\mathcal{O}(\mathcal{A}) \geq 2$ for algorithms of type 2. For algorithms of type 3 the proof is analogous to that for type 2. Thus we have shown that $\mathcal{O}(\mathcal{A}) \geq 2$ for all exploration algorithms.

4 Conclusion

We showed algorithms to explore a known tree by a swarm of agents with an arbitrary range d in optimal time. We also showed that for an unknown tree the overhead of DFS (which can be executed by a single agent) is the smallest possible for any range d of the exploring swarm. When the tree is known, we observed that there is an optimal algorithm using only m agents, where m is the number of leaves in the tree. Two open questions follow from our research. The first is: what is the minimum number of agents in a swarm with range d that are sufficient to explore a given tree in optimal time? The second question concerns generalizing our results for arbitrary graphs; more precisely, construct an optimal algorithm to explore a given graph by a swarm of agents with range d in optimal time. Note that in the scenario of unknown graphs the fact that the overhead of DFS is the best possible remains valid for arbitrary graphs, as the overhead of DFS is 2 for the class of all graphs as well.

References

1. Albers, S., Henzinger, M.R.: Exploring unknown environments. SIAM J. Comput. 29, 1164–1188 (2000)
2. Averbakh, I., Berman, O.: A heuristic with worst-case analysis for minimax routing of two traveling salesmen on a tree. Discr. Appl. Math. 68, 17–32 (1996)
3. Bender, M.A., Fernandez, A., Ron, D., Sahai, A., Vadhan, S.: The power of a pebble: exploring and mapping directed graphs. In: Proc. 30th Ann. Symp. on Theory of Computing, STOC 1998, pp. 269–278 (1998)

4. Bender, M.A., Slonim, D.: The power of team exploration: Two robots can learn unlabeled directed graphs. In: Proc. 35th Ann. Symp. on Foundations of Computer Science, FOCS 1994, pp. 75–85 (1994)
5. Betke, M., Rivest, R., Singh, M.: Piecemeal learning of an unknown environment. Machine Learning 18, 231–254 (1995)
6. Chalopin, J., Flocchini, P., Mans, B., Santoro, N.: Network Exploration by Silent and Oblivious Robots. In: Thilikos, D.M. (ed.) WG 2010. LNCS, vol. 6410, pp. 208–219. Springer, Heidelberg (2010)
7. Das, S., Flocchini, P., Kutten, S., Nayak, A., Santoro, N.: Map construction of unknown graphs by multiple agents. Theoretical Computer Science 385, 34–48 (2007)
8. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. J. of Graph Theory 32, 265–297 (1999)
9. Dessmark, A., Pelc, A.: Optimal graph exploration without good maps. Theoretical Computer Science 326, 343–362 (2004)
10. Devismes, S.: Optimal exploration of small rings. In: Proc. 3rd Int. Workshop on Reliability, Availability, and Security, WRAS 2010 (2010)
11. Devismes, S., Petit, F., Tixeuil, S.: Optimal Probabilistic Ring Exploration by Semi-synchronous Oblivious Robots. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 195–208. Springer, Heidelberg (2010)
12. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. Journal of Algorithms 51, 38–63 (2004)
13. Duncan, C.A., Kobourov, S.G., Kumar, V.S.A.: Optimal constrained graph exploration. In: Proc. 12th Ann. ACM-SIAM Symp. on Discrete Algorithms, SODA 2001, pp. 807–814 (2001)
14. Fleischer, R., Trippen, G.: Exploring an Unknown Graph Efficiently. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 11–22. Springer, Heidelberg (2005)
15. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Remembering without memory: tree exploration by asynchronous oblivious robots. Theoretical Computer Science 411, 1544–1557 (2010)
16. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Computing Without Communicating: Ring Exploration by Asynchronous Oblivious Robots. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 105–118. Springer, Heidelberg (2007)
17. Fraigniaud, P., Gasieniec, L., Kowalski, D., Pelc, A.: Collective tree exploration. Networks 48, 166–177 (2006)
18. Frederickson, G.N., Hecht, M.S., Kim, C.E.: Approximation algorithms for some routing problems. SIAM J. Comput. 7, 178–193 (1978)
19. Lamani, A., Potop-Butucaru, M.G., Tixeuil, S.: Optimal Deterministic Ring Exploration with Oblivious Asynchronous Robots. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 183–196. Springer, Heidelberg (2010)
20. Panaite, P., Pelc, A.: Exploring unknown undirected graphs. Journal of Algorithms 33, 281–295 (1999)

Crash Resilient and Pseudo-Stabilizing Atomic Registers*

Shlomi Dolev[1], Swan Dubois[2], Maria Gradinariu Potop-Butucaru[3], Sébastien Tixeuil[4]

[1] Ben-Gurion University of the Negev, Israel
dolev@cs.bgu.ac.il
[2] EPFL, Switezerland
swan.dubois@epfl.ch
[3] UPMC Sorbonne Universités, France
[4] UPMC Sorbonne Universités & IUF, France
{maria.potop-butucaru,sebastien.tixeuil}@lip6.fr

Abstract. We propose a crash safe and pseudo-stabilizing algorithm for implementing an atomic memory abstraction in a message passing system. Our algorithm is particularly appealing for multi-core architectures where both processors and memory contents (including stale messages in transit) are prone to errors and faults. Our algorithm extends the classical fault-tolerant implementation of atomic memory that was originally proposed by Attiya, Bar-Noy, and Dolev (ABD) to a stabilizing setting where memory can be initially corrupted in an arbitrary manner. The original ABD algorithm provides no guaranties when started in such a corrupted configuration. Interestingly, our scheme preserves the same properties as ABD when there are no transient faults, namely the linearizability of operations. When started in an arbitrarily corrupted initial configuration, we still guarantee eventual yet suffix-closed linearizability.

Keywords: Fault-Tolerance, Pseudo-Stabilization, Atomic Register.

1 Introduction

Distributed computing theory has proven extremely relevant in the daily practice of current networked systems. The important properties in today's distributed systems include availability, reliability, serviceability, and fault-tolerance. The multi-core systems for example have to be able to mask the unexpected yet

* The research of the first author has been supported by the Ministry of Science and Technology, the Institute for Future Defense Technologies Research named for the Medvedi, Shwartzman and Gensler Families, the Israel Internet Association, the Lynne and William Frankel Center for Computer Science at Ben-Gurion University, Rita Altura Trust Chair in Computer Science, Israel Science Foundation (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium and MAFAT.

The research of the other authors has been supported in part by ANR project SHAMAN.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 135–150, 2012.

possible faults of processors and memory transient errors. In these architectures applying the classical technique consisting in restarting the system anytime an error or a fault occurs (at least once a day in current systems, but at least once every few minutes –or even seconds– in forecast exascale supercomputers) attains the limits both in terms of energy cost and the time spent in rebooting the system. In these particular systems, fault recovery mechanisms that rely on the paradigm that combines self-stabilization and fault-tolerance techniques at the application level are particularly appealing. *Self-stabilization* [8] is a versatile technique that permits forward recovery from any kind of *transient* fault (*i.e.* there exists a point in the execution after which there is no fault), while *Fault-tolerance* [15] is traditionally used to mask the effect of a limited number of *permanent* faults. Providing core building blocks for application designers (such as atomic memory construction) that are highly resilient to various kinds of failures is essential for the next generation of those systems. However, making distributed systems tolerant to both transient and permanent faults proved difficult [3,17] as impossibility results are expected in many cases.

Related Works. In the context of self-stabilization, the simulation of an atomic single-writer single-reader shared register in a message-passing system was presented in [12]. This simulation does not address the multiple readers case, and does not consider that crash faults of processors may occur in the system during execution. More recent work [11,19] focused on self-stabilizing simulation of shared registers using shared registers with weaker properties than atomicity, and still do not consider crash faults. Self-stabilizing timestamps implementations using single writer multiple readers atomic registers were suggested in [1,13], and assume that there already exists a shared memory abstraction. Most related to our work are [2], where a crash-fault tolerant and "practically" stabilizing scheme for simulating atomic memory in a message passing system is presented. There, practically means that after stabilization, the linearizability is guaranteed for practically infinite time (say time required for a process to execute 2^{64} steps). Still, in every infinite execution suffix of [2], linearizability is violated infinitely often, leaving open the question of suffix-closed linearizability guaranteeing algorithms that are both stabilizing and crash resilient.

Our contribution. In this paper, we answer positively to the open question of [2]. In more details, we propose a crash-safe and pseudo-stabilizing algorithm for implementing an atomic memory abstraction in a message passing system (provided that the writer does not crash before the first "stabilized" read, see below). Pseudo-stabilization guarantees that, starting from any configuration, any execution contains a suffix satisfying linearizability. Hence, pseudo-stabilization is stronger than practical stabilization since we ensure the closure of linearizability.

Our algorithm extends the classical fault-tolerant implementation of atomic memory that was originally proposed in [4] to a stabilizing setting where memory can be initially corrupted in an arbitrary manner. Note that the original algorithm of [4] provides no guarantees when started in such corrupted configuration. Interestingly, we preserve the same properties as the [4] scheme when

there are no transient faults, namely the linearizability of the operations. Additionally, when started in a corrupted initial configuration the algorithm still guarantees eventual yet suffix closed linearizability.

In the current paper, the writer has the major responsibility for updating the last value, unlike [4] where readers assist each other to spread the most up-to-date value. Note that when the system is started in an arbitrary configuration and the writer is crashed before the stabilization, this cascade-like update may lead to executions where the specification is never verified unless an additional mechanism is used. In [2] we used an epoch-based technique in order to circumvent this drawback. However, the solution proposed in [2] respects a weaker specification (*i.e.*, practically stabilization) while the current work respects the pseudo-stabilization specifications.

2 Model and Definitions

This section is devoted to the presentation of the background of this paper. First, we present the distributed system and fault-tolerance model in Sections 2.1 and 2.2, we specify formally our problem in Section 2.3. Finally, we present in details the ABD simulation on which our protocol is built in Section 2.4.

2.1 Message Passing Model

A message-passing distributed system consists of n vertices (*a.k.a.* processes), $v_0, v_1, v_2, \ldots, v_{n-1}$, connected by communication links through which messages are sent and received. Two vertices connected through a communication link are referred in the following as neighboring vertices. The communication graph of the distributed system is assumed to be fully connected (*i.e.* any pair of vertices are neighboring vertices).

We assume in the following that the capacity of each communication link is bounded and that its capacity is c packets (*i.e.* low level messages). We assume that c is known to the protocol. Note that in the scope of self-stabilization, where the system copes with an arbitrary starting configuration, the initial content of each communication link may be arbitrary.

The channels are *unreliable and non-FIFO* (*i.e.* packets may not follow the FIFO order and may be lost). Additionally, their delivery time is unbounded - the system is *asynchronous*. That is, any non lost packet is received in a finite but unbounded time. Each communication link is weakly fair in the sense that if the sender sends infinitely often a packet on the channel, then the receiver receives this packet an infinite number of time. Sending a packet to a channel whose capacity is exhausted (*i.e.* the channel already contains c packets) results in losing a packet (either a packet already in the channel or the packet being sent).

As we deal with arbitrary initial corruptions, a channel may initially contain up to c *ghost packets* (*i.e.* packets that have never been sent and contain arbitrary content).

A vertex is modeled by a state machine that executes steps. Channels are modeled as sets (rather than queues to reflect the non-FIFO order). For example, the c-bounded channel (i, j) (used to send messages from v_i to v_j) is modeled by a c-sized set denoted by s_{ij}.

In each step, a vertex changes its local state (*i.e.* the state of its local memory), and executes a single communication operation, which is either a *send* operation or a *receive* operation. The communication operation changes the state of an attached channel. In case the communication operation is a send operation from v_i to v_j then s_{ij} is a union of s_{ij} in the previous state with the sent packet. If the obtained union does not respect the bound $|s_{ij}| \leq c$ then an arbitrary message in the obtained union is deleted. In case the communication operation is a receive operation of a (non null) packet m (m must exist in s_{ji} of the previous state), then m is removed from s_{ji}. A receive operation by p_i from p_j may result in a null packet even when the s_{ji} is not empty, thus allowing unbounded delay for any particular packet. Packet losses are modeled by allowing spontaneous packet removals from the set.

A configuration of the system is the product of the local states of processes in the system and of their incident channels. An execution is a sequence of configurations, $\sigma = (\gamma_1, \gamma_2, \ldots)$ such that γ_i, $i > 1$, is obtained from γ_{i-1} when at least one process in the system executes a step. We assume that executions are fully asynchronous.

Finally, we assume that the distributed system is simultaneously subject to transient (*i.e.* of finite duration) faults and to (permanent) crash faults (*i.e.* faults in which affected processes stop to execute steps). The number of crash faults is bounded by a constant f. Transient faults may be arbitrary in nature but there exists a point of the execution after that they no longer occur. Hence, we assumed that the processes local state and channels contents are arbitrary in the initial configuration of the system (and that transient faults no longer corrupt the system during the execution).

2.2 Pseudo-Stabilization and Fault-Tolerance

In this paper, we focus on joint tolerance to transient and crash faults. The classical approach for such a tolerance is fault-tolerant self-stabilization (FTSS for short) [3,17] that ensures that the distributed system stabilizes to its specification in a finite time from any arbitrary initial configuration in spite of crash faults. This strong fault tolerance property leads to numerous impossibility results, see *e.g.* [5]. Hence, we choose in this paper a weaker fault tolerance definition, called pseudo-stabilization [6], in which any execution contains a suffix satisfying the specification. Note that, contrarily to self-stabilization, it is not required that this suffix is reached in a finite time.

Definition 1 (Fault-tolerant pseudo-stabilization [7]). *A distributed protocol π is f-fault-tolerant and pseudo-stabilizing (f-ftps for short) for specification spec if and only if starting from any arbitrary configuration every execution of π involving at most f crashed vertices has a suffix satisfying spec.*

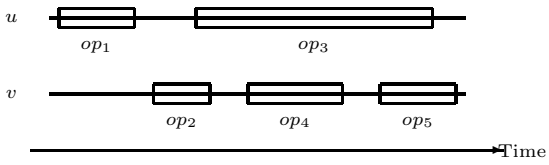


Fig. 1. In this example, op_1 happens before op_2 while op_3 is concurrent with op_2, op_4, and op_5. Operation op_2 and op_4 are consecutive.

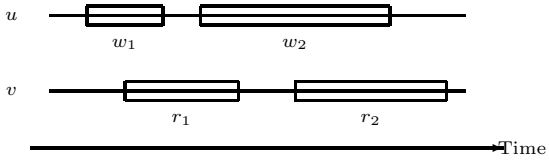


Fig. 2. If r_2 returns the value written by w_1 and r_1 returns the value written by w_2, we have a new/old inversion

2.3 Problem and Specification

In this paper, we emulate an atomic register on top of a message passing system. Registers have been introduced by Lamport [20,21] as a model of communication between vertices of a distributed system. A register is a variable (over a domain D) shared by all vertices of the distributed system that provides two operations: a read operation that returns the value of the register to the invoking vertex and a write operation that allows the invoking vertex to modify the value of the register. Given a register, we call readers the vertices that are able to invoke the read operation of the register and writers the vertices that are able to invoke the write operation of the register. In the following, we consider only single-writer registers. As readers of a register may be distinct from its writer, read and write operations may be interleaved in some executions of the distributed system. Then, we must clarify the result of read operations in such cases. Lamport [20,21] distinguishes three types of registers according to read operation properties: safe, regular and atomic. In the following, we focus on the strongest one, the atomic register.

Note that read and write operations on the register are not instantaneous. Each operation starts when a vertex invokes it and ends when it returns. We say that an operation op_1 happens before an operation op_2 if op_1 ends before op_2 starts. Two operations op_1 and op_2 are concurrent if they satisfy: op_1 does not happen before op_2 and op_2 does not happen before op_1. Two operations op_1 and op_2 are consecutive if op_1 is the most recent operation that happens before op_2. See Figure 1 for an illustration. We introduce now new/old inversions. Consider two consecutive read operations r_1, r_2 and two consecutive write operations w_1, w_2 such that r_1 is concurrent with both w_1 and w_2 and r_2 is concurrent only with w_2 (see Figure 2). We say that a new/old inversion occurs when r_2 returns the value written by w_1 and r_1 returns the value written by w_2.

The writer that is supplied with two operations: *read* and *write* while other vertices, the readers, are supplied with only one operation: *read*. Each *read* invocation needs no parameter and returns a value from D, the domain of the register. Each *write* invocation needs a parameter from D and returns no value. We say that a value v is written to the register when the operation $write(v)$ returns. Intuitively, an atomic register is a register such that all its read and write operations appear as if they have been executed sequentially, this sequential total order respecting the real time order of the operations. More formally, we can define it as follows.

Specification 1 ($spec_{ARS}$)**.** *An execution σ satisfies $spec_{ARS}$ if and only if it complies with the following two properties:*

Regularity: *Each read operation returns either the value written by the most recent write operation that happens before it or a value written by a concurrent write operation.*

No new/old Inversion: *If a read operation r returns a value written by a concurrent write operation w then no read operation that happens after r returns a value written by a write operation that happens before w.*

2.4 The ABD Simulation

This section aims to present in details the fault-tolerant single-writer multi-reader atomic register ABD simulation provided by Attiya, Bar-Noy, and Dolev [4]. Their assumptions on the distributed system follow. They assume a complete identified communication graph (*i.e.* each process has a distinct identifier) and an asynchronous distributed system subject to a minority of crash faults (that is, $2n > f$). Vertex v_0 (also denoted w in the sequel) is the writer (that is, it can invoke both the write and the read operation) while vertices from v_1 to v_{n-1} are readers (that is, they can invoke the read operation only).

In the following, we present only the bounded ABD simulation (the unbounded version makes use of natural numbers to label values of the register and can be easily derived from the bounded version). In this simulation, the authors assume the existence of a sequential bounded labeling system [18]. Israeli and Li defined in [18] time-stamps as "numerical labels which enable a system to keep track of temporal precedence relation among its data elements". Labels are elements of a set enhanced with a total antisymmetric binary relation (to compare labels) and a function to compute a new label given a set of existing labels.

The ABD simulation works as follows. First, they define a communication primitive, called **Communicate**, that ensures the communication by quorum. This primitive broadcasts a given message to all vertices and waits until getting an acknowledgment for a majority of them (it is always possible since at most $\frac{n}{2} - 1$ vertices may crash in any execution). Note that this communication primitive is designed to deal with the properties of the considered message passing model (non reliable and non FIFO communication links).

A label (from the sequential bounded labeling system) is associated to each value of the register. As the labeling system is bounded, the writer must take in

account all existing labels in the distributed system before computing a new one to ensure correctness. Indeed, the new label does not depend only of the writer label as in the unbounded version. Note that the set of gathered labels may be greater and contains obsolete labels.

To reach this goal, the **Write** operation operates as follow. The writer collects (via the primitive **Communicate**) the existing labels in the distributed system (readers send labels that they have for the writer and the most recent labels that they have sent to other vertices). The writer computes then a new label greater than each label it collected. The problem is that the primitive **Communicate** ensures only the collect from a majority of vertices. In consequence, any correct vertex must ensure that its labels are stored at a majority (at least) of vertices at any time. In this way, the writer is able to gather all existing labels when it collects labels from any majority.

To this end, whenever a vertex adopts a new label (that it believes to be the maximum label of the writer), it invokes a procedure **Record** that stores this label and all the recent labels it has sent to other vertices using the primitive **Communicate**. A vertex receiving a recording message simply stores all the labels in its memory. In response to a query from the writer, a reader sends all labels it has stored. This implies that no label may be lost (since a majority of vertices stores these labels). Note that, to avoid chain reaction where a recording message causes other recording messages, vertices ignore the labels carried by recording messages even if their label is greater than their current writer label.

However, when the environment faces both crashes and transient corruptions of the memory the ABD simulation fails to satisfy its specification. This fact is due to the building blocks that compose the ABD simulation: the communication primitive and the labeling scheme and also to the way the labels are included in the viable set. First, the primitive **Communicate** is not resilient to an arbitrary initial content of communication links. Second, the underlying labeling scheme used by the ABD simulation may be unable to compute a new label greater than the existing ones when started in an arbitrary configuration. Finally, the ABD simulation itself cannot deal with arbitrary initialization of labels since some initially corrupted labels may remain unknown to the writer and may be included infinitely often in the **Read** function decision sets.

The next section presents two recent achievements in the area of self-stabilization that allow us to bypass the problems related to the communication primitive and the labeling scheme. Section 4 extends the ABD simulation in order to manage also corrupted labels that have not been generated by the scheme itself but are present in the system due to some transient memory corruptions.

3 Necessary Tools

3.1 Data-Link Protocol

This section sums up the contributions of [10] in which we provided a data-link protocol that ensures optimal fault resiliency above bounded, non-reliable but

fair, non-FIFO communication channels. The main goal is to provide a communication protocol between two vertices that allows us to neglect the actual characteristics of the communication channel. The specification we provide in this paper is borrowed from [22] but we adapt it to the stabilizing context. In particular, we introduce the idea to bound the number of lost, duplicated, ghost and re-ordered messages by some constants.

Consider a system of two vertices v_i and v_j. A distributed application needs to send some messages from v_i to v_j. We say that the application layer of v_i sends a message when it requests the communication protocol to carry this message to v_j. This message is delivered to v_j when the communication protocol releases this message to the application layer of v_j. A ghost message is a message delivered to v_j whereas v_i did not send it previously (due to the arbitrary content of communication channels in the initial configuration). A duplicated message is a message that is delivered several times to v_j whereas v_i sent it only once. A message is lost when v_i sends it but v_j never delivers it. A message m is reordered when it is delivered to v_j before a message m' whereas m has been sent after m' by v_i. Intuitively, the goal of a data-link protocol is to provide a communication black box that ensures there is no lost, duplicated, ghost, or reordered messages during any execution. In the sequel, we formally specify the data-link problem.

Specification 2 (Data-link communication). *For any non negative integers α, β, γ, and δ, the* ***$(\alpha, \beta, \gamma, \delta)$-Stabilizing Data-Link communication*** *over c-bounded channels satisfies the following properties starting from an arbitrary configuration (with v_i and v_j being respectively the sender and the receiver) for any execution σ:*

*- α-**Loss:** The first α messages sent by v_i (in the worst case) may be lost.*
*- β-**Duplication:** The first β messages delivered to v_j (in the worst case) may be duplicated ones.*
*- γ-**Creation:** The first γ messages delivered to v_j (in the worst case) may be ghost messages.*
*- δ-**Reordering:** The first δ messages delivered to v_j (in the worst case) may be reordered.*

In [10], we proved that it is impossible to perform a $(\alpha, \beta, \gamma, \delta)$-Stabilizing Data-Link communication with $\beta = 0$, $\gamma = 0$, or $\delta = 0$. We also provided a data-link protocol (called $\mathcal{SDL}$) that achieves this optimal fault-resiliency.

In the following of this paper, we reuse this data-link protocol that provides to each vertex several functions. For each neighbor v_j, a vertex v_i is supplied with two functions: $\mathcal{SDL}$**-Send**$_j(m)$ that allows v_i to send messages to v_j using $\mathcal{SDL}$ and **DeliverMessage**$_i(m)$ that allows v_i to receive messages sent by v_j using $\mathcal{SDL}$.

3.2 Bounded Labeling Scheme

To the best of our knowledge, any existing bounded labeling system including the scheme used in the ABD simulation ([18,9,16]) does not tolerate corrupted initial

Algorithm 1. $\mathcal{PSARS}$: FTPS single-writer multi-reader atomic register simulation (read operation for any vertex v_i, write operation for the writer $w = v_0$).

Variables:
L_i: a matrix $n \times n$ with the following constraints:
- For any $j \neq k$, the element $L_i[j,k]$ contains two fields: $L_i[j,k].sent$ and $L_i[j,k].ack$. The first field is the last label that v_j sent to v_k in the last **Read** operation of v_j known at v_i. The second field contains the last label known at v_i sent by v_j to v_k when v_j replied to the v_k label request.
- For any j, the element $L_i[j,j]$ has two fields. The field $L_i[j,j].value$ provides information on the last label of the writer known by v_j. The second field $L_i[j,j].conflict$ gives information on a label that conflicts with the current label of a vertex and that may be not known at the writer.
$label\_set_i$: a set of labels
Functions:
MaxLabel: returns the maximum label (according to $\prec$) of the label set supplied as parameter if it exists, $\perp$ otherwise
Next: returns a label greater than (according to $\prec$) any label of the set given as parameter
PickValue: returns an arbitrary element of any circuit (according to $\prec$) of the label set supplied as parameter if possible, $\perp$ otherwise

Read$_i$()

```
01: label_set_i := ReadQuorum_i(read)
02: if MaxLabel(label_set_i) ≠ ⊥ then
03:   if L_i[i,i].value ≺ MaxLabel(label_set_i)
04:     L_i[i,i].value := MaxLabel(label_set_i)
05:     L_i[i,i].conflict := ⊥
06:     WriteQuorumPromote_i()
07:   WriteQuorumRecord_i()
08:   return L_i[i,i].value
09: else
10:   L_i[i,i].conflict := PickValue(label_set_i)
11:   WriteQuorumRecord_i()
12:   return abort
```

Write$_0$()

```
01: label_set_0 := ReadQuorum_0(write)
02: L_0(0,0).value := Next(label_set_0)
03: WriteQuorumPromote_0()
```

configurations. We defined and provided in [2] for the first time a stabilizing bounded labeling system: for any subset of at most k labels, there exists a label that dominates each label of the subset. In this way, we are ensured that a stabilizing bounded labeling system can deal with any arbitrary initialization since it is always possible to compute a label greater than the existing ones. We can define formally a stabilizing bounded labeling system in the following way:

Definition 2 (Stabilizing bounded labeling system). *A k-stabilizing bounded labeling system ($k \geq 2$) is a triplet $(L, \prec, next)$ where L is a finite set, $\prec$ is a total antisymmetric binary relation over L and $next$ is a function $next : L^k \to L$ such that:*

$$\forall L' \subseteq L, |L'| \leq k \Rightarrow \forall \ell \in L', \ell \prec next(L')$$

4 Our FTPS Simulation

This section proposes our extension to the ABD simulation that can tolerate, in addition to permanent crash faults, any transient memory corruption. We present a fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation over the message passing model. As far as we know, it is the first

time when a simulation with such strong guarantees is designed. Note that our previous work, [2], proposed a simulation that satisfies a weaker property than the pseudo-stabilization. That is, in each infinite run of system the atomicity specification is violated infinitely often. The significant amelioration of our current simulation stems from guaranteeing that each infinite run of the system has an infinite suffix where the atomicity specification is satisfied. First, we describe our distributed protocol in Section 4.1. We prove its correctness and provide its space complexity in Section 4.2.

4.1 Distributed Protocol

As we previously claimed, our distributed protocol is the pseudo-stabilizing version of the ABD simulation presented in details in Section 2.4. In this section, we explain first the differences between our simulation and the ABD simulation. Then, we present formally our distributed protocol. Note that, for the sake of simplicity, we ignore the actual value of the register and we concentrate only on the label associated to it (as in [4]).

Recall that we assume an asynchronous distributed system simultaneously subject to transient and (permanent) crash faults (with a maximal number of crashed vertices f such that $2n > f$). The communication graph is complete and identified. One vertex is distinguished to be the writer. We denote this vertex by $w = v_0$. Vertices from v_1 to v_{n-1} are readers. We also assume that any pair of vertices are able to communicate using the data-link protocol defined in Section 2. More precisely, if a vertex v_i has a message m to send to v_j, it invokes $\mathcal{SDL}$-**Send**$_j(m)$. The data-link protocol delivers this message to v_j by invoking **DeliverMessage**$_i(m)$. Finally, we assume the existence of a stabilizing bounded labeling system as the one described in Section 2. This labeling system provides a set of labels L and two functions. The first one, **Next**, computes a label greater than (according to $\prec$) any label of the set given as parameter. The second one, **MaxLabel**, returns the maximum label (according to $\prec$) of the label set supplied as parameter if this maximum exists, $\perp$ otherwise. Note that **MaxLabel** returns $\perp$ when there exists a circuit in the set of labels supplied as parameter (that is, there exists a subset of labels $\ell_0, \ldots, \ell_t$ such that $\ell_0 \prec \ell_1 \prec \ldots \prec \ell_t \prec \ell_0$).

Our distributed protocol makes use of a similar data structure as the ABD simulation. Each vertex v_i stores an $n \times n$ label matrix L_i. For any $j \neq k$, the element $L_i[j,k]$ contains the same fields as in the ABD simulation: $L_i[j,k].sent$ and $L_i[j,k].ack$. The ith row $L_i[i]$ is updated dynamically by v_i according to messages it sends while other rows $L_i[j]$ ($j \neq i$) are updated by messages that v_i received from v_j (that is, $L_i[j]$ is the latest view of v_i on $L_j[j]$). Each element $L_i[i,j]$ (for $j \neq i$), contains two fields: $L_i[i,j].sent$ and $L_i[i,j].ack$ that store respectively the last label that v_i sent to v_j and the last label acknowledged by v_j to v_i.

The only difference with the ABD simulation matrix is that, for any j, the element $L_i[j,j]$ contains now two fields: $L_i[j,j].value$ and $L_i[j,j].conflict$. The field $L_i[j,j].value$ provides the last label of the writer known by v_j. In particular $L_i[i,i].value$ contains the last label of the writer that the v_i is aware. Note that

this field is equivalent to the field $L_i[j,j]$ of the ABD simulation. The second field $L_i[j,j].conflict$ gives information on a label that conflicts with the current label of a vertex and that may be not known at the writer. This field is used to avoid that some initially corrupted label remains unknown to the writer but is included infinitely often in **Read** function decision set.

Our distributed protocol is composed of two primitives: **Read** (for any vertex) and **Write** (only for the writer v_0). When a reader v_i invokes its **Read** primitive, it collects first the labels of at least a majority of vertices and computes the maximum with **MaxLabel**. Two cases can appear:

1) MaxLabel returns a label. This value (if it exceeds the current label of the reader) is recorded in the distributed system in order to refresh the views of the other vertices on the last label of v_i. Note that, after the reception of this new value, a vertex updates the corresponding entry in its matrix. Vertex v_i finishes its **Read** operation by promoting its value in the distributed system. Upon the reception of the value to be promoted, the vertex v_j compares its current label with the label of the received value. If its local value is obsolete (the local label is less than the received label), then v_j adopts the new value and pushes it in the distributed system.

2) MaxLabel returns bottom whenever the maximum cannot be computed (when the set of collected labels contains a circuit). Then, the **Read** operation aborts. The circuit in the label set may have been introduced either by a corrupted label present in the system at the initialization or by the writer that computed the next label based on partial information from the non stabilized system. Then, the reader changes its $L_i[i,i].conflict$ field to one of the labels that form a circuit. The idea is to help in revealing all the corrupted labels. Indeed, the conflicting value is then recorded in the matrices of a majority of vertices that prevents such conflicting values to disturb further **Read** operations. This case is the main difference with the ABD simulation.

The **Write** operation is similar to the one of the ABD simulation. When the writer invokes this primitive, it first collects the latest labels in the system (by asking any majority of vertices), then computes its next label using the **Next** function. Finally it starts a promotion of the new value in the distributed system.

Algorithms 1 and 2 provide the formal implementation of our fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation.

4.2 Proof of Correctness

This section is devoted to the proof of the fault-tolerant pseudo-stabilization of $\mathcal{PSARS}$ for $spec_{ARS}$. According to properties of our data-link protocol described in Section 2, we know that any execution has an infinite suffix in which no ghost, duplicated or re-ordered messages are delivered (since there is only a finite number of communication links in the distributed system). We can conclude that any execution has an infinite suffix in which any delivered message was actually sent. For the sake of simplicity, we consider only such suffixes of executions in the sequel of this proof. Note that this assumption does not restrict the generality of the proof since we want to prove the pseudo-stabilization of our

Algorithm 2. $\mathcal{PSARS}$: Auxiliary functions (for any vertex v_i).

Notations:
For any j, the notation $L_i[j]$ represents the jth row of the matrix L_i.
Variables:
$return\_set_i$: a set of labels
$read\_answer_i$: array of n booleans
$record\_answer_i$: array of n booleans
$promote\_answer_i$: array of n booleans

ReadQuorum$_i(type)$

01: $read\_answer_i := [0, 0, \ldots, 0]$
02: $read\_answer_i[i] := 1$
03: $return\_set_i := \emptyset$
04: foreach $j \in \{0, \ldots, n-1\} \setminus \{i\}$ do
05: $\quad \mathcal{SDL}$-**Send**$_j(Inquiry(type))$
06: while $|\{j, read\_answer_i[j] = 1\}| \leq n/2$ do
07: $\quad$ wait
08: return $(return\_set_i)$

upon **DeliverMessage**$_j(Inquiry(type))$
09: if $type =' read'$ then
10: $\quad \mathcal{SDL}$-**Send**$_j(Answer\_Read(L_i[i, i]))$
11: $\quad L_i[i, j].ack := L_i[i, i].value$
12: $\quad$ **WriteQuorumRecord**$_i()$
13: else
14: $\quad \mathcal{SDL}$-**Send**$_j(Answer\_Write(L_i))$

upon **DeliverMessage**$_j(Answer\_Read(L_j[j, j]))$
15: $L_i[j, j] := L_j[j, j]$
16: $read\_answer_i[i] := 1$
17: $return\_set_i := return\_set_i \cup L_i$

upon **DeliverMessage**$_j(Answer\_Write(L_j))$
18: $L_i[j] := L_j[j]$
19: $read\_answer_i[i] := 1$
20: $return\_set_i := return\_set_i \cup L_i \cup L_j$

WriteQuorumPromote$_i()$

01: $promote\_answer_i := [0, 0, \ldots, 0]$
02: $promote\_answer_i[i] := 1$
03: foreach $j \in \{0, \ldots, n-1\} \setminus \{i\}$ do
04: $\quad \mathcal{SDL}$-**Send**$_j(Promote(L_i[i, i]))$
05: while $|\{j, promote\_answer_i[j] = 1\}| \leq n/2$
06: $\quad$ wait
07: foreach $promote\_answer_i[j] \neq 0$ do
08: $\quad L_i[i, j].sent := L_i[i, i].value$

upon **DeliverMessage**$_j(Promote(L_j[j, j]))$
10: if $L_i[i, i].value \prec L_j[j, j].value$ then
11: $\quad L_i[i, i] := L_j[j, j]$
12: $\quad$ **WriteQuorumRecord**$_i()$
13: $\quad \mathcal{SDL}$-**Send**$_j(Ack\_Promote())$

upon **DeliverMessage**$_j(Ack\_Promote())$
14: $promote\_answer_i[j] := 1$

WriteQuorumRecord$_i()$

01: $record\_answer_i := [0, 0, \ldots, 0]$
02: $record\_answer_i[i] := 1$
03: foreach $j \in \{0, \ldots, n-1\} \setminus \{i\}$ do
04: $\quad \mathcal{SDL}$-**Send**$_j(Record(L_i[i]))$
05: while $|\{j, record\_answer_i[j] = 1\}| \leq n/2$
06: $\quad$ wait

upon **DeliverMessage**$_j(Record(L_j[j]))$
07: $L_i[j] := L_j[j]$
08: $\mathcal{SDL}$-**Send**$_j(Ack\_Record())$

upon **DeliverMessage**$_j(Ack\_Record())$
09: $record\_answer_i[j] := 1$

distributed protocol (that is, only the existence of an infinite suffix satisfying the specification, not the finiteness of a prefix that does not satisfy the specification).

The main difficulty in proving our atomic register simulation comes from the presence of corrupted labels (due to the arbitrary initialization of matrices) in the distributed system that may disturb the good functioning of the distributed protocol.

The key idea of our proof is to show that the writer includes in its decision set (records) all the viable labels in the system (defined below). A label ℓ is *viable* and in the responsibility of vertex v_i if it satisfies one of the following properties:

- $L_i[i, i].value = \ell$ or $L_i[i, i].conflict = \ell$
- $L_i[i, k].sent = \ell$ or $L_i[i, k].ack = \ell$

- there is a vertex v_j such that $L_j[i]$ contains ℓ in one of the fields *sent*, *ack*, *value* or $conflict$.

A viable label is *recorded* if this label is stored in the writer matrix or the matrix of any majority of vertices. In the following, we show that any label in the responsibility of a vertex eventually becomes recorded. Note that once a label is stored in the matrix of the writer or in the matrix of a majority of vertices, this label is included in the computation of the new label of the writer and it does not generate new conflicts.

This observation motivates the following necessary assumption for the fault-tolerant pseudo-stabilization of $\mathcal{PSARS}$: if the writer crashes in an execution, then this crash must happen after the first stabilized **Write** invocation (that is, a **Write** invocation during which the label set supplied to **Next** includes all the viable labels in the distributed system). In other words, an execution has an infinite suffix that satisfies $spec_{ARS}$ if the writer does not crash during this execution or if the writer crashes after the first stabilized **Write** invocation (we cannot provide any properties in the contrary case). In the sequel of this section, we consider only such executions. Otherwise, corrupted labels may generate incoherent read outputs. Note that when started in a correct state this assumption is not necessary and the behavior of our simulation is exactly the same as the ABD's simulation. Also note that the ABD simulation cannot cope with corrupted labels.

Lemma 1. *Any execution of $\mathcal{PSARS}$ has an infinite suffix where every **Read** invocation does not abort if $n > 2f$.*

Lemma 2. *Any execution of $\mathcal{PSARS}$ has an infinite suffix where, for any vertex, the labels in its responsibility become recorded either at the writer or in a majority, or are never included in the label set of a read operation if $n > 2f$.*

From now, a viable label refers only to labels that do not stay forever out of the computation.

Lemma 3. *Any execution of $\mathcal{PSARS}$ has an infinite suffix that satisfies the regularity property of $spec_{ARS}$ if $n > 2f$.*

Proof. Let σ be an infinite execution of $\mathcal{PSARS}$. Following Lemma 1 and Lemma 2, σ contains an infinite suffix, σ', where no **Read** invocation aborts and any **Write** operation includes in its decision set all the viable labels in the distributed system. By contradiction, assume there is a vertex v_i such that its **Read** invocations return an obsolete label infinitely often in σ'.

That is, there exists a **Read** invocation r by v_i such that the label returned by r is either a corrupted label or a label corresponding to a previous write but not the most recent. In σ', r returns the output value of **MaxLabel** invoked over the set of labels returned by **ReadQuorum**.

Let w_1 and w_2 be two **Write** operations such that w_1 happens before w_2 and r. Since w_1 happens before r then the label computed by w_1 is promoted and recorded in at least a majority of vertices and is greater than any label in

the distributed system. When r starts invoking **ReadQuorum** two cases may appear: (i) w_2 did not modify the writer label and did not start the promotion of the new label via **WriteQuorumPromote** or (ii) w_2 executed **WriteQuorumPromote**. In the first case, w_1's label is the largest label in the distributed system. When r invokes the **ReadQuorum**, it gets w_1's label (otherwise w_1 is not terminated) and returns this label. Hence, r cannot return a value older than the one written by w_1. In the second case, some vertices contacted during the **ReadQuorum** execution may send the w_1's label, other vertices the w_2's label. Since the label computed in w_2 is greater than the label computed in w_1, **MaxLabel** invoked in r returns w_2's label. Hence, r returns the last written value, that contradicts its construction.

Lemma 4. *Any execution of $\mathcal{PSARS}$ has an infinite suffix that satisfies the no new/old inversion property of $spec_{ARS}$ if $n > 2f$.*

Proof. Let σ be an execution of $\mathcal{PSARS}$. Following Lemmas 1 and 3, σ has an infinite suffix, σ', that satisfies the regularity property of $spec_{ARS}$ and in which any **Read** invocation does not abort. In the following, we prove that σ' does not violate the new/old inversion property of $spec_{ARS}$.

Consider two **Write** operations w_1 and w_2 in σ' such that w_1 happens before w_2. Consider also two **Read** operations r_1 and r_2 such that r_1 happens before r_2 and w_1 happens before r_1 (following the transitivity of the relation "happens before", w_1 also happens before r_2). Assume that r_1 and r_2 are concurrent with w_2 and that a new/old inversion happens. That is, r_1 returns the label ℓ_2 written by w_2 and r_2 returns the label ℓ_1 written by w_1.

Since r_1 happens before r_2, then r_1 executes the following actions (before the start of r_2): it modifies its local label to ℓ_2, it also executes **WriteQuorumPromote** in order to help w_2 to push its label in the distributed system and finally it executes **WriteQuorumRecord** in order to inform the distributed system on its new value. Since **WriteQuorumPromote** returns before r_1 finishes, then the label ℓ_2 is already adopted by at least a majority of vertices. That is, since $\ell_2 \succ \ell_1$ (w_1 happens before w_2), then ℓ_2 replaces ℓ_1 in the matrices of at least a majority of vertices and also a majority of vertices proceeds to the record of their new label.

We assumed r_2 returns ℓ_1. Since r_1 happens before r_2 then r_2 starts its **ReadQuorum** after r_1 returned, in particular after r_1 completed its **WriteQuorumPromote** operation. This implies that ℓ_2 is the label adopted by at least a majority of vertices and at least one vertex in this majority responds while r_2 invokes its **ReadQuorum**. That is, r_2 collects at least one label ℓ_2 and since $\ell_2 \succ \ell_1$, r_2 should return this value. This contradicts the assumption r_2 returns ℓ_1. It follows that σ' satisfies the no new/old inversion property of $spec_{ARS}$.

Lemma 5. *$\mathcal{PSARS}$ requires $O(n^5 \times log_2(n))$ bits per vertex. Consequently, the total amount of memory on the distributed system is in $O(n^6 \times log_2(n))$ bits.*

Proof. Note that the set *label_set* which is the input of **Next** contains $2n^3$ labels. Hence, following [2], one label needs $O(n^3 \times log_2(n))$ bits to be stored. Since any vertex must store $2n^2$ labels, we have the result.

Theorem 1. *$\mathcal{PSARS}$ is a f-ftps distributed protocol for $spec_{ARS}$ provided that $n > 2f$ and that the writer can crash only after its first stabilized **Write** invocation. It requires $O(n^6 log_2(n))$ bits of memory on the whole distributed system.*

5 Conclusion

We presented a distributed solution for implementing a shared register in a network where processors communicate by exchanging messages. To our knowledge, this is the first such construction to be both pseudo-stabilizing and fault tolerant. Note that our simulation verifies also the eventual linearizability specification [14,23]. Differently from the eventual linearizable simulations proposed so far our simulation tolerates initial corrupted memory. Also, we do not reorder operations nor maintain locally the history of the system execution.

We expect future research to tackle the following open issues. A generalization to the multi-writer (and multi-reader) case looks challenging. Indeed, previous transformers for the crash fault model do handle memory corruption, and the multiplicity of writers enable the possibility that fake writers (*i.e.* stale writer identifiers) are initially present in the network.

References

1. Abraham, U.: Self-stabilizing timestamps. Theoretical Computer Science 308(1-3), 449–515 (2003)
2. Alon, N., Attiya, H., Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Pragmatic Self-stabilization of Atomic Memory in Message-Passing Systems. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 19–31. Springer, Heidelberg (2011)
3. Anagnostou, E., Hadzilacos, V.: Tolerating Transient and Permanent Failures (Extended Abstract). In: Schiper, A. (ed.) WDAG 1993. LNCS, vol. 725, pp. 174–188. Springer, Heidelberg (1993)
4. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. Journal of the ACM 42(1), 124–142 (1995)
5. Beauquier, J., Kekkonen-Moneta, S.: Fault-tolerance and self stabilization: impossibility results and solutions using self-stabilizing failure detectors. IJSS 28(11), 1177–1187 (1997)
6. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. DC 7(1), 35–42 (1993)
7. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Stabilizing leader election in partial synchronous systems with crash failures. JPDC 70(1), 45–58 (2010)
8. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. CACM 17(11), 643–644 (1974)
9. Dolev, D., Shavit, N.: Bounded concurrent time-stamping. SIAM J. on Comp. 26(2), 418–455 (1997)
10. Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Stabilizing data-link over non-fifo channels with optimal fault-resilience. IPL 111(18), 912–920 (2011)
11. Dolev, S., Herman, T.: Dijkstra's Self-Stabilizing Algorithm in Unsupportive Environments. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 67–81. Springer, Heidelberg (2001)

12. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE TPDS 8(4), 424–440 (1997)
13. Dolev, S., Kat, R.I., Schiller, E.M.: When consensus meets self-stabilization. JCSC 76(8), 884–900 (2010)
14. Fekete, A., Gupta, D., Luchangco, V., Lynch, N., Shvartsman, A.: Eventually-serializable data services. TCS 220(1), 113–156 (1999)
15. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of ACM 32(2), 374–382 (1985)
16. Gawlick, R., Lynch, N., Shavit, N.: Concurrent Timestamping Made Simple. In: Dolev, D., Rodeh, M., Galil, Z. (eds.) ISTCS 1992. LNCS, vol. 601, pp. 171–183. Springer, Heidelberg (1992)
17. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance (preliminary version). In: PODC 1993, pp. 195–206 (1993)
18. Israeli, A., Li, M.: Bounded time-stamps. DC 6(4), 205–209 (1993)
19. Johnen, C., Higham, L.: Fault-Tolerant Implementations of Regular Registers by Safe Registers with Applications to Networks. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 337–348. Springer, Heidelberg (2008)
20. Lamport, L.: On interprocess communication. Part i: Basic formalism. DC 1(2), 77–85 (1986)
21. Lamport, L.: On interprocess communication. Part ii: Algorithms. DC 1(2), 86–101 (1986)
22. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)
23. Serafini, M., Dobre, D., Majuntke, M., Bokor, P., Suri, N.: Eventually linearizable shared objects. In: PODC 2010, pp. 95–104 (2010)

Directed Graph Exploration

Klaus-Tycho Förster and Roger Wattenhofer

Computer Engineering and Networks Laboratory,
ETH Zurich, 8092 Zurich, Switzerland
{k-t.foerster,wattenhofer}@tik.ee.ethz.ch

Abstract. We study the problem of exploring all nodes of an unknown directed graph. A searcher has to construct a tour that visits all nodes, but only has information about the parts of the graph it already visited. The goal is to minimize the cost of such a tour. In this paper, we present upper and lower bounds for both the deterministic and the randomized online version of exploring all nodes of directed graphs. Our bounds are sharp or sharp up to a small constant, depending on the specific model. Essentially, exploring a directed graph has a multiplicative overhead linear in the number of nodes. If one wants to search for just a node in unweighted directed graphs, a greedy algorithm with quadratic multiplicative overhead can only be improved by a factor of at most two. We were also able to show that randomly choosing a starting point does not improve lower bounds beyond a small constant factor.

Keywords: online algorithms, graph exploration, mobile agents and autonomous robots.

1 Introduction

The hotel concierge promised that this tourist attraction is easy to find, just a short drive in your car, and she was right. However, how do you now get back to your hotel, in this cursed city full of one-way streets? After finally being back at your hotel, totally exhausted, you have a hunch that one-way streets render navigation more difficult, but is it true?!

In this paper we quantitatively analyze navigation problems in unknown directed graphs from a worst-case perspective. We present a whole flurry of tight upper and lower bounds, showing that directed graphs exhibit a penalty in the order of the number of nodes of the graph.

Navigation problems in directed graphs are not restricted to the playful introductory example of one-way streets. Staying in the car context, if we are for instance interested in minimizing gasoline cost, any hill-side city becomes directed, as driving downhill is virtually free, whereas driving uphill may incur a high cost. As such, when applying a cost measure, edges of a graph must often be represented by two directed edges with an appropriate cost.

The most important applications for investigating navigation in directed graphs are however beyond street networks. In computer networks, for instance,

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 151–165, 2012.

directed graphs have for instance been studied in the context data aggregation [29], routing [33], or traversing social networks [34]. Brass et. al. [7] compared the exploration of directed graphs to exploring the state space of a finite automaton, where the states are nodes and the transitions are edges. Deng and Papadimitriou [15] proposed the exploration of directed graphs as a model for learning, for example for a newborn: current states can be detected by sensor information (like eyes or ears) and possible actions leading to other states are known, but it is not known what the situation will be at a not yet explored state. And last not least, exploring an unknown graph is considered one of the fundamental problems in robotics [11,25]. Because of all these applications, directed graph exploration will be the main focus in this paper. In addition, we look at other navigation problems, such as searching for a node, which turn out to be related to exploration.

1.1 Model

We only consider the common model of strongly connected directed graphs [1,14,15,25,30], since a searcher else might get stuck right away (Section 8). We call a graph explored, if a searcher starting from some node s has visited all nodes and returned to s. The cost of such an online exploration tour is measured by the total sum of the weight of the traversed edges. It is allowed (and might be necessary) to visit nodes multiple times, but if we traverse an edge again it costs the same as for the first time. The competitive ratio of a tour is measured by the ratio of the cost of the tour divided by the cost of a tour of minimum cost. The competitive ratio of an algorithm is measured by the largest competitive ratio of all tours generated over all input graphs. For randomized algorithms, it is the largest expected competitive ratio.

For ease of notation, in the remainder of our paper a graph $G = (V, E)$ has $|V| = n \geq 6$ nodes and $|E| = m$ edges. All nodes have unique IDs, and all edges have non-negative weights. A searcher has unlimited computational power and memory and may only traverse edges from tail to head. Upon arriving at a node v, the following information is made available: all outgoing incident edges including their weight, plus the IDs (cf. [27,31]) of the corresponding nodes at the head of these edges. Graph exploration is an online problem since only partial information about the graph is available [9,10]. For other exploration models, e.g. unique edge names or information about incoming edges, we refer to Section 8.

1.2 Results

In our paper we give the first matching lower and upper bounds for the competitive exploration of an unknown directed graph. Our results are sharp for both the weighted and the unweighted case. For randomized exploration, our results only have a gap of less than four. We prove similar results for various commonly used graph classes, like planar or complete graphs or bounding different parameters like degree or diameter. We also discuss changes in the model,

like randomly choosing a starting position or more powerful searchers. We are able to show that in all these cases, the exploration of unknown directed graphs has a multiplicative overhead of $\Theta(n)$.

In a similar fashion, searching for a single node has $\Theta(n^2)$ overhead if all edges have unit weight (see Section 6). Furthermore, we look at the impact of randomly choosing a starting point. It turns out that even the best possible starting node can decrease any lower bound only by a factor of at most four.

To the best of our knowledge, sharp results regarding deterministic and randomized exploration of directed graphs have not yet been published. We summarize our main results in Table 1.

Table 1. Short overview of our main results: In the weighted general case we only need to use two different edge weights to achieve the bounds. A randomized starting node can only decrease our lower bounds by a factor of four.

competitivity / type of graph	lower bound	upper bound	multiplicative gap
(deterministic) general*[c]	$n-1$	$n-1$	sharp
(randomized) general*[+c]	$\frac{n}{4}$	$n-1$	≤ 4
(determ.) unweighted general*	$\frac{n}{2}+\frac{1}{2}-\frac{1}{n}$	$\frac{n}{2}+\frac{1}{2}-\frac{1}{n}$	sharp
(random.) unweighted general*	$\frac{n}{8}+\frac{3}{4}-\frac{1}{n}$	$\frac{n}{2}+\frac{1}{2}-\frac{1}{n}$	≤ 4
(deterministic) euclidean planar	$n-2-\bar{\epsilon}$	$n-1$	$\leq 1.25+\epsilon$
(randomized) euclidean planar	$\frac{n}{4}-\bar{\epsilon}$	$n-1$	$\leq 4+\epsilon$
(d.) unit weight euclidean planar	$\frac{n}{4}+\frac{1}{2}-\frac{2}{n}$	$\frac{n}{2}+\frac{1}{2}-\frac{1}{n}$	≤ 2
(r.) unit weight euclidean planar	$\frac{n}{8}+\frac{3}{4}-\frac{1}{n}$	$\frac{n}{2}+\frac{1}{2}-\frac{1}{n}$	≤ 4

* also applies to planar graphs and graphs that satisfy the triangle inequality

[c] also applies to complete graphs and graphs with any diameter from 1 to $n-1$

[+] also applies to graphs with any maximum incoming/outgoing degree from 2 to $n-1$ and to graphs with any minimum incoming/outgoing degree from 1 to $n-1$

2 Related Work

The offline variant, i.e. where all information about the graph is available to the algorithm, of directed graph exploration is the asymmetric travelling salesperson problem, where it is allowed to visit nodes multiple times. Unlike the undirected case, there is no known polynomial approximation algorithm with constant approximation ratio [3]. An approximation ratio of $O(\log n)$ was achieved in [22], the constant was improved over time, e.g. [8,28]; the best result known to us is $\frac{2}{3}\log_2 n$ [19]. There exists a result of $O(\log n/\log\log n)$ for the randomized

case [2]. If only the edge weights 1 and 2 are allowed, it is approximable with a ratio of 17/12 [37], with a NP-hard lower bound of $2805/2804 - \epsilon$ [18]. An online variant of asymmetric TSP is as follows: A searcher knows the graph, but the nodes to visit get determined during the runtime by an adversary [3].

More closely related to the online exploration of all nodes of directed graphs is the online exploration of all nodes of undirected graphs. While a greedy algorithm achieves a competitive ratio of $\Theta(\log n)$ [35], it is not known if a constant competitive ratio for general graphs is possible [31]. For cycles there is an algorithm with a sharp competitive ratio of $\frac{1+\sqrt{3}}{2}$, while for trees depth-first search is optimal [32]. Recently, the best known lower bound for general graphs was improved from $2 - \epsilon$ [32] to $5/2 - \epsilon$ [16]. For planar graphs a sophisticated variant of depth-first search named ShortCut by Kalyanasundaram and Pruhs achieves a competitive ratio of 16 [27]. Their result was recently extended for graphs of genus g to $16(1 + 2g)$ [31]. If there are just k different edge weights, there exists an algorithm with competitive ratio $2k$ [31]. Fleischer et. al. considered the problem of searching just for a node instead of a tour in [23]. They model their searcher as "blind", meaning that it can only sense the outgoing edges, but not any incoming edges or adjacent neighbors. They use the example of a modified clique to show a lower bound on the cost of $\Omega(n^2)$ for unit weights, since a blind searcher might visit nearly all edges.

Another related problem is the exploration of all edges of a strongly connected directed graph. Here the difficulty of the problem depends on another parameter, introduced by Kutten [30]: the eulerian deficiency d of a graph, which is the minimum amount of edges that need to be added to make the graph eulerian. A graph is eulerian, if there exists a path that visits all edges exactly once. If a graph is eulerian, then it can be traversed in an online fashion with at most $2m$ edge traversals [14], which directly implies at most $4m$ edge traversals in the undirected case, see for example [1]. For $d = 1$, a ratio of 4 is optimal [15]. An upper bound only dependent polynomially in d for the directed case was given by Fleischer and Trippen [25], their algorithm is $O(d^8)$-competitive. There exists also a lower bound of $\Omega(d)$-competitivity for the deterministic case and a lower bound of $\Omega(\frac{d}{\log d})$-competitivity for the randomized case [14,15]. Furthermore, graph exploration has also been considered with restricted memory models or multiple searchers, see for example [4,6,12,13,17,20,21].

There seems to be no known randomized algorithm for the exploration of graphs (wether it be just nodes or edges) that gives better bounds than the known deterministic algorithms. Experimental studies of randomized algorithms for exploring all edges and nodes of a strongly connected directed graph have been done in [24].

The similar sounding term graph searching, which was first discussed by Breisch and Parsons (cf. [5]), stands for another problem: A number of agents has to capture an intruder, or as formulated in the original papers, a party of searchers has to find a person lost in a cave. For an overview of other online navigation tasks we refer to [10].

3 Lower Bounds for General Graphs

We note that in this section we only use the weights 0 and 1 in the weighted case for lower bounds. If only integers of size at least one are allowed as edge weights, then analog results can be achieved by replacing 0 with 1 and 1 with $\lceil 1/\epsilon \rceil$ for arbitrarily small $\epsilon > 0$. Furthermore, the unique names of nodes in the remainder of the paper are just fixed for the convenience of the reader, an adversary can permute them in any way it desires – therefore an online algorithm can derive no further information from just the unique name of an unexplored node. Also the graphs used in the lower bounds are planar and satisfy the triangle inequality.

3.1 Deterministic Online Algorithms

Theorem 1. *No deterministic online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed weighted graphs than* $n-1$.

Proof. Consider the graph in Figure 1. A searcher using any deterministic online algorithm starting at node v_n cannot differentiate between the nodes v_1,v_2,..., v_{n-1}, they all look the same, since it can only see the outgoing edges from v_n and the nodes at the end of these edges. In the worst case, the searcher chooses to visit the node v_{n-1} first, then is forced to go back to v_n, then to visit v_{n-2} and so on, until it visits v_1 and then returns to v_n. The cost of this route is $n-1$, while an optimal tour first visits v_1 and then goes to v_n, inducing a total cost of just 1. This yields a competitive ratio of $n-1$ for any deterministic online algorithm. □

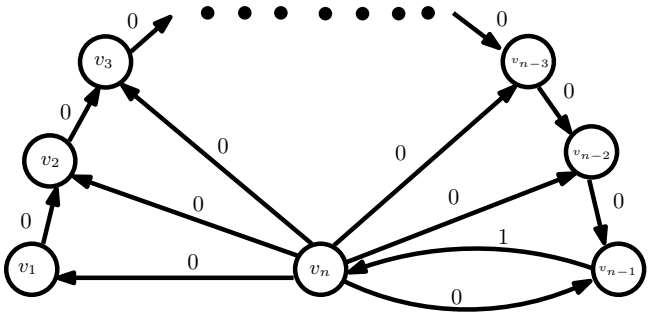


Fig. 1. In this graph the starting node s is v_n in the lower middle of the image. A deterministic algorithm can get tricked into first visiting v_{n-1}, then v_{n-2} and so on.

3.2 Randomized Online Algorithms

A randomized searcher can explore the graph in Figure 1 with much lower expected costs: In average it chooses a node in the "middle" of the so far yet unvisited nodes when being at v_n, therefore visiting the starting node only about $O(\ln(n))$-times. However, we can reach nearly the same lower bounds with the graph from Figure 2 as in the deterministic case:

Theorem 2. *No randomized online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed weighted graphs than* $\frac{n}{4}$.

Proof. Consider the graph in Figure 2 and let the number of nodes n be even. If one wants to consider odd n, then the same results can be achieved by removing the node $v_{\frac{n}{2}}$ and updating the graph accordingly. Let us assume a searcher using any randomized online algorithm starting from v_n visits a node v_i, with $1 \leq i \leq \frac{n}{2} - 2$, for the first time: then it cannot differentiate the two outgoing edges. An adversary can choose the IDs so that a good edge is picked with a probability of at most $p = 0.5$. Thus the decisions at the nodes v_1 to v_{i-1} do not yield any useful information about how to pick the outgoing edges at v_i. Therefore the expected amount of choosing a wrong outgoing edge is $0.5\left(\frac{n}{2} - 2\right)$. A wrongly chosen edge when visiting v_i for the first time induces a cost of 1, since the searcher has to follow the unique way back to v_i, traversing the edge from v_{n-1} to v_n with cost 1. This results in an expected cost of $0.5\left(\frac{n}{2} - 2\right) = \frac{n}{4} - 1$ to explore the node $v_{\frac{n}{2}-1}$. Once reaching the node $v_{\frac{n}{2}-1}$ for the first time, the searcher is forced to go back to v_n, resulting in another cost of 1. Since an optimal tour has a cost of 1, this yields the lower bound of $\frac{n}{4}$. □

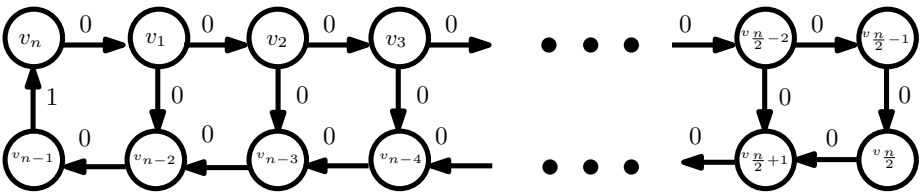


Fig. 2. In this graph the starting node s is v_n in the upper left corner. Upon arriving at each of the nodes $v_1, v_2, \ldots v_{\frac{n}{2}-2}$ for the first time, a randomized algorithm gets tricked into taking the wrong edge with probability at least 0.5. If n is odd, then the lower right node $v_{\frac{n}{2}}$ can be removed to achieve the lower bound.

3.3 Starting Node

While the examples of the graphs in the Figures 1 and 2 lead to a high lower bound for the competitive ratio, this is only true because the online algorithm is

forced to start at the node v_n. Starting at node v_1 in Figure 1 or at node $v_{\frac{n}{2}}$ in Figure 2 leads to a competitive ratio of 1. If the starting node were to be chosen randomly, the expected ratio is $O(\sqrt{n})$ for both cases. This raises the question if a random starting node can lead to a better competitive ratio. However this is not the case, there is still a lower bound of $\Omega(n)$:

Theorem 3. *Even if taking the best result from all possible n starting nodes, no deterministic online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed weighted graphs than $n/4$. The same holds for randomized online algorithms with a competitive ratio of $n/16$.*

Proof. We start with the deterministic case. We again take the graph from Figure 1, but draw it two times as G and G' with their respective starting nodes v_n and v'_n. We now connect these graphs by adding an edge from v_n to v'_n and back, both with weight 0 – resulting in a graph G'' with $2n$ nodes. Without loss of generality we can assume that a starting node from G' is chosen. No deterministic online algorithm can achieve a better worst-case cost on exploring G than $n-1$, since the old graph G can only be entered by the edge from v'_n to v_n. On the other hand, the graph G' can be explored with a cost of just 1. An optimal offline algorithm will just have a cost of 2 for exploring the whole graph, no matter what starting node is chosen. Since the graph has $2n$ nodes, this leads to a lower bound of $n/4$. We can apply the same arguments to the randomized case using the graph in Figure 2, giving a lower bound of $n/16$. □

4 Upper Bounds for General Graphs

In the undirected case, it is not known yet if there is an algorithm with a better competitive ratio than $O(\log n)$ [35]. A greedy approach reaches this competitive ratio of $O(\log n)$ [35], but the same algorithm has a competitivity of $\Omega(\log n)$ even on planar unweighted graphs [26]. Thanks to our strong lower bounds, a greedy algorithm has a sharp competitive ratio in the directed case:

Theorem 4. *A greedy algorithm achieves a competitive ratio of $n-1$ for exploring all nodes of strongly connected directed weighted graphs.*

Proof. Given any graph $G = (V, E)$, let us fix an optimal tour OPT. The tour OPT can be viewed as a concatenation of n paths, that visit the nodes of the graph in the following order: $s = v_0^o, v_1^o, v_2^o, \ldots, v_{n-1}^o, v_n^o = s$. We name the path from v_i^o to v_{i+1}^o as w_{i+1}^o with $0 \leq i \leq n-1$. The walk $W_{i,j}^o$ from v_i^o to v_j^o (with $v_i^o \neq v_j^o$) in OPT consists of the concatenation of $w_{i+1}^o, w_{i+2}^o, \ldots, w_j^o$ for $i < j$ or of $w_{i+1}^o, w_{i+2}^o, \ldots, w_n^o, w_1^o, \ldots, w_{j-1}^o, w_j^o$ for $i > j$. For each $W_{i,j}^o$ with $i \neq j$ it holds that $W_{i,j}^o$ is the concatenation of at most $(n-1)$ different paths w_r^o with $1 \leq r \leq n$. Let us assume the greedy algorithm proceeds as follows: upon reaching a node v_k^g for the first time, find a shortest path w_{k+1}^g from the current node to a unknown node v_{k+1}^g in the outgoing neighborhood of the so far explored nodes. This path w_{k+1}^g has at most the weight of the concatenated paths from

v_k^g to v_{k+1}^g in OPT. Let us assume it has heavier weight: then there is a cheaper path from v_k^g to v_{k+1}^g that also visits another not yet explored node v_q before visiting v_{k+1}^g. However by the choice of v_{k+1}^g, then v_q is the same node as v_{k+1}^g, which leads to a contradiction. If we sum this up for all n paths $w_1^g, \ldots, w_{n-1}^g$ from the greedy algorithm plus the shortest path w_n^g from v_{n-1}^g to $s = v_n^g$, a first simple upper bound is $n \cdot |OPT|$. However, each path w_r^o with $1 \leq r \leq n$ from OPT only gets used at most $(n-1)$ times in the upper bound. This leads to an upper bound of $(n-1) \cdot |OPT|$ on the cost of a tour produced by the greedy algorithm. □

A combination of Theorem 1, 2 and 4 yields the following corollary:

Corollary 1. *The result of Theorem 4 cannot be improved by any other deterministic online algorithm. For randomized online algorithms, only a improvement by a factor of at most 4 is possible.*

Furthermore, the authors of [16] also studied the problem of advice complexity for exploring undirected graphs. They showed that there is a family of graphs where a searcher needs to be given at least $\Omega(n \ln(n))$ bits of information (from an all-knowing outside source before starting) to explore the graphs with optimal cost. We note that a greedy algorithm in any directed or undirected graph can solve the graph exploration problem optimally with $O(n \ln(n))$ bits. We apply the arguments from above and give a list of the nodes from an optimal tour OPT in the order they first appear in OPT to the searcher. Since the ID of every node is of size $O(\ln(n))$ bits, the lower bound of $\Omega(n \ln(n))$ from [16] is a sharp bound of $\Theta(n \ln(n))$ bits for both directed and undirected graphs.

5 Unweighted Graphs

An unweighted graph is a graph where the edges have no edge weights, i.e. the cost is the same for all edges. For our purposes, this is the same as assigning the edge weight 1 to every edge. The lower bounds are lower, but we will see that the upper bounds also go down:

Theorem 5. *No online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed unweighted graphs than* $\frac{n}{2} + \frac{1}{2} - \frac{1}{n}$ *(deterministic) or* $\frac{n}{8} + \frac{3}{4} - \frac{1}{n}$ *(randomized) .*

Proof. Consider the graph in Figure 1 for the deterministic case and assign all edges a weight of 1. A deterministic online algorithm starting at v_n first visits v_{n-1}, then v_{n-2} etc. in the worst case. Exploring v_{n-1} and going back to v_n has a cost of 2, for v_{n-2} it is 3, ..., for v_1 it is n. Summed up this yields $2+3+\ldots+n = \frac{n^2}{2} + \frac{n}{2} - 1$. Since an optimal tour has cost n, this gives a lower bound for the competitive ratio of $\frac{n}{2} + \frac{1}{2} - \frac{1}{n}$.

Consider the graph in Figure 2 for the randomized case and assign all edges an edge weight of 1. Now we can apply the same argument as in the weighted case, but the induced cost by each wrong decision is not 1, but 4 for v_1, 6 for

v_2, ..., $n-2$ for $v_{\frac{n}{2}-2}$. Since the previous decisions are useless for the current decision, each of these wrong decisions happens with a probability of at least 0.5. Furthermore, independently of these decisions, the last exploration tour starting at v_n will visit all nodes exactly once in this example. This gives a lower cost bound of $0.5\left(\left(\frac{n}{2}-2\right)^2+3\left(\frac{n}{2}-2\right)\right)+n=\frac{n^2}{8}+\frac{3n}{4}-1$. An optimal tour has cost n, resulting in a lower bound for the competitive ratio of $\frac{n}{8}+\frac{3}{4}-\frac{1}{n}$. □

Theorem 6. *A greedy algorithm achieves a competitive ratio of $\frac{n}{2}+\frac{1}{2}-\frac{1}{n}$ for exploring all nodes of strongly connected directed unweighted graphs.*

Proof. We prove this upper bound by summing up the costs to reach the first newly explored node, the second newly explored node, ..., the $(n-1)$th (and last) newly explored node. Let us assume that, beside the starting node, we have explored $(k-2)$ additional nodes and have just reached the $(k-1)$th new node v_{k-1} for the first time. Since the graph is strongly connected, there is always at least one new node reachable from the current node in the neighborhood of the so far explored subgraph – unless every node has been visited already. If we pick the new node v_k as a unexplored one we can reach with as few edge-traversals as possible, then we induce a cost of at most k. A shortest path from v_{k-1} to v_k will by definition not include another unexplored node v_u, since then v_u had been chosen as v_k. Furthermore, the path will not include any node twice. This gives an upper bound of k for the length of the path from v_{k-1} to v_k. In order to get back to the starting node once all nodes are explored, a shortest path can again visit at most all other $n-2$ nodes before reaching the starting node, giving an upper bound of $n-1$ for this last path. If we sum this up we get an upper bound of $1+2+3+\ldots+(n-2)+(n-1)+(n-1)=-1+\sum_{i=1}^{n} i=\frac{n^2}{2}+\frac{n}{2}-1$. An optimal tour has cost at least n, giving a competitive ratio of at most $(\frac{n^2}{2}+\frac{n}{2}-1)/n=\frac{n}{2}+\frac{1}{2}-\frac{1}{n}$. □

Combining the results of Theorem 5 and Theorem 6 yields:

Corollary 2. *The result of Theorem 6 cannot be improved by any other deterministic online algorithm. For randomized online algorithms, only a improvement by a factor of at most 4 is possible.*

6 Searching a Node

Instead of generating a tour, one can also change the model, and find just one specific node v and then stop. However an adversary can place this node in such a way that it is found last. The searcher does not need to return to the start, but searching for a node is still costly:

Theorem 7. *Searching for a node in strongly connected directed weighted graphs has an arbitrarily large competitive ratio for any deterministic or randomized online algorithm and can induce arbitrarily large additive costs.*

Proof. We start with the deterministic case. In Figure 1, a node v_{n+1} can be added that is connected to v_1 with two edges of weight 0. Since an optimal algorithm finds this node with cost 0, any deterministic node search algorithm has an arbitrarily bad competitive ratio, since it induces positive costs. The same holds for randomized algorithms if the same construction is applied at node $v_{\frac{n}{2}-1}$ in Figure 2. We can apply the same thought for arbitrarily large additive costs by replacing the edge weight of 1 with an arbitrarily large value. □

If we consider the model of unit weight edges, then the situation changes:

Theorem 8. *Any online algorithm for searching a node in strongly connected directed unweighted graphs has a lower bound of* $\frac{(n-1)^2}{4} - \frac{(n-1)}{4} - \frac{1}{2}$ *(deterministic) or* $\frac{(n-1)}{4} + \frac{1}{2} + \frac{2}{(n-1)}$ *(randomized) for its competitive ratio.*

Proof. An optimal offline algorithm has a cost of 2 to find v_{n+1} in the modified graph from Figure 1 with unweighted edges (v_n to v_1 to v_{n+1}). Any deterministic online algorithm finds v_{n+1} last in the worst case, producing a cost of at least (see the proof of Theorem 5) $\frac{n^2}{2} + \frac{n}{2} - 1 - n$. The searcher does not have to go back to the start, so $(-n)$ is added at the end. Since this graph has $(n+1)$ nodes, a lower bound for the competitive ratio of any deterministic node search algorithm is $\frac{(n-1)^2}{4} - \frac{(n-1)}{4} - \frac{1}{2}$.

For the randomized case we use the modified graph from Figure 2 and search for the node $v_{\frac{n}{2}}$. An optimal algorithm finds $v_{\frac{n}{2}}$ after $\frac{n}{2}$ steps (v_n to v_1 ... to $v_{\frac{n}{2}-1}$ to v_{n+1}). Any randomized algorithm needs at least an expected cost of (see the proof of Theorem 5) $\frac{n^2}{8} - \frac{n}{4} - 1 + \frac{n}{2}$. This leads to a competitive ratio of $\left(\frac{n^2}{8} + \frac{n}{4} - 1\right) / \frac{n}{2} = \frac{n}{4} - \frac{2}{n} + \frac{1}{2}$. □

For an upper bound we can again use the greedy algorithm:

Theorem 9. *A greedy algorithm searching for a node in strongly connected directed unweighted graphs has a competitive ratio of* $\frac{n^2}{4} - \frac{n}{4}$.

Proof. A greedy algorithm finds the searched node last in the worst case with a cost of at most $\frac{n^2}{2} - \frac{n}{2}$ (see the proof of Theorem 6). If the node were to be directly reachable from the starting node, then an online algorithm can find it in one step. Therefore we can use 2 as the minimal cost needed for an offline algorithm when computing an upper bound for the competitive ratio. This leads to a competitive ratio of $\frac{n^2}{4} - \frac{n}{4}$. □

Combining Theorem 8 and Theorem 9 yields the following corollary:

Corollary 3. *Any deterministic online algorithm searching for a node in strongly connected directed unweighted graphs can improve the competitive ratio of a greedy algorithm by a factor of 3 at most.*

Proof. The quotient of the upper and lower bounds from Theorem 9 and 8 for $n \geq 4$ (for $n \leq 3$ any node search takes two steps at most) has a global maximum in the range $n \in [4, \infty)$ at $n = 4$ with value 3. □

Let us now come back to the situation mentioned at the start of our introduction. How expensive can going back to your hotel be? Essentially, it is the same as searching for a node – just that this node is the only one that has an outgoing edge to your hotel. For the deterministic case, we again use the graph from Figure 1 with unweighted edges. We add a hotel-node v_h and add a directed edge from v_h to v_n (the node with the tourist attraction) and one directed edge from v_1 to v_h. Going back to your hotel is now the same as searching the node v_1 with one additional step back. The same construction can be used for the randomized case with the graph from Figure 2 with unweighted edges. We add a hotel-node v_h and add an outgoing edge from v_h to v_n (again, the node with the tourist attraction) and one outgoing edge from $v_{\frac{n}{2}}$ to v_h.

7 Lower Bounds for Special Cases

When we add directed edges with arbitrarily high weights to a given graph, then using these edges in any online algorithm will not improve the weight of an obtained tour. An online algorithm has now more information about the graph (for example about the number of nodes), but we can add these edges in such a way to our lower bound graphs in Figure 1 and 2 that the searcher gains no useful information. For example, if we turn the graph from Figure 1 into a complete graph by adding all missing edges with arbitrarily high weights, then these new edges do not help a searcher on deciding what node to explore next when visiting v_n, since all possibilities look the same except for their ID – unless the searcher decides to use an expensive edge. Due to space constraints, we omit the proofs of the Theorems 10 and 11 in this section:

Theorem 10. *For graphs of any diameter from 1 to $n-1$ or complete graphs with eulerian deficiency of $d = 0$, no online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed weighted graphs than $n-1$ (deterministic) or $n/4$ (randomized).*

We can apply the same line of thought to the graph in Figure 2:

Theorem 11. *For graphs of any maximum (minimum) incoming/outgoing degree from 2(1) to $n-1$, no online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed weighted graphs than $n/4$.*

A graph is called euclidean, if its nodes can be embedded into the euclidean plane with the edge weights being equivalent to the length of the straight edge in the embedding [36].

Theorem 12. *No online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed weighted planar euclidean graphs than $n-2-\epsilon^*$ (deterministic) or $n/4-\epsilon^*$ (randomized) for any $\epsilon^* > 0$.*

Proof. We again consider the graph in Figure 2 for the randomized case. If we replace all edge weights with a fixed $\epsilon^r > 0$, then it can be embedded as a planar euclidean graph like shown in the figure. To reach the lower bound for

competitivity, we replace both edge weights of the incoming and the outgoing edge for v_{n-1} with $1/2$. Now let us consider a circle with radius $\frac{1}{2}$ through the nodes v_n and v_{n-2}, with the nodes v_1 and v_{n-3} not being inside the circle. If we place v_{n-1} in the center of the circle, we have a proper planar euclidean embedding of the constructed graph. By choosing ϵ^r to be small enough, for example $\epsilon^r < \epsilon^*/n^2$, we reach a lower bound of $n/4 - \epsilon^*$.

For the deterministic case we consider the graph in Figure 1. Let us fix a $\epsilon^d > 0$ and construct a cycle of radius ϵ^d with v_n being in the center of the cycle and placing the nodes v_1 to v_{n-1} with distance ϵ^d/n on the cycle. Like in the randomized case, we construct another circle of radius $\frac{1}{2}$ through the nodes v_n and v_{n-2}, with v_1 and v_{n-3} not being inside the circle. We now place v_{n-1} in the middle of that cycle, which means that the edge weights of both the incoming and the outgoing edges are $1/2$. We remove the edge from v_n to v_{v-1}, since it has no longer the same weight than the other outgoing edges from v_n. All other edge weights are now $\leq \epsilon^d$. Notice that a deterministic algorithm can now only be tricked $n-2$ times. If we choose $\epsilon^d \leq \epsilon^*/n^2$, we reach a lower bound of $n - 2 - \epsilon^*$. □

A similar result also holds if all edge weights have to be of unit weight:

Corollary 4. *No online algorithm can achieve a better competitive ratio on exploring all nodes of strongly connected directed unit weight planar euclidean graphs than* $\frac{n}{4} + \frac{1}{2} - \frac{2}{n}$ *(deterministic) or* $\frac{n}{8} + \frac{3}{4} - \frac{1}{n}$ *(randomized).*

Proof. We use the graph from Figure 2 (see Theorem 5) and set all edge weights to 1, which results in a unit weight euclidean planar graph. □

8 Other Exploration Models

Unique Edge Names: Our results also hold if the searcher cannot see the name of nodes at the end of incident outgoing edges, but just the unique name of both incoming and outgoing edges. When two nodes v_i and v_j are visited by the searcher, it knows the name of all incident edges for v_i and v_j, therefore also the subgraph that is spanned by v_i and v_j. If the searcher is at a node v_i and does not know where an incident outgoing edge ends, then the node at the end of that edge has not been explored yet. In other words, the searcher has visited all nodes if and only if it knows where each edge ends and starts. Since our greedy algorithms do not utilize node names when selecting the next node to be explored, but just try to get to a unexplored node as cheap as possible, our upper bounds still apply. This holds as well for our lower bound examples if we use this modified exploration model: every time we trick any online algorithm into making a wrong decision, we give a set of options to choose from that look exactly the same for the online searcher.

Incoming Edges: Let us assume that the searcher does not just see the names of the nodes at the end of incident outgoing edges, but also the names of the

nodes at the other end of incident incoming edges. Our upper bound still applies, since the algorithms can just choose to ignore that additional information. For the lower bound however, we can no longer use the graphs from Figure 1 and Figure 2. For example when starting on the graph in Figure 1, the node v_{n-1} now can be differentiated from the nodes $v_1, \ldots, v_{n-2}$. Also when visiting v_{n-2}, the searcher can now differentiate v_{n-3} from $v_1, \ldots, v_{n-4}$, since there is an edge from v_{n-3} to v_{n-2}. We can fix this problem by hiding this information with adding additional nodes. For the example in Figure 1, we add $n-1$ additional nodes. For $1 \leq i \leq n-1$, remove the edge from v_i to v_{i+1}, add a new node v_i^+ between them and add a edge from v_i to v_i^+ and from v_i^+ to v_{i+1}. The edge weights of the two new edges is one half of the edge weight of the removed edge. This decreases the lower bound by a factor of less than 2. We fix the graph in Figure 2 in a similar way. We add $\frac{n}{2} - 3$ nodes between the nodes $v_{\frac{n}{2}+1}$ to v_{n-2}. For $\frac{n}{2} + 1 \leq i \leq n-2$, remove the edge from v_i to v_{i+1} add a new node v_i^+ between them and add a edge from v_i to v_i^+ and from v_i^+ to v_{i+1}. The edge weights of the two new edges are one half of the edge weight of the removed edge. This decreases the lower bound by a factor of less than 1.5.

Connectivity: When exploring directed graphs (for both cases of just nodes or nodes and edges), usually only strongly connected variants are considered, see for example [1,14,15,25,30]. This ensures that every node is reachable from the starting node and that the searcher can return to the starting node from every node. If the directed graph is not strongly connected, then any deterministic online algorithm can already get stuck after visiting the first new node, even though an offline algorithm can visit every other node and just skip this one. Similar graphs can be constructed for the randomized case. Consider a directed cycle, where each node has an outgoing edge to the same node v – which has outgoing degree of 0. The starting point is only reached again by the searcher with a probability of $(0.5)^{n-2}$ for $n \geq 3$. Already for $n = 12$ this gives just a chance of < 0.001 to return to the start.

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments.

References

1. Albers, S., Henzinger, M.R.: Exploring Unknown Environments. SIAM J. Comput. 29(4), 1164–1188 (2000)
2. Asadpour, A., Goemans, M.X., Madry, A., Gharan, S.O., Saberi, A.: An O(log n/log log n)-approximation Algorithm for the Asymmetric Traveling Salesman Problem. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 379–389. Society for Industrial and Applied Mathematics, Philadelphia (2010)
3. Ausiello, G., Bonifaci, V., Laura, L.: The on-line asymmetric traveling salesman problem. J. Discrete Algorithms 6(2), 290–298 (2008)

4. Baldoni, R., Bonnet, F., Milani, A., Raynal, M.: Anonymous graph exploration without collision by mobile robots. Inf. Process. Lett. 109(2), 98–103 (2008)
5. Flocchini, P., Fraigniaud, P., Santoro, N.: Capture of an intruder by mobile agents. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002, pp. 200–209. ACM, New York (2002)
6. Bender, M.A., Fernandez, A., Ron, D., Sahai, A., Vadhan, S.P.: The Power of a Pebble: Exploring and Mapping Directed Graphs. Inf. Comput. 176(1), 1–21 (2002)
7. Brass, P., Gasparri, A., Cabrera-Mora, F., Xiao, J.: Multi-robot tree and graph exploration. In: Proceedings of the 2009 IEEE International Conference on Robotics and Automation, ICRA 2009, pp. 495–500. IEEE Press, Piscataway (2009)
8. Bläser, M.: A new approximation algorithm for the asymmetric TSP with triangle inequality. ACM Transactions on Algorithms 4(4), 47:1–47:15 (2008)
9. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press, Cambridge (1998)
10. Bermann, P.: On-line Searching and Navigation. In: Fiat, A., Woeginger, G.J. (eds.) Online Algorithms 1996. LNCS, vol. 1442, pp. 232–241. Springer, Heidelberg (1998)
11. Burgard, W., Moors, M., Fox, D., Simmons, R.G., Thrun, S.: Collaborative Multi-Robot Exploration. In: Proceedings of the 2000 IEEE International Conference on Robotics and Automation, ICRA 2000, pp. 476–481. IEEE, San Francisco (2000)
12. Chalopin, J., Flocchini, P., Mans, B., Santoro, N.: Network Exploration by Silent and Oblivious Robots. In: Thilikos, D.M. (ed.) WG 2010. LNCS, vol. 6410, pp. 208–219. Springer, Heidelberg (2010)
13. Das, S., Flocchini, P., Kutten, S., Nayak, A., Santoro, N.: Map construction of unknown graphs by multiple agents. Theor. Comput. Sci. 385(1-3), 34–48 (2007)
14. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph (Extended Abstract). In: Proceedings of the 31st Annual Symposium on Foundations of Computer Science, FOCS 1990, vol. I, pp. 355–361. IEEE Computer Society, St. Louis (1990)
15. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. J. Graph Theory 32(3), 265–297 (1999)
16. Dobrev, S., Královič, R., Markou, E.: Online Graph Exploration with Advice. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 267–278. Springer, Heidelberg (2012)
17. Dynia, M., Łopuszański, J., Schindelhauer, C.: Why Robots Need Maps. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 41–50. Springer, Heidelberg (2007)
18. Engebretsen, L.: An Explicit Lower Bound for TSP with Distances One and Two. Algorithmica 35(4), 301–318 (2003)
19. Feige, U., Singh, M.: Improved Approximation Ratios for Traveling Salesperson Tours and Paths in Directed Graphs. In: Charikar, M., Jansen, K., Reingold, O., Rolim, J.D.P. (eds.) APPROX and RANDOM 2007. LNCS, vol. 4627, pp. 104–118. Springer, Heidelberg (2007)
20. Fraigniaud, P., Ilcinkas, D.: Digraphs Exploration with Little Memory. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 246–257. Springer, Heidelberg (2004)
21. Fraigniaud, P., Gasieniec, L., Kowalski, D.R., Pelc, A.: Collective tree exploration. Networks 48(3), 166–177 (2006)
22. Frieze, A.M., Galbiati, G., Maffioli, F.: On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. Networks 12(1), 23–39 (1982)

23. Fleischer, R., Kamphans, T., Klein, R., Langetepe, E., Trippen, G.: Competitive Online Approximation of the Optimal Search Ratio. SIAM J. Comput. 38(3), 881–898 (2008)
24. Fleischer, R., Trippen, G.: Experimental Studies of Graph Traversal Algorithms. In: Jansen, K., Margraf, M., Mastrolli, M., Rolim, J.D.P. (eds.) WEA 2003. LNCS, vol. 2647, pp. 120–133. Springer, Heidelberg (2003)
25. Fleischer, R., Trippen, G.: Exploring an Unknown Graph Efficiently. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 11–22. Springer, Heidelberg (2005)
26. Hurkens, C.A.J., Woeginger, G.J.: On the nearest neighbor rule for the traveling salesman problem. Oper. Res. Lett. 32(1), 1–4 (2004)
27. Kalyanasundaram, B., Pruhs, K.: Constructing Competitive Tours from Local Information. Theor. Comput. Sci. 130(1), 125–138 (1994)
28. Kaplan, H., Lewenstein, M., Shafrir, N., Sviridenko, M.: Approximation Algorithms for Asymmetric TSP by Decomposing Directed Regular Multigraphs. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2003, pp. 56–65. IEEE Computer Society, Washington, DC (2003)
29. Kuhn, F., Oshman, R.: The Complexity of Data Aggregation in Directed Networks. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 416–431. Springer, Heidelberg (2011)
30. Kutten, S.: Stepwise construction of an efficient distributed traversing algorithm for general strongly connected directed networks or: Traversing one way streets with no map. In: Computer Communication Technologies for the 90's, Proceedings of the Ninth International Conference on Computer Communication, ICCC 1988, pp. 446–452. International Council for Computer Communication, Elsevier (1988)
31. Megow, N., Mehlhorn, K., Schweitzer, P.: Online Graph Exploration: New Results on Old and New Algorithms. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 478–489. Springer, Heidelberg (2011)
32. Miyazaki, S., Morimoto, N., Okabe, Y.: The Online Graph Exploration Problem on Restricted Graphs. IEICE Transactions 92-D(9), 1620–1627 (2009)
33. Prakash, R.: Unidirectional links prove costly in wireless ad hoc networks. In: Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, DIALM 1999, pp. 15–22. ACM, New York (1999)
34. Ribeiro, B.F., Wang, P., Murai, F., Towsley, D.: Sampling directed graphs with random walks. In: Proceedings of the IEEE INFOCOM 2012, pp. 1692–1700. IEEE, Orlando (2012)
35. Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M.: An Analysis of Several Heuristics for the Traveling Salesman Problem. SIAM J. Comput. 6(3), 563–581 (1977)
36. Sedgewick, R., Vitter, J.S.: Shortest Paths in Euclidean Graphs. Algorithmica 1(1), 31–48 (1986)
37. Vishwanathan, S.: An Approximation Algorithm for the Asymmetric Travelling Salesman Problem with Distances One and Two. Inf. Process. Lett. 44(6), 297–302 (1992)

Lattice Completion Algorithms for Distributed Computations

Vijay K. Garg*

Parallel and Distributed Systems Lab,
Department of Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, TX 78712
garg@ece.utexas.edu
http://www.ece.utexas.edu/~garg

Abstract. A distributed computation is usually modeled as a finite partially ordered set (poset) of events. Many operations on this poset require computing meets and joins of subsets of events. The lattice of normal cuts of a poset is the smallest lattice that embeds the poset such that all meets and joins are defined. In this paper, we propose new algorithms to construct or enumerate the lattice of normal cuts. Our algorithms are designed for distributed computing applications and have lower time or space complexity than those of existing algorithms. We also show applications of this lattice to the problems in distributed computing such as finding the extremal events and detecting global predicates.

1 Introduction

A distributed computation is usually modeled as a set of events ordered by the partial order relation called the happened-before [Lam78] relation. This relation can be tracked using Mattern [Mat89] and Fidge's vector clocks [Fid89] which provide an efficient implicit representation of the poset of events that happened in a distributed computation. There are numerous applications in distributed systems such as distributed debugging [CM91, GW94], and recovery of distributed programs [SY85], that track the happened-before relation using vector clocks.

Since the joins and meets are always defined for lattices but may not exist for a general poset, there are many fundamental and practical advantages of working with lattices rather than posets. Given any poset, there are usually two ways to complete it — completion by consistent cuts (or ideals) and completion by normal cuts. The lattice of consistent cuts captures the notion of consistent global states in a distributed computation and has been discussed extensively in the distributed computing literature [Mat89, CM91, GM01]. The lattice of normal

* Supported in part by the NSF Grants CNS-1115808, CNS-0718990, CNS-0509024, and Cullen Trust for Higher Education Endowed Professorship.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 166–180, 2012.

cuts has not received much attention in distributed computing. For a poset P, its completion by normal cuts, or Dedekind-Macneille (DM) completion, denoted by $L_{DM}(P)$ is the smallest lattice that has P as its suborder [DP90]. Fig. 1(i) shows a distributed computation with four events. Its completion by normal cuts and consistent cuts is shown in Fig. 1(ii) and (iii), respectively.

The lattice of normal cuts is generally much smaller in size than the lattice of consistent cuts. In the extreme case, the lattice of consistent cuts may be exponentially bigger in size than the lattice of normal cuts. We show in this paper that some global predicates can be detected on the lattice of normal cuts instead of consistent cuts, thereby providing an exponential reduction in the complexity of detecting them.

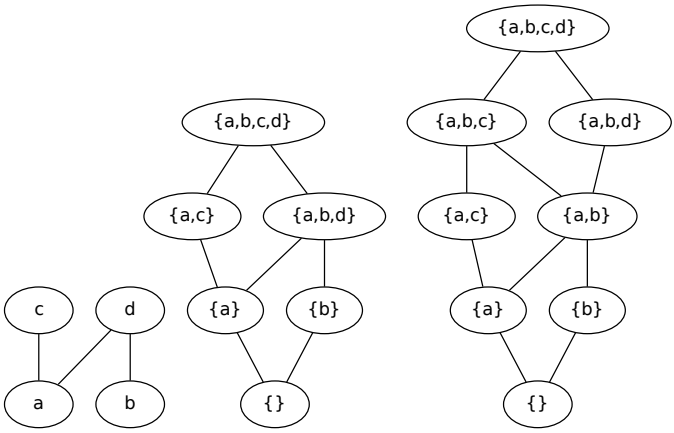


Fig. 1. (i) The original poset. (ii) Its lattice of normal cuts (iii) Its lattice of consistent cuts.

In this paper, we also discuss algorithms for constructing and enumerating $L_{DM}(P)$ for a distributed computation given as a finite poset P with implicit representation (i.e., represented using vector clocks). There has been extensive research in algorithms for the problems of DM-completion [NR99, NR02, GK98], construction of concept lattices [Gan84], construction of maximal antichain lattice [JRJ94], and construction of union-closed family of sets [NR99, NR02]. Our work differs in principally two ways. First, our focus is on implicit representation of posets and lattices. Most of the earlier work builds explicit cover relation of the lattice, whereas we represent the lattice implicitly using vector clocks. We note here that [Gar13] also uses vector clocks but for the lattice of maximal antichains. Second, our work is targeted towards distributed computing traces. For distributed computing traces, it is natural to assume that the number of events generated by a single process is significantly more than the number of processes, i.e., the width of the poset is much smaller than the height of the poset

corresponding to the computation. Also, most events in a distributed computation are internal to the process, i.e., they do not have any interaction with other processes. The computational complexity of our algorithm is explicitly dependent on the width of the poset, and the number of message receive events in a distributed system.

There are principally two classes of algorithms for generation of lattices. Incremental algorithms take as input a poset P and its lattice completion L, and output the lattice completion of the poset P extended with an element x. The algorithms by Ganter and Kuznetsov [GK98] and Lourine and Raynaud [NR99, NR02] fall in this class. These algorithms store the entire lattice. The other class of algorithms, frequently used in concept analysis [GW97], only require enumeration of all elements of the concept lattice. They do not require storage of the entire lattice (which may be exponentially bigger than the poset itself). The algorithm by Ganter [Gan84] falls in this class. It enumerates all elements of the lattice in a lexicographical order. To distinguish between these two classes of lattice generation, we refer to the first class of algorithms as the lattice *construction* and the second class of algorithms as the lattice *enumeration*. In this paper, we propose algorithms for both lattice construction and lattice enumeration adapted to distributed computations.

Table 1. Algorithms for Lattice Construction and Enumeration of Normal Cuts

Algorithm	Incremental	Time Complexity	Space Complexity
Ganter and Kuznetsov [GK98]	Yes	$O(mn^3)$	$O(mn \log n)$
Nourine and Raynaud [NR99, NR02]	Yes	$O(mn^2)$	$O(mn \log n)$
Algorithm IDML [this paper]	Yes	$O(rwm \log m)$	$O(mw \log n)$
BFS [this paper]	No	$O(mw^2(w + \log w_L))$	$O(w_L w \log n)$
DFS [this paper]	No	$O(mw^3)$	$O(h_L w \log n)$
Lexical by Ganter [Gan84]	No	$O(mn^3)$	$O(n \log n)$

Table 2. The notation used in the paper

Symbol	Definition	Symbol	Definition
n	size of the poset P	m	size of the normal cuts lattice L
w	width of the poset P	r	number of elements with more than one lower cover
h_L	height of the lattice L	w_L	width of the lattice L.

We first propose an incremental Dedekind-Macneille lattice construction algorithm called IDML which compares favorably with the algorithms proposed by Nourine and Raynoud [NR99, NR02] for distributed computing. Let the size of the poset be n and the size of the DM-lattice be m, then the algorithm by Nourine and Raynoud takes $O(n^2 m)$ time. The IDML algorithm takes $O(rwm \log m)$

time where w is the width of the poset and r is the number of receive events in the computation. For typical distributed computations, our algorithm has significantly smaller time complexity. Moreover, Nourine and Raynoud's algorithm require building a special structure called a lexicographic tree with space complexity $O(mn \log n)$. Our incremental algorithm uses a balanced binary search tree of all the lattice elements with the space complexity $O(mw \log n)$.

For lattice enumeration, the existing algorithms use *lexicographical* enumeration of the lattice [Gan84]. In this paper, we propose techniques for breadth-first (BFS) and depth-first (DFS) enumeration of lattices. It is important to note that the algorithms for BFS and DFS enumeration of lattices are different from the standard graph-based BFS and DFS enumeration because our algorithms cannot store the explicit graph corresponding to the lattice. Hence, the usual technique of marking the visited nodes is not applicable. BFS-enumeration and DFS-enumeration may be semantically more meaningful and useful in distributed computing than lexical enumeration. For example, while searching for an event with a given property in a distributed computation, it is more useful to find one at the lowest level of the lattice. Note that BFS, DFS and lexical algorithms for enumeration of the lattice of consistent cuts (but not for the lattice of normal cuts) have already been proposed in the distributed computing literature. For example, BFS enumeration has been proposed by Cooper and Marzullo [CM91], DFS enumeration by Alagar and Venkatesan [AV01], and Lexical enumeration by Garg [Gar03]. Due to different structure of these lattices, the technique for BFS and DFS enumeration is quite different. For example, the problem of determining if an element of the lattice has already been enumerated is different for the two lattices. Table 1 summarizes the time and space complexity of the lattice construction and enumeration algorithms.

The ability to construct or enumerate the lattice of normal cuts has wide applications in many areas. We discuss distributed computing applications in Section 6. It has applications in other areas such as formal concept analysis [GW97] but will not be discussed in this paper.

2 Background: Posets with Implicit Representation

We assume that the reader is familiar with the basic concepts of posets and lattices [DP90]. A partially ordered set (or *poset*) is a pair $P = (X, \leq)$ where X is a set and $\leq$ is a reflexive, antisymmetric, and transitive binary relation on X. A *subposet* of P is a subset of X whose order relation is restriction of P to the subset. If either $x \leq y$ or $y \leq x$, we say that x and y are *comparable*; otherwise, we say x and y are *incomparable*. For any two elements x and y, y covers x if $x < y$ and $\forall z \in X : x \leq z < y$ implies $z = x$. A subset $Y \subseteq X$ is called an *antichain (chain)*, if every distinct pair of points from Y is incomparable (comparable) in P. The *width (height)* of a poset is defined to be the size of a largest antichain (chain) in the poset.

Given a subset $Y \subseteq X$, the *meet* of Y, if it exists, is the greatest lower bound of Y and the *join* of Y is the least upper bound. An element is *join-irreducible (meet-irreducible)* if it cannot be expressed as the join (meet) of other elements. A poset $P = (X, \leq)$ is a *lattice* if joins and meets exist for all finite subsets of X. It is a *complete lattice* if joins and meets exists for all subsets of X. The largest element of a lattice is called the *top* element.

Let P be a poset with a given chain partition of width w. In a distributed computation, P is the set of events executed under the happened-before relation where a chain corresponds to a total order of events executed on a single process. In such a poset, every element e can be identified with a tuple (i, k), the k^{th} event on the i^{th} chain. In this paper, we keep the order relation implicit using vector clock [Mat89] as explained next. For $e \in P$, let $D[e]$, the *down-set* of e be the elements in P, that are less than or equal to e. The set $D[e]$ can equivalently be captured using a vector $e.V$ such that $e.V[i] = j$ iff there are exactly j elements on chain i that are less than or equal to e. It is easy to verify that $e \leq f$ iff $e.V \leq f.V$. Fig. 2(i) and (ii) show a poset and corresponding vector clocks.

A subset Q is a consistent cut (an order ideal) of P if it satisfies the constraint that if f is in Q and e is less than or equal to f, then e is also in Q. For any element $e \in P$, $D[e]$ is always a consistent cut and is called a *principal ideal*. Any consistent cut Q of P can be represented using a simple vector $Q.V$ with the interpretation that $Q.V[i] = j$ iff exactly j smallest elements of chain i are in Q. Note that we have used vectors for representing events as well as set of events. Given two consistent cuts Q and R, their intersection (union) is simply the component-wise minimum (maximum) of the vectors for Q and R.

Just as we have constructed vectors using the down-sets, we can also use the dual *up-sets*. For $e \in P$, let $U[e]$, the *up-set* of e be the elements in P, that are greater than or equal to e. The notion of order filters which are duals of order ideals can similarly be defined.

3 Lattice Completion of a Computation

In this section, we discuss lattice-completion of a computation via normal cuts. Given $Q \subseteq P$, the set of lower bounds of Q, denoted by Q^l is given by

$$\{x \in P | \forall e \in Q : x \leq e\}$$

In Fig. 1, $\{c, d\}^l = \{a\}$. When Q is empty, Q^l is trivially the entire set P. When Q is singleton $\{e\}$, then it simply corresponds to $D[e]$. In general, we can compute Q^l using D as follows:

$$Q^l = \cap_{e \in Q} D[e]$$

It is easy to verify that Q^l is alway a consistent cut because $D[e]$ is a consistent cut for any e, and consistent cuts are closed under intersection. Similarly, the set of upper bounds of Q, denoted by Q^u can be computed. In Fig. 1, $\{a, b\}^u = \{d\}$. The set $(\{a, b\}^u)^l = \{a, b, d\}$.

Definition 1 (Normal Cut). *[DP90] A set $Q \subseteq P$ is a* normal cut *if $(Q^u)^l = Q$.*

We will use the simpler notation Q^{ul} for $(Q^u)^l$. It is easily shown [DP90] that computing Q^{ul} for any $Q \subseteq P$ is a closure operator, i.e., (1) $Q \subseteq Q^{ul}$ (it is extensive), (2) $Q_1 \subseteq Q_2 \Rightarrow Q_1^{ul} \subseteq Q_2^{ul}$ (it is monotone) (3) $(Q^{ul})^{ul} = Q^{ul}$ (it is idempotent). It is easy to verify that principal ideals are always normal. Indeed, if $Q = D[e]$, then $Q^u = U[e]$ and $Q^{ul} = D[e] = Q$. Since normal cuts correspond to a closure operator, they are closed under intersection.

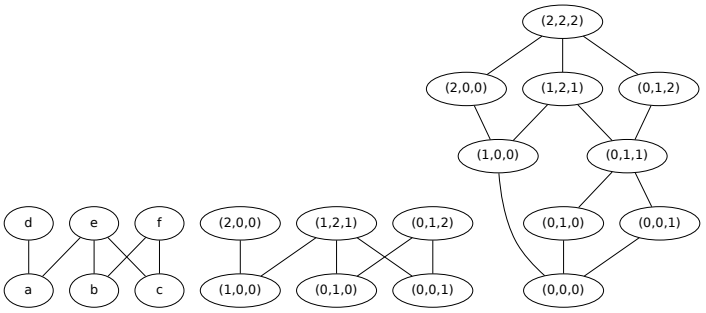


Fig. 2. (i) The original poset. (ii) Equivalent representation using Vector Clocks (iii) Its Lattice of normal cuts.

Definition 2 (Dedekind–MacNeille Completion of a Poset). *For a given poset $P = (X, \leq)$, the Dedekind–MacNeille completion of P is the poset formed with the set of all the normal cuts of P under the set inclusion. Formally,*

$$DM(P) = (\{A \subseteq X : A^{ul} = A\}, \subseteq).$$

For the poset in Figure 1(i), the set of all normal cuts is:
$\{\{\}, \{a\}, \{b\}, \{a, c\}, \{a, b, d\}, \{a, b, c, d\}\}$.
The poset formed by these sets under the $\subseteq$ relation is shown in Figure 1(ii). This new poset is a complete lattice. The meet of normal cuts is same as the set intersection. The join of a set of normal cuts Q is defined as the meet of all the normal cuts that that are greater than or equal to all the normal cuts in Q. For example, the join of $\{a\}$ and $\{b\}$ is $\{a, b, d\}$ because it is the meet of all normal cuts which contain both $\{a\}$ and $\{b\}$. Our original poset P is embedded in this new structure such that x is mapped to the set $D[x]$.

For the poset in Fig. 2, the lattice of normal cuts has 9 cuts:
$\{\{\}, \{a\}, \{b\}, \{c\}, \{a, d\}, \{b, c\}, \{a, b, e, c\}, \{b, c, f\}, \{a, b, c, d, e, f\}\}$. Figure 2(iii) shows these 9 cuts in the vector clock representation. The lattice of consistent cuts has 19 elements (not shown in the figure).

4 IDML: An Incremental Algorithm for Lattice Completion

Let P be a poset and L be its Dedekind-Macneille lattice completion. In this section, we present a new incremental algorithm for lattice completion in implicit representation in which both P and L are represented using vectors. Suppose that a new element x is added to P with the constraint that x is not less than or equal to any of the existing elements. Our goal is to compute the lattice completion, L' of $P' = P \cup \{x\}$ given P and L. When P is a singleton, then its completion is itself. By adding one element at a time in any linear order that is consistent with the partial order, we can use the incremental algorithm for lattice completion of any poset.

Our incremental strategy for the lattice completion is as follows. We show that all the elements of L other than the top element of L are also contained in L'. The top element of L would either be retained or modified for L'. Therefore, except for the top, our algorithm will simply add elements to L to obtain L'.

Lemma 1. *Let S be a normal cut of $P = (X, \leq)$ such that $S \neq X$. Then S is also a normal cut of $P' := P \cup \{x\}$ where x is a maximal element of P'.*

Proof. Let $T = S^u$ in P. This implies that $T^l = S$ in P because S is a normal cut of P. Since $S \neq X$, T is nonempty (because if T is empty, $T^l = X$ which is not equal to S).

If $S \subseteq D[x]$, then S^u in P' equals $T \cup \{x\}$. We need to show that $S^{ul} = S$ in P', i.e., $(T \cup \{x\})^l = S$ in P'. The set $(T \cup \{x\})^l = T^l \cap D[x]$. Since x is a maximal element, we know that $x \notin T^l$. Since $T^l = S$ and $S \subseteq D[x]$, $T^l \cap D[x] = S$. Hence, S is a normal cut of P'.

If $S \not\subseteq D[x]$, then S^u in P' equals T. Since T is nonempty, and $T^l = S$ in P, we get that $T^l = S$ in P' as well. Hence, S is a normal cut of P'.

Our algorithm for DM-construction is shown in Fig. 3. Whenever a new element x arrives, we carry out three steps. In step 1, we process Y, the top element of L; in step 2, we add a normal cut corresponding to the principal ideal of x; and, in step 3, we process the remaining elements of L. The goal of step 1 is to ensure that L' has a top element. The goal of step 2 is to ensure that all principal ideals of P' are in L'. The goal of step 3 is to ensure that L' is closed under intersection. In step 3, we first check if x covers more than one element. If it does not, then we do not have to go over all normal cuts in L because of the following claim.

Lemma 2. *If x covers at most one element in P, then for any normal cut $W \in L$, $min(W, D[x]) \in L$ assuming $\{\} \in L$.*

Proof. If x does not cover any element of P, then $D[x] = \{x\}$ and $W \cap D[x] = \{\}$ which is assumed to be in L. Now suppose that x covers just one element y, then $D[x] = \{x\} \cup D[y]$. Therefore, $W \cap D[x] = W \cap D[y]$. Since both W and $D[y]$ are normal cuts of L, so is $W \cap D[y]$.

Input: a nonempty finite poset P, its DM-completion L, element x
Output: L' := DM-completion of $P \cup \{x\}$

$D[x]$:= the vector clock for x;
$Y := top(L)$;
$newTop := max(D[x], Y)$;
// **Step 1**: Ensure that L' has a top element
  if $Y \in P$ then $L' := L \cup \{newTop\}$;
  else $L' := (L - Y) \cup \{newTop\}$;
// **Step 2**: Ensure that $D[x]$ is in L'
  if $(D[x] \neq newTop)$ then $L' := L' \cup \{D[x]\}$;
// **Step 3**: Ensure that all meets are defined
  if x does not cover any element in P then
    $L' := L' \cup \{\mathbf{0}\}$; // add zero vector
  else if x covers more than one element in P then
    for all normal cuts $W \in L$ do
      if $min(W, D[x]) \notin L'$ then $L' := L' \cup min(W, D[x])$;

Fig. 3. Incremental Algorithm IDML for DM-construction

We now show the correctness of the algorithm, i.e., L' is precisely the DM-lattice for P'.

Theorem 1. *The algorithm IDML computes DM-completion of P' assuming that L is a DM-completion of P.*

Proof. We first show that all cuts included in L' are normal cuts of P'. In step 1, we add to L' all cuts of L except possibly $top(L)$, and $max(D[x], Y)$. All elements of L except possibly $top(L)$ are normal cuts of P' from Lemma 1. The cut $max(D[x], Y)$ is a cut of P', because it includes all elements of P'. In step 2, we add cut $D[x]$ to L' which is a normal cut of P' because it is a principal ideal of P'. In step 3, we only add cuts of the form $min(W, D[x])$. Since both W and $D[x]$ are normal cuts of P', and the set of normal cuts is closed under intersection, we get that $min(W, D[x])$ is also a normal cut of P'.

We now show that all normal cuts of P' are included in L'. Let S be a normal cut of P'. Let Q be the set of all principal ideals of P'. By our construction, L' includes all principal ideals of P' (because of step 2). It is sufficient to show that L' is closed under joins and meets. Since we have the top element in L', it is sufficient to show closure under meets. Let S and T be two normal cuts in L'. If both S and T are in L, then $S \cap T$ is in L and therefore also L'. Now, assume that $S \in L' - L$. Therefore, $S = W \cap D[x]$ for some $W \in L$. If $T \in L$, then $S \cap T = W \cap D[x] \cap T = (W \cap T) \cap D[x]$. Since $(W \cap T) \in L$, we get that $S \cap T \in L'$ because of step 3. If $T \in L' - L$, then it can be written as $W' \cap D[x]$ for some $W' \in L$. Therefore, $S \cap T = W \cap D[x] \cap W' \cap D[x] = (W \cap W') \cap D[x]$. Since $(W \cap W') \in L$, we again get that $S \cap T \in L'$. Since L' contains all principal ideals of P' and is closed under meet and join, we get that all normal cuts of P' are included in L'.

Note that our algorithm also gives an easy proof for the following claim.

Lemma 3. *The number of normal cuts of $P \cup \{x\}$ is at most twice the number of normal cuts of P plus two.*

Proof. For every cut in L, we add at most one more cut in Step 3 of the algorithm. Further, we add at most one cut in step 1 and one additional cut in Step 2.

We now discuss the time complexity of the IDML algorithm. Let m be the size of the lattice L. The time complexity of the IDML algorithm is dominated by step 3. Assuming that L is kept in a sorted order (for example, in the lexicographically sorted order) in a balanced binary search tree, the operation of checking whether $min(W, S) \in L$ can be performed in $O(w \log m)$, where w is the width of the poset P'. For any element for which we traverse the lattice L, we take $O(wm \log m)$ time. If the element x covers only one element (or no elements), then we take $O(w \log m)$ time. Suppose that there are r events in the poset that cover at least two events. In a distributed computation, only *receive* events would have this property. Then, to compute DM-Lattice of a poset P, we can repeatedly invoke IDML algorithm in any total order consistent with P. Therefore, we can construct DM-lattice of a poset P of width w with r elements of lower cover of size at least two in $O(rwm \log m)$. The algorithm by Nourine and Raynoud [NR99, NR02] takes $O(n^2 m)$ time. Since $n \leq m \leq 2^n$, our algorithm takes time $O(rwn \log n)$ when $m = O(n)$.

We also note here that given a poset P, to construct its DM-lattice, we can restrict our attention to its subposet of irreducible elements because DM-completion of P is identical to DM-completion of the subposet containing all its join and meet irreducibles [DP90].

5 Traversal Based Algorithms for DM-Completion

In some distributed computing applications, we may be interested not in storing the DM-Lattice but simply enumerating all the elements of the lattice or storing only those elements of the lattice that satisfy a given property. Recall that the size of the DM-Lattice may be exponential in the size of the poset in the worst case. Algorithm IDML has space complexity of $O(mw \log n)$ to store the lattice L (there are m elements in the lattice, and each element is represented using a w dimensional vector of entries of size $O(\log n)$). We now give an algorithm BFS-DML that does not require storing the entire lattice.

5.1 Breadth First Search Enumeration of Normal Cuts

The algorithm BFS-DML views the lattice as a directed graph and generates its elements in the breadth-first-order. It is different from the traditional BFS algorithm on a graph because we do not store the graph or keep data that is proportional to the size of the graph (such as the nodes already visited). Let $Layer(k)$ be the set of nodes in the graph that are at distance k from the bottom

element of the lattice. Let w_L be the size of the largest set $Layer(k)$. Then, the space required by BFS-DML is $O(w_L w \log n)$.

The algorithm BFS-DML is shown in Figure 4. The set $\mathcal{S}$ is used to store the set of nodes that have been generated but have not been explored yet. The set is kept in a balanced binary tree so that it is easy to check if some element is already contained in the set. We maintain the invariant that the set $\mathcal{S}$ contains only the normal cuts of the poset P. The elements in the binary search tree are compared using the function *levelCompare* shown in Fig. 4. For any vector a corresponding to a consistent cut, the function $a.sum()$ returns the number of events in the consistent cut. At lines (1) and (2) of the function *levelCompare*, we define a consistent cut to be smaller than the other if it has fewer elements. Lines (3)-(5) impose a lexicographic order on all consistent cuts with equal number of elements. As a result, the function *levelCompare* imposes a total order on the set of all consistent and normal cuts.

The main BFS traversal of normal cuts, shown in lines (1) to (6), exploits the fact that there is a unique least normal cut that contains any consistent cut. The algorithm removes normal cuts from $\mathcal{S}$ in the *levelCompare* order. Let H be the smallest vector in this order (line 2). It finds all consistent cuts reachable from H by executing a single event e (line 4). We define an event e to be *enabled* in H if $H \cup \{e\}$ is a consistent cut. It adds all normal cuts that corresponds to "closure" of consistent cuts $H \cup \{e\}$ at line (5). We need to ensure that no normal cut is enumerated twice. At line (6), we check if a normal cut is already part of $\mathcal{S}$. It can be shown that this check is sufficient to ensure that no normal cut is enumerated twice (due to the definition of *levelCompare* and the BFS order of traversal).

We now discuss the complexity of the BFS algorithm. At line (4), since there are w processes, there can be at most w events enabled on any normal cut H. Checking whether an event e is enabled in H requires that the events that happened-before e in poset P are included in H. This check requires $O(w)$ comparisons in the worst case (using vector clocks).

To find the smallest normal cut containing $Q := H \cup \{e\}$, we simply compute Q^{ul}. Since $f \leq g$ is equivalent to $U[g] \subseteq U[f]$, we can restrict our attention to maximal elements of Q, i.e.,

$$Q^u = \cap_{f \in maximal(Q)} U[e].$$

Since P is represented using w chains, there are at most w maximal elements and therefore we can compute Q^u in $O(w^2)$ operations. We now take $R := Q^u$ and compute R^l, again using $O(w^2)$ operations. Thus, step (5) can be implemented in $O(w^2)$.

To check if the resulting normal cut K is not in $\mathcal{S}$, we exploit the tree structure of $\mathcal{S}$ to perform it in $O(w \log |\mathcal{S}|)$ which is $O(w \log w_L)$ in the worst case. Hence the total time complexity of Algorithm BFS is $O(mw(w^2 + w \log w_L)) = O(mw^2(w + \log w_L))$. The main space complexity of the BFS algorithm is the data structure $\mathcal{S}$ which is $(w_L w \log n)$. Note that the size of $\mathcal{S}$ is proportional to the size of the layer of the lattice in BFS enumeration (w_L) and is much smaller than the size of the lattice m used in the IDML algorithm.

```
    Input: a finite poset P
    Output: Breadth First Enumeration of elements of DM-completion of P
    G := bottom element ;
    S := Ordered Set of VectorClocks initially {G} with levelCompare order;
(1)   while (S is notEmpty)
(2)       H := remove the smallest element from S;
(3)       output(H);
(4)       foreach event e enabled in H do;
(5)           K := the smallest normal cut containing Q := H ∪ {e};
(6)           if K is not in S, then add K to S;

      int function levelCompare(VectorClock a, VectorClock b)
(1)       if (a.sum() > b.sum()) return 1;
(2)       else if (a.sum() < b.sum()) return -1;
(3)       for (int i = 0; i < a.size(); i++)
(4)           if (a[i] > b[i]) return 1;
(5)           if (a[i] < b[i]) return -1;
(6)       return 0;
```

Fig. 4. Algorithm BFS-DML for BFS Enumeration of DM-Lattice

5.2 Depth First Search Enumeration of Normal Cuts

Another useful technique to enumerate elements of the lattice is based on the depth first search order. In BFS enumeration, the storage required is proportional to the width of the lattice whereas in DFS enumeration the storage required is proportional to the height of the lattice. Given any poset with n elements, the width of its lattice of normal cuts may be exponential in the size of the poset, but the height is always less than or equal to n. Hence, the DFS enumeration may result in exponential savings in space.

The algorithm for DFS enumeration is shown in Fig. 5. From any normal cut, we explore all enabled events to find the normal cuts. There are at most w enabled events and for each event it takes $O(w^2)$ time to compute the normal cut K at line (3). Since we are not storing the enumerated elements explicitly, we need a method to ensure that the same normal cut is not visited twice. For example, in Fig. 1, the normal cut $\{a, b, d\}$ is reachable from $\{a\}$ as well as $\{b\}$. Let $pred(K)$ be the set of all normal cuts that are covered by K in the lattice. We use the total order *levelCompare* defined in Section 5.1 on the set $pred(K)$. We make a recursive call on K from the normal cut G iff G is the maximum normal cut in $pred(K)$ in the *levelCompare* order. Line (4) finds the maximum predecessor M using the traversal on the dual poset P^d. The dual of a poset $P = (X, \leq)$ is defined as follows. In the poset P^d, $x \leq y$ iff $y \leq x$ in P. It is easy to verify that S is a normal cut in P iff S^u is a normal cut in P^d. The function *get-Max-Predecessor*, shown in Fig. 5 uses expansion of a normal cut in the poset P^d to find the maximum predecessor.

```
    Input: a finite poset P, starting state G
    Output: DFS Enumeration of elements of DM-completion of P
(1)     output(G);
(2)     foreach event e enabled in G do
(3)         K := smallest normal cut containing Q := G ∪ {e};
(4)         M := get-Max-predecessor(K) ;
(5)         if M = G then
(6)             DFS-NormalCuts(K);

    function VectorClock get-Max-predecessor(K) {
    //takes K as input vector and returns its maximum predecessor normal cut
(1)     H = MinimalUpperBounds(K); // H := K^u
(2)     // find the maximal predecessor using normal cuts in the dual poset
(3)     foreach event f enabled in the cut H in P^d do
(4)         temp_f := H − {f}; // advance on event f in P^d from cut H;
(5)         // get the set of lower bounds on temp_f
(6)         pred := MaximalLowerBounds(temp_f) using H^l;
(7)          if (levelCompare(pred, maxPred) = 1) then maxPred = pred;
(8)     return maxPred;
```

Fig. 5. Algorithm DFS-DML for BFS Enumeration of DM-Lattice

The function *get-Max-predecessor* works as follows. At line (1), we compute $H = K^u$, which is the normal cut in P^d corresponding to K. Our goal is to compute all predecessors of K in P which corresponds to all successors of H in P^d. To find successors of H, we consider each event f enabled in H in P^d. At line (4), we compute the consistent cut $temp_f$. The closure of $temp_f$ in P^d equals $temp_f^{ul}$ in P^d. Equivalently, we can compute $temp_f^{lu}$ in P. The closed set $temp_f^{lu}$ in P^d corresponds to the closed set $temp_f^{lul}$ in P. However, we know that $temp_f^{lul}$ is equal to $temp_f^{l}$. Therefore, by computing $temp_f^{l}$ for each f enabled in H, we get all the predecessors of K in P^d. Since there can be w events enabled in H in P^d, and it takes $O(w^2)$ time to compute each predecessor, it would take $O(w^3)$ to determine the maximum predecessor. However, since $temp_f$ and H differ on a single event, we can compute $temp_f^{l}$ using $H^l = K$ in $O(w)$ time. By this observation, the complexity of computing max-predecessor reduces to $O(w^2)$, and the total time complexity to determine whether K can be inserted is $O(w^2)$.

In line (5) of DFS-DML, we traverse K using recursive DFS call only if M equals G. Since the complexity of step (3) and step (4) is $O(w^2)$, the overall complexity of processing a normal cut G is $O(w^3)$ due to the *foreach* at line (2). Since there are m normal cuts, we get the total time complexity of DFS-DML algorithm as $O(mw^3)$.

The main space requirement of the DFS algorithm is the stack used for recursion. Every time the recursion level is increased, the size of the normal cut increases by at least 1. Hence, the maximum depth of the recursion is n. Therefore, the space requirement is $O(nw \log n)$ bits because we only need to store

vectors of dimension w at each recursion level. Hence, the DFS algorithm takes significantly less space than the BFS algorithm.

6 Applications of Normal Cuts in Distributed Systems

6.1 Finding the Meet and Join of Events

Suppose that there are two events x and y on different processes that correspond to faulty behavior. It is natural to determine the largest event, z, in the computation that could have affected both x and y. The event z is simply the meet of events x and y if it exists in the underlying computation. For example, in Fig. 2(a), suppose that the faulty events are $\{d, e\}$. In this case, the "root" cause of faults of these events could be event a. In the vector clock representation, the root cause is $(1, 0, 0)$ in the DM-Lattice. Now consider the case when the set of faulty events is $\{e, f\}$. In this case, the underlying computation does not have a unique maximum event that affects both e and f. It can be seen in Fig. 2(a) that both the events b and c could be the "root" cause of the events e and f. This is exactly what we would get from the lattice of normal cuts. The largest normal cut that is smaller than both events e with vector clock $(1, 2, 1)$ and event f with vector clock $(0, 1, 2)$ equals the vector $(0, 1, 1)$ which correctly identifies the set of events that affect both e and f.

Dually, we may be interested in the smallest event z that happened-after a subset of events. In a distributed system, an event z can have the knowledge of event x only if x happened-before event z. If two events x and y happened on different processes, the minimum event z that knows about both x and y corresponds to their join.

6.2 Detecting Global Predicates in Distributed Systems

A global predicate on a distributed computation is a boolean function B defined on the set of consistent cuts of the computation. If B is true on a consistent cut G, then we denote it as $B(G)$. The problem of detecting a global predicate *possibly* : B corresponds to determining if there exists a consistent cut G in the computation that satisfies B. The global predicate detection problem is NP-complete [CG98] even for the restricted case when the predicate B is a singular 2CNF formula of local predicates [MG01]. The key problem is that the lattice of consistent cuts L_{CGS} may be exponential in the size of the poset. The lattice of normal cuts, L_{DM} of a poset P is a suborder of the L_{CGS} (every normal cut is consistent, but every consistent cut may not be normal). Its size always lies between the size of the poset P and the size of the lattice of consistent cuts of P. In particular, it may be exponentially smaller than L_{CGS}. We now show that a class of predicates can be efficiently detected by traversing the lattice of normal cuts rather than L_{CGS}.

The class of predicates we discuss are based on the idea of knowledge in a distributed system [HM84]. We define knowledge predicates based on the happened-before relation. We use the notation $G[i]$ to refer to events of G on process i.

Definition 3. *Given a distributed computation, or equivalently a poset $(P, \leq)$, we say that every one knows the predicate B in the consistent cut G, if there exists a consistent cut H such that H satisfies B and for every process i there exists an event e in $G[i]$ such that all events in H happened before e. Formally, $E(B, G) \equiv \exists H : B(H) \wedge \forall i \exists e \in G[i] : \forall f \in H : f \leq e$.*

We also define $E(B) \equiv \exists G : E(B, G)$

Intuitively, the above definition says that a predicate is known to everyone in the system if every process has a consistent cut in its past in which B was true. The definition captures the fact that in a distributed system, a process can know about remote events only through a chain of messages.

We now show that instead of traversing L_{CGS} we can traverse L_{DM} to detect $E(B)$ for any global predicate B.

Theorem 2. *Let B be any global predicate and G be a consistent cut such that $E(B, G)$. Then, there exists a normal cut N such that $E(B, N^u)$.*

Proof. Since everyone knows B in G, by the definition of "everyone knows", we get that there exists a consistent cut $H \subseteq G$ such that B is true in H and every process in G knows H. Let $\mathcal{K}$ be the set of all consistent cuts that know H. The set is nonempty because $G \in \mathcal{K}$. Furthermore, it is easy to show that the set $\mathcal{K}$ is closed under intersection. The least element K of the set $\mathcal{K}$ corresponds to the minimal elements of the filter H^u. Hence, we conclude that $E(B, K)$.

Define N to be the consistent cut corresponding to H^{ul}. It is clear that N is a normal cut because it corresponds to the closure of H. Moreover, $N^u = H^{ulu} = H^u = K$. The first equality holds by the definition of N and the second equality holds due to properties of u and l operators. Since K equals N^u, from $E(B, K)$ we get that $E(B, N^u)$.

7 Conclusions and Future Work

We have proposed algorithms for the construction and enumeration of the lattice of normal cuts of a poset of a distributed computation. We have also shown their application to distributed computing.

It is clear that enumeration or construction of a lattice of size m in which each element is represented using $w \log n$ bits requires $\Omega(mw \log n)$ time. The problem of finding an algorithm that matches the lower bound is open.

Acknowledgements. I am thankful to Bharath Balasubramanian for discussions on the topic.

References

[AV01] Alagar, S., Venkatesan, S.: Techniques to tackle state explosion in global predicate detection. IEEE Transactions on Software Engineering 27(8), 704–714 (2001)

[CG98] Chase, C.M., Garg, V.K.: Detection of global predicates: Techniques and their limitations. Distributed Computing 11(4), 191–201 (1998)

[CM91] Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, pp. 163–173 (May 1991)

[DP90] Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)

[Fid89] Fidge, C.J.: Partial orders for parallel debugging. In: Proc. of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, vol. 24(1), pp. 183–194 (January 1989)

[Gan84] Ganter, B.: Two basic algorithms in concept analysis. Technical Report 831, Techniche Hochschule, Darmstadt (1984)

[Gar03] Garg, V.K.: Enumerating global states of a distributed computation. In: Intl Conf. on Parallel and Distributed Computing and Systems, pp. 134–139 (November 2003)

[Gar13] Garg, V.K.: Maximal antichain lattice algorithms for distributed computations. In: Proc. of Distributed Computing and Networking - 14th International Conference, ICDCN 2013 (January 2013)

[GK98] Ganter, B., Kuznetsov, S.O.: Stepwise Construction of the Dedekind-MacNeille Completion. In: Mugnier, M.-L., Chein, M. (eds.) ICCS 1998. LNCS (LNAI), vol. 1453, pp. 295–302. Springer, Heidelberg (1998)

[GM01] Garg, V.K., Mittal, N.: On slicing a distributed computation. In: 21st Intnatl. Conf. on Distributed Computing Systems, ICDCS 2001, pp. 322–329. IEEE, Washington (2001)

[GW94] Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. on Parallel and Distributed Systems 5(3), 299–307 (1994)

[GW97] Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations, 1st edn. Springer-Verlag New York, Inc., Secaucus (1997)

[HM84] Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. In: Kameda, T., Misra, J., Peters, J., Santoro, N. (eds.) PODC, pp. 50–61. ACM (1984)

[JRJ94] Jourdan, G.-V., Rampon, J.-X., Jard, C.: Computing on-line the lattice of maximal antichains of posets. Order 11, 197–210 (1994)

[Lam78] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. of the ACM 21(7), 558–565 (1978)

[Mat89] Mattern, F.: Virtual time and global states of distributed systems. In: Proc. of the Intl. Workshop on Parallel and Distributed Algorithms, pp. 215–226 (1989)

[MG01] Mittal, N., Garg, V.K.: On detecting global predicates in distributed computations. In: 21st Intnatl. Conf. on Distributed Computing Systems, ICDCS 2001, pp. 3–10. IEEE, Washington (2001)

[NR99] Nourine, L., Raynaud, O.: A fast algorithm for building lattices. Inf. Process. Lett. 71(5-6), 199–204 (1999)

[NR02] Nourine, L., Raynaud, O.: A fast incremental algorithm for building lattices. J. Exp. Theor. Artif. Intell. 14(2-3), 217–227 (2002)

[SY85] Strom, R.E., Yemeni, S.: Optimistic recovery in distributed systems. ACM Trans. Comput. Syst. 3(3), 204–226 (1985)

Optimal Broadcast in Shared Spectrum Radio Networks*

Mohsen Ghaffari[1], Seth Gilbert[2], Calvin Newport[3], and Henry Tan[3]

[1] MIT
ghaffari@mit.edu
[2] National University of Singapore
seth.gilbert@comp.nus.edu.sg
[3] Georgetown University
{cnewport,ztan}@cs.georgetown.edu

Abstract. This paper studies single hop broadcast in a single hop shared spectrum radio network. The problem requires a source to deliver a message to n receivers, where only a polynomial upper bound on n is known. The model assumes that in each round, each device can participate on 1 out of $\mathcal{C} \geq 1$ available communication channels, up to $t < \mathcal{C}$ of which might be disrupted, preventing communication. This disruption captures the unpredictable message loss that plagues real shared spectrum networks. The best existing solution to the problem, which comes from the systems literature, requires $O\big(\frac{\mathcal{C}t}{\mathcal{C}-t} \log n\big)$ rounds. Our algorithm, by contrast, solves the problem in $O\big(\frac{\mathcal{C}}{\mathcal{C}-t} \lceil \frac{t}{n} \rceil \log n\big)$ rounds, when $\mathcal{C} \geq \log n$, and in $O\big(\frac{\mathcal{C}}{\mathcal{C}-t} \log n \cdot \log\log n\big)$ rounds, when $\mathcal{C}$ is smaller. It accomplishes this improvement by deploying a self-regulating relay strategy in which receivers that already know useful information coordinate themselves to efficiently assist the source's broadcast. We conclude by proving these bounds tight for most cases.

1 Introduction

Consider a wireless device, which we call the *source*, with a message to send to a group of nearby devices, which we call the *receivers*. If this network had a dedicated communication channel, the problem would be easily solved: the receivers could simply wait on this channel for the source to broadcast its message. Unfortunately, in practice, this assumption almost never holds. Most wireless networking now takes place in *shared spectrum networks* where a group of communication frequencies are shared, in an uncoordinated manner, by multiple different networks, protocols, and unrelated sources of interference. The 2.4 GHz band, for example, is used by 802.11, Bluetooth, Zigbee, many different types of sensor network motes, cordless phones, baby monitors, and some types of car alarm sensors. Not surprisingly, interference between these competing devices is common [8].

The challenge faced by our source is now more pronounced. It can no longer use a fixed channel to communicate, because that channel might be disrupted by other users of the same spectrum. It must instead start sifting through the channels, seeking its receivers amidst this churning sea of electromagnetic noise. This problem, which we

* This research was supported by Singapore NUS FRC grant R-252-000-443-133 and the Ford Motor Company University Research Program.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 181–195, 2012.

call *Single-hop Shared-spectrum Broadcast* (SSB), comes from the systems literature, where it is well-studied [9–12, 16–21]. In practice, SSB algorithms are typically used to send a session key from a master device to its slaves. This key can then be used to configure more traditional disruption-resilient coding techniques such as frequency-hopping spread spectrum (FHSS) or direct sequence spread spectrum (DSSS). A well-known example of an SSB solution is the pairing protocol used by Bluetooth. (See [17] for more on the practical motivation driving this problem.)

Results. Following the lead of past theory work on shared spectrum, we formally describe this setting using the t*-disrupted* network model [2–7, 13, 14], which assumes devices have access to $\mathcal{C} \geq 1$ communication channels, with up to $t < \mathcal{C}$ disrupted by outside sources of interference. In this model, in each round, each device chooses a single channel on which to participate. The set of up to t disrupted channels is chosen arbitrarily and can change from round to round.

The best existing SSB algorithm [20], when analyzed in this model, solves the problem in $O\big(\frac{\mathcal{C}t}{\mathcal{C}-t} \log n\big)$ rounds, with high probability (i.e., at least $1 - 1/N$, where N is a known polynomial upper bound on n). In Section 3, we describe *pandemic broadcast*, a pair of algorithms which solve the problem in $O\big(\frac{\mathcal{C}}{\mathcal{C}-t}\lceil\frac{t}{n}\rceil \log n\big)$ rounds, for $\mathcal{C} \geq \log n$, and in $O\big(\frac{\mathcal{C}}{\mathcal{C}-t} \log n \cdot \log\log n\big)$ rounds for smaller $\mathcal{C}$. In the $\mathcal{C} \geq \log n$ setting, our solution is a factor of t faster than the existing solution when $t \leq n$, and a factor of n faster when $t > n$. For the small $\mathcal{C}$ setting, this advantage reduces only slightly to $t/\log\log n$. Finally, in Section 4, we prove our solutions optimal (within $\log\log n$ factors) for most relevant cases.

Because the SSB problem is drawn directly from the systems literature, our solutions can be applied directly back to these systems, making it a good example of distributed algorithm theory helping to improve real world wireless networks.

Intuition. The core insight driving our algorithm is that it exploits the parallelism inherent in having multiple available communication channels. The best existing solution [20] allows devices to only receive information from the source. In contrast, our solutions, which we call *Pandemic Broadcast Algorithms*, allow devices to relay information on behalf of the source, eventually converging on a state where there is a single relayer per channel, on a constant fraction of the channels. In this state, the remaining uninformed devices quickly learn new information.

The main challenge in implementing this idea is the unknown number of receivers. If too few devices relay, then not much advantage is gained. On the other hand, if too many devices relay, they clog the channels with collisions. The pandemic broadcast algorithms overcome these issues with a self-regulating process, inspired by infectious disease propagation. In more detail, the algorithms rely on two-stage infection. The first stage aggressively infects receivers, turning them into relayers. It guarantees at least $\mathcal{C}$ relayers, but may end up producing many more before it dies out. The second stage has these relayers reduce their numbers back down to around $\mathcal{C}$, allowing them to efficiently infect the remaining receivers. This reduction relies on a pair of distributed estimation subroutines, interesting in their own right. The first routine, which requires at least $\log n$ channels, gains efficiency by moving estimation from the time domain to the channel domain. In more detail, it uses $\log n$ channels to estimate the number of relayers in a single round instead of using $\log n$ rounds to estimate the relayers using a

single channel. The second routine, which works for small number of channels (and is slightly slower), uses the source to emulate collision detection, allowing the relayers to run an efficient estimation routine on a single (changing) channel.

Related Work. The SSB problem was introduced in [19], which described an algorithm that delivers $k \geq 1$ messages in $O\left(\frac{\mathcal{C}^2}{\mathcal{C}-t} k \log n\right)$ rounds.[1] In [18,20], erasure coding on the packets, and more clever use of the channels when t was small, improved the result to $O\left(\frac{\mathcal{C}t}{\mathcal{C}-t}(\log n + k)\right)$ rounds. In this paper, we focus on the case where $k = 1$, yielding $O\left(\frac{\mathcal{C}t}{\mathcal{C}-t} \log n\right)$ as the most relevant comparable result. It was recently suggested in [21] that relaying could be used to speed up SSB. However, the algorithms presented in [21] make strong assumptions. The algorithm described here uses the general idea of relaying from [21], but differs in essentially all other details. Concurrent to the series of systems papers cited above, another series looked at solving SSB using uncoordinated spread spectrum coding techniques [9–12,16]. This approach is less immediately applicable as it requires modifications to the radio.

On the theory side, the shared spectrum model we use to study the SSB problem, sometimes called the *t-disrupted model*, has been previously used to study all-to-all gossip [5–7], pairwise node discovery [13], leader election [2,3], and multihop broadcast (with collision detection) [4]. In [14], it was shown how to simulate a reliable channel in this shared spectrum setting, simplifying the development and analysis of reliable algorithms (though often at the cost of added time complexity). We underscore that the t-disrupted model does a good job of describing real shared spectrum networks by noting that it is cited in some of the systems SSB papers [19,20].

The broadcast problem is well-studied in the the classical wireless model where devices shares a single dedicated undisrupted channel. The seminal result of Bar-Yehuda et al. [1], for example, solves broadcast in $O((D + \log n) \log n)$ for a multihop network of D hops, which is (near) optimal. The single hop broadcast problem, however, is trivially solved in this undisrupted setting, as the distinguished source can broadcast without disruption or contention. In our shared spectrum model, the broadcast algorithm of [4] can be adapted to solve our problem in $O\left(\frac{\mathcal{C}}{\mathcal{C}-t} t \log n \log (n/t)\right)$ rounds, assuming collision detectors and sufficiently large $\mathcal{C}$. This solution, even though it uses collision detectors (which our algorithms do not), is still a factor of t slower than ours under comparable conditions.

2 Model and Problem

We model a single hop synchronous wireless network consisting of $\mathcal{C} \geq 1$ communication channels and $n \geq 2$ devices. We assume each device runs the same randomized *algorithm*, and we refer to each individual device executing this algorithm as a *process*. All processes start an execution together in round 1. In each round, each process i chooses a single channel $c \in \{1, ..., \mathcal{C}\}$ on which to participate. Concurrently, an abstract *interference adversary* chooses up to t, $1 \leq t < \mathcal{C}$, channels to *disrupt*.

[1] The existing SSB papers cited here are from the systems literature and therefore do not analyze the time complexity of their algorithms asymptotically, in the way that is standard for the theory literature. We calculated the time complexities shown here based on how their algorithms would perform in our formal model with our parameters.

We model this adversary as an arbitrary randomized algorithm that receives no inputs during an execution. Therefore, its disruption strategy *can* be based on the algorithm executed by the processes (if, for example, processes hard-code a frequency hopping pattern in their definition, the adversary can disrupt that pattern). On the other hand, it *cannot* base its disruption on the random bits used by processes during the execution or the content of messages sent. This captures the reality of disruption in shared spectrum networks which tends to fall somewhere between random and malicious.[2]

A process i participating on channel c during round r receives a message m if and only if: (1) c is not disrupted in r; and (2) only one process broadcasts on c during r, and it broadcasts m (i.e., concurrent broadcasts on a channel leads to collision). To make our upper bound as strong as possible, we assume processes cannot distinguish collisions, disruption, and silence. We assume that processes know t and a polynomial upper bound on n, denoted N, but not n itself.

Formally, the SSB problem assumes a single *source* with a message to send to the remaining processes, which we call *receivers*. We say an algorithm solves the SSB problem in $f(n, \mathcal{C}, t)$ rounds if it guarantees that the source delivers the message to all receivers in $f(n, \mathcal{C}, t)$ rounds, with high probability, i.e., probability at least $1 - 1/n$.

3 Upper Bounds

In this section, we describe and analyze a pair of SSB algorithms, called *pandemic broadcast algorithm 1* (PBA1) and *pandemic broadcast algorithm 2* (PBA2). We use PBA1 when $\mathcal{C} \geq \log N$ and PBA2 when $\mathcal{C} < \log N$, where N is the aforementioned polynomial upper bound on n. In most real shared spectrum networks, $\mathcal{C}$ will be typically larger than $\log N$, as such PBA2 is presented mainly for completeness.[3] We begin, in Section 3.1, by describing these algorithms for the case where $t \leq 0.05 \times \mathcal{C}$, which we call the *low-disruption* regime. Later, in Section 3.2, we show how to simulate these protocols, with an overhead factor of $\frac{\mathcal{C}}{\mathcal{C}-t}$, for the case where $t > 0.05 \times \mathcal{C}$, which we call the *high-disruption* regime.

3.1 Low-Disruption Regime

We begin by studying the case where no more than a constant fraction of the channels can be disrupted concurrently. That is, in this section we assume $t \leq 0.05 \times \mathcal{C}$. It follows from this assumption that $\mathcal{C} \geq 20t$. Our algorithms only need the first $20t$ channels so we assume without loss of generality that $\mathcal{C} = 20t$. (Notice, there is nothing special about the constant 0.05—or the other constants used in our upper bounds. We fix specific values only to gain concreteness in the analyses that follow.)

[2] For example, imagine your network is running a MAC protocol with a hard-coded frequency hopping pattern. If an unrelated network nearby happens to run the same MAC protocol, it will end up generating highly-correlated interference. This is more damaging than random interference, but at the same time is not literally malicious.

[3] For example, Bluetooth divides the 2.4 GHz shared spectrum network into 79 channels. Unless N is an exceptionally large overestimate, we can assume that $\mathcal{C} \geq \log N$ for such a configuration.

Algorithm 1. One phase of Pandemic Broadcast Algorithm Prototype, run @ process u

▷ *odd* rounds

1: select a channel uniformly at random, out of the first $20t$ channels.
2: **if** $u ==$source **then**
3: BROADCAST(m)
4: **else**
5: LISTEN
6: **if** received m **then** $broadcaster_u \leftarrow true$

▷ *even* rounds

7: select a channel uniformly at random, out of the first $20t$ channels.
8: **if** $broadcaster_u$ **then**
9: **with probability** 0.2 **do** BROADCAST(m)**, otherwise** LISTEN
10: **else**
11: LISTEN
12: **if** received m **then** $broadcaster_u \leftarrow true$

To aid intuition, we begin by explaining a simplified SSB algorithm that we call the *pandemic broadcast prototype* (PBP). This protocol assumes that $\mathcal{C} \geq n/2$ (a strong assumption). We present it for the sake of exposition, as this strong assumption removes several difficulties faced by the more general setting. Once we explain this algorithm we move on to describing and analyzing our main upper bound results, PBA1 and PBA2, which we present as generalizations of the prototype. As mentioned, we will generalize these algorithms to work for more disruption (i.e., larger t) in Section 3.2.

Pandemic Broadcast Prototype. The PBP algorithm (detailed in Algorithm 1), works as follows. In each round, each process is either a *broadcaster* or a *receiver*. Initially, the source is the only broadcaster, but as processes receive the message from a broadcaster, they too become broadcasters. The algorithm divides rounds into *phases*, each consisting of two rounds. In all rounds, all processes choose their channels with uniform independent randomness, out of the first $20t$ channels. In the first round of a phase, only the source broadcasts. In the second round of a phase, each broadcaster decides to broadcast with independent probability 0.2. We prove the following:

Theorem 1. *If $t \leq 0.05 \times \mathcal{C}$ and $\frac{n}{2} \leq \mathcal{C}$, then the pandemic broadcast prototype algorithm solves the SSB problem in $O\left(\frac{t}{n} \log n\right)$ rounds.*

Proof. We consider the execution in three stages. For the first stage, we focus on the first round of the phase, in which only the source broadcasts. As long as at least $\frac{n}{2}$ processes remain as receivers, the probability, p_r, that at least 1 receiver chooses the same channel as the source is bounded as: $1 - (1 - \frac{1}{\mathcal{C}})^{n/2} \geq 1 - e^{-n/2\mathcal{C}}$. Given that $\frac{n}{2} \leq \mathcal{C}$, we have $\frac{n}{\mathcal{C}} \leq 2$. It follows that the above probability is at least $\frac{n}{4\mathcal{C}}$. Now, the channel chosen by source is disrupted with probability at most $\frac{t}{\mathcal{C}} = \frac{1}{20}$. Therefore, in each odd round, with probability at least $\frac{n}{5\mathcal{C}}$, at least one receiver receives the message from the source. Hence, using a Chernoff bound, we get that after $O(\frac{\mathcal{C}}{n} \log n) = O(\frac{t}{n} \log n)$ rounds, number of the broadcasters is $\Omega(\log n)$.

For the second stage, we focus on the even rounds. For this, we prove that as long as number of broadcasters is less than $n/2$, in every $\Theta(\frac{t}{n})$ rounds, the number of the broadcasters doubles with high probability. This, proves that after $O(\frac{t}{n} \log n)$ rounds, the number of broadcasters is at least $\frac{n}{2}$. To this end, consider an arbitrary even round

r and suppose that the set of broadcasters at this round is B_r. For this round, we call a channel *active* if at least one broadcaster selects it. We first show that, with high probability, the number of active channels is at least $\frac{|B_r|}{4}$.

For each channel j, the probability that channel j is *active* is $p^j_{active} = 1 - (1 - \frac{1}{\mathcal{C}})^{|B_r|} \geq 1 - e^{-\frac{|B_r|}{\mathcal{C}}}$. Since $\frac{|B_r|}{\mathcal{C}} \leq \frac{n}{\mathcal{C}} \leq 2$, we get that $p^j_{active} \geq \frac{|B|}{3\mathcal{C}}$. Hence, overall, the expected number of active channels is at least $\frac{|B_r|}{3}$. Moreover, note that for any two channels j_1 and j_2, the two events of respectively j_1 or j_2 being active are negatively correlated. This is because, for instance, given that j_1 is not active, the number of broadcaster distributed over other channels goes up which means that the probability that j_2 is active increases. One can easily generalize this argument and see that for any subset S of channels, the probability of the event that all the channels in S are active together is at most equal to the product of the probabilities of each of the channels in S being active. Hence, because of this generalized form of negative-correlation, Chernoff bound holds in this case [15]. Therefore, since $|B_r| = \Theta(\log n)$, we can use the Chernoff bound and infer that, with high probability, at least $\frac{|B_r|}{4}$ channels are active.

Next, let us call a channel *promising* if it is active and the number of broadcasters which chose it is at most 5. Using a simple pigeon-hole principle, we get that at most $\frac{|B_r|}{5}$ active channels have more than 4 broadcasters. Thus, at least $\frac{|B_r|}{20}$ channels are promising, with high probability.

Now we say a channel is *good* if (a) exactly one broadcaster transmits on it, (b) it is not disrupted, (c) it has at least one receiver process listening to it. For a promising channel j that has $b_j \in [1, 5]$ broadcasters on it, we have: First, the probability that (a) holds, is at least $\frac{1}{5}(1 - \frac{1}{5})^4 > 0.08$. Second, the probability that (b) holds is at least $\frac{\mathcal{C}-t}{t} \geq 0.95$. Finally, the probability that (c) holds is at least $1 - (1 - \frac{1}{\mathcal{C}})^{n-|B_r|} \geq 1 - e^{-\frac{n-|B_r|}{\mathcal{C}}} \geq 1 - e^{-\frac{n/2}{\mathcal{C}}} \geq \frac{n}{4\mathcal{C}}$. Thus, we conclude that every promising channel is *good* with probability at least $\Omega(\frac{n}{\mathcal{C}})$. Moreover, we again have the generalized form of negative correlation. For instance, for two promising channels j_1 and j_2, the events of them being *good* are negatively correlated. This is because, for example if j_1 is not good, then we know that it lacks at least one of properties (a) to (c). But for j_2, (a) holds with probability at least 0.1, independent of what happens on j_1, and also j_1 lacking (b) or (c) just makes j_2 more likely to have (b) or (c), respectively.

Since there are at least $\frac{|B_r|}{20}$ promising channels in round r, we get that, in round r, the expected number of *good* channels is at least $\Theta(\frac{n|B_r|}{\mathcal{C}})$. Similarly, since the number of broadcasters is non-decreasing, we see that in the $\Theta(\frac{\mathcal{C}}{n}) = \Theta(\frac{t}{n})$ even rounds starting with round r, the expected number of good channels is at least $2|B_r|$. Using a Chernoff bound again, which holds because of the aforementioned generalized version of negative correlation, and since $|B_r| = \Omega(\log n)$, we get that after $\Theta(\frac{t}{n})$ rounds, at least $|B_r|$ new broadcasters are recruited, with high probability. In other words, with high probability, the number of broadcasters at least doubles in every $\Theta(\frac{t}{n})$ rounds. This completes the proof of the second stage.

Thus far, we have settled the cases of stages 1 and 2 and we know that after $O(\frac{t}{n} \log n)$ rounds, the number of broadcasters is at least $\frac{n}{2}$. For the third stage, consider an arbitrary process v that remains a receiver by the end of second stage. We show that in $O(\frac{t}{n} \log n)$ rounds after the second stage, v gets the message m with high probability.

For this, similar to above we see that in each even round of third stage, at least $\frac{n}{40}$ channels are promising. Now in each such round, v chooses a channel at random. Thus the probability that this channel is (i) promising, (ii) has exactly one broadcaster, and (iii) is not disrupted is at least $\frac{n/40}{\mathcal{C}} \times \frac{1}{5}(1-\frac{1}{5})^4 \times \frac{\mathcal{C}-t}{\mathcal{C}} = \Theta(\frac{n}{\mathcal{C}}) = \Theta(\frac{n}{t})$. Thus, in $O(\frac{t}{n}\log n)$ even rounds, v has received the message m with high probability. Hence, by a union bound, by that time all the nodes have received it with high probability.

Generalizing the Prototype. We begin by asking what happens when we run PBP for $\mathcal{C} < n/2$. The first stage from our analysis still works, and in $O(\log n)$ rounds, we recruit $\Omega(\log n)$ receivers. In the second stage, however, the doubling process stops when the number of broadcasters passes $\mathcal{C}$, at which point they might start causing collisions, slowing down future recruitment. This creates problems as now, in the third stage of the analysis, the proof breaks down due to this contention.

To solve this problem, it would be sufficient to provide the broadcasters an estimate $\tilde{B}$ of $|B|$, as they could then reduce their broadcast probability to minimize collisions, regardless of their numbers (i.e., reducing down to around $\mathcal{C}$ broadcasters is optimal, as this allows a constant number per channel). An easy way to determine $\tilde{B}$ is to try $\log n$ exponentially growing guesses, one of which would be close to the actual size of B. This approach, however, has a slow-down factor of $\Theta(\log n)$, resulting in a $\Theta(\log^2 n)$ factor in the time complexity—which is too slow.

To avoid this overhead we need more efficient estimation routines. In the next two sections, *we present two algorithms that generalize PBP by implementing efficient broadcaster estimation routines: PBA1 and PBA2.* The PBA1 algorithm assumes $\mathcal{C} > \log n$ and leverages this channel diversity to gain efficiency. The PBA2 algorithm, by contrast, has fewer channels to work with. It leverages the presence of a distinguished source (which breaks symmetry in an important way) to achieve an estimation that is slower than PBA1, but still faster than the $\log n$ overhead of our simple suggestion from above.

Pandemic Broadcast Algorithm 1. As mentioned, the PBA1 algorithm, detailed in Algorithm 2, can be understood as a generalization of PBP. In more detail, we now increase the size of a phase to include the following 5 rounds: The first two rounds are the same as in PBP. In the next two rounds, broadcasters find an estimate of $|B|$ that is in $[\frac{|B|}{4}, 4|B|]$, with at least a nonzero constant probability (described below). We are able to accomplish this in only 2 rounds by moving guesses from $\log n$ consecutive rounds to $\log n$ channels during the same round. In the final round, broadcasters sub-sample themselves using this estimate and then, similar to the second round, broadcast in uniformly chosen channels. We show that $O(\lceil \frac{t}{n} \rceil \log n)$ phases are enough for delivering the message to every process, yielding the total complexity of $O(\lceil \frac{t}{n} \rceil \log n)$ rounds.

The core novelty of PBA1, therefore, is the 2-round estimation subroutine. This routine consists of a *test* and a *report* segment. In the test segment, each broadcaster v chooses one of the channels using an exponential probability distribution. Then broadcaster v, having picked channel f, decides to transmit or listen with probability 0.5. In this *test* segment, any broadcaster that listens to a channel f and receives a message on that channel, estimates $|B|$ to be 2^{f+1}. In the report segment, all broadcasters choose

Algorithm 2. Pandemic Broadcast Algorithm 1, run @ broadcaster process u

for $phase = 1$ to $\Theta(\log n)$ **do**

  select a channel uniformly at random, out of the first $20t$ channels. ▷ **source's broadcast round**
  if $u ==$source **then**
    BROADCAST(m)
  else
    LISTEN
    if received m **then** $broadcaster_u \leftarrow true$

  select a channel uniformly at random, out of the first $20t$ channels. ▷ **simple relaying round**
  if $broadcaster_u$ **then**
    with probability 0.2 **do** BROADCAST(m)**, otherwise** LISTEN
  else
    LISTEN
    if received m **then** $broadcaster_u \leftarrow true$

  if $broadcaster_u$ **then**
    $est \leftarrow \mathcal{C}$; $estimationFlag \leftarrow false$ ▷ **test segment of estimation**
    select random channel f (of source) from an exponential probability distribution
    with probability 0.5 **do** BROADCAST(m) **, otherwise** LISTEN
    if received message m **then**
      $est \leftarrow 2^{f+1}$; $estimationFlag \leftarrow true$
    select channel 1 (of source). ▷ **report segment of estimation**
    if $estimationFlag$ **then**
      with probability 0.05 **do** BROADCAST(est)**, otherwise** LISTEN
    else
      LISTEN
  else ▷ for receivers to keep them in synch with broadcasters
    select a channel uniformly at random, out of the first $20t$ channels.
    LISTEN
    LISTEN

  select a channel uniformly at random, out of the first $20t$ channels. ▷ **final relaying round**
  if $broadcaster_u$ **then**
    with probability $\min\{\frac{\mathcal{C}}{est}, 0.2\}$ **do** BROADCAST(m)**, otherwise** LISTEN
  else
    LISTEN
    if received m **then** $broadcaster_u \leftarrow true$

the same channel, and every broadcaster that made an estimate in the previous segment broadcasts their estimate. Any broadcaster that receives such an estimate adopts it as their est of $|B|$. If a broadcaster learns no estimate, it uses the default value of $\mathcal{C}$.

Intuitively, if $|B| = 2^{f+1}$, then we expect a constant number of processes to choose f, leading to a constant probability of a single process broadcasting and a small number receiving its messages. Because the channel selection probabilities grow exponentially, we can show that the probability that the same happens on other frequencies, sums to a constant. Therefore, with a constant probability, we have a single process reporting an estimate, and the estimate is correct.

A wrinkle here is that the adversary might choose to consistently jam the channel corresponding to the right estimate, or it might jam the reporting channel. To avoid this, we recall that only broadcasters participate in this estimate. Therefore, all participants have received a message from the source. We assume this message can contain sufficiently many bits (or a seed to a pseudo-random number generator) so that in each round of the estimation routine, the broadcasters can shift the channels (circular-shift) by a random

amount, unknown to the adversary. In the pseudo-code of Algorithm 2, in the estimation rounds, broadcasters choose their channels using the random shift provided by source in the initial message.[4] Therefore, the probability that these key channels are disrupted is the same as that of a random channel being disrupted. Formally, we prove:

Theorem 2. *If $t \leq 0.05 \times \mathcal{C}$, and $\log n \leq \mathcal{C}$, then the pandemic broadcast algorithm 1 solves the SSB problem $O(\lceil \frac{t}{n} \rceil \cdot \log n)$ rounds.*

Proof (Proof Outline). The analysis of the case where $\mathcal{C} \geq \frac{n}{2}$ is as done in Theorem 1, by focusing only on the first two rounds of each phase, and noticing that in that case, each phase has only 5 rounds. For this case, we proved time complexity of $O(\frac{t}{n} \cdot \log n)$ rounds in Theorem 1. On the other hand, for the case where $\mathcal{C} \in [\log n, \frac{n}{2}]$, we show that in $O(\log n)$ rounds, the message is delivered to every node. For this, we prove the lemmas 1, 2, and 3. Please see the full paper for the proofs. We remark that, the main change, where the effect of estimation part comes in, is studied in Lemma 3.

Lemma 1. *If $t \leq 0.05 \times \mathcal{C}$ and $\mathcal{C} \in [\log n, \frac{n}{2}]$, after $O(\log n)$ phases of PBA1, the number of broadcasters is at least $\Theta(\log n)$.*

Lemma 2. *If $t \leq 0.05 \times \mathcal{C}$ and $\mathcal{C} \in [\log n, \frac{n}{2}]$, after $O(\log n)$ phases of PBA1, the number of broadcasters is at least $\mathcal{C}$.*

Lemma 3. *If $t \leq 0.05 \times \mathcal{C}$ and $\mathcal{C} \in [\log n, \frac{n}{2}]$, after $O(\log n)$ phases of PBA1, all receivers receive the message m, with high probability.*

Pandemic Broadcast Algorithm 2. If $\mathcal{C} \leq \log n$, we no longer can assume that we have at least $\log n$ channels and this prevents us from running the 2-round estimation routine of PBA1. In this case, we use Pandemic Broadcast Algorithm 2 (PBA2). This algorithm is divided into three explicit parts. In the first part, we grow the number of broadcasters to at least $\mathcal{C}$, in $\Theta(\log n)$ rounds, using the PBP strategy. We then stop the recruitment and move on to the second part, where we run a new estimation subroutine that estimates the number of recruited broadcasters to within a factor of 2, with high probability, in $O(\log n \cdot \log \log n)$ rounds (detailed below). In the final part, we use this estimate to sub-sample the broadcasters, by having each broadcaster now broadcast with probability $\min\{\frac{\mathcal{C}}{est}, 0.2\}$. This part runs for $\Theta(\log n)$ rounds, by the end of which the remaining processes have all received the message with high probability. Formally, this gives us the following:

Theorem 3. *If $t \leq 0.05 \times \mathcal{C}$ and $\mathcal{C} \leq \frac{n}{2}$, then the pandemic broadcast algorithm 2 solves the SSB problem in $O(\log n \cdot \log \log n)$ rounds.*

Returning to the algorithm details, notice that the first part of PBA2 is the same as running $\Theta(\log n)$ phases of PBP, and thus its correctness follows from Lemmas 1 and 2. Similarly, the third part is similar to running PBP with the addition that in the second

[4] Because $O(\log n)$ rounds are sufficient to solve the problem, this would require $O(\log n \log t)$ total bits, which could fit under the standard assumption of messages holding a polylogarithmic number of bits.

round of each phase, broadcasters decide to transmit with probability $\min\{\frac{\mathcal{C}}{est}, 0.2\}$, where est is the estimate of $|B|$ to within a factor of 2. An argument similar to that given in proof of Lemma 3 establishes that $\Theta(\log n)$ rounds are sufficient for all remaining receivers to become broadcasters. Therefore, we are left to describe and analyze the estimation part of the algorithm—which is where we turn our attention next.

Estimation Part of PBA2: We change the role of some broadcasters to *mirrors* using the following rule: the processes which received the message in the first $\Theta(\log n)$ rounds of part 1 become *mirrors* while the other processes which received the message remain *broadcasters*[5]. The constants in the asymptotic notations are selected such that, with high probability, at least one mirror exists and there are at least $\frac{\mathcal{C}}{2}$ broadcasters. The core idea is to estimate the number of broadcasters using the help of mirrors.

Theorem 4. *If $t \leq 0.05 \times \mathcal{C}$ and $\mathcal{C} \leq \frac{n}{2}$, and there is at least one mirror, then Estimation Algorithm (presented in Algorithm 4) produces a 2-approximation for the number of broadcasters, with high probability and in $O(\log n \cdot \log\log n)$ rounds.*

In the Estimation Algorithm, our basic tool is a simple probabilistic comparison method called *ApproxCompare*, which compares the number of broadcasters with a given threshold X and outputs a response in form '*larger*' or '*smaller*'. We say that the output is *correct* in case it is 'larger' if the number of broadcasters is greater than $1.4X$, and in case it is 'smaller' if if the number of broadcasters is less than $X/1.4$. On the other hand, if the number of broadcaster nodes is in $[X/1.4,\ 1.4X]$, we do not expect any guarantee from the output. Next, we explain this comparison method and show that its output is *correct* with high probability. But before that, let us finish the story of the estimation algorithm considering a black-box algorithm ApproxCompare. To get a 2-estimation, it is enough to compare the number of broadcasters with thresholds 2^i for $i \in [\log N]$, using ApproxCompare. Moreover, we can use a binary search over these thresholds, to speed up this process. The pseudo-code presented in Algorithm 4 realizes this idea, with some special care about anomalies possible due to incomplete guarantee of the aforementioned definition of correct output. The related correctness and the time-complexity are studied in the proof of Theorem 4.

ApproxCompare Algorithm: As presented in Algorithm 3, *ApproxCompare* procedure is comprised of $200 \log n$ phases. Throughout these phases, both mirrors and broadcasters always work on channel 1 (of the source). Each broadcaster has a variable $counter$ initially set to zero. In each *phase*, there are two rounds, namely a *test* round and a *report* round. In the test round, the source transmits message m; also each broadcaster transmits message m with probability $p_X = 1 - 2^{-1/X}$ and remains silent otherwise. In this round, mirrors only listen. Then, in the report round, the source again transmits message m. However this time, each mirror that did not receive a message transmits and broadcasters all listen. Each broadcaster that does not receive a message in the report round increments its counter. After all phases are finished, each broadcaster outputs '*larger*' if its counter is more than half of the number of phases, i.e., if it did not receive anything back in the majority of report rounds. Otherwise, it outputs '*smaller*'.

[5] This change remains in effect for the third part as well.

Algorithm 3. ApproxCompare(X) — at process u

```
1:  counter ← 0
2:  for i:=1 to 200 log n do                                   ▷ Θ(log n) phases

3:      if u == source then                                    ▷ Test Round
4:          BROADCAST(m) on randomly shifted channel 1
5:      else if broadcaster_u then
6:          with probability p_X = 1 − 2^{−1/X} do
7:              BROADCAST(m) on channel 1 (of source)
8:          otherwise
9:              LISTEN to channel 1 (of source)
10:     else if mirror_u then
11:         LISTEN

12:     if u == source then                                    ▷ Report Round
13:         BROADCAST(m) on a randomly shifted channel 1
14:     else if mirror_u then
15:         if received a message in test round then
16:             BROADCAST(m) on channel 1 (of source)
17:         else
18:             LISTEN to channel 1 (of source)
19:     else if broadcaster_u then
20:         LISTEN
21:         if did not received a message then counter ← counter + 1

22: if counter ≥ 100 log n then
23:     return 'larger'
24: else
25:     return 'smaller'
```

Algorithm 4. Estimation Algorithm

```
1:  upperLog ← log N
2:  lowerLog ← 0
3:  while upperLog − lowerLog > 1 do

4:      midLog = ⌊(lowerLog + upperLog)/2⌋
5:      res_1 ← ApproxCompare(2^{midLog−1})
6:      res_2 ← ApproxCompare(2^{midLog})
7:      res_3 ← ApproxCompare(2^{midLog+1})

8:      switch (res_1, res_2, res_3) do
9:          case (*, 'smaller', 'smaller')
10:             upperLog ← midLog
11:         case ('larger', 'larger', *)
12:             lowerLog ← midLog
13:         case ('smaller', *, 'larger')
14:             return 2^{midLog}
15:         default case:
16:             return 2^{midLog}

17: return 2^{lowerLog}
```

Lemma 4. *If $t \leq 0.05 \times \mathcal{C}$ and $\mathcal{C} \leq \frac{n}{2}$, and there is at least one mirror, then each call to ApproxCompare procedure gives a* correct *response with high probability.*

Proof. If the number of broadcasters is greater than or equal to $1.4X$, then in the test round of each phase, the probability that at least one broadcaster transmits is $1-(1-p_X)^{|B|} \geq 1-(1-p_X)^{2X} = 1-2^{-1.4} > 0.62$. If in the test round of a given phase, at least one broadcaster transmits, then the transmission of these broadcasters collides with the transmission of the source and thus, mirrors receive no messages. Hence, in the report round of that phase, mirrors all transmit and therefore, once again due to collision with the source's transmission, broadcasters receive no messages. In such a case, broadcasters increment their counter. Hence, we get that if the number of broadcasters is greater than or equal to $1.4X$, in each test, the counter of each broadcaster is incremented with probability at least 0.62. Using Hoeffding's inequality, we can infer that after $200 \log n$ phases, with high probability, the counter of each broadcaster is greater than $100 \log n$ and therefore, response of the approximate comparison is '*larger*'.

On the other hand, if the number of broadcasters is less than or equal to $X/1.4$, then in the test round of each phase, the probability that no broadcaster transmits is $(1-p_X)^{|B|} \geq (1-p_X)^{X/(2)} = 2^{-1/1.4} > 0.60$. If in the test round of a given phase, no broadcaster transmits, then in that round, the mirrors receive the transmission of the source with probability at least $\frac{\mathcal{C}-t}{\mathcal{C}} \geq 0.95$. In that case, in the report round, no mirrors transmits and therefore, broadcasters receive the transmission of the source again with probability at least $\frac{\mathcal{C}-t}{\mathcal{C}} \geq 0.95$. If all of these events happen, broadcasters do not increment their counter. Hence, we get that if the number of broadcasters is less than or equal to $X/1.4$, in each test, the probability that counter of each broadcaster is incremented is at most $1-0.6 \times 0.95 \times 0.95 < 0.45$. Using Hoeffding's inequality, we can infer that after $200 \log n$ phases, with high probability, the counter of each broadcaster is less than $100 \log n$ and therefore, response of the approximate comparison is '*smaller*'.

Proof (Proof of Theorem 4). First, for the time-complexity analysis, notice that there are $\log n$ comparison thresholds and therefore, the binary search over these threshold values as presented in Algorithm 4 requires just $O(\log \log n)$ comparisons. Since each comparison takes $\Theta(\log n)$ rounds, the total time complexity becomes $\Theta(\log n \cdot \log \log n)$.

Now, we analyse the correctness. Consider an arbitrary turn of the while loop in Algorithm 4. We make three calls to ApproxCompare to take into account the fact that when the number of broadcasters is within a 1.4 factor of the comparison threshold, we do not get any guarantee from Lemma 4. Since $1.4^2 < 2$, at most only one of the three thresholds $2^{midLog-1}$, 2^{midLog}, $2^{midLog+1}$ is within 1.4 factor of the number of broadcasters. Thus, noting Lemma 4, we know that with high probability, the output of at most one of the three calls to ApproxCompare in this turn is not true. The case is clear if all the three responses are true. If only the response of the comparison to $2^{midLog-1}$ is not true, then we know that the number of broadcasters is within a 1.4 factor of $2^{midLog-1}$ and thus, less than 2^{midLog}. In this case, we get a response of 'smaller' from the other two comparisons and therefore, following case presented in line 9 of Algorithm 4, the binary search moves in the correct direction. Similarly, if only the response of the comparison to $2^{midLog+1}$ is not true, then we know that the number of broadcasters is within a 1.4 factor of $2^{midLog+1}$ and thus, greater than 2^{midLog}.

In this case, we get response of 'larger' from the other two comparisons and therefore, following case presented in line 11 of Algorithm 4, the binary search moves in the correct direction. In the last case, if the response to the comparison to 2^{midLog} is not true, then the number of broadcasters is within a 1.4 factor of 2^{midLog}. In this case, Algorithm 4 returns 2^{midLog} as the final estimation, which is clearly a 2-factor estimation. Finally, we know from Lemma 4 that with high probability, no other case happens.

3.2 High-Disruption Regime

In Section 3.1, we presented the PBA1 and PBA2 algorithms, which work when t is not too large compared to $\mathcal{C}$. Here we generalize for any $t < \mathcal{C}$.

Our approach is to use $\Theta\left(\frac{\mathcal{C}}{\mathcal{C}-t}\right)$ rounds to simulate one *abstract* round of PBA1 or PBA2 (in particular, $\frac{3\mathcal{C}}{\mathcal{C}-t}$ rounds will prove sufficient). To simulate abstract round r of one of these low-disruption algorithms, we first let broadcasters choose their channel, and whether or not they broadcast, according to logic of the respective algorithm. They then use *these same fixed choices* for the $\Theta\left(\frac{\mathcal{C}}{\mathcal{C}-t}\right)$ simulation rounds that follow. In each of these simulation rounds, these broadcasters permute their channels using the common random bits from the source message. Therefore, all broadcasters that choose the same channel, will be on the same channel for all $\Theta\left(\frac{\mathcal{C}}{\mathcal{C}-t}\right)$ rounds, but this channel will randomly change from round to round. The receivers can continue to choose random channels on which to receive, throughout this period.

We now prove that this simulation strategy allows PBA1 and PBA2 to work in the high-disruption setting at the cost of slow down factor $\frac{\mathcal{C}}{\mathcal{C}-t}$.

Theorem 5. *The pandemic broadcast algorithm 2, augmented with the simulation strategy, solves the SSB problem in* $O(\frac{\mathcal{C}}{\mathcal{C}-t}\log n \cdot \log\log n\})$ *rounds, for* $\mathcal{C} \leq \log n$ *and* $t < \mathcal{C}$*; the pandemic broadcast algorithm 1, augemented with the simulation strategy, solves the SSB problem in* $O(\frac{\mathcal{C}}{\mathcal{C}-t}\lceil\frac{t}{n}\rceil \log n)$ *rounds, for* $\log n \leq \mathcal{C}$ *and* $1 \leq t < \mathcal{C}$.

Proof. Consider abstract round r of one of these SSB algorithms augmented with the simulation strategy. We fix the broadcasters channel and broadcast choices at the beginning of this abstract round, and stick with these choices for the simulation rounds that follow. Assume that these fixed choices include a channel c with either less than or more than 1 broadcaster. In the low-disruption setting, no process would receive a message on c. Notice that the same holds here for the $\Theta\left(\frac{\mathcal{C}}{\mathcal{C}-t}\right)$ channels mapped to c throughout the simulation rounds.

Now assume that these fixed choices include a channel c with exactly one broadcaster. It follows that in our simulation of this abstract round, there will be at least one simulation round where the channel mapped to c is undisrupted, with constant probability. This follows because in each such round, c is disrupted with probability at most $\frac{t}{\mathcal{C}}$. Therefore we experience at least one undisrupted simulation round with probability at least $1-(\frac{t}{\mathcal{C}})^{\frac{\mathcal{C}}{3\mathcal{C}-t}} = 1-(1-\frac{\mathcal{C}-t}{\mathcal{C}})^{\frac{3\mathcal{C}}{\mathcal{C}-t}} \geq 1-\frac{1}{e^3} > 0.95$.

We are, therefore, in effect simulating our low-disruption algorithms in a new type of network model where the adversary disrupts each channel with some independent disruption probability of no more than 0.05. Though PBA1 and PBA2 were originally

analyzed in a model with an arbitrary adversary that jams up to $t \leq 0.05 \times \mathcal{C}$ channels, it is easy to verify that the same arguments work in the more well-behaved model simulated here, in which we fix the interference adversary to choose its $t \leq 0.05 \times \mathcal{C}$ channels randomly. Therefore, the same correctness holds under the same conditions, in exchange for the slow down factor of $\Theta\left(\frac{\mathcal{C}}{\mathcal{C}-t}\right)$ caused by the simulation rounds.

4 Lower Bounds

We now present our lower bounds. For the case where $t = O(n)$, we can prove our solution optimal (within $\log \log n$ factors). This bound focuses on showing that it takes a while for the source to choose a non-disrupted channel (clearly, broadcast cannot complete before the source lands on a non-disrupted channel for the first time). For larger t, the task gets more difficult. In this case, the $\lceil \frac{t}{n} \rceil$ term in our time complexity becomes relevant. Proving this term necessary requires that we bound the behavior of the receivers—a difficult task because they can potentially coordinate in advance of receiving the source message, creating dependencies that thwart straightforward lower bound arguments. Below, we present the lower bound for this difficult case under the assumptions that we are in the low disruption regime and using *regular* algorithms [2,3]: An SSB algorithm is called *regular* if for each process u, there exists a probability distribution π_u over the channels, i.e., $\pi_u : \{1, 2, \ldots, \mathcal{C}\} \to [0, 1]$, such that the following holds: as long as u has not received the message, in each round r, process u does not transmit and moreover, it selects the channel to which it listens to using the distribution π_u. Once u receives the message, its behavior is no longer restricted. Note that all our algorithms satisfy this *regularity* assumption. It is unclear whether it is the upper or lower bound that would improve in the absence of these properties. We leave that question as interesting future work. Please see the full paper for the proof.

Theorem 6. *Every solution to the SSB problem requires* $\Omega\left(\frac{\mathcal{C}}{\mathcal{C}-t} \log n\right)$ *rounds. In the low disruption regime, every* regular *algorithm for the SSB problem also requires* $\Omega\left(\frac{\mathcal{C}}{\mathcal{C}-t} \lceil \frac{t}{n} \rceil \log n\right)$ *rounds.*

References

1. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. Journal of Computer and System Sciences 45(1), 104–126 (1992)
2. Daum, S., Gilbert, S., Kuhn, F., Newport, C.: Leader Election in Shared Spectrum Radio Networks. In: Proceedings of the International Symposium on Principles of Distributed Computing (2012)
3. Dolev, S., Gilbert, S., Guerraoui, R., Kuhn, F., Newport, C.: The wireless synchronization problem. In: Proc. 28th Symp. on Principles of Distributed Computing, PODC, pp. 190–199 (2009)
4. Dolev, S., Gilbert, S., Khabbazian, M., Newport, C.: Leveraging Channel Diversity to Gain Efficiency and Robustness for Wireless Broadcast. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 252–267. Springer, Heidelberg (2011)

5. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Gossiping in a Multi-channel Radio Network: An Oblivious Approach to Coping with Malicious Interference (Extended Abstract). In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 208–222. Springer, Heidelberg (2007)
6. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Secure Communication Over Radio Channels. In: Proceedings of the International Symposium on Principles of Distributed Computing (2008)
7. Gilbert, S., Guerraoui, R., Kowalski, D., Newport, C.: Interference-Resilient Information Exchange. In: The Proceedings of the Conference on Computer Communication (2009)
8. Gummadi, R., Wetherall, D., Greenstein, B., Seshan, S.: Understanding and Mitigating the Impact of RF Interference on 802.11 Networks. In: Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM, pp. 385–396 (2007)
9. Jin, T., Noubir, G., Thapa, B.: Zero Pre-Shared Secret Key Establishment in the Presence of Jammers. In: Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (2009)
10. Liu, A., Ning, P., Dai, H., Liu, Y.: USD-FH: Jamming-Resistant Wireless Communication using Frequency Hopping with Uncoordinated Seed Disclosure. In: Proceedings of the IEEE International Conference on Mobile Ad Hoc and Sensor Systems (2010)
11. Liu, A., Ning, P., Dai, H., Liu, Y., Wang, C.: Defending DSSS-Based Broadcast Communication Against Insider Jammers via Delayed Seed-Disclosure. In: Proceedings of the IEEE Annual Computer Security Applications Conference (2010)
12. Liu, Y., Ning, P., Dai, H., Liu, A.: Randomized Differential DSSS: Jamming-Resistant Wireless Broadcast Communication. In: The Proceedings of the Conference on Computer Communication (2010)
13. Meier, D., Pignolet, Y.A., Schmid, S., Wattenhofer, R.: Speed Dating Despite Jammers. In: Krishnamachari, B., Suri, S., Heinzelman, W., Mitra, U. (eds.) DCOSS 2009. LNCS, vol. 5516, pp. 1–14. Springer, Heidelberg (2009)
14. Newport, C.: Distributed Computation on Unreliable Radio Channels. Ph.D. thesis. MIT (2009)
15. Panconesi, A., Srinivasan, A.: Randomized distributed edge coloring via an extension of the chernoff–hoeffding bounds. SIAM J. Comput. 26(2), 350–368 (1997), http://dx.doi.org/10.1137/S0097539793250767
16. Pöpper, C., Strasser, M., Čapkun, S.: Jamming-Resistant Broadcast Communication without Shared Keys. In: Proceedings of the USENIX Security Symposium (2009)
17. Popper, C., Strasser, M., Čapkun, S.: Anti-Jamming Broadcast Communication using Uncoordinated Spread Spectrum Techniques. IEEE Journal on Selected Areas in Communications 28(5), 703–715 (2010)
18. Slater, D., Tague, P., Poovendran, R., Matt, B.: A Coding-Theoretic Approach for Efficient Message Verification over Insecure Channels. In: Proceedings of the ACM Conference on Wireless Network Security (2009)
19. Strasser, M., Capkun, S., Popper, C., Čagalj, M.: Jamming-Resistant Key Establishment using Uncoordinated Frequency Hopping. In: IEEE Symposium on Security and Privacy (2008)
20. Strasser, M., Pöpper, C., Čapkun, S.: Efficient Uncoordinated FHSS Anti-Jamming Communication. In: Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (2009)
21. Xiao, L., Dai, H., Ning, P.: Jamming-Resistant Collaborative Broadcast Using Uncoordinated Frequency Hopping. IEEE Transactions on Forensics and Security 7(1), 297–309 (2012)

Attack-Resilient Multitree Data Distribution Topologies

Sascha Grau*

Technische Universität Ilmenau, Germany
sascha.grau@tu-ilmenau.de

Abstract. We consider a scenario of information broadcast where a source node distributes data in parallel over a fixed number of trees spanning over a large audience of nodes. The trees used for data dissemination are called distribution topology. Particular implementations of this scenario are peer-to-peer live streaming systems. Encoding data partially redundant, nodes are satisfied as long as they receive packets in at least a certain portion of trees. Otherwise, they are called *isolated*.

We study distribution topologies limiting the worst-case consequences of attacks suddenly removing nodes from the trees. In particular, we aim to minimize the maximum possible number of isolated nodes for each number of removed nodes. We show necessary conditions on distribution topologies closely approximating this goal. Then, we demonstrate that the attack-resilience of topologies adhering to these conditions is characterized by specific matrices that have to be Orthogonal Arrays of maximum strength. The computational complexity of finding such matrices for arbitrary dimensions is a long-standing research problem. Our results show that finding representatives of the studied distribution topologies is at least as hard as this problem.

Keywords: network topologies, dependability, P2P, orthogonal arrays.

1 Introduction

In many applications data shall be reliably broadcast from a resource-restricted source to a large audience of nodes. Applying multiple description coding [1] or error-correcting codes [2], it is possible to split each block of data into k subblocks, such that the reception of a certain portion of these subblocks already satisfies the participating nodes (i.e. they can restore the original data to satisfactory degree).

Distributing each of the k subblocks from node to node over a distinct tree rooted at the source, a data distribution system is obtained which is tolerant to failures. Furthermore, the number of participants in such a system can scale independently from resource restrictions of the source. Popular implementations of such approaches can be found in peer-to-peer live streaming systems like [3–5].

* This work was supported by the *Deutsche Forschungsgemeinschaft* under grant number KU 658/10-2.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 196–208, 2012.

Due to their spreading application and growing importance, such data distribution systems are target of attacks. Abstracting from technical details, these attacks can often be modeled as a removal of nodes from the system. The consequences of such a removal can be measured as damage and depend on the layout of the distribution topology, i.e., the trees used for data dissemination. This motivated the study of distribution topologies minimizing the maximum damage that is achievable on them.

Here, different measures of damage can be of interest. In the past, distribution topologies minimizing notions of system-wide damage, like the global number of disturbed source-to-node paths, have been identified [3, 6]. However, in many applications a damage measure based on the user-perceived quality of the data distribution service is more relevant. This corresponds to counting the number of nodes that are no longer satisfied since they lost too many paths from the source.

In the following, distribution topologies minimizing this kind of damage are called *attack-resilient*. Despite their relevance, the author is not aware of any analytical study of such topologies or of related network design problems based on a similarly generalized concept of connectivity. Current applications resort to following rules of thumb, as building 'diverse trees' [5]. Some insights for scenarios considering random node removal instead of worst-case attacks were obtained by simulation in [7].

Contribution. In this document, we introduce *forward-stable* distribution topologies and show that they closely approximate attack-resilient topologies in situtations where the number of nodes considerably exceeds the number of source neighbors. This is a usual condition in applications of multitree data distribution topologies. We find necessary and sufficient requirements for forward-stable topologies and show that they can be characterized by matrices representing certain successor relations in the trees. By showing that these matrices have to be Orthogonal Arrays of maximum possible strength, we discover connections to design and coding theory. In particular, we show that the identification of an efficient construction scheme for forward-stable topologies would solve several long-standing open problems in these areas.

Structure of This Document. In Section 2, we specify our system model and formalize the notion of attack-resilient distribution topologies. Section 3 introduces and motivates an alternative damage measure which is then used in Section 4 to define forward-stable distribution topologies. Their properties are studied in-depth in the following Subsections. Finally, Section 5 concludes this document.

2 System Model and Attack-Resilient Topologies

In our system model, a source s distributes data to a set $V = \{1, \ldots, n\}$ of nodes. Each block of data is encoded into k subblocks and a node is satisfied as long as it receives *more* than $k - z$ such subblocks, for a fixed $z \in \{1, \ldots, k\}$ (see [1, 2] for suitable encoding schemes). Otherwise, the node is called *isolated*.

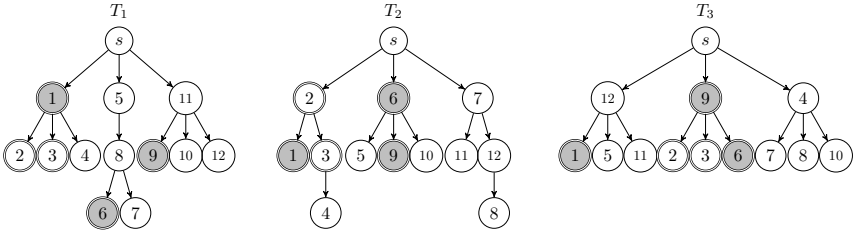


Fig. 1. Attack $X = \{1, 6, 9\}$ on this topology $\mathcal{T} \in \mathbb{T}(12, 3, 3)$ leads to $\mathrm{b}^{\mathcal{T}}(X, 2) = 5$ (attacked nodes gray, isolated nodes double-lined)

Each subblock is distributed over one of k distribution trees (also called *stripes*). Those have node set $\{s\} \cup V$, are rooted at s, and are directed towards the leafs. A *distribution topology* is a k-tupel $\mathcal{T} = (T_1, \ldots, T_k)$ of stripes. The nodes that are adjacent to the source in stripe T_i of $\mathcal{T}$ are the *heads* $H_i^{\mathcal{T}}$. The nodes $H^{\mathcal{T}} = \bigcup_{i \in \{1,\ldots,k\}} H_i^{\mathcal{T}}$ are the *heads of* $\mathcal{T}$.

We assume that the maximum degree of source node s is limited to a value of Ck, for $C \in \mathbb{N}$ and $n \geq Ck$. The class of all distribution topologies with k trees, node set $V = \{1, \ldots, n\}$ and source degree limit Ck is denoted as $\mathbb{T}(n, C, k)$.

The data distribution over a topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ can be disturbed in a number of ways. In this document, we study the consequences of sudden removals of nodes. Such events are common, especially in peer-to-peer systems with their unreliable and vulnerable participants. Considering the worst-case, we assume that the sets of removed nodes are the result of a malicious planning process. For this reason, they are termed as *attacks*. Note that we do not account for a removal of the source node, since it would always result in a non-functional distribution topology. Furthermore, in practical applications it is usual to take special measures to safeguard source functionality.

When a node v is removed from topology $\mathcal{T}$, in each stripe T_i with $i \in \{1, \ldots, k\}$, the paths between s and all nodes in the subtree rooted at v become disturbed. The set of nodes in this subtree is denoted as *successor set* $\mathrm{succ}_i^{\mathcal{T}}(v)$ and contains v. For node sets X, we correspondingly define $\mathrm{succ}_i^{\mathcal{T}}(X) = \bigcup_{v \in X} \mathrm{succ}_i^{\mathcal{T}}(v)$. Figure 2 gives an example.

Assuming that a node is isolated by the loss of at least z paths from the source, the number of nodes isolated by attack X is counted as *damage*

$$\mathrm{b}^{\mathcal{T}}(X, z) := \left| \bigcup_{I \subseteq \{1,\ldots,k\}, |I|=z} \bigcap_{i \in I} \mathrm{succ}_i^{\mathcal{T}}(X) \right|. \tag{1}$$

Figure 1 shows an example in which nodes are isolated by the loss of at least 2 paths from the source.

Given an arbitrary class $\mathbb{T}(n, C, k)$, we are generally interested in finding topologies $\mathcal{T} \in \mathbb{T}(n, C, k)$ minimizing the maximum damage that can occur for every possible number x of removed nodes and every value of threshold z. Note that for $x \geq Cz$, it is possible to remove all heads of z stripes (the ones with

the least number of heads) and isolate all nodes. Hence, the maximum damage on topologies in $\mathbb{T}(n, C, k)$ can only differ for $x < Cz \leq Ck$. This leads to the following definition.

Definition 1. *A topology* $\mathcal{T} \in \mathbb{T}(n, C, k)$ *is* attack-resilient*, if for all* $z \in \{1, \ldots, k\}$*, all* $x \in \{1, \ldots, Ck\}$*, and all* $\mathcal{C} \in \mathbb{T}(n, C, k)$*, it holds that*

$$\max_{X \subseteq V, |X|=x} \mathrm{b}^{\mathcal{T}}(X, z) \leq \max_{X \subseteq V, |X|=x} \mathrm{b}^{\mathcal{C}}(X, z).$$

3 An Approximative Damage Measure

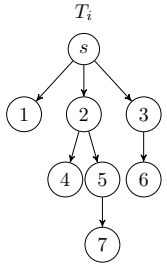
The function $\mathrm{b}^{\mathcal{T}}(X, z)$ used to characterize attack-resilient topologies counts nodes of two different kinds. On the one hand, it considers all removed nodes in the set X. On the other hand, it counts nodes positioned in subtrees below removed nodes in at least z stripes. Furthermore, there are nodes falling into both categories. This superposition of different causes of damage complicates an analyis. For this reason, we choose to study a slightly altered notion of damage. At first, we define the *forward successor set* of a node v in stripe T_i of $\mathcal{T}$:

$$\mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) := \begin{cases} \mathrm{succ}_i^{\mathcal{T}}(v) & \text{, if } |\mathrm{succ}_i^{\mathcal{T}}(v)| > 1 \text{ or } v \in H_i^{\mathcal{T}} \\ \emptyset & \text{, otherwise.} \end{cases} \tag{2}$$

It is equal to the successor set in most cases, but is empty if v is neither head nor has children in T_i. Again, this definition extends to node sets: $\mathrm{succ}_i^{\mathcal{T}\rightarrow}(X) = \bigcup_{v \in X} \mathrm{succ}_i^{\mathcal{T}\rightarrow}(v)$. Figure 2 provides an example.

For $\mathcal{T} \in \mathbb{T}(n, C, k)$, $z \in \{1, \ldots, k\}$, and attacks $X \subseteq V$, we define the corresponding damage function as *forward damage*

$$\mathrm{bf}^{\mathcal{T}}(X, z) := \left| \bigcup_{I \subseteq \{1,\ldots,k\}, |I|=z} \bigcap_{i \in I} \mathrm{succ}_i^{\mathcal{T}\rightarrow}(X) \right|. \tag{3}$$



(a) A tree T_i from a topology $\mathcal{T}$

X	$\mathrm{succ}_i^{\mathcal{T}}(X)$	$\mathrm{succ}_i^{\mathcal{T}\rightarrow}(X)$
$\{1\}$	$\{1\}$	$\{1\}$
$\{3\}$	$\{3, 6\}$	$\{3, 6\}$
$\{4\}$	$\{4\}$	$\emptyset$
$\{3, 5, 6\}$	$\{3, 5, 6, 7\}$	$\{3, 5, 6, 7\}$

(b) Successor and forward successor sets

Fig. 2. Different concepts of successor sets

Since it holds that $\mathrm{succ}_i^{\mathcal{T}}(X) = X \cup \mathrm{succ}_i^{\mathcal{T}\rightarrow}(X)$, we observe that

$$\mathrm{bf}^{\mathcal{T}}(X,z) \leq \mathrm{b}^{\mathcal{T}}(X,z) \leq \mathrm{bf}^{\mathcal{T}}(X,z) + |X|. \tag{4}$$

The maximum value of both, possible damage and forward damage, is n on each topology $\mathcal{T} \in \mathbb{T}(n,C,k)$ if at least Cz nodes may be removed (removing the heads of z stripes). Together with Equation (4), we obtain the following theorem.

Theorem 1. *For every topology $\mathcal{T} \in \mathbb{T}(n,C,k)$, $z \in \{1,\ldots,k\}$, and $x \in \{1,\ldots,n\}$, it holds that*

$$\max_{\substack{X\subseteq V\\|X|=x}} \mathrm{bf}^{\mathcal{T}}(X,z) \leq \max_{\substack{X\subseteq V\\|X|=x}} \mathrm{b}^{\mathcal{T}}(X,z) \leq \max_{\substack{X\subseteq V\\|X|=x}} \mathrm{bf}^{\mathcal{T}}(X,z) + \min(Cz-1,x).$$

In applications of multitree data distribution topologies, we usually have $n \gg Ck$. Furthermore, the maximum achievable forward damage for threshold z on each topology $\mathcal{T} \in \mathbb{T}(n,C,k)$ is in $\Omega(\frac{n}{Cz})$ if at least z nodes are removed. Consequently, with growing node numbers, the maximum possible forward damage *dominates* the value of the maximum possible damage.

4 Forward-Stable Distribution Topologies

Theorem 1 motivates the study of distribution topologies minimizing maximum forward damage for all numbers of removed nodes and thresholds z. In the following, we will distinguish between different levels of this resilience concept by restricting the possible sets of removed nodes. For this, we introduce the *t-restricted attacks* $\chi(\mathcal{T},t)$ for each $\mathcal{T} \in \mathbb{T}(n,C,k)$ and $t \in \{1,\ldots,k\}$. An attack $X \subseteq V$ satisfies $X \in \chi(\mathcal{T},t)$, if there is a set $I \subseteq \{1,\ldots,k\}$ of t stripe indices such that each $v \in X$ either has forward successors in at least one of the stripes I, or it has no forward successors at all. Thus, if topology $\mathcal{T}$ has inner-node disjoint stripe trees, $\chi(\mathcal{T},t)$ is the set of all attacks containing inner-nodes from *at most* t stripes and an arbitrary number of nodes that are leaf in all stripes. The definition ensures that $\chi(\mathcal{T},t-1) \subseteq \chi(\mathcal{T},t)$ is true and that $\chi(\mathcal{T},k)$ equals the power set $\mathcal{P}(V)$ of V. Furthermore, for each $t \in \{1,\ldots,k\}$, the set $\chi(\mathcal{T},t)$ contains *all* subsets of V that have cardinality up to t.

Now, we can define t-forward-stable and forward-stable distribution topologies.

Definition 2. *A topology $\mathcal{T} \in \mathbb{T}(n,C,k)$ is called* t-forward-stable*, if for all $z \in \{1,\ldots,k\}$, $x \in \{1,\ldots,n\}$, and $\mathcal{C} \in \mathbb{T}(n,C,k)$, it holds that*

$$\max_{X\in\chi(\mathcal{T},t),|X|=x} \mathrm{bf}^{\mathcal{T}}(X,z) \leq \max_{X\in\chi(\mathcal{C},t),|X|=x} \mathrm{bf}^{\mathcal{C}}(X,z).$$

If $\mathcal{T}$ is t-forward-stable for all $t \in \{1,\ldots,k\}$, it is called forward-stable*.*

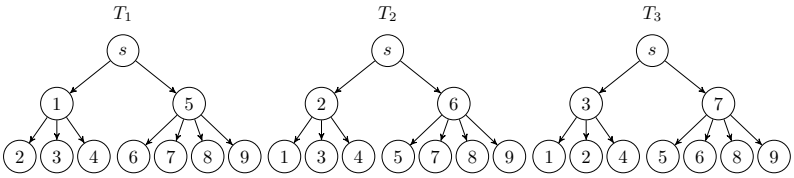


Fig. 3. A distribution topology $\mathcal{C} \in \mathbb{T}(9, 2, 3)$ as in the proof of Lemma 1

Consequently, a topology $\mathcal{T}$ is t-forward-stable, if it minimizes the maximum possible forward damage that is achievable by t-restricted attacks (for all attack cardinalities and thresholds z), while forward-stable topologies are t-forward-stable for all possible values of t. As we have seen in Section 3, the latter closely approximate attack-resilient topologies.

In the following, we show necessary and sufficient requirements for (t-)forward-stable topologies. Furthermore, we give a notion of the computational complexity of finding a forward-stable topology in a given class $\mathbb{T}(n, C, k)$. In particular, we show that a corresponding oracle could be used to efficiently determine so-called Orthogonal Arrays of given dimension and maximum strength. The latter is a notorious problem in both design and coding theory [2, 8].

4.1 Basic Requirements

Lemma 1. *A t-forward-stable topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ with $t \in \{1, \ldots, k\}$ has the following properties:*

1. $\forall v \in V\colon |\{i \in \{1, \ldots, k\} \mid \mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) \neq \emptyset\}| \leq 1$
2. $\forall v \in V\colon |\bigcup_{i \in \{1,\ldots,k\}} \mathrm{succ}_i^{\mathcal{T}\rightarrow}(v)| \leq \lceil \frac{n}{C} \rceil$

Proof. We compare $\mathcal{T}$ with a topology $\mathcal{C} \in \mathbb{T}(n, C, k)$ that has Ck distinct heads in total and C heads per stripe. In each stripe $i \in \{1, \ldots, k\}$, all nodes $V \setminus H_i^{\mathcal{C}}$ are leafs below the heads $H_i^{\mathcal{C}}$. They are grouped such that each head $h \in H_i^{\mathcal{C}}$ satisfies $|\mathrm{succ}_i^{\mathcal{C}\rightarrow}(h)| \in \{\lceil n/C \rceil, \lfloor n/C \rfloor\}$. Figure 3 gives an example of such a topology.

Since it is t-forward-stable with $t \geq 1$, topology $\mathcal{T}$ should minimize the maximum possible forward-damage for attacks of cardinality 1 and all values of z. However, if $\mathcal{T}$ lacks one of the mentioned properties, we show that, for certain z, there are attacks of cardinality 1 on $\mathcal{T}$ that achieve more forward-damage than any such attack can achieve on $\mathcal{C}$:

1. Assume there is $v \in V$ and two distinct stripes $i, j \in \{1, \ldots, k\}$, such that $\mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) \neq \emptyset$ and $\mathrm{succ}_j^{\mathcal{T}\rightarrow}(v) \neq \emptyset$. Then, it holds that $v \in \mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) \cap \mathrm{succ}_j^{\mathcal{T}\rightarrow}(v)$. In contrast, for all $w \in V$ there is no pair i, j of distinct stripes of $\mathcal{C}$ such that $\mathrm{succ}_i^{\mathcal{C}\rightarrow}(w) \cap \mathrm{succ}_j^{\mathcal{C}\rightarrow}(w) \neq \emptyset$. It follows that $\max_{u \in V} \mathrm{bf}^{\mathcal{T}}(\{u\}, 2) \geq 1$ and $\max_{u \in V} \mathrm{bf}^{\mathcal{C}}(\{u\}, 2) = 0$. Consequently, $\mathcal{T}$ is not t-forward-stable.

2. Assume that $\exists v \in V\colon |\bigcup_{i\in\{1,\dots,k\}} \mathrm{succ}_i^{\mathcal{T}\rightarrow}(v)| > \lceil \frac{n}{C} \rceil$. For every topology $\mathcal{D} \in \mathbb{T}(n, C, k)$, the definition of forward damage guarantees that

$$\max_{X\subseteq V, |X|=1} \mathrm{bf}^{\mathcal{D}}(X, 1) = \max_{u\in V} \Big| \bigcup_{i\in\{1,\dots,k\}} \mathrm{succ}_i^{\mathcal{D}\rightarrow}(u) \Big|. \tag{5}$$

In $\mathcal{C}$, this value is $\lceil \frac{n}{C} \rceil$, whereas it is higher in $\mathcal{T}$. Again, $\mathcal{T}$ is not t-forward-stable. □

The first property ensures the construction of inner-node disjoint stripe trees. The second one corresponds to a balanced distribution of successors to the heads of each stripe. Both are frequent optimization goals in peer-to-peer live streaming systems such as [4] and [3]. Note that topologies from $\mathbb{T}(n, C, k)$ with both properties will have C unique heads per stripe.

Additionally, such topologies have another interesting property.

Lemma 2. *Let $\mathcal{T} \in \mathbb{T}(n, C, k)$ satisfy the requirements of Lemma 1. For all $z \in \{1, \dots, k\}$ and each $X \subseteq V$, there exists an attack $Y \subseteq H^{\mathcal{T}}$ with $\mathrm{bf}^{\mathcal{T}}(Y, z) \geq \mathrm{bf}^{\mathcal{T}}(X, z)$ and $|Y| = \min(|X|, Cz)$.*

Proof. The stripe trees of topology $\mathcal{T}$ are inner-node disjoint. Therefore, the node sets $V_i := \{v \in V \mid \mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) \neq 0\}$ for $i \in \{1, \dots, k\}$ together with set $V_0 := V \setminus \bigcup_{i\in\{1,\dots,k\}} V_i$ form a partition of V.

Since each T_i is a tree, it holds that $\mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) \subseteq \mathrm{succ}_i^{\mathcal{T}\rightarrow}(u)$ for each $v \in \mathrm{succ}_i^{\mathcal{T}\rightarrow}(u)$. Hence, for each stripe T_i the set $\{\mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) \mid v \in V_i\}$ is a *laminar family of sets*. In particular, the forward successor sets of the heads $H_i^{\mathcal{T}}$ are the only sets that are not subsets of others.

Now, let $X \subseteq V$ be an arbitrary attack on $\mathcal{T}$. If $|X| \geq Cz$, then *all* nodes can be isolated by attacking the (at most) Cz heads of z stripes with the smallest number of heads. Otherwise, set $X := X \setminus V_0$, and let

$$Y' := \{h \in H^{\mathcal{T}} \mid \exists i \in \{1, \dots, k\}, v \in V_i \cap X\colon \mathrm{succ}_i^{\mathcal{T}\rightarrow}(v) \subseteq \mathrm{succ}_i^{\mathcal{T}\rightarrow}(h)\}.$$

Due to the node partition and set laminarity, it holds that $|Y'| \leq |X|$. Furthermore, we have $\forall i \in \{1, \dots, k\}\colon \mathrm{succ}_i^{\mathcal{T}\rightarrow}(X) \subseteq \mathrm{succ}_i^{\mathcal{T}\rightarrow}(Y')$ and therefore $\mathrm{bf}^{\mathcal{T}}(X, z) \leq \mathrm{bf}^{\mathcal{T}}(Y', z)$ for all $z \in \{1, \dots, k\}$. No superset $Y \subseteq H^{\mathcal{T}}$ with $Y' \subseteq Y$ and $|Y| = |X|$ can create less forward-damage. □

We see, that on every topology with the properties given in Lemma 1, a maximum value of forward damage can always be achieved by removing only heads. Consequently, the optimization of their forward successor sets is the key to finding forward-stable topologies.

4.2 A Matrix Representation and Orthogonal Arrays

For every distribution topology $\mathcal{T}$, there is a convenient matrix representation of its heads' forward successor sets.

Definition 3. *Let $\mathcal{T} \in \mathbb{T}(n, C, k)$ be given. Using per stripe $i \in \{1, \ldots, k\}$ a bijection $\sigma_i \colon H_i^{\mathcal{T}} \to \{1, \ldots, |H_i^{\mathcal{T}}|\}$, the* matrix $M^{\mathcal{T}}$ of forward successor sets of the heads $H^{\mathcal{T}}$ *is an $n \times k$ matrix $M^{\mathcal{T}} = (m_{vi})$, such that*

$$m_{vi} = \sigma_i(j) \Leftrightarrow v \in \operatorname{succ}_i^{\mathcal{T}\rightarrow}(j).$$

For $v \in V$, $M^{\mathcal{T}}[v] = (m_{v1}, \ldots, m_{vk})$ denotes the v-th row of $M^{\mathcal{T}}$.

Consequently, the i-th entry of the v-th row of $M^{\mathcal{T}}$ encodes the head supplying node v in stripe i. Its numeric value is determined by bijection σ_i. As an example for this definition, Figure 4(b) shows a matrix corresponding to the topology in Figure 4(a).

Reusing the bijections σ_i from $M^{\mathcal{T}}$, we can also transform attacks on the heads of $\mathcal{T}$ into sets of k-dimensional vectors. In their i-th position, these vectors contain entries from $\{0, \ldots, |H_i^{\mathcal{T}}|\}$.

Definition 4. *Let topology $\mathcal{T} \in \mathbb{T}(n, C, k)$, matrix $M^{\mathcal{T}}$, and the the corresponding bijections $\sigma_i \colon H_i^{\mathcal{T}} \to \{1, \ldots, |H_i^{\mathcal{T}}|\}$ for $i \in \{1, \ldots, k\}$ be given.*

The vector attack $\sigma(X)$ *for an attack $X \subseteq H^{\mathcal{T}}$ contains each vector $\mathbf{y} \in \{0, \ldots, Ck\}^k$ such that for all $i \in \{1, \ldots, k\}$ either $\sigma_i^{-1}(\mathbf{y}_i) \in X$ or $(\mathbf{y}_i = 0) \wedge (X \cap H_i^{\mathcal{T}} = \emptyset)$ is true.*

Due to its definition, $\sigma(X)$ will contain $\prod_{i=1}^{k} \min(1, |X \cap H_i^{\mathcal{T}}|)$ vectors. In position i, such a vector either contains the value $\sigma_i(h)$ for some $h \in X \cap H_i^{\mathcal{T}}$ or the value 0 if $X \cap H_i^{\mathcal{T}} = \emptyset$.

Using vector attacks, the forward damage $\mathrm{bf}^{\mathcal{T}}(X, z)$ of an attack $X \subseteq H^{\mathcal{T}}$ on $\mathcal{T}$ can be determined by counting row vectors of $M^{\mathcal{T}}$ that are in *Hamming Distance* at most $k - z$ to an element of $\sigma(X)$. With $\mathrm{d}(\cdot, \cdot)$ as the Hamming Distance function, we can write

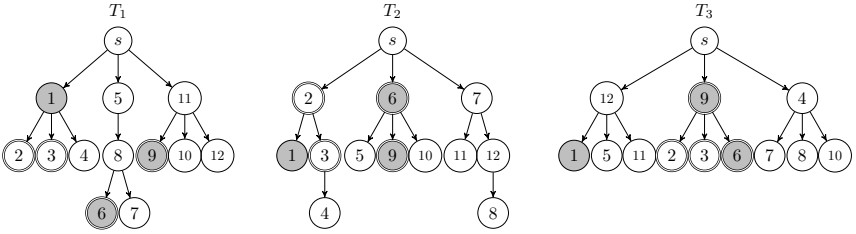
$$\begin{aligned} \mathrm{bf}^{\mathcal{T}}(X, z) &= \left| \bigcup_{I \subseteq \{1,\ldots,k\}, |I|=z} \bigcap_{i \in I} \operatorname{succ}_i^{\mathcal{T}\rightarrow}(X) \right| \\ &= \left| \left\{ v \in V \;\middle|\; \exists I \subseteq \{1, \ldots, k\}, |I| = z, \forall i \in I \colon \sigma_i^{-1}(m_{vi}) \in X \right\} \right| \\ &= \left| \left\{ v \in V \;\middle|\; \exists \mathbf{x} \in \sigma(X) \colon \mathrm{d}(M[v], \mathbf{x}) \leq k - z \right\} \right|. \end{aligned} \tag{6}$$

Figure 4(c) gives a graphical example.

Next, we introduce a special class of matrices, the *Orthogonal Arrays* [8].

Definition 5. *For $n, k, C \in \mathbb{N}$ and $t \in \{0, \ldots, k\}$, an $n \times k$ matrix M with entries $m_{vi} \in \{1, \ldots, C\}$ is called an* Orthogonal Array $\mathrm{OA}(n, k, C, t)$ *if in every $n \times t$ submatrix M' consisting of t complete columns of M, each $\mathbf{x} \in \{1, \ldots, C\}^t$ appears exactly $\lambda := \frac{n}{C^t}$ times as a row.*

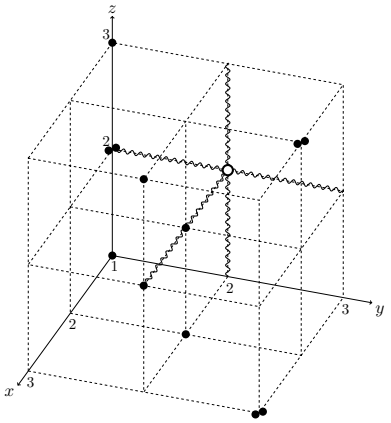
An $\mathrm{OA}(n, k, C, t)$ is said to have *strength* t. It minimizes the maximum frequency of a row vector in each of its t-column submatrices. Every Orthogonal Array of strength $t > 1$ also has strength $t - 1$. The strength of a given $n \times k$ matrix is computable in time $O(n^2 k)$ [8, Chapter 4.4]. Figure 5 shows an $\mathrm{OA}(18, 3, 3, 2)$.



(a) Attack $X = \{1, 6, 9\}$ on topology $\mathcal{T} \in \mathbb{T}(12, 3, 3)$ leads to $\mathrm{bf}^{\mathcal{T}}(X, 2) = 4$ (attacked nodes gray, nodes suffering forward-damage double-lined)

$$M^{\mathcal{T}} = \begin{pmatrix} 1\,1\,1 \\ 1\,1\,2 \\ 1\,1\,2 \\ 1\,1\,3 \\ 2\,2\,1 \\ 2\,2\,2 \\ 2\,3\,3 \\ 2\,3\,3 \\ 3\,2\,2 \\ 3\,2\,3 \\ 3\,3\,1 \\ 3\,3\,1 \end{pmatrix}$$

(b) $M^{\mathcal{T}}$ for $\sigma_1(1) = \sigma_2(2) = \sigma_3(12) = 1$, $\sigma_1(5) = \sigma_2(6) = \sigma_3(9) = 2$ and $\sigma_1(11) = \sigma_2(7) = \sigma_3(4) = 3$



(c) Rows of $M^{\mathcal{T}}$ (dots) in Hamming distance ≤ 1 (snaked) from vector attack $\sigma(X) = \{(1, 2, 2)\}$ (circled) correspond to nodes $\{2, 3, 6, 9\}$

Fig. 4. A distribution topology $\mathcal{T}$, a corresponding matrix $M^{\mathcal{T}}$, and forward damage due to the removal of node set $X = \{1, 6, 9\}$ from $\mathcal{T}$

Lemma 3. *For every* $\mathrm{OA}(n, k, C, t)$ M *with* $n \geq Ck$ *and strength* $t \geq 1$*, there is a topology* $\mathcal{T} \in \mathbb{T}(n, C, k)$ *with* $M^{\mathcal{T}} = M$ *that satisfies the requirements of Lemma 1.*

Proof. We construct a suitable topology $\mathcal{T}$ of depth 2. For the use as heads $H^{\mathcal{T}}$, we determine the indices of Ck suitable rows of M. For this, construct a bipartite graph $G = (\{1, \ldots, n\} \dot\cup (\{1, \ldots, C\} \times \{1, \ldots, k\}), E)$. Its node set contains the nodes $V = \{1, \ldots, n\}$ of $\mathcal{T}$ and head positions (i, j). A head position (i, j) corresponds to the role as i-th head in stripe j of $\mathcal{T}$. The edge set E satisfies $\{v, (i, j)\} \in E \Leftrightarrow M[v]_j = i$.

For each node u, let $N(u)$ be the set of u's neighbors in G. Since M has k columns, each node $v \in V$ satisfies $|N(v)| = k$. Since M has strength at least 1, each head position (i, j) has $|N((i, j))| = n/C$. Due to Hall's Theorem

$$\begin{pmatrix} 1&1&1&2&2&2&3&3&3&1&1&1&2&2&2&3&3&3\\ 1&2&3&1&2&3&1&2&3&1&2&3&1&2&3&1&2&3\\ 1&2&3&2&1&3&3&1&2&1&3&2&3&2&1&2&3&1 \end{pmatrix}$$

Fig. 5. Transpose of an OA$(18, 3, 3, 2)$ with $\lambda = 2$

(cmp. [9]) there is a matching covering all head positions in G, if it holds that $\forall S \subseteq \{1, \ldots, C\} \times \{1, \ldots, k\}\colon |\bigcup_{u \in S} N(u)| \geq |S|$. This is the case in G. For each possible subset S of head positions, there are $|S| \cdot n/C$ edges to nodes from V. Since these $|\bigcup_{u \in S} N(u)|$ nodes have $|\bigcup_{u \in S} N(u)| \cdot k$ edges in total and since $n/C \geq k$, we obtain

$$|S| \cdot \frac{n}{C} \leq |\bigcup_{u \in S} N(u)| \cdot k \quad \Rightarrow \quad |S| \leq |\bigcup_{u \in S} N(u)|. \tag{7}$$

Hence, a maximum matching R in G connects each head position with a unique node from V. For each $\{v, (i, j)\} \in R$, we use v as head in stripe T_j of $\mathcal{T}$, define $\sigma_j(v) := i$, and set $\mathrm{succ}_j^{\mathcal{T}}(v) := \{u \in V \mid M[u]_j = i\}$. In each stripe of the emerging topology $\mathcal{T}$, every node is either head or child of a head. The matching R guarantees that we have $|H^{\mathcal{T}}| = Ck$ and that each head forwards in only one stripe. The defined bijections σ_j with $j \in \{1, \ldots, k\}$ establish $M^{\mathcal{T}} = M$. Since M is of strength at least 1, for all $j \in \{1, \ldots, k\}$ each head $v \in H_j^{\mathcal{T}}$ satisfies $|\mathrm{succ}_j^{\mathcal{T}\rightarrow}(v)| = n/C$. All other forward successor sets are empty. □

A matrix $M^{\mathcal{T}}$ of high strength is beneficial for the forward-stability of $\mathcal{T}$.

Theorem 2. *A topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ is t-forward-stable, if it has the properties of Lemma 1 and $M^{\mathcal{T}}$ is an* OA(n, k, C, t).

Proof (sketch). For reasons of space, we can only give a proof sketch. See [10, Theorem 5.3.14] for all details.

Given a topology $\mathcal{T} \in \mathbb{T}(n, C, k)$, we call each vector $\mathbf{x} \in \{0, \ldots, Ck\}^k$ satisfying $\forall i \in \{1, \ldots, k\}\colon \mathbf{x}_i \leq |H_i^{\mathcal{T}}|$ an *attack distribution for* $\mathcal{T}$ and say that an attack $X \subseteq H^{\mathcal{T}}$ *follows* $\mathbf{x}$ if $\forall i \in \{1, \ldots, k\}\colon |X \cap H_i^{\mathcal{T}}| = \mathbf{x}_i$ holds.

If $\mathcal{T}$ has the properties given in Lemma 1 and $M^{\mathcal{T}}$ has strength t, then for each threshold $z \in \{1, \ldots, k\}$ the forward damage of all attacks $X \in \chi(\mathcal{T}, t)$ on $\mathcal{T}$ following the same attack distribution $\mathbf{x}$ is equal. Furthermore, the value of this forward damage on $\mathcal{T}$ gives a lower bound on the average (and maximum) forward damage of attacks following $\mathbf{x}$ on other topologies from $\mathbb{T}(n, C, k)$. Consequently, for each $z \in \{1, \ldots, k\}$ and each $\mathcal{C} \in \mathbb{T}(n, C, k)$ on which attacks with distribution $\mathbf{x}$ exist, there is $Y \in \chi(\mathcal{C}, t)$ following $\mathbf{x}$ with $\mathrm{bf}^{\mathcal{T}}(X, z) \leq \mathrm{bf}^{\mathcal{C}}(Y, z)$.

If there is no attack with distribution $\mathbf{x}$ on $\mathcal{C}$, then a suitable distribution $\mathbf{x}'$ can be found by adapting $\mathbf{x}$ with regard to the number of heads available in $\mathcal{C}$. Thus, for each $\mathcal{C} \in \mathbb{T}(n, C, k)$ and each attack $X \in \chi(\mathcal{T}, t)$, we can find an attack $Y \in \chi(\mathcal{C}, t)$ creating at least the same forward damage on $\mathcal{C}$ as X does on $\mathcal{T}$. Consequently, $\mathcal{T}$ is t-forward-stable. □

Next, we show that the matrix $M^{\mathcal{T}}$ of a forward-stable topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ must necessarily be an Orthogonal Array of maximum possible strength.

Theorem 3. *If an* $\mathrm{OA}(n, k, C, t)$ *exists, then for every* t'*-forward-stable distribution topology* $\mathcal{T} \in \mathbb{T}(n, C, k)$ *with* $t' \geq t$, $M^{\mathcal{T}}$ *is an* $\mathrm{OA}(n, k, C, t)$.

Proof. Topology $\mathcal{T}$ must have the properties listed in Lemma 1. Furthermore, assume that $\mathcal{T}$ is not an $\mathrm{OA}(n, k, C, t)$. If $\mathcal{T}$ were t'-forward-stable, it had to minimize maximum forward-damage for attacks of cardinality t and threshold $z = t$. We show that under the above assumption, this is not the case. For this, let $\mathcal{C} \in \mathbb{T}(n, C, k)$ be a topology with the properties listed in Lemma 1 and $M^{\mathcal{C}}$ being an $\mathrm{OA}(n, k, C, t)$ (the existence of $\mathcal{C}$ is guaranteed by Lemma 3).

Set $z = t$ and study the possible forward-damage of attacks of cardinality t. Due to Lemma 2, it suffices to consider attacks removing only heads. Such attacks may target heads from less than t different stripes. This would lead to forward-damage of 0 on both $\mathcal{T}$ and $\mathcal{C}$ since they have inner-node disjoint stripes. Alternatively, attacks can target one head from each stripe of a combination of t stripes. In this case, the maximum possible forward-damage on $\mathcal{T}$ and $\mathcal{C}$ equals the maximum row frequency in $M^{\mathcal{T}}$ resp. $M^{\mathcal{C}}$ over all possible restrictions to t columns (cmp. Equation (6)). An attack achieving this damage contains the heads corresponding to the entries in the respective columns of the most frequent row vector. Since $\mathcal{C}$ is an $\mathrm{OA}(n, k, C, t)$ but $\mathcal{T}$ is not, this frequency is smaller on $\mathcal{C}$ than on $\mathcal{T}$. Hence, $\mathcal{T}$ is not t-forward-stable and, thus, not t'-forward-stable. □

Summing up, this subsection has shown that – given the basic properties identified in Lemma 1 – the forward-stability of a distribution topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ is characterized by its matrix $M^{\mathcal{T}}$. In particular, if an $\mathrm{OA}(n, k, C, t)$ exists, it is necessary and sufficient that $M^{\mathcal{T}}$ is such an Orthogonal Array to obtain a t-forward-stable topology. To reach a maximum level of forward-stability, $M^{\mathcal{T}}$ must be an Orthogonal Array of maximum possible strength t. This observation is used in Subsection 4.3 to provide a notion of the computational complexity of finding forward-stable distribution topologies.

4.3 Hardness of Finding Forward-Stable Topologies

For given parameters $n, C, k \in \mathbb{N}$, let $\hat{\mathrm{t}}(n, C, k)$ be the maximum value t such that an $\mathrm{OA}(n, k, C, t)$ exists. If $\hat{\mathrm{t}}(n, C, k)$ is efficiently computable, it is also possible to use binary search to efficiently determine extremal values for the parameter k of Orthogonal Arrays.

However, resolving the computational complexity of finding such extremal parameters and finding Orthogonal Arrays featuring them are long-standing open problems in design theory (cmp. [8, p.32]). A special case in coding theory is the *MDS conjecture* [2, 11] which claims to specify the maximum length of MDS codes. Its disputed part was first stated in 1955 [12].

We show that finding an efficient construction strategy for t-forward-stable distribution topologies would resolve many of the above questions.

Theorem 4. *Let $\mathcal{O}$ be an oracle returning a t-forward-stable topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ on input (n, k, C, t) if one exists.*

- *If one exists, an $\mathrm{OA}(n, k, C, t)$ can be constructed by one call to $\mathcal{O}$ plus $O(nk)$-time post-processing.*
- *The function $\hat{\mathrm{t}}(n, k, C)$ can be evaluated by $\lceil \log(k) \rceil$ calls to $\mathcal{O}$ plus $O(n^2 k)$-time post-processing.*

Proof. Due to the Theorems 2 and 3, there is a t-forward-stable topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ if an $\mathrm{OA}(n, k, C, t)$ exists. In this case, $M^{\mathcal{T}}$ must be an $\mathrm{OA}(n, k, C, t)$. Using input (n, k, C, t), such a $\mathcal{T}$ is obtained by one call to $\mathcal{O}$. The information necessary to return the $n \times k$ matrix $M^{\mathcal{T}}$ can be gathered by a traversal of all stripe trees. This needs time $O(nk)$.

Applying binary search, we need $\lceil \log(k) \rceil$ oracle calls to find the maximum $t' \in \{0, \ldots, k\}$ such that a t'-forward-stable topology $\mathcal{T} \in \mathbb{T}(n, C, k)$ exists. By Theorem 3, $M^{\mathcal{T}}$ must be an $\mathrm{OA}(n, k, C, \hat{\mathrm{t}}(n, k, C))$. The strength of $M^{\mathcal{T}}$ can be determined in time $O(n^2 k)$. □

In the light of these results, the goal of identifying efficient construction schemes for forward-stable distribution topologies turns out to be a challenging task. Advancements would lead to a breakthrough in multiple connected fields of research.

Until then, it is possible to make use of the large number of constructions and catalogues for Orthogonal Arrays that are already available [8]. However, most of them are specific for certain parameter combinations and not of provably maximum strength. All algorithmic approaches known to the author that try to find Orthogonal Arrays with given parameters rely on metaheuristics and local search schemes (e.g., [13, 14]).

5 Conclusion

In this document, we studied multitree data distribution topologies aiming to minimize the maximum number of nodes that can be isolated by an attack. In particular, this minimization should hold for every possible number of removed nodes and every level of redundancy in data encoding. We introduced the notion of forward-stable multitree data distribution topologies and showed that they closely approximate this goal if the number of nodes considerably exceeds the number of possible source neighbors. This is a common condition in applications of the studied topologies.

We found basic requirements for forward-stable distribution topologies and pointed out that the resilience of topologies adhering to these requirements is captured by a matrix representation of their heads' forward successor sets. We showed that such a topology is t-forward-stable if its matrix is an Orthogonal Array of strength t. Furthermore, the use of Orthogonal Arrays of maximum strength is necessary for forward-stable topologies. This result allowed to connect the problem of finding forward-stable topologies to long-standing open problems in design and coding theory.

Since for higher numbers of nodes, attack-resilient and forward-stable topologies must be very similar, this also provides a notion of hardness of finding attack-resilient distribution topologies. The identified topologies and results are relevant for data distribution applications such as peer-to-peer live streaming systems. Furthermore, the studied model could also be applied to certain data aggregation tasks in wireless sensor networks.

References

1. Goyal, V.: Multiple description coding: compression meets the network. IEEE Signal Proc. Mag. 18(5), 74–93 (2001)
2. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland Mathematical Library (1993)
3. Brinkmeier, M., Schaefer, G., Strufe, T.: Optimally DoS Resistant P2P Topologies for Live Multimedia Streaming. IEEE T. Parall. Distr. 20(6), 831–844 (2009)
4. Castro, M., Druschel, P., Kermarrec, A.M., Nandi, A., Rowstron, A., Singh, A.: Splitstream: high-bandwidth multicast in cooperative environments. SIGOPS Oper. Syst. Rev. 37, 298–313 (2003)
5. Padmanabhan, V.N., Wang, H.J., Chou, P.A., Sripanidkulchai, K.: Distributing streaming media content using cooperative networking. In: NOSSDAV 2002, pp. 177–186. ACM, New York (2002)
6. Grau, S., Fischer, M., Schäfer, G.: On the Dependencies between Source Neighbors in Optimally DoS-stable P2P Streaming Topologies. In: IEEE International Conference on Distributed Computing Systems 2011, ICDCS, pp. 121–130 (2011)
7. Dán, G., Fodor, V.: Stability and performance of overlay multicast systems employing forward error correction. Perform. Eval. 67, 80–101 (2010)
8. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: Orthogonal Arrays: Theory and Applications. Springer, New York (1999)
9. Diestel, R.: Graph Theory, 3rd edn. Graduate Texts in Mathematics, vol. 173. Springer, Heidelberg (2005)
10. Grau, S.: On the Stability of Distribution Topologies in Peer-to-Peer Live Streaming Systems. PhD thesis, Technische Universität Ilmenau, Germany (2012)
11. Roth, R.M.: Introduction to Coding Theory. Cambridge University Press (2006)
12. Segre, B.: Curve razionali normali e k-archi negli spazi finiti. Ann. Math. Pura Appl. (39), 357–359 (1955)
13. Nguyen, N.K., Liu, M.Q.: An algorithmic approach to constructing mixed-level orthogonal and near-orthogonal arrays. Comput. Stat. Data An. 52, 5269–5276 (2008)
14. Xu, H.: An Algorithm for Constructing Orthogonal and Nearly Orthogonal Arrays with Mixed Levels and Small Runs. Technometrics 44, 356–368 (2002)

On the Complexity of Distributed Broadcasting and MDS Construction in Radio Networks*

Tomasz Jurdzinski[1] and Dariusz R. Kowalski[2]

[1] Institute of Computer Science, University of Wrocław, Poland
[2] Department of Computer Science, University of Liverpool, United Kingdom

Abstract. We study two fundamental problems in the model of undirected radio networks: broadcasting and construction of a Minimal Dominating Set (MDS). The network is ad hoc, in the sense that initially nodes know only their own ID and the IDs of their neighbors. For both problems, we provide deterministic distributed algorithms working in $O(D\sqrt{n}\log^6 n)$ communication rounds, and complement them by a close lower bound $\Omega(\sqrt{Dn\log(n/D)})$, where n is the number of nodes and D is the radius of the radio network. Our work provides several novel algorithmic methods for overcoming the impact of collisions in radio networks, and shrinks the gap between the lower and the upper bounds for the considered problems from polynomial to polylogarithmic, for networks with small (polylogarithmic) radius.

Keywords: radio networks, broadcasting, minimal dominating set, distributed algorithms.

1 Introduction

Radio Networks model a communication environment where simultaneous message transmissions in a close proximity result in signal interference, and no message is successfully delivered. This model has been successfully used since early 80s in the context of Local Access Networks, wireless networks, multi-bus and multi-core topologies (c.f., [4,9]), for obtaining and analyzing many algorithmically non-trivial and applicable solutions. Even though some of them have been later analyzed in more complex models, radio networks are still widely used for their simplicity and suitability for design and (preliminary) analysis of communication algorithms.

In the radio network model, c.f., [4], the core assumption is that a transmitted message reaches all neighbors of the transmitting node v, however it could be successfully heard by a neighbor w only if w is not transmitting and v is the only transmitting neighbor of w at a time. We consider the setting *without collision detection*, i.e., the case when no neighbor transmits is indistinguishable from the case when at least two neighbors transmit. We use notation n for the number

* This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/G023018/1].

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 209–223, 2012.

of nodes in the network, D for the radius of the network (with respect to some distinguished node, called a source), and $N = \text{poly}(n)$ for the range of node ids. We consider two fundamental problems: Broadcasting and construction of Minimal Dominating Set (MDS). We seek for time-efficient deterministic distributed solutions for these problems.

Previous results. Bar-Yehuda et al. [1] claimed that the time complexity of deterministic broadcasting in ad hoc radio networks is $\Omega(n)$ even for networks of radius 2. Kowalski and Pelc [10] proved that it is not the case: they showed a deterministic algorithm that accomplishes broadcast in $O(n^{2/3} \log n)$ rounds in any network of radius 2, and another algorithm that completes broadcast in $o(n)$ rounds in networks of radius $o(\log \log n)$. On the other hand, a lower bound $\Omega(Dn^{1/4})$ was proved in [10] for broadcasting in networks with radius D, which proves an exponential gap between the overhead in this model and the model with randomization, see the next paragraph. Brito and Vaya [3] improved this bound to $\Omega(n^{1/2})$, still leaving the gap between the lower bound and the best known upper bound of magnitude $n^{1/6} \log n$.

The first efficient randomized solution in the ad hoc radio model, working in expected time $O(D \log n + \log^2 n)$, was presented by Bar-Yehuda et al. [1]. A lower bound $\Omega(D \log(n/D) + \log^2 n)$ on expected time of randomized broadcast was given by Kushilevitz and Mansour [12], and the matching algorithm was developed by Kowalski and Pelc [11] and by Czumaj and Rytter [8].

The problem of constructing a Minimal Dominating Set (MDS) is closely related to Broadcasting in the model of radio networks, and many of the developed techniques and results for broadcasting also hold for MDS. In particular, we are not aware of any separate result on the time complexity of MDS in radio networks that would not be obtained in the context of broadcasting.

Our results. We strengthen the lower bound $\Omega(\sqrt{n})$ on deterministic distributed broadcasting for networks of radius 2 to $\Omega(\sqrt{n \log n})$, which justifies that the complexity of the problem is asymptotically larger than $\sqrt{n}$. For D-hop networks, the lower bound takes the form of $\Omega(\sqrt{Dn \log(n/D)})$. These bounds are easily extended to the problem of deterministic distributed construction of a MDS. We also provide two broadcasting algorithms: one for networks with radius 2, which works in $O(\sqrt{n} \log^6 n)$ communication rounds, and the other for networks of radius D, working in $O(D\sqrt{n} \log^6 n)$ rounds. The former algorithm improves over the best known $O(n^{2/3} \log n)$ time broadcasting algorithm, and thus shrinks the gap between the lower and upper bounds from polynomial to polylogarithmic for networks of radius 2. The latter algorithm extends the range of diameters admitting sublinear $o(n)$-rounds algorithms from $o(\log \log n)$ to polynomial, which is a double exponential improvement, c.f., [10]. It also shrinks the gap between upper and lower bounds from polynomial to polylogarithmic for networks of polylogarithmic radius. Finally, we show how to adapt these algorithms for constructing a MDS in asymptotically same round complexity.

Previous sublinear time deterministic algorithms for broadcasting propagated messages layer-by-layer in such a way that each node followed its own schedule, sometimes coordinated by the source. These method incurred a substantial

communication overhead on each hop. We introduce more complex clustering mechanism of bipartite graphs, which allows to form collaborative groups of nodes, with the goal to inform their neighbors, c.f., Phase 3 of algorithm $\mathcal{A}_1$. We show how to efficiently build such clusters and simultaneously maintain short intra- and inter-cluster communication schedules, all in a deterministic distributed way. This clustering combined with a greedy schedule of selecting nodes with certain properties and with the centralized schedule of Chlamtac-Weinstein [5], results in substantial improvement of time complexity, especially for shallow networks (i.e., networks of small diameter). An example of novel algorithmic techniques used for efficient clustering is a new way of constructing transmission schedules, by taking a product of selectors and adaptively maintained minimum ID of cluster nodes, which results in a large portion of inter-cluster point-to-point successful communication.

Due to space limit, the missing proofs are deferred to the full version of the paper.

2 Preliminaries

We consider radio networks defined as an undirected connected reachability graph $G(V, E)$ whose nodes have distinct labels belonging to the set $[N] = \{1, \ldots, N\}$, where N is polynomially large with respect to the number of stations $n = |V|$; both n and N are known to all stations prior the computation. In the broadcasting problem, a distinguished node with label 1 is called a *source*. We define the radius D of a network as the largest distance from the source to any node of the network, where distance between nodes denotes the length of the shortest path connecting them. Initially each node has no knowledge about the topology of the underlying network, except of the information about IDs of its neighbors — we call it *local knowledge*.

It is assumed that time is divided into discrete time steps, called *rounds*, all nodes start simultaneously, they have access to the central clock, and work in rounds. A message sent at round t by a node u is sent to all its neighbors. However, a neighbor v of u receives this message if u is its only neighbor transmitting in round t. If v does not receive any message at time t, then either none of its neighbors has transmitted at round t, or at least two have. However, v is not able to distinguish between these two events; such model characteristic is typically called a model with *no collision detection*.

Communication protocol. A communication protocol specifies — for each node $v \in [N]$, the set of neighbors of node v, each round t and all messages received by node v before round t — whether node v transmits a message at round t, and if yes, what is the content of this message. The goal of any *broadcast protocol* is to deliver a message originally stored in the source, also called the *broadcast message* or the *source message*, to all nodes of the network, by transmitting and successful receptions of this message along the underlying radio network. We say that a station is *informed* at time t of an execution of a broadcasting protocol if that station received the broadcast message until round t, and it is *uninformed* otherwise. We consider a *non-spontaneous* model, i.e., a node (except the source)

may act as a transmitter only if it has received a message earlier. We assume that each time a station sends a message, it encloses its ID and information containing its whole history of communication (from which one can deduce its knowledge about the network). Our algorithms, however, will use only at most polynomial, in n, number of bits, in addition to the source message.

Graph-based notation. Throughout this paper, N denotes the range of identifiers of nodes, n is the actual size of the graph of the network. Each time we refer to a symmetric graph $G(V, E)$, we mean the graph with unique identifiers in the range $[N]$ of its nodes. Given a symmetric graph $G(V, E)$, $\Gamma_G(v)$ denotes the set of neighbors of v in G, and $d_G(v) = |\Gamma_G(v)|$ (the subscript G is omitted when it is clear from the context). For a graph $G(V, E)$ with distinguished source node s, $L_i \subset V$ denotes the set of nodes in distance i from s (thus, in particular, $L_0 = \{s\}$ and L_1 is equal to the set of neighbors of s). Moreover, we denote $n_i = |L_i|$ for each $i \geq 0$. A *dominating set* in graph G is a set of nodes such that every node in the network is in this set or has a neighbor in this set. A dominating set is *minimal* if after removing any node from it the resulting set would not be dominating.

Selectors. We use combinatorial structures, called selectors, which play crucial role in many deterministic communication algorithms for radio networks. We say that a family $\mathcal{F} = (F_1, \ldots, F_f)$ of sets *hits* a set X if $|F_i \cap X| = 1$ for some $i \in [f]$. Moreover $\mathcal{F}$ *hits* X at x if $F_i \cap X = \{x\}$ for some $i \in [f]$.

Definition 1. *A family $\mathcal{F} = (F_1, \ldots, F_f)$ of subsets of $[N]$ is a (N, k, r)-*selector *if for any set $X \subseteq [N]$ of size k there is $X' \subseteq X$ of size $\min\{r + 1, k\}$ such that $\mathcal{F}$ hits X at each element of X'.*

We say that f is the *size* of a family $\mathcal{F} = (F_1, \ldots, F_f)$. Several (almost) tight bounds on the size of optimal selectors have been established for various parameters, c.f., [7,2,6]. For our lower bound arguments, we need the following result.

Theorem 1. *[7] Let $\mathcal{F}$ be a $(N, k, 1)$-selector, where $N > 2$ and $2 \leq k \leq n/64$. Then, $|\mathcal{F}| \geq \frac{k}{24} \log \frac{N}{k}$.*

On the other hand, we apply the following upper bound in our algorithm(s) for broadcasting in radio networks.

Theorem 2. *[2] For any integers $N \geq k \geq r \geq 1$, there exists a (N, k, r)-selector of size $O(\min(N, \frac{k^2}{k-r+1} \log \frac{N}{k}))$.*

Though the above result is only existential, efficient algorithms constructing (N, k, r)-selectors of size $O(\min(N, \frac{k^2}{k-r+1} \text{polylog}(N)))$ are known as well [6].

Corollary 1. *For any integers $N \geq k \geq 1$ and a real constant $\varepsilon > 0$, there exists a $(N, k, (1 - \varepsilon)k)$-selector of size $O(\min(N, k \log \frac{N}{k}))$.*

For our purposes, we need a bit stronger property defined below.

Definition 2. *A family $\mathcal{F} = (F_1, \ldots, F_f)$ of subsets of $[N]$ is a* linear $(N, k, 1-\varepsilon)$-selector *if for any set $X \subseteq [N]$ such that $k/2 < |X| \leq k$, there is $X' \subseteq X$ of size at least $\min(|X|, (1-\varepsilon)|X| + 1)$ such that $\mathcal{F}$ hits X at each element of X'.*

Thus, on one hand, definition of linear selectors concerns only the case where $r = (1-\varepsilon)k$ in general selectors. On the other hand, we require that the property of being hit by $\mathcal{F}$ at many elements holds not only for sets of size k but for all sets of size in the range $(k/2, k]$. Using Corollary 1, one can easily prove the following statement.

Corollary 2. *For any integers $N \geq k \geq 1$ and a real constant $1 > \varepsilon > 0$, there exists a linear $(N, k, 1-\varepsilon)$-selector of size $O(\min(N, k \log(N/k)))$.*

(A, B)-broadcast protocol under known topology of graph $G(A \cup B, E)$. Let A and B be disjoint subsets of V such that all nodes in A have the same message M. Then a protocol which makes message M known to all nodes $v \in B$ having a neighbor in A is called *(A, B)-broadcast protocol*.

Theorem 3. *[5] Let a radio network be modeled by a graph $G(V, E)$, where IDs of stations belong to $[N]$, and let $A, B \subset V$ be such that $A \cap B = \emptyset$, all nodes in A have the same message M and they* know *the topology of the subgraph of G spanned on $A \cup B$ (i.e., the graph $G(A \cup B, E \cap (A \cup B)^2)$). Then the elements of A can compute (A, B)-broadcast protocol that informs all nodes in B in time $O(\log^2 N)$.*

Communication schedules. An (oblivious) *communication schedule* of length f is a family of sets $\mathcal{S} = (S_1, \ldots, S_f)$, where $S_i \subseteq [N]$ for every $i \in [f]$. The length of such communication schedule is denoted by $|\mathcal{S}| = f$. An *execution* of the communication schedule $\mathcal{S}$ is a protocol in which station v transmits in round j iff $v \in S_j$. An *execution* of the communication schedule $\mathcal{S} = (S_1, \ldots, S_f)$ for r rounds is a communication protocol in which station v transmits in round $j \in [r]$ iff $v \in S_{1+(j-1) \mod f}$, i.e., we apply the communication schedule which consists of consecutive repetitions of $\mathcal{S}$. An *execution* of the communication schedule $\mathcal{S}$ *on the set* X (for r rounds) is a protocol in which station v transmits in round $j \in [|S|]$ (resp., $j \in [r]$) iff $v \in X \cap S_j$ (resp., $v \in X \cap S_{1+(j-1 \mod |S|)}$).

3 Lower Bound

In this section we prove a lower bound $\Omega(\sqrt{n \log n})$ for deterministic broadcasting with local knowledge on networks with radius 2, and its generalized version $\Omega(\sqrt{Dn \log(n/D)})$ for network of radius D.

Theorem 4. *Every deterministic broadcasting protocol for networks of radius 2 works in time $\Omega(\sqrt{n \log n})$.*

The idea of the proof is as follows.[1] Consider a class of networks of radius 2, in which nodes in the middle layer are not connected among themselves and are conceptually partitioned into groups. Some groups are connected to a single node in the last layer, some do not have any neighbor in the last layer. Assume that the size of a single group is around k, for some $k \leq n$. In order to choose a successful transmitter from a random group (to inform their unique neighbor in the last layer) without help of the source, a lower bound $\Omega(k \log(N/k))$ applies, c.f., [7]. On the other hand, there are $\Theta(n/k)$ groups, and intuitively the source could not help all the groups (by speeding-up the process of obtaining a successful transmission) in time asymptotically smaller than n/k, provided it can help one group at a time. We show formally that no other faster scenario could happen except the combination of the two described above. Therefore, all nodes in the last layer obtain the source message in time asymptotically not smaller than $\min_{k \leq n} \max\{k \log(N/k), n/k\}$, which is $\Omega(\sqrt{n \log n})$ for $k = \sqrt{n/\log n}$. One could concatenate the above construction and repeat the arguments $\Theta(D)$ times, by putting the source node of the next radius 2 component in the last informed node of the previously built part of the network. Here, network layers have size $\Theta(n/D)$, and optimal parameter k should be set to $k = \sqrt{n/(D \log(n/D))}$, in order to get broadcasting time of $\Omega(D\sqrt{(n/D)\log(n/D)}) = \Omega(\sqrt{nD \log(n/D)})$.

Corollary 3. *Every deterministic protocol requires $\Omega(\sqrt{Dn \log(n/D)})$ rounds to accomplish broadcast on networks with local knowledge and radius D.*

4 Broadcasting Algorithm in Networks of Radius 2

In this section we develop algorithm $\mathcal{A}_1$, whose complexity differs from the lower bound by only a polylogarithmic multiplicative factor. It will also be a subroutine for the protocol broadcasting in networks of any radius, c.f., Section 5.

4.1 Description of Algorithm $\mathcal{A}_1$

Testing and election subroutines. First we define two auxiliary problems for a radio network $G(V, E)$ with distinguished source s, where each station knows its neighbors. Recall that L_k denotes the set of nodes at distance k from the source. Assume that a set of stations $A \subseteq L_k$ is defined such that each station has a unique key in range $[R]$, for some R such that $\log R = O(\log n)$, and it knows whether it belongs to A. However, no station knows which other stations belong to A. The *k-layer emptiness testing* problem is to learn whether A is empty, that is, all nodes in $\bigcup_{i=0}^{k} L_i$ should know at the end of the protocol whether $A = \emptyset$. The *k-layer election* problem is to decide whether A is empty and, if $A \neq \emptyset$, to choose the element in A with the largest value of the key. That is, all nodes in $\bigcup_{i=0}^{k} L_i$ should know at the end of the protocol either that $A = \emptyset$ or the ID of the element of A with the largest key.

[1] Although the general framework of the proof is similar to the one in [3], we analyze slightly different class of networks to obtain an additional factor $\sqrt{\log n}$ in the lower bound formula.

Theorem 5. *[10] Consider a symmetric radio network with distinguished source node s and with no collision detection where each station knows its neighbors. Then,*

1. *there exists a protocol of time $O(1)$ solving 1-layer emptiness testing;*
2. *there exists a protocol of time $O(\log n)$ solving 1-layer election problem.*

Introduction to algorithm $\mathcal{A}_1$. Below, we present an algorithm $\mathcal{A}_1$ broadcasting in networks of radius 2. It consists of four *Phases*. Each time we check in algorithm $\mathcal{A}_1$ whether a subset of L_1 is empty or we choose an element of this subset, the appropriate protocol for 1-layer emptiness or 1-layer election from Theorem 5 is applied. Notice that, when one node v from L_1 is chosen, it can pass any message M to all elements of $L_1 \cup \{s\}$ in two rounds: first v sends this message to the source s, then s sends M to all elements of L_1.

During execution of algorithm $\mathcal{A}_1$ we conceptually *delete*, or *remove*, some nodes from the network, which means that these nodes are switched off (i.e., become idle) in the following parts of the algorithm. Therefore, all references to the network graph, layers L_1, L_2 and to the sets of neighbors of nodes (i.e., to $\Gamma(v)$ and $d(v)$, for a station v) in the following description of the algorithm will be made with respect to the values of these parameters after removal of *deleted* nodes and edges adjacent to them from the network reachability graph. Each time we will remove nodes from the network during Phases 1 and 4 of algorithm $\mathcal{A}_1$, all nodes in L_1, as well as the source s, will be aware of this fact and will send this information in their messages. However, in general, it is sufficient that non-removed neighbors of a removed node v know about the deletion of v (this issue becomes nontrivial in Phase 3).

High-level description of $\mathcal{A}_1$. The idea of the algorithm is as follows. We gradually decrease the size of the network graph by removing some nodes from it, i.e., by deciding that some nodes remain idle and do not participate in the further part of the algorithm; each station is aware whether it is removed or not. However, an invariant will be maintained that a node from L_1 can be removed only when all its neighbors in L_2 are informed, and a node from L_2 can be removed only when it is informed already. Next we describe Phases 1-4.

Phase 1. Using 1-layer election we first eliminate all nodes from L_1 that have at least $\sqrt{n}$ neighbors in L_2. More precisely, we delete some nodes from L_1, together with their neighbors in L_2, such that in the resulted graph (i.e., after these deletions), no node in L_1 has more than $\sqrt{n}$ neighbors in L_2. Since each such node eliminates at least $\sqrt{n}$ nodes from the graph, and since it can be chosen in $O(\log n)$ rounds (see Theorem 5), Phase 1 requires $O(\sqrt{n} \log n)$ rounds. Moreover, thanks to connection to the source, all stations from L_1 are aware of the deleted nodes, and therefore they know their neighborhood in the remaining network graph.

Phase 2. When there are no more nodes in L_1 with at least $\sqrt{n}$ (remaining) neighbors in L_2, we cannot continue choosing the remaining nodes in L_1 sequentially (to inform their neighbors in L_2), since this might require $\omega(\sqrt{n})$ rounds.

Instead, some nodes in L_2 can be informed in parallel. To this aim, we execute a sequence of linear $(N, 2^i, 1/2)$-selectors, for consecutive $i = 0, 1, \ldots, (1/2)\log n$, on nodes in L_1, which ensures that all stations from L_2 of degree at most $\sqrt{n}$ are informed (Phase 2), c.f., Corollary 2. Indeed, if X is a set of neighbors of a node v and $2^{i-1} < |X| \leq 2^i$, then at least half of neighbors of v will be heard by v during the execution of $(N, 2^i, \frac{1}{2})$-selector. Hence, the degrees of all stations from L_2 which were not informed are larger than $\sqrt{n}$ after Phase 2.

Phase 3. If stations from L_1 knew which of their neighbors are *not informed*, we could choose sequentially (as we will do later in Phase 4) stations from L_1 with the largest number of uninformed neighbors in L_2 and remove them from the graph together with their neighbors. Such a process would inform all stations in $O(\sqrt{n}\log n)$ rounds, since we can benefit from the fact that removed stations from L_2 "eliminate" many edges of the graph (recall that their degrees are larger than $\sqrt{n}$).

Unfortunately, we do not know whether the task of acquiring such a knowledge by the considered stations in L_1 is feasible in $O(\sqrt{n}\ \mathrm{polylog}(n))$ rounds. However, in Phase 3 we design a protocol which achieves similar goal with slightly relaxed knowledge requirements. Namely, we require that in the sub-network remaining after Phase 3, the nodes in $L_1 \cup L_2$ with degree smaller than $\sqrt{n}$ constitute only small isolated connected components (here by small we understand $O(\sqrt{n})$) and each station knows its whole component. This gives stations a knowledge about uninformed neighbors in L_2 and will allow informing all uninformed nodes in L_2 (i.e., those with degrees at least $\sqrt{n}$) in $O(\sqrt{n}\log n)$ rounds later in Phase 4, by using a greedy process similar to the one in Phase 1.

In order to trim the network graph to obtain the desired property at the end of Phase 3, we keep building a specific clustering allowing efficient propagation of knowledge inside each cluster, and simultaneously we uncover nodes that gather large information about its surrounding (i.e., information about $\Omega(\sqrt{n})$ remaining nodes that are reachable through the intra-cluster communication in $O(\sqrt{n}\ \mathrm{polylog}(n))$ rounds). The uncovered node delivers the information about its surrounding to all nodes in L_1 via the source, and thus the nodes in this surrounding also become uncovered. In the process of building the clustering, we keep joining clusters in a way guarantying fast intra-cluster communication, until they become big (and then uncovered) or isolated. Then, at the end of Phase 3, a short $O(\log^2 n)$ broadcasting schedule is designed locally for all nodes in L_1 uncovered in Phase 3, so that they can successfully inform all their neighbors in L_2, among which some may be still not informed. (This follows from the fact that some nodes in L_1 are uncovered by another member of their clusters, as a part of its surrounding, so they might not have had an opportunity to transmit successfully.) The details of Phase 3 include several novel algorithmic techniques and synchronization between them, and therefore they are deferred to the full version of the paper. Below we describe a high-level idea of how the clusters are joined and how uncovering is done.

Initially each node participating in Phase 3 constitutes a single cluster. Suppose we are given a partition of participating nodes into connected clusters, each

of them is not big and provides intra-cluster communication schedule that allows exchanging point-to-point messages between any two nodes v, w in the cluster in $O(\sum_{i \leq k} d(v_i)\ \text{polylog}(n))$ rounds, where $v = v_1, \ldots, v_k = w$ is a path between v and w in the cluster. It can be argued that any two nodes in the cluster can therefore communicate in $O(\sqrt{n}\ \text{polylog}(n))$ rounds. Consider a single node in a cluster. It learns the minimum ID of nodes in its cluster in $O(\sqrt{n}\ \text{polylog}(n))$ rounds, and then it locally computes the product of its selector schedule and the minimum ID. More precisely, the local transmission schedule of a node is defined as follows: whenever the node belongs to the currently considered set in the selector family, it performs a sequence of silences/transmissions corresponding to the 0-1 representation of the hold minimum ID; otherwise it stays idle for $\log N$ rounds. It can be shown that when using the obtained schedules, several clusters exchange messages and join into bigger clusters, in $O(\sqrt{n}\ \text{polylog}(n))$ rounds. This is because selectors combined with the minimum IDs of the clusters (to which nodes belong) assure that a constant fraction of *inter-cluster* edges will propagate a message successfully. After joining into bigger clusters, nodes interleave their previous intra-cluster schedules with the newly computed ones, which, as we will show, preserves the required property of fast intra-cluster communication with respect to the new clusters. This invariant assures that every such joining operation lasts $O(\sqrt{n}\ \text{polylog}(n))$ rounds. Because after each of them a constant fraction of inter-cluster edges become intra-cluster edges, this process can be continued no more than $\log m = O(\log n)$ times, where m is the number of edges in the graph. This gives $O(\sqrt{n}\ \text{polylog}(n))$ bound on the length of joining processes in Phase 3.

The above joining process can be applied only to small clusters. Therefore, once a surrounding of a node in L_1 becomes big (i.e., the cluster itself has become big after the last merge), it participates in the process of electing nodes in its cluster such that each of them will cover $\Omega(\sqrt{n})$ remaining nodes in the network (we say that a node v *covers* other uncovered node if v has knowledge that this node belongs to the network and it knows some edge adjacent to it). This is done through the source by using election procedure, c.f., Theorem 5. After that the uncovered parts of the network (which, as we will show, cover all newly created big clusters), are conceptually removed from the graph of participating nodes, and the joining process described above can be resumed with respect to the remaining small clusters. The process of uncovering components takes $O(\sqrt{n}\ \text{polylog}(n))$ rounds in total, by arguments similar to the one used for Phase 1.

At the end of Phase 3, the remaining nodes switch to Phase 4, while the nodes in L_1 that have been uncovered (together with their neighbors) in Phase 3 compute a short $O(\log^2 n)$ broadcast schedule to inform all their neighbors. For this purpose, a centralized algorithm from [5] is applied, as all nodes in L_1 share the same knowledge about uncovered nodes. All together: joining clusters, uncovering components and final broadcast schedule, take $O(\sqrt{n}\ \text{polylog}(n))$ rounds.

Phase 4. The source sequentially elects elements of L_1 with largest remaining neighborhoods.

The structure of Algorithm $\mathcal{A}_1$ is as follows:

Algorithm $\mathcal{A}_1$

Phase 1
While the set $X = \{v \mid v \in L_1 \text{ and } d(v) \geq \sqrt{n}\}$ is not empty:

1. choose $v \in X$ with the largest ID, using the protocol for 1-layer election;
2. v transmits a message and informs L_1 about $\Gamma(v)$ via the source;
3. remove $(\Gamma(v) \cap L_2) \cup \{v\}$ from the graph.

Phase 2
Execute the sequence of linear $(N, 2^i, \frac{1}{2})$-selectors, for consecutive $i = 0, 1, \ldots, \frac{\log n}{2}$ on nodes of L_1.

Phase 3
This phase removes some number of nodes from L_1 and L_2. As the result, we obtain the network with properties (a)–(d) specified in Lemma 1.

Phase 4
While $X = \{v \mid v \in L_1 \text{ and } \Gamma(v) \cap L_2 \neq \emptyset\}$ is not empty:

1. choose $v \in \{x \in L_1 \mid |\Gamma(x) \cap L_2| = \max_{w \in L_1} |\Gamma(w) \cap L_2|\}$ with the largest ID, using the protocol for 1-layer election and IDs $(|\Gamma(x) \cap L_2|, x)$ with lexicographic ordering;
2. v transmits a message and informs L_1 about $\Gamma(v)$ via the source;
3. remove $(\Gamma(v) \cap L_2) \cup \{v\}$ from the graph.

4.2 Analysis of Algorithm $\mathcal{A}_1$

Properties of Phases 1 and 2 are quite straightforward, therefore we will state them later in the proof of the final theorem. Now we focus on the properties of Phase 3, and based on them we analyze the complexity of Phase 4.

Lemma 1. *Time complexity of Phase 3 is $O(\sqrt{n} \log^6 n)$. Moreover, the graph $G(V, E)$ corresponding to the network at the end of Phase 3 satisfies:*

(a) $\Gamma_G(v) \leq \sqrt{n}$ for each $v \in L_1$;
(b) $\Gamma_G(v) > \sqrt{n}$ for each $v \in L_2$;
(c) each station $v \in L_1$ knows IDs of its neighbors from L_2 in G;
(d) each station deleted from the network is informed.

Using the properties stated in Lemma 1 we can analyze time complexity of Phase 4. Let $\mathcal{E}_1(n)$ be maximum of time complexities of 1-layer emptiness testing and 1-layer election problem. Although $\mathcal{E}_1(n) = O(\log n)$ according to Theorem 5, we present complexity analysis of $\mathcal{A}_1$ explicitly specifying the number of executions of election and emptiness testing, since we will apply this result for broadcasting in networks with larger diameter.

Proposition 1. *All elements of L_2 become informed after at most $(2\frac{n_1}{\sqrt{n}} + 1)\log n$ executions of steps $1-3$ of Phase 4, where $n_1 = |L_1|$. That is, time complexity of Phase 4 is $O((\frac{n_1}{\sqrt{n}} + 1) \cdot \mathcal{E}_1(n) \log n)$.*

Theorem 6. *The algorithm $\mathcal{A}_1$ performs broadcasting in radio networks of radius 2 in time $O(\sqrt{n}\log^6 n + \mathcal{E}_1(n) \cdot \frac{n_1+n_2}{\sqrt{n}} \cdot \log n) = O(\sqrt{n}\log^6 n)$.*

Proof. (Sketch) Since the above claimed time complexity of $\mathcal{A}_1$ corresponds to the time complexity of Phase 3 stated in Lemma 1, it remains to analyze Phases 1, 2 and 4. Time of Phase 2 is $O(\sum_{i=1}^{(\log n)/2} 2^i \log n) = O(\sqrt{n}\log n)$, according to Corollary 2. Phase 1 consists of at most $1+n_2/\sqrt{n}$ calls of the election procedure, where $n_2 = |L_2|$, since each execution of the election (but the last one) deletes at least $\sqrt{n}$ stations from L_2. Finally, time complexity of Phase 4 is $O(\mathcal{E}_1(n) \cdot \log n \cdot \frac{n_1}{\sqrt{n}})$, as stated in Proposition 1.

As for correctness of Algorithm $\mathcal{A}_1$, it follows from Lemma 1 and the fact that a node $v \in L_1$ is deleted in Phase 1 or Phase 4 only when all its neighbors are informed, while a node $v \in L_2$ is deleted only when it is informed. □

Finally, we make an additional observation, which will be useful for designing an extension of protocol $\mathcal{A}_1$ to multi-hop networks.

Corollary 4. *After execution of $\mathcal{A}_1$, the stations from L_1 can build an (L_1, L_2)-broadcast protocol working in time $O(\log^2 n)$.*

Proof. (Sketch) All nodes from L_1 can compute an (L_1', L_2')-broadcast protocol S_1 of required size, where $L_1' \subseteq L_1$ and $L_2' \subseteq L_2$ are the nodes uncovered in Phases 1, 3 and 4 (c.f., Theorem 3). The graph spanned on all remaining nodes can be partitioned into connected components such that there are no edges between these connected components in the original network, and each node v knows its whole connected component $G(v)$; it follows from the structure of Phase 3, that only small components that cannot merge into bigger ones remain at the end of this phase. Therefore, each node v can compute a $(L_1 \cap G(v), L_2 \cap G(v))$-broadcast protocol. Since there are no edges between the components, the schedules for all components can be executed simultaneously without causing additional collisions, forming a new protocol S_2. Concatenation of S_1 and S_2 gives a (L_1, L_2)-broadcast protocol working in time $O(\log^2 n)$. □

5 Broadcasting in Networks with Any Radius $1 \leq D \leq n$

In this section we describe a deterministic algorithm accomplishing broadcast in time $O(D\sqrt{n}\log^6 n)$ on any network of radius D. The algorithm work in stages. After the kth stage of the algorithm, for $k \in [D]$, where D is the radius of the network, the following properties will be satisfied:

(P1) All nodes from $\bigcup_{i=0}^{k} L_i$ are informed and each node $v \in L_i$, for $i \leq k$, knows its layer i.

(P2) For each $i \in [k-1]$, the protocol $\textsc{Send}_i$ is constructed, which performs (L_{i-1}, L_i)-broadcast in time $O(\log^2 n)$, i.e., if all nodes in L_i have the same message M, the protocol $\textsc{Send}_i$ makes M known to all nodes of L_{i+1} in time $O(\log^2 n)$.

(P3) For each $i \in [k-1]$, the protocol $\textsc{Test}_i$ is constructed which solves the i-layer emptiness testing problem in time $O(i \log^2 n)$.

The term "protocol is constructed" means here that each node knows its schedule in some protocol solving the appropriate communication problem.

Observe that, after application of Algorithm $\mathcal{A}_1$, the above statements are satisfied for $k = 2$ (i.e., (P1) follows from Theorem 6 and (P2) follows from Corollary 4, and (P3) follows from Theorem 5). Assume that the above properties are satisfied for $k \geq 2$. First, we would like to show how the protocol $\textsc{Test}_k$ can be build without any communication in the network, assuming $\textsc{Send}_i$ and $\textsc{Test}_i$ are known for $i < k$. Below, we assume that $A \subseteq L_k$ is the set of stations for which we test emptiness.

Procedure $\textsc{Test}_k(A)$

1: nodes from L_{k-2} execute protocol $\textsc{Send}_{k-2}$ with the same (arbitrary) message M_1; at the same time, each element of $A \subseteq L_k$ sends a message M_2 in each of $|\textsc{Send}_{k-2}|$ rounds different from M_1, where $|\textsc{Send}_{k-2}|$ denotes the time of $\textsc{Send}_{k-2}$;
2: each station $v \in L_{k-1}$ which could not hear a message M_1 from L_{k-2} in the preceding $|\textsc{Send}_{k-2}|$ rounds belongs to the set A';
3: execute $\textsc{Test}_{k-1}(A')$, let R be the result of this execution known to all elements of L_{k-1};
4: execute $\textsc{Send}_{k-1}$ with the message R.

Assume that time of $\textsc{Send}_i$ is at most $c_1 \log^2 n$ and time of $\textsc{Test}_i$ is at most $c_2 i \log^2 n$ for each $i < k$ and $c_2 > 2c_1$. Then, time of the above protocol is at most $2c_1 \log^2 n + c_2(k-1)\log^2 n < c_2 k \log^2 n$ which shows that time of $\textsc{Test}_k$ is $O(k \log^2 n)$.

Procedure $\textsc{Elect}_k$. Using the protocol $\textsc{Test}_k$, one can build a protocol $\textsc{Elect}_k$ solving the kth layer election problem, i.e., chooses an element of $A \subseteq L_k$ with the largest key (keys are polynomial wrt n), provided A is not empty. Such a protocol requires $\log n$ execution of $\textsc{Test}_k$, since it gradually decreases A using binary selection. Therefore, the complexity of protocol $\textsc{Elect}_k$ is $O(k \log^3 n)$.

Procedure $\textsc{Inform}_k$. Algorithm $\mathcal{A}_1$ relies on the fact that all elements of L_1 are connected to the source and therefore, once an element $v \in L_1$ is elected, it can pass any message M to all elements of L_1 in two rounds (through the source). We need a counterpart of this possibility in the case when a node $v \in L_k$ for $k > 1$ wants to pass a message M to all other elements of L_k. Such a message can be first sent to the source in $k-1$ rounds in the following way. Assume that each station v stores $prec(v)$, id of the station which informed v. In order to send a message from $v_0 \in L_k$ to s in k rounds, $v_i = prec(v_{i-1})$ sends a message from

L_{k-i+1} to L_{k-i} in the ith round, for $i \in [k]$. Then, the message is transmitted from the source to L_k by the application of $\textsc{Send}_0, \textsc{Send}_1, \ldots, \textsc{Send}_{k-1}$. We call such a protocol $\textsc{Inform}_k$. Note that its time complexity is $O(k \log^2 n)$ by (P2).

Algorithm $\mathcal{A}_k$. Equipped with the protocols $\textsc{Send}_k$, $\textsc{Elect}_k$ and $\textsc{Inform}_k$, we are ready to transmit the broadcasted message from L_k to L_{k+1}. Namely, we mimic the algorithm $\mathcal{A}_1$ in the following way:

(a) nodes from L_k work as the elements of L_1 in $\mathcal{A}_1$;
(b) newly informed nodes and stations in $L_k \cup L_{k-1}$ work as the elements of L_2 in $\mathcal{A}_1$ (nodes informed during this execution, which do not belong to L_{k-1}, learn that they belong to L_{k+1});
(c) each time emptiness of some subset of L_1 should be checked in $\mathcal{A}_1$, the procedure $\textsc{Test}_k$ is applied;
(d) each time an element from some subset of L_1 should be chosen in $\mathcal{A}_1$, the procedure $\textsc{Elect}_k$ is applied;
(e) each time a message M from $v \in L_1$ should be transmitted through the source to the whole L_1, the procedure $\textsc{Inform}_k$ is used.

One subtle issue is that our presentation of Algorithm $\mathcal{A}_1$ utilized the fact that nodes in layer L_1 know which of their neighbors are in which layer. A corresponding property may not be true after moving to the next layers. Therefore, in order to apply algorithm $\mathcal{A}_1$, after the adaptation described in the above items (a)–(e), for propagating the broadcast message from L_k to L_{k+1}, a few more subtle technical fixes in Phase 3 are needed (they do not, however, change the general structure of the algorithm and its analysis). Let $\mathcal{A}_k$ denote algorithm $\mathcal{A}_1$ modified as described in (a)–(e).

Procedure $\textsc{Send}_k$. It can be argued that the knowledge about the nodes collected during the execution of $\mathcal{A}_1$ is sufficient for designing a (L_1, L_2)-broadcast protocol of size $O(\log^2 n)$ (c.f., Corollary 4). This property generalizes to $\mathcal{A}_k$, since the information acquired by L_k about L_{k+1} corresponds to the information about L_2 known to L_1 during the execution of $\mathcal{A}_1$. That is, the nodes in L_k can build a (L_k, L_{k+1})-broadcast protocol $\textsc{Send}_k$ of size $O(\log^2 n)$ after the execution of $\mathcal{A}_k$. Thus, (P1)–(P3) are satisfied after the execution of $\mathcal{A}_k$. Based on the constructions of $\mathcal{A}_k, \textsc{Test}_k, \textsc{Send}_k$, and $\textsc{Elect}_k$, we obtain the following broadcast algorithm $\mathcal{B}$:

Algorithm $\mathcal{B}$

1: The source sends the broadcasted message.
2: **for** $k = 2, 3, \ldots$ **do**
3: Execute $\mathcal{A}_k$;
4: Build $\textsc{Test}_k, \textsc{Send}_k$, and $\textsc{Elect}_k$;
5: Execute $\textsc{Test}_k(L_k)$ in order to check whether L_k is empty;
6: If L_k is empty, finish the algorithm.

Let us stress here that deletion of nodes in phases 1–4 of $\mathcal{A}_k$ applies only to the execution of $\mathcal{A}_k$ — the deleted nodes are restored after that.

Theorem 7. *Algorithm $\mathcal{B}$ completes broadcasting in time $O(D\sqrt{n}\log^6 n)$ in any n-node radio network of radius D.*

Proof. (Sketch) The above discussion justifies the fact that properties (P1)-(P3) are satisfied in consecutive stages defined by the for loop of algorithm $\mathcal{B}$. Moreover, $\textsc{Elect}_k$ works in time $O(k\log^3 n)$ for each k, as discussed earlier. Therefore, the number of rounds in the kth stage of the algorithm is

$$O\left(\sqrt{n}\log^6 n + (k\log^3 n)\cdot\frac{n_{k-1}+n_k+n_{k+1}}{\sqrt{n}}\right) ,$$

due to Theorem 6 (recall that nodes from L_{k-1} and L_{k+1} play the role of L_2 in the execution of $\mathcal{A}_k$).

The test of emptiness of L_k in line 6 guarantees that the algorithm finishes its work only after informing all nodes in the Dth layer, where D is the radius of the network (recall that, after execution of $\textsc{Test}_k$ on the set A, all elements of $\bigcup_{i=1}^{k} L_i$ know the result of the test).

Observe that each execution of $\textsc{Inform}_k$ in Algorithm $\mathcal{A}_k$ (e.g., in step 2 of Phase 1 or Phase 4), for $k \in [D]$, is preceded by an execution of $\textsc{Elect}_k$. Hence, the executions of $\textsc{Inform}_k$, for $k \in [D]$, have no impact on the asymptotic complexity of the algorithm (as the complexity of $\textsc{Inform}_k$ is asymptotically smaller than the complexity of $\textsc{Elect}_k$). Thus, the time complexity of algorithm $\mathcal{B}$ is

$$O\left(D\cdot\sqrt{n}\log^6 n + \sum_{k=1}^{D-1} k\log^3 n\frac{n_{k-1}+n_k+n_{k+1}}{\sqrt{n}}\right) = O(D\sqrt{n}\log^6 n) . \qquad \square$$

6 From Broadcasting to Minimal Dominating Set

Observe that the lower bound $\Omega(\sqrt{Dn\log(n/D)})$ on broadcasting can be extended in a natural way to the problem of distributed construction of MDS, since at least one node in the last component of the network used in the proof of the lower bound on broadcasting (c.f., Theorem 4 and Corollary 3) must be reached by the message initiated by the source. Indeed, otherwise all elements of the last component must have decided whether they belong to MDS based merely on the information about their neighbors in a graph, which is insufficient for some network topologies.

Algorithms $\mathcal{A}_k$ and $\mathcal{B}$ could be used as black boxes to obtain MDS in a distributed way in asymptotically the same number of rounds. In the beginning, the broadcasting algorithm $\mathcal{B}$ is run. It is enough to compute sets MDS_k, being the intersection of the final MDS with layer L_k of the network, after (and based on) the execution of $\mathcal{A}_k$, where $k = 3i+1$ for non-negative integers i not larger than $(D-2)/3$. Assume that the execution of algorithm $\mathcal{A}_k$ has just finished. First, all nodes that end up Phase 3 in small components without outside neighbors, apply a centralized greedy schedule to select a MDS for the component. Next, nodes that were elected by the source during Phases 1, 3 and 4 check, one after another in the reversed order to the one they were elected in the execution of $\mathcal{A}_k$,

whether they have neighbors that have not been dominated yet and whether they have neighbors already selected to the dominating set; both checks are done by using procedure TEST. If the first question is answered affirmative or the second one is answered negative, the node includes itself to the dominating set.

It follows directly from the properties of broadcasting and the above greedy selection made from the broadcasting nodes, that the above algorithm computes a dominating set, and no node can be removed without violating the domination property. In terms of round complexity, the MDS algorithm mimics some operations that occurred in the original execution of the broadcast algorithm $\mathcal{B}$, and therefore its time complexity is (asymptotically) upper-bounded by the time complexity of algorithm $\mathcal{B}$. Thus the following result holds.

Theorem 8. *Every distributed solution building a MDS requires* $\Omega(\sqrt{Dn\log\frac{n}{D}})$ *rounds on some radio networks of radius* D*. There exists a distributed algorithm constructing a MDS in* $O(D\sqrt{n}\log^6 n)$ *on any radio network of radius* D*.*

References

1. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. J. Comput. Syst. Sci. 45(1), 104–126 (1992)
2. De Bonis, A., Gąsieniec, L., Vaccaro, U.: Generalized Framework for Selectors With Applications in Optimal Group Testing. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 81–96. Springer, Heidelberg (2003)
3. Brito, C.F., Vaya, S.: Improved lower bound for deterministic broadcasting in radio networks. Theor. Comput. Sci. 412(29), 3568–3578 (2011)
4. Chlamtac, I., Kutten, S.: Tree-based broadcasting in multihop radio networks. IEEE Trans. Computers 36(10), 1209–1223 (1987)
5. Chlamtac, I., Weinstein, O.: The wave expansion approach to broadcasting in multihop radio networks. IEEE Transactions on Communications 39(3), 426–433 (1991)
6. Chlebus, B.S., Kowalski, D.R.: Almost Optimal Explicit Selectors. In: Liśkiewicz, M., Reischuk, R. (eds.) FCT 2005. LNCS, vol. 3623, pp. 270–280. Springer, Heidelberg (2005)
7. Clementi, A.E.F., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: SODA, pp. 709–718 (2001)
8. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. J. Algorithms 60(2), 115–143 (2006)
9. Goldberg, L.A., Jerrum, M., Leighton, F.T., Rao, S.: Doubly logarithmic communication algorithms for optical-communication parallel computers. SIAM J. Comput. 26(4), 1100–1119 (1997)
10. Kowalski, D.R., Pelc, A.: Time of deterministic broadcasting in radio networks with local knowledge. SIAM J. Comput. 33(4), 870–891 (2004)
11. Kowalski, D.R., Pelc, A.: Broadcasting in undirected ad hoc radio networks. Distributed Computing 18(1), 43–57 (2005)
12. Kushilevitz, E., Mansour, Y.: An omega(d log (n/d)) lower bound for broadcast in radio networks. SIAM J. Comput. 27(3), 702–712 (1998)

On the Impact of Identifiers on Local Decision*

Pierre Fraigniaud[1,**], Magnús M. Halldórsson[2,***], and Amos Korman**

[1] CNRS and University Paris Diderot, France
[2] ICE-TCS, School of Computer Science, Reykjavik University, Iceland
{pierre.fraigniaud,amos.korman}@liafa.univ-paris-diderot.fr,
mmh@ru.is

Abstract. The issue of identifiers is crucial in distributed computing. Informally, identities are used for tackling two of the fundamental difficulties that are inherent to deterministic distributed computing, namely: (1) *symmetry breaking*, and (2) *topological information gathering*. In the context of *local computation*, i.e., when nodes can gather information only from nodes at bounded distances, some insight regarding the role of identities has been established. For instance, it was shown that, for large classes of *construction* problems, the role of the identities can be rather small. However, for the identities to play no role, some other kinds of mechanisms for breaking symmetry must be employed, such as edge-labeling or sense of direction. When it comes to local distributed *decision* problems, the specification of the decision task does not seem to involve symmetry breaking. Therefore, it is expected that, assuming nodes can gather sufficient information about their neighborhood, one could get rid of the identities, without employing extra mechanisms for breaking symmetry. We tackle this question in the framework of the $\mathcal{LOCAL}$ model.

Let LD be the class of all problems that can be *decided* in a constant number of rounds in the $\mathcal{LOCAL}$ model. Similarly, let LD* be the class of all problems that can be decided at constant cost in the anonymous variant of the $\mathcal{LOCAL}$ model, in which nodes have no identities, but each node can get access to the (anonymous) ball of radius t around it, for any t, at a cost of t. It is clear that LD* $\subseteq$ LD. We conjecture that LD* = LD. In this paper, we give several evidences supporting this conjecture. In particular, we show that it holds for *hereditary* problems, as well as when the nodes know an arbitrary upper bound on the total number of nodes. Moreover, we prove that the conjecture holds in the context of *non-deterministic* local decision, where nodes are given certificates (independent of the identities, if they exist), and the decision consists in verifying these certificates. In short, we prove that NLD* = NLD.

Keywords: Distributed complexity, locality, identities, decision problems, symmetry breaking, non-determinism.

* This work is supported by the *Jules Verne* Franco-Icelandic bilateral scientific framework.
** Additional support from ANR project DISPLEXITY, and INRIA project GANG.
*** Supported by Iceland Research Foundation grant-of-excellence 90032021.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 224–238, 2012.

1 Introduction

1.1 Background and Motivation

The issue of identifiers is crucial in distributed computing [2, 33]. Indeed, the correct operation of deterministic protocols often relies on the assumption that each processor u comes with with a unique *identity*, $\mathrm{Id}(u)$ [9]. Informally, in network computing, such an identity assignment is crucial for tackling two of the fundamental difficulties that are inherent to distributed computing, namely: (1) *symmetry breaking*, and (2) *topological information gathering*.

The use of identities for tackling the above two difficulties is illustrated well in the context of *local* algorithms [30, 32]. Indeed, in the $\mathcal{LOCAL}$ model [35], an algorithm that runs in t communication rounds, assuming an identity assignment, can be viewed as composed of two parts: first, collecting at each node u, the ball $B(u,t)$ of radius t around it (together with the inputs of nodes), and second, deciding the output at u based solely on the information in $B(u,t)$. To achieve these two tasks, one should first obtain the ball $B(u,t)$, which may not be possible if the underlying graph is anonymous (i.e., without identities). Moreover, even if obtaining the ball is possible, e.g., if the structure of the graph allows it, the absence of unique identities given to the nodes may prevent the algorithm from breaking symmetry. For example, in the absence of unique identities, it is impossible to design a distributed deterministic coloring algorithm, even for the symmetric connected graph composed of two nodes only. In fact, to the best of our knowledge, all algorithms in the $\mathcal{LOCAL}$ model are designed assuming the presence of pairwise distinct identities or some other type of node-labeling or edge-labeling, including, e.g., sense of direction [5, 22, 29, 31, 33, 34].

The seminal paper of Naor and Stockmeyer [33] provides an important insight regarding the role of identities in local computation. Informally, they show that, even though identities are necessary, in many cases the actual values of identities is not crucial, and only their relative order matters. Specifically, [33] shows that for a particular class of problems, called LCL (for *Locally Checkable Languages*), if there exists a local algorithm that, for any identity assignment, constructs an instance of a problem in LCL in constant number of rounds, then there exists an *order invariant*[1] algorithm for that problem that runs in the same number of rounds. LCL restricts its concern to graphs with constant maximum degree, and to problems with a constant number of inputs. The assumption on the size of the inputs of problems in LCL was shown necessary in [21], by exhibiting a natural problem that is locally checkable, has unbounded input size, can be solved in 1 round with identities, but cannot be solved in constant time by any order invariant algorithm. The role of identities can also be gauged by comparing their impact to that of "orientation mechanisms". For instance, Göös et al. [20] have shown that for a large class of optimization problems, called PO-checkable

[1] Essentially, an order invariant algorithm uses the actual values of the identities only to impose an ordering between the nodes, that is, it behaves the same for any two identity assignments that preserve the total order between the nodes. For more details refer to [33].

problems, local algorithms do not benefit from any kind of identifiers: if a PO-checkable optimization problem can be approximated with a local algorithm, the same approximation factor can be achieved in anonymous networks if the network is provided with a port-numbering and an orientation.

The discussion above involved distributed *construction* tasks, including, e.g., graph coloring [5, 29, 30, 33, 34], maximal independent set [30, 34], and maximal matching [22, 31]. When it comes to distributed *decision* tasks [13, 14], symmetry breaking issues do not however seem to play a role. Informally, a decision task requires the nodes to "collectively decide" whether the given instance (i.e., a graph with inputs to the nodes) satisfies some specific properties. For instance, deciding coloring requires, given a colored graph, to check whether this graph is properly colored. The meaning of "collectively decide" is as follows. On a legal instance, all nodes should output "yes", and on an illegal one, at least one node should output "no". Note that it is not really important whether this node is unique or not; hence, this specification does not inherently require any symmetry breaking. Therefore, assuming that each node u can obtain the ball $B(u,t)$, it makes sense that the assumption of having an identity assignment may not be crucial for achieving correct decision.

1.2 Model and Objectives

We tackle the question of whether identities play a role in decision problems in the framework of the aforementioned $\mathcal{LOCAL}$ model [35], which is a standard distributed computing model capturing the essence of locality. Recall that, in this model, processors are nodes of a connected network $G = (V(G), E(G))$, have pairwise distinct identities, and have inputs. More formally, a *configuration* is a triplet $(G, \mathbf{x}, \mathrm{Id})$ where G is a connected graph, every node $v \in V(G)$ is assigned as its *local input* a binary string $\mathbf{x}(v) \in \{0,1\}^*$, and $\mathrm{Id}(v)$ denotes the identity of node v. (In some problems, the local input of every node is empty, i.e., $\mathbf{x}(v) = \epsilon$ for every $v \in V(G)$, where ϵ denotes the empty binary string). Processors are woken up simultaneously, and computation proceeds over the input configuration $(G, \mathbf{x}, \mathrm{Id})$ in fault-free synchronous *rounds* during which every processor exchanges messages of unlimited size with its neighbors in the underlying network G, and performs arbitrary individual computations on its data. In many cases, the running time of an algorithm is measured with respect to the size n of G: the running time of an algorithm is defined as the maximum number of rounds it takes to terminate at all nodes, over all possible n-node networks. Similarly to [21, 33], we consider algorithms whose running time is independent of the size of the network, that is they run in constant time.

Let $B(u,t)$ be the ball centered at u, of radius t, excluding the edges between two nodes at distance exactly t from u. As mentioned before, without loss of generality, any algorithm running in time $t = O(1)$ in the $\mathcal{LOCAL}$ model consists of:

1. Collecting (in t rounds) at every node u the structure of the ball $B(u,t)$ together with all the inputs $\mathbf{x}(v)$ and identities $\mathrm{Id}(v)$ of these nodes, and,

2. Performing some individual computation at every node. (Note that we do not insist on efficient computations, as long as they involve functions that are computable; Of course, in practice, we seek for polynomial-time algorithms, but our results do not rely on this assumption).

We define the *anonymous* $\mathcal{LOCAL}$ model similarly to the $\mathcal{LOCAL}$ model, except that nodes have no identities. More precisely, an input configuration in the anonymous $\mathcal{LOCAL}$ model is just a pair $(G, \mathbf{x})$. An algorithm running in time $t = O(1)$ in the anonymous $\mathcal{LOCAL}$ model consists of:

1. Getting at every node u a snapshot of the structure of the ball $B(u, t)$ together with all the inputs of the nodes in this ball, and,
2. Performing some individual computation at every node.

Note that the *anonymous* $\mathcal{LOCAL}$ model does not explicitly involve communications between nodes. Instead, it implicitly assumes that the underlying network supports the snapshot operation. Clearly, this model is not stronger than the $\mathcal{LOCAL}$ model, and possibly even strictly weaker, since a node u can no longer base its individual computation on the identities of the nodes in the ball $B(u, t)$. One can think of various other "anonymous" models, i.e., which do not involve node identities. In particular, there is a large literature on distributed computing in networks without node identities, where symmetry breaking is enabled thanks to locally disjoint port numbers (see, e.g., [18]). We consider the anonymous $\mathcal{LOCAL}$ model to isolate the role of node identities from other symmetry breaking mechanisms.[2] Our aim is to compare the power of the anonymous $\mathcal{LOCAL}$ model with the standard $\mathcal{LOCAL}$ model in order to capture the impact of identities on local distributed decision.

Recall from [13] that a *distributed language* is a decidable collection $\mathcal{L}$ of configurations. (Since an undecidable collection of configurations remains undecidable in the distributed setting too, we consider only decidable collections of configurations). A typical example of a language is

$$\texttt{Coloring} = \{(G, \mathbf{x}) \mid \forall v \in V(G), \forall w \in N(v), \mathbf{x}(v) \neq \mathbf{x}(w)\} \ ,$$

where $N(v)$ denotes the (open) neighborhood of v, that is, all nodes at distance exactly 1 from v. Still following the terminology from [13], we say that a distributed algorithm A *decides* a distributed language $\mathcal{L}$ if and only if for every configuration $(G, \mathbf{x})$, every node of G eventually terminates and outputs "yes" or "no", satisfying the following decision rules:

- if $(G, \mathbf{x}) \in \mathcal{L}$, then each node outputs "yes";
- if $(G, \mathbf{x}) \notin \mathcal{L}$, then at least one node outputs "no".

[2] In some sense, the anonymous $\mathcal{LOCAL}$ model is the strongest model among all models without node identities. Indeed, there are network problems that can be solved in the anonymous $\mathcal{LOCAL}$ model which cannot be solved in the aforementioned model that is based on locally disjoint port numbers. A simple example is to locally detect the absence of a 3-node cycle.

In the (non-anonymous) $\mathcal{LOCAL}$ model, these two rules must be satisfied for every identity assignment. That is, all processes must output "yes" on a legal instance, independent of their identities. And, on an illegal instance, at least one node must output "no", for every identity assignment. Note that this node may potentially differ according to the identity assignment. Some languages can be decided in constant time (e.g., `Coloring`), while others can easily be shown not to be decidable in constant time (e.g., "is the network planar?"). In contrast to the above examples, there are some languages whose status is unclear. To elaborate on this, consider the particular case where it is required to decide whether the network belongs to some specified family $\mathcal{F}$ of graphs. If this question can be decided in a constant number of communication rounds, then this means, informally, that the family $\mathcal{F}$ can somehow be characterized by relatively simple conditions. For example, a family $\mathcal{F}$ of graphs that can be characterized as consisting of all graphs having no subgraph from $\mathcal{C}$, where $\mathcal{C}$ is some specified finite set of graphs, is obviously decidable in constant time. However, the question of whether a family of graphs can be characterized as above is often non-trivial. For example, characterizing cographs as precisely the graphs with no induced P_4, attributed to Seinsche [36], is not easy, and requires nontrivial usage of modular decomposition.

We are now ready to define one of our main subjects of interest, the classes LD and LD*. Specifically, LD (for *local decision*) is the class of all distributed languages that can be decided by a distributed algorithm that runs in a constant number of rounds in the $\mathcal{LOCAL}$ model [13]. Similarly, LD*, the anonymous version of LD, is the class of all distributed languages that can be decided by a distributed algorithm that runs in a constant number of rounds in the anonymous $\mathcal{LOCAL}$ model. By definition, LD$^* \subseteq$ LD. We conjecture that

$$\text{LD}^* = \text{LD}.$$

In this paper, we provide several evidences supporting this conjecture. In addition, we investigate the *non-deterministic* version of these classes, and prove that they coincide. More specifically, a distributed *verification* algorithm is a distributed algorithm A that gets as input, in addition to a configuration $(G, \mathbf{x})$, a global *certificate vector* $\mathbf{y}$, i.e., every node v of a graph G gets as input two binary strings, an input $\mathbf{x}(v) \in \{0,1\}^*$ and a certificate $\mathbf{y}(v) \in \{0,1\}^*$. A verification algorithm A verifies $\mathcal{L}$ if and only if for every input configuration $(G, \mathbf{x})$, the following hold:

- if $(G, \mathbf{x}) \in \mathcal{L}$, then there exists a certificate $\mathbf{y}$ such that every node outputs "yes";
- if $(G, \mathbf{x}) \notin \mathcal{L}$, then for every certificate $\mathbf{y}$, at least one node outputs "no".

Again, in the (non-anonymous) $\mathcal{LOCAL}$ model, these two rules must be satisfied for every identity assignment, but the certificates must be the same regardless of the identities. We now recall the class NLD, for *non-deterministic local decision*, as defined in [13]: it is the class of all distributed languages that can be verified in a constant number of rounds in the $\mathcal{LOCAL}$ model. Similarly, we define NLD*,

the anonymous version of NLD, as the class of all distributed languages that can be verified in a constant number of rounds in the anonymous $\mathcal{LOCAL}$ model. By definition, $\text{NLD}^* \subseteq \text{NLD}$.

1.3 Our Results

In this paper, we give several evidences supporting the conjecture $\text{LD}^* = \text{LD}$. In particular, we show that it holds for languages defined on paths, with a finite set of input values. More generally, we show that the conjecture holds for *hereditary* languages, that is, languages closed under node deletion. Regarding arbitrary languages, and arbitrary graphs, we prove that the conjecture holds assuming that every node knows an upper bound on the total number of nodes in the input graph. (This upper bound can be arbitrary, and may not be the same for all nodes).

Moreover, we prove that equality between non-anonymous decision and anonymous decision holds in the context of *non-deterministic* local decision, where nodes are given certificates (independent of the identities, if they exist), and the decision consists in verifying these certificates. More precisely, we prove that $\text{NLD}^* = \text{NLD}$. This latter result is obtained by characterizing both NLD and NLD^*.

1.4 Related Work

The question of how to locally decide (or verify) languages has received quite a lot of attention recently. Inspired by classical computation complexity theory, it was suggested in [13] that the study of decision problems may lead to new structural insights also in the more complex distributed computing setting. Indeed, following that paper, which focused on the $\mathcal{LOCAL}$ model, efforts were made to form a fundamental computational complexity theory for distributed decision problems in various other aspects of distributed computing [13, 15–17].

The classes LD, NLD and BPLD defined in [13] are the distributed analogues of the classes P, NP and BPP, respectively. The paper provides structural results, developing a notion of local reduction and establishing completeness results. One of the main results is the existence of a sharp threshold for randomization, above which randomization does not help (at least for hereditary languages). More precisely, the BPLD classes were classified into two: below and above the randomization threshold. In [14], the authors show that the hereditary assumption can be lifted if we restrict our attention to languages on path topologies. These two results from [13, 14] are used in the current paper in a rather surprising manner. The authors in [14] then "zoom" into the spectrum of classes below the randomization threshold, and defines a hierarchy of an infinite set of BPLD classes, each of which is separated from the class above it in the hierarchy.

The precise knowledge of the number of nodes n was shown in [13] to be of large impact on non-deterministic decision. Indeed, with such a knowledge every language can be decided non-deterministically in the model of NLD. We note,

however, that the knowledge of an arbitrary upper bound on n (as assumed here in one of our results) seems to be a much weaker assumption, and, in particular, will not suffice for non-deterministically deciding all languages. In the context of construction problems, it was shown in [28] that in many case, the knowledge of n (or an upper bound on n) is not essential.

The original theoretical basis for non-determinism in local computation was laid by the theory of *proof-labeling schemes* (PLS) [19, 24–26] originally defined in [26]. As mentioned, this notion resembles the notion of NLD, but differs in the role identities play. Specifically, in PLS the designer of the algorithm may base the certificates' (called labels in the terminology of PLS) construction on the given identity assignment. In contrast, in the model of NLD, the certificates must be the same regardless of the identities of nodes. Indeed, this difference is significant: while every language can be verified by a proof labeling scheme, not every language belongs to NLD [13]. These notions also bear some similarities to the notions of *local computation with advice* [7, 10–12], *local detection* [1], *local checking* [4], or *silent stabilization* [8]. In addition, as shown later on, the notion of NLD is related also to the theory of *lifts* or *covers* [2, 3].

Finally, the classification of decision problems in distributed computing has been studied in several other models. For example, [6] and [23] study specific decision problems in the $\mathcal{CONGEST}$ model. In [25], the authors study MST verification in the PLS sense but under the $\mathcal{CONGEST}$ model of communication. In addition, decision problems have been studied in the asynchrony discipline too, specifically in the framework of *wait-free computation* [16, 17] and *mobile agents computing* [15]. In the wait-free model, the main issues are not spatial constraints but timing constraints (asynchronism and faults). The main focus of [17] is deterministic protocols aiming at studying the power of the "decoder", i.e., the interpretation of the results. While this paper essentially considers the AND-checker (since a global "yes" corresponds to all processes saying "yes"), [17] deals with other interpretations, including more values (not only "yes" and "no"), with the objective of designing checkers that use the smallest number of values.

2 Deterministic Decision

We conjecture that $\mathrm{LD} = \mathrm{LD}^*$. A support to this conjecture is that it holds for a large class of languages, namely for all *hereditary* languages, that is languages closed under node deletion. For instance, `Coloring` and `MIS` are hereditary, as well as all languages corresponding to hereditary graph families, such as planar graphs, interval graphs, forests, chordal graphs, cographs, perfect graphs, etc.

Theorem 1. $\mathrm{LD}^* = \mathrm{LD}$ *for hereditary languages.*

To prove the theorem, it is sufficient to show that $\mathrm{LD} \subseteq \mathrm{LD}^*$ for hereditary languages. This immediately follows from the statement and proof of Theorem 3.3 in [13]. Indeed, let A be a non-anonymous local algorithm deciding $\mathcal{L}$.

This deterministic algorithm is in particular a randomized algorithm, with success probabilities $p = 1$ for legal instances, and $q = 1$ for illegal instance. That is, algorithm A is a $(1, 1)$-decider for $\mathcal{L}$, according to the definition in [13]. Since $\mathcal{L}$ is hereditary, and since $p^2 + q > 1$, the existence of A implies the existence of a specific deterministic anonymous local algorithm D for $\mathcal{L}$. Indeed, the algorithm D described in the proof of Theorem 3.3 in [13] is in fact anonymous: it simply collects the ball $B(u, t)$ of radius t around each node u for some constant t, and u then decides "yes" or "no" according to whether $B(u, t) \in \mathcal{L}$ or not, regardless of the identities.

A similar proof, based on Theorem 4.1 in [14], enables to establish the following:

Theorem 2. $\mathrm{LD}^* = \mathrm{LD}$ *for languages defined on the set of paths, with a finite set of input values.*

Another evidence supporting the conjecture $\mathrm{LD} = \mathrm{LD}^*$ is that it holds assuming that nodes have access to a seemingly weak oracle. Specifically, this oracle, denoted $\mathbf{N}$, simply provides each node with an arbitrarily large upper bound on the total number of nodes in the actual instance. (It is not assumed that all the upper bounds provided to nodes are the same). We denote by $\mathrm{LD}^{*\mathbf{N}}$ the class of languages that can be decided by an anonymous local algorithm having access to oracle $\mathbf{N}$, and we prove the following:

Theorem 3. $\mathrm{LD}^* \subseteq \mathrm{LD} \subseteq \mathrm{LD}^{*\mathbf{N}}$.

Proof. We just need to prove that $\mathrm{LD} \subseteq \mathrm{LD}^{*\mathbf{N}}$. Let $\mathcal{L} \in \mathrm{LD}$, and let A be a local (non-anonymous) algorithm deciding $\mathcal{L}$. Assume that the running time of A is t. We transform A into an anonymous algorithm A' deciding $\mathcal{L}$ in time t, assuming each node u in a given input G has an access to the oracle $\mathbf{N}$, i.e., it knows an arbitrary upper bound n_u on the number of nodes in G. Algorithm A' works as follows. Each node u collects the ball $B(u, t)$ of radius t around it. Then, for every possible assignment of identities to the nodes of $B(u, t)$ taken from the range $[1, n_u]$, node u simulates the behavior of the non-anonymous algorithm A on the ball $B(u, t)$ with the corresponding identities. If, in one of these simulations, algorithm A decides "no", then A' decides "no". Otherwise, A' decides "yes".

We now prove the correctness of A'. If the input $(G, \mathbf{x}) \in \mathcal{L}$, then A accepts it for every identity assignment to the nodes of G. Therefore, since, for every node u, every possible identity assignment to the nodes of the ball $B(u, t)$ can be extended to an identity assignment to all the nodes of G, all the simulations of A by u return "yes", and hence A' accepts $\mathcal{L}$ as well. On the other hand, if $(G, \mathbf{x}) \notin \mathcal{L}$ then A rejects it for every identity assignment to the nodes of G. That is, for every identity assignment to the nodes of G, at least one node u decides "no". (Note that, this node u may be different for two different identity assignments). Let us fix one identity assignment Id to the nodes of G, in the range $[1, n]$, and let u be one node that decides "no" for Id. Let $B_{\mathrm{Id}}(u, t)$ be

the ball $B(u,t)$ with the identities of the nodes given by Id. In A', since u tries all possible identity assignments of the ball $B(u,t)$ in the range $[1, n_u]$ with $n \leq n_u$, in one of its simulations of A, node u will simulate A on $B_{\mathrm{Id}}(u,t)$. In this simulation, node u decides "no", and hence algorithm A' rejects $\mathcal{L}$ as well. □

Note that the inclusion $\mathrm{LD} \subseteq \mathrm{LD}^{*\mathbf{N}}$ holds when one imposes no restrictions on the individual sequential running time. However, the transformation of a (non-anonymous) local algorithm into an anonymous local algorithm as described in the proof of Theorem 3 is very expensive in terms of individual computation. Indeed, the number of simulations of the original local algorithm A by each node u can be as large as $\binom{n_u}{n_B}$ where n_u is the upper bound on n given by the oracle $\mathbf{N}$, and n_B is the number of nodes in the ball $B(u,t)$. This bound can be exponential in n even if the oracle provides a good approximation of n (even if it gives precisely n). It would be nice to establish $\mathrm{LD} \subseteq \mathrm{LD}^{*\mathbf{N}}$ by using a transformation not involving a huge increase in the individual sequential computation time.

3 Non-deterministic Decision

In the previous section, we have seen several evidences supporting the conjecture that $\mathrm{LD}^* = \mathrm{LD}$, but whether it holds or not remains to be proved. In this section, we turn our attention to the non-deterministic variants of these two classes, and show that they coincide. More formally, we have:

Theorem 4. $\mathrm{NLD}^* = \mathrm{NLD}$.

Proof. To prove $\mathrm{NLD}^* = \mathrm{NLD}$, it is sufficient to prove $\mathrm{NLD} \subseteq \mathrm{NLD}^*$. To establish this inclusion, we provide a sufficient condition for NLD^*-membership, and prove that it is a necessary condition for NLD-membership.

Let $I = (G, \mathbf{x})$ and $I' = (G', \mathbf{x}')$ be two input instances. A *homomorphism* from I to I' is a function $f : V(G) \to V(G')$ that preserves the edges of G as well as the inputs to the nodes. Specifically,

$$\{u, v\} \in E(G) \Rightarrow \{f(u), f(v)\} \in E(G'),$$

and f maps every node $u \in V(G)$ to a node $f(u) \in V(G')$ satisfying

$$\mathbf{x}'(f(u)) = \mathbf{x}(u).$$

For instance, assuming the nodes have no inputs, and labeling the nodes of the n-node cycle C_n by consecutive integers from 0 to $n-1$, modulo n, then the map $f : C_8 \to C_4$ defined by $f(u) = u \bmod 4$ is a homomorphism. The trivial map $g : C_8 \to K_2$ defined by $g(u) = u \bmod 2$, where K_2 is the 2-node clique, is also a homomorphism. To establish conditions for NLD- and NLD^*-membership, we require the involved homomorphisms to preserve the local neighborhood of a node, and define the notion of *t-local isomorphism*.

Let t be a positive integer. We say that I is t-local isomorphic to I' if and only if there exists an homomorphism f from I to I' such that, for every node $v \in V(G)$, f restricted to $B_G(v,t)$ is an isomorphism from $B_G(v,t)$ to $B_{G'}(f(v),t)$. We call such a homomorphism f a t-local isomorphism.

Note that a homomorphism is not necessarily a 1-local isomorphism. For instance, the aforementioned map $f : C_8 \to C_4$ defined by $f(u) = u \bmod 4$ is a 1-local isomorphism, but the map $g : C_8 \to K_2$ defined by $g(u) = u \bmod 2$ is not a 1-local isomorphism. To be a 1-local isomorphism, a homomorphism should also insure isomorphism between the balls of radius 1. Also observe that any t-local isomorphism $f : G \to G'$ is onto (because if a node of G' has no pre-image, then neither do its neighbors have a pre-image, since homomorphisms preserve edges, and so forth). To avoid confusion, it is thus useful to keep in mind that, informally, a t-local isomorphism goes from a "larger" graph to a "smaller" graph.

Definition 1. *For positive integer t, we say that $\mathcal{L}$ is* t-closed under lift *if, for every two instances I, I' such that I is t-local isomorphic to I', we have:*

$$I' \in \mathcal{L} \Rightarrow I \in \mathcal{L}.$$

So, informally, Defintion 1 states that, for a language $\mathcal{L}$ to be t-closed under lift, if a "smaller" instance I' is in $\mathcal{L}$ then any "larger" instance I that is a lift of I', i.e., satisfying that I is t-local isomorphic to I', must also be in $\mathcal{L}$. The following lemma gives a sufficient condition for NLD*-membership.

Lemma 1. *Let $\mathcal{L}$ be a language. If there exists $t \geq 1$ such that $\mathcal{L}$ is t-closed under lift, then $\mathcal{L} \in$ NLD*.*

Proof. Let $\mathcal{L}$ be a language, and assume that there exists $t \geq 1$ such that $\mathcal{L}$ is t-closed under lift. We describe an anonymous non-deterministic local algorithm A deciding $\mathcal{L}$, and performing in t rounds. The certificate of each node v is a triple $\mathbf{y}(v) = (i, G', \mathbf{x}')$ where G' is an n-node graph with nodes labeled by distinct integers in $[1,n] = \{1,\ldots,n\}$, $i \in [1,n]$, and $\mathbf{x}'$ is an n-dimensional vector. Informally, the certificates are interpreted by A as follows. The graph G' is supposed to be a "map" of G, that is, G' is interpreted as an isomorphic copy of G. The integer i is the label of the node in G' corresponding to node v in G. Finally, $\mathbf{x}'$ is interpreted as the input of the nodes in G'.

The algorithm A performs as follows. Every node v gets $B_G(v,t)$, the ball of radius t around it; hence, in particular, it collects all the certificates of all the nodes at distance at most t from it. Then, by comparing its own certificate with the ones of its neighbors, it checks that the graph G', and the input $\mathbf{x}'$ in its certificate, are identical to the ones in the certificates of its neighbors. It also verifies consistency between the labels and the nodes in its ball of radius t. That is, it checks whether the labels and inputs in the certificate of the nodes in $B_G(v,t)$ are as described by its certificate. Whenever a node fails to pass any of these tests, it outputs "no". Otherwise it output "yes" or "no" according to whether $(G', \mathbf{x}') \in \mathcal{L}$ or not, respectively. (This is doable because we are considering languages that are decidable in the usual sense of sequential computation).

We show that A performs correctly. If $(G, \mathbf{x}) \in \mathcal{L}$, then by labeling the nodes in G by distinct integers from 1 to $|V(G)|$, and by providing the node v labeled i with $\mathbf{y}(v) = (i, G, \mathbf{x})$, the algorithm A output "yes" at all nodes, as desired. Consider now a instance $I = (G, \mathbf{x}) \notin \mathcal{L}$. Assume, for the purpose of contradiction that there exists a certificate $\mathbf{y}$ leading all nodes to output "yes". Let $f : V(G) \to V(G')$ be defined by $f(v) = i$ where i is the label of v in its certificate. Since $\mathbf{y}$ passes all tests of A, it means that (1) $\mathbf{y}(v) = (i, G', \mathbf{x}')$ where the instance $I' = (G', \mathbf{x}')$ is the same for all nodes, (2) f restricted to $B_G(v, t)$ is an isomorphism from $B_G(v, t)$ to $B_{G'}(f(v), t)$, for every node v, and (3) $(G', \mathbf{x}') \in \mathcal{L}$. In view of (2), I is t-local isomorphic to I'. Therefore, (3) implies that $I = (G, \mathbf{x}) \in \mathcal{L}$, because $\mathcal{L}$ is t-closed under lift. This is in contradiction with the actual hypothesis $(G, \mathbf{x}) \notin \mathcal{L}$. Thus, for each certificate $\mathbf{y}$, there must exist at least one node that outputs "no". As a consequence, A is a non-deterministic algorithm deciding $\mathcal{L}$, and thus $\mathcal{L} \in \text{NLD}^*$. ◇

The following lemma shows that the aforementioned sufficient condition for NLD^*-membership is a necessary condition for NLD-membership.

Lemma 2. *Let $\mathcal{L}$ be a language. If $\mathcal{L} \in \text{NLD}$, then there exists $t \geq 1$ such that $\mathcal{L}$ is t-closed under lift.*

Proof. Let $\mathcal{L}$ be a language in NLD, and let A be a non-deterministic (non-anonymous) local algorithm deciding $\mathcal{L}$. Assume, for the purpose of contradiction that, for any integer $t \geq 1$, $\mathcal{L}$ is *not* t-closed under lift. That is, for any t, there exist two input instances I, I' such that I is t-local isomorphic to I', with $I \notin \mathcal{L}$ and $I' \in \mathcal{L}$. Assume that A runs in t rounds. Without loss of generality, we can assume that $t \geq 1$. Let $I = (G, \mathbf{x}) \notin \mathcal{L}$ and $I' = (G', \mathbf{x}') \in \mathcal{L}$ satisfying I is t-local isomorphic to I'. Since $I' \in \mathcal{L}$, there exists a certificate $\mathbf{y}'$ such that when A is running on I' with certificate $\mathbf{y}'$, every node output "yes" for every identity assignment. Since I is t-local isomorphic to I', there exists an homomorphism $f : I \to I'$ such that, for every node $v \in G$, f restricted to $B_G(v, t)$ is an isomorphism from $B_G(v, t)$ to $B_{G'}(f(v), t)$. Let $\mathbf{y}$ be the certificate for I defined by $\mathbf{y}(v) = \mathbf{y}'(f(v))$. Consider the execution of A running on I with certificate $\mathbf{y}$, and some arbitrary identity assignment Id.

Since A performs in t rounds, the decision at each node v is taken according to the inputs, certificates, and identities in the ball $B_G(v, t)$, as well as the structure of this ball. By the nature of the homomorphism f, and by the definition of certificate $\mathbf{y}$, the structure, inputs and certificates of the ball $B_G(v, t)$, are identical to the corresponding structure, inputs and certificates of the ball $B_{G'}(f(v), t)$. Balls may however differ in the identities of their nodes. So, let v_0 be the node in G deciding "no" for $(G, \mathbf{x})$ with certificate $\mathbf{y}$. There exists such a node since $I \notin \mathcal{L}$. Let $v'_0 = f(v_0)$, and assign the same identities to the nodes in $B_{G'}(v'_0, t)$ as their corresponding nodes in $B_G(v_0, t)$. Arbitrarily extend this identities to an identity assignment Id' to the whole graph G'. By doing so, the two balls are not only isomorphic, but every node in $B_G(v_0, t)$ has the same input, certificate and identity as its image in $B_{G'}(v'_0, t)$. Therefore, the decision taken by A at $v_0 \in G$ under Id

is the same as its decision at $v'_0 \in G'$ under Id'. This is in contradiction to the fact that v_0 decides "no" while v'_0 decides "yes". ◇

Lemmas 1 and 2 together establish the theorem. □

The proof of Lemma 1 also provides an upper bound on the size of the certificates for *graph languages* in NLD, that is, for languages in NLD with no input. (This includes, e.g., recognition of interval graphs, and recognition of chordal graphs). Indeed, given $\mathcal{L} \in$ NLD, Algorithm A in the proof of Lemma 1 verifies $\mathcal{L}$ using a certificate at each node which is essentially an isomorphic copy of the input instance $(G, \mathbf{x})$, with nodes labeled by consecutive integers in $[1, n]$. If $\mathcal{L}$ is a graph language, then there is no input $\mathbf{x}$, and thus the size of the certificates depends only on the size of the graph. More precisely, we have:

Corollary 1. *Let $\mathcal{L} \in$ NLD be a graph language. There exists an algorithm verifying $\mathcal{L}$ using certificates of size $O(n^2)$ bits at each node of every n-node graph in $\mathcal{L}$.*

We now argue that the above bound is tight, that is, we prove the following.

Proposition 1. *There exists a graph language $\mathcal{L} \in$ NLD such that every algorithm verifying $\mathcal{L}$ requires certificates of size $\Omega(n^2)$ bits.*

Proof. Recall that [26] showed that there exists a graph language for which every proof labeling scheme (PLS) requires labels of size $\Omega(n^2)$ bits (the proof of this latter result appears in a detailed version [27]). Still in the context of PLS, [19] showed that this lower bound holds for two *natural* graph families: specifically, [19] showed that verifying symmetric graphs requires labels of size $\Omega(n^2)$ bits, and verifying non-3 colorable graphs requires almost the same size of labels, specifically, $\Omega(n^2/\log n)$ bits. Note that the certificate size required for verifying a language in NLD is at least as large as the minimum label size required for verifying the language via a proof labeling scheme. Unfortunately, however, one cannot obtain our claim directly from the aforementioned results since it turns out that neither of the two graph languages (namely, symmetric graphs and non-3 colorable graphs) belongs to NLD.

We therefore employ an indirect approach. Specifically, consider a graph G. We say that H is a *seed* of G if there exists a 1-local isomorphism from G to H. Suppose $\mathcal{F}$ is a family of graphs. Let Seed-$\mathcal{F}$ denote the family of graphs G, for which there exists a seed of G that belongs to $\mathcal{F}$. Then, by definition, Seed-$\mathcal{F}$ is 1-closed under lift. Indeed, assume that there is a 1-local isomorphism g from G' to G, and let $H \in \mathcal{F}$ be a seed of G that belongs to $\mathcal{F}$. Then let f be the 1-local isomorphism from G to H. We have that $f \circ g$ is a 1-local isomorphism from G' to H, because, for every $u \in V(G')$, $B_{G'}(u, 1)$ is isomorphic to $B_G(g(u), 1)$, which in turn is isomorphic to $B_H(f(g(u)), 1)$. Thus H is also a seed of G'. Seed-$\mathcal{F}$ is therefore in NLD. Now, in the proof of corollary 2.2 in [27], the authors construct, for every integer n, a family $\mathcal{F}_n$ of n-node graphs that requires proof labels of size $\Omega(n^2)$. Note that for every prime integer n', a graph G of size n'

belongs to $\mathcal{F}_{n'}$ if and only if it belongs to Seed-$\mathcal{F}_{n'}$. Therefore, there exists a graph language, namely, Seed-$\mathcal{F}_n$, that requires certificates of size $\Omega(n^2)$ bits (at least for prime n's). □

4 Conclusion

Again, in this paper, we provide some evidences supporting the conjecture $\text{LD}^* = \text{LD}$. For instance, Theorem 3 shows that if every node knows any upper bound on the number of nodes n, then all languages in LD can be decided in the anonymous $\mathcal{LOCAL}$ model. One interesting remark about the $\mathcal{LOCAL}$ model is that it is guaranteed that at least one node has an upper bound on n. This is for instance the case of the node with the largest identity. In the anonymous $\mathcal{LOCAL}$ model, however, there is no such guarantee. Finding a language whose decision would be based on the fact that one node has an upper bound on n would disprove the conjecture $\text{LD}^* = \text{LD}$. Nevertheless, it is not clear whether such a problem exists.

In this paper, we also prove that $\text{NLD}^* = \text{NLD}$, that is, our conjecture holds for the non-deterministic setting. It is worth noticing that [13] proved that there exists an NLD-complete problem under the local one-to-many reduction. It is not clear whether such a problem exists for NLD^*. Indeed, the reduction in the completeness proof of [13] relies on the aforementioned guarantee that, in the $\mathcal{LOCAL}$ model, at least one node has an upper bound on n.

References

1. Afek, Y., Kutten, S., Yung, M.: The local detection paradigm and its applications to self stabilization. Theoretical Computer Science 186(1-2), 199–230 (1997)
2. Angluin, D.: Local and Global Properties in Networks of Processors. In: Proc. Twelfth ACM Symp. on Theory of Computing, STOC, pp. 82–93 (1980)
3. Amit, A., Linial, N., Matousek, J., Rozenman, E.: Random lifts of graphs. In: Proc. 12th ACM-SIAM Symp. on Discrete Algorithms, SODA, pp. 883–894 (2001)
4. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-Stabilization By Local Checking and Correction. In: Proc. IEEE Symp. on the Foundations of Computer Science, FOCS, pp. 268–277 (1991)
5. Barenboim, L., Elkin, M.: Distributed $(\Delta + 1)$-coloring in linear (in delta) time. In: Proc. 41st ACM Symp. on Theory of Computing, STOC, pp. 111–120 (2009)
6. Das Sarma, A., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., Wattenhofer, R.: Distributed Verification and Hardness of Distributed Approximation. In: Proc. 43rd ACM Symp. on Theory of Computing, STOC (2011)
7. Dereniowski, D., Pelc, A.: Drawing maps with advice. Journal of Parallel and Distributed Computing 72, 132–143 (2012)
8. Dolev, S., Gouda, M., Schneider, M.: Requirements for silent stabilization. Acta Informatica 36(6), 447–462 (1999)
9. Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. on Programming Languages and Systems 5, 66–77 (1983)

10. Fraigniaud, P., Gavoille, C., Ilcinkas, D., Pelc, A.: Distributed Computing with Advice: Information Sensitivity of Graph Coloring. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 231–242. Springer, Heidelberg (2007)
11. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Communication algorithms with advice. J. Comput. Syst. Sci. 76(3-4), 222–232 (2008)
12. Fraigniaud, P., Korman, A., Lebhar, E.: Local MST computation with short advice. In: Proc. 19th ACM Symp. on Parallelism in Algorithms and Architectures, SPAA, pp. 154–160 (2007)
13. Fraigniaud, P., Korman, A., Peleg, D.: Local Distributed Decision. In: Proc. 52nd Annual IEEE Symposium on Foundations of Computer Science, FOCS, pp. 708–717 (2011)
14. Fraigniaud, P., Korman, A., Parter, M., Peleg, D.: Randomized Distributed Decision, http://arxiv.org/abs/1207.0252
15. Fraigniaud, P., Pelc, A.: Decidability Classes for Mobile Agents Computing. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 362–374. Springer, Heidelberg (2012)
16. Fraigniaud, P., Rajsbaum, S., Travers, C.: Locality and Checkability in Wait-Free Computing. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 333–347. Springer, Heidelberg (2011)
17. Fraigniaud, P., Rajsbaum, S., Travers, C.: Universal Distributed Checkers and Orientation-Detection Tasks (submitted, 2012)
18. Fusco, E.G., Pelc, A.: Communication Complexity of Consensus in Anonymous Message Passing Systems. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 191–206. Springer, Heidelberg (2011)
19. Göös, M., Suomela, J.: Locally checkable proofs. In: Proc. 30th ACM Symp. on Principles of Distributed Computing, PODC (2011)
20. Göös, M., Hirvonen, J., Suomela, J.: Lower bounds for local approximation. In: Proc. 31st Symposium on Principles of Distributed Computing, PODC (2012)
21. Hasemann, H., Hirvonen, J., Rybicki, J., Suomela, J.: Deterministic Local Algorithms, Unique Identifiers, and Fractional Graph Colouring. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 48–60. Springer, Heidelberg (2012)
22. Hanckowiak, M., Karonski, M., Panconesi, A.: On the Distributed Complexity of Computing Maximal Matchings. SIAM J. Discrete Math. 15(1), 41–57 (2001)
23. Kor, L., Korman, A., Peleg, D.: Tight Bounds For Distributed MST Verification. In: Proc. 28th Int. Symp. on Theoretical Aspects of Computer Science, STACS (2011)
24. Korman, A., Kutten, S.: Distributed verification of minimum spanning trees. Distributed Computing 20, 253–266 (2007)
25. Korman, A., Kutten, S., Masuzawa, T.: Fast and Compact Self-Stabilizing Verification, Computation, and Fault Detection of an MST. In: Proc. 30th ACM Symp. on Principles of Distributed Computing, PODC (2011)
26. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Distributed Computing 22, 215–233 (2010)
27. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Detailed version, http://ie.technion.ac.il/~kutten/ps-links/ProofLabelingSchemes.ps
28. Korman, A., Sereni, J.S., Viennot, L.: Toward More Localized Local Algorithms: Removing Assumptions Concerning Global Knowledge. In: Proc. 30th ACM Symp. on Principles of Distributed Computing, PODC, pp. 49–58 (2011)

29. Kuhn, F.: Weak graph colorings: distributed algorithms and applications. In: Proc. 21st ACM Symp. on Parallel Algorithms and Architectures, SPAA, pp. 138–144 (2009)
30. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comput. 21(1), 193–201 (1992)
31. Lotker, Z., Patt-Shamir, B., Rosen, A.: Distributed Approximate Matching. SIAM J. Comput. 39(2), 445–460 (2009)
32. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. 15, 1036–1053 (1986)
33. Naor, M., Stockmeyer, L.: What can be computed locally? SIAM J. Comput. 24(6), 1259–1277 (1995)
34. Panconesi, A., Srinivasan, A.: On the Complexity of Distributed Network Decomposition. J. Algorithms 20(2), 356–374 (1996)
35. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM (2000)
36. Seinsche, D.: On a property of the class of n-colorable graphs. J. Combinatorial Theory, Ser. B 16, 191–193 (1974)

Black Hole Search and Exploration in Unoriented Tori with Synchronous Scattered Finite Automata

Euripides Markou[1,⋆] and Michel Paquette[2,⋆⋆]

[1] Department of Computer Science and Biomedical Informatics, University of Central Greece, Lamia, Greece
emarkou@ucg.gr
[2] Department of Computer Science, Vanier College, Montréal, Canada
michel.paquette@vaniercollege.qc.ca

Abstract. We consider the problem of locating a black hole in a synchronous, anonymous, and unoriented torus network using mobile agents. A black hole is a harmful network node that destroys any agent visiting it without leaving any trace. The objective is to locate the black hole using as few agents as possible. We present here an almost optimal deterministic algorithm for synchronous (partially) unoriented tori using five scattered agents with constant memory and three identical tokens. We also study the exploration problem of a safe (i.e., without black holes) unoriented torus. While it has been previously shown that there is no universal algorithm for one agent with constant memory and any constant number of tokens which can explore all cubic planar graphs, we give here the first algorithm which enables a finite automaton with two tokens to explore (without termination detection) any totally unoriented torus and we prove optimality on the number of tokens.

Keywords: Distributed Algorithms, Fault Tolerance, Black Hole Search, Anonymous Networks, Mobile Agents, Finite State Automata.

1 Introduction

The exploration of an unknown graph by one or more mobile agents is a classical problem initially formulated in 1951 by Shannon [23] and it has been extensively studied since then (e.g., see [2,9,16]). In 1967, during a talk at Berkeley, Rabin [21] conjectured that no finite automaton with a constant number of pebbles (or tokens) can explore all graphs (a pebble is a marker that can be dropped at

⋆ E. Markou has been co-financed by the European Union (European Social Fund-ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)-Research Funding Program: THALIS-UOA-GeomComp.

⋆⋆ Part of this work was done during a visit from this author at the University of Central Greece, financed by grants from the University of Central Greece and Cégep International (Programme de soutien à la mobilité enseignante.)

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 239–253, 2012.

and removed from nodes). The first step towards a formal proof of Rabin's conjecture is generally attributed to Budach [4], for an agent without pebbles. Blum and Kozen [3] improved Budach's result by proving that *three* agents cannot cooperatively perform exploration of all graphs. In 1979, Kozen [19] proved that *four* cooperative agents cannot explore all graphs. Finally, in 1980, Rollik [22] gave a complete proof of Rabin's conjecture. More precisely, Rollik proved that no finite set of finite automata can cooperatively perform exploration of all cubic planar graphs. Since a finite automaton is more powerful than a pebble (a pebble does not have states, or a transition function), Rabin's conjecture is a corollary of Rollik's theorem.

Recently, the exploration problem has also been studied in unsafe networks which contain malicious hosts of a highly harmful nature, called *black holes*. A black hole is a node which contains a stationary process destroying all mobile agents visiting this node, without leaving any trace. In the *Black Hole Search (BHS)* problem the goal for the agents is to locate the black hole within finite time. In particular, at least one agent has to survive knowing all edges leading to the black hole. Without the knowledge of the size of the network, the only way of locating a black hole is to have at least one agent visiting it. However, since any agent visiting a black hole vanishes without leaving any trace, the location of the black hole must be deducted by some communication mechanism employed by the agents. Four such mechanisms have been proposed in the literature: a) the *whiteboard* model in which there is a whiteboard at each node of the network on which the agents can leave messages, b) the *pure token* model where the agents carry tokens which they can leave at nodes, c) the *enhanced token* model in which the agents can leave tokens at nodes or edges, and d) the *time-out mechanism* (only for synchronous networks) in which at least two agents gather at a node u, and then one agent explores a new node and returns to u to inform the other agents who wait.

The *whiteboard* model provides the most powerful inter-agent communication mechanism. Since, in this model, access to a whiteboard is provided in mutual exclusion, this model could also provide the anonymous agents with a symmetry-breaking mechanism: if the agents start at the same node, they can get distinct identities and then the distinct agents can assign different labels to all nodes. Hence in this model, if the agents are initially co-located, both the agents and the nodes can be assumed to be non-anonymous without any loss of generality.

In asynchronous networks and given that all agents initially start at the same safe node, the Black Hole Search problem has been studied under the whiteboard model (e.g., [11,12]), the *enhanced token* model (e.g., [10]) and the *pure token* model (e.g., [1,14]). In these models it has been proven that the problem can be solved with a minimal number of agents performing a polynomial number of moves. It has been also shown that in an asynchronous network the number of the nodes in the network must be known to the agents otherwise the problem is unsolvable ([12]). If the graph topology is unknown, at least $\Delta + 1$ agents are needed, where Δ is the maximum node degree in the graph ([11]). Furthermore the network should be 2-connected. It is also not possible to answer the question

of *whether* one black hole exists in the asynchronous network. With scattered agents (not initially located at the same node) in asynchronous networks, the problem has been investigated for the ring topology ([12]) and for arbitrary topologies ([15]) in the whiteboard model while in the *enhanced token* model it has been studied for rings ([13]) and for some interconnected networks ([24]).

The situation in synchronous networks is dramatically different. Under this assumption, two co-located distinct agents can discover one black hole in any graph (provided that the graph can be explored) by using the time-out mechanism, without the need of whiteboards or tokens. Moreover the network does not have to be 2-connected anymore, as in asynchronous networks, and furthermore it is now possible to answer the question of whether a black hole actually exists or not in the network. No knowledge about the number of nodes is needed. Hence, with co-located distinct agents, the issue is not the feasibility but the time efficiency of black hole search. The issue of efficient black hole search has been studied in synchronous networks without whiteboards or tokens, only using the time-out mechanism (e.g., [7,8,17,18]) under the condition that all distinct agents start at the same node. However when the agents are scattered in the network, the time-out mechanism is not sufficient anymore: the agents have to gather at the same node in order to use the time-out mechanism.

While the whiteboard model is commonly used in unsafe networks, the token model has been mostly used in the exploration of safe networks. The *pure* token model can be implemented with $O(1)$-bit whiteboards for a constant number of agents and a constant number of tokens (since the only information which is stored at a node's whiteboard is the number of tokens placed at that node), while the *enhanced* token model can be implemented having a $O(\log \Delta)$-bit whiteboard on a node with degree Δ (since in that case the information stored at a node's whiteboard is the number of tokens placed at each port of that node). In the whiteboard model, the capacity of each whiteboard is assumed to be of at least $\Omega(\log n)$ bits, where n is the number of nodes of the network (since in that case the information stored at a whiteboard includes labels of nodes).

In all previous papers studying the Black Hole Search problem under a token model apart from [1,6,5,14], the authors have used the *enhanced* token model with agents having non-constant memory. The weakest *pure* token model has been used in [1,14] for co-located agents with non-constant memory in asynchronous networks. The first results for scattered agents with constant memory and *pure* tokens appeared in [6] for synchronous unoriented rings and [5] for synchronous oriented tori. In [5] it has been proven that three scattered agents with constant memory and with two tokens each can locate the black hole in any synchronous oriented torus.

2 Our Results

We study the Black Hole Search problem (BHS) for scattered identical anonymous agents with constant memory in synchronous anonymous *unoriented* torus

networks under the *pure token* model. We use the same model as in [6,5] but we focus on *unoriented* torus topologies. Throughout the paper we discuss four cases of unoriented tori:

- *type 0*: the agents do not agree on anything regarding the orientation;
- *type 1*: the agents perceive orthogonal axes but they do not agree on which axis is horizontal and which is vertical;
- *type 2*: the agents agree on which axis is horizontal and which is vertical, but there is no consensus on the orientation of each axis;
- *type 3*: the agents agree on which axis is horizontal and which is vertical and they also agree on the orientation in one of the axes.

For the BHS problem we show the results presented in Table 1:

Table 1. Summary of results for BHS in synchronous unoriented tori

Torus orientation	**# agents**	**# tokens**	**Black Hole Search**
Type 0	any constant	1	Impossible
	4	any constant	
Type 1 or 2	4	any constant	Impossible
	5	1	
	5	3	Algorithm $UBHS_{5,3}$
Type 3	3	any constant	Impossible
	4	1	
	5	3	Algorithm $UBHS_{5,3}$

We also show the following results for the exploration problem of a safe unoriented torus:

- There is no universal algorithm for any constant number of agents with one movable token solving the exploration problem (even without termination detection) in all unoriented tori.
- There is a universal algorithm for the exploration (without termination detection) of any unoriented torus (type 0) by one agent with constant memory and two movable tokens.

This last result is optimal on the number of tokens and it is somewhat surprising since it had been proven, in [22], that an agent with any constant number of tokens cannot explore all cubic planar graphs. Although it has been proved in [3] that exploration can be done with a constant number of tokens in partial grids (grids with missing nodes and edges) with *sense of direction*, our result shows that the impossibility result of [22] is not robust enough to resist in highly structured graphs, like tori, without *sense of direction*.

Due to space limitations some proofs have been omitted and will appear in the full version of the paper.

3 Our Model

Our model consists of $k \geq 2$ anonymous and identical mobile agents that are initially placed at distinct nodes of an anonymous, synchronous torus network. Each agent consistently evaluates direction across the torus (except in type 0 tori); agents do not necessarily evaluate the same directions for West, East, North or South. In *type* 0 tori, we make no assumption on the way each direction is interpreted. For *type* 1*,* 2*, and* 3 tori, however, we assume that each such function evaluates West and East to be opposites on the same axis and orthogonal to North and South (also opposites on the same axis). Each mobile agent has a constant number t of identical pure tokens which can be placed at any node visited by the agent. We call a token *movable* if it can be moved by any mobile agent to any node of the network, otherwise we call the token *unmovable* in the sense that, once released, it can occupy only the node in which it has been released. A token at a given node is visible to all agents on the same node, but is not visible to agents at other locations. While our negative results hold even when the agents have the capability to communicate when they are at the same node (which is the usual assumption in previous works), our protocol works even when the agents cannot directly communicate at all, regardless of their location. The agents follow the same deterministic algorithm and begin execution at the same time, at the same initial state. At any single time unit, a mobile agent occupies a node of the network and may 1) stay there or move to an adjacent node, 2) detect the presence of one or more tokens at the node it is occupying and 3) release/take one or more tokens to/from the node it is occupying. When two or more agents located at the same node attempt to see and/or change the node configuration (release or take tokens) at the same time, they do it by mutual exclusion. We give more details on the mutual exclusion mechanism, later in this section.

Formally we consider a mobile agent as a finite Moore automaton $\mathcal{A} = (\mathcal{S}, S_0, \Sigma, \Lambda, \delta, \phi)$, where $\mathcal{S}$ is a set of $\sigma \geq 2$ states; S_0 is the *initial* state; Σ is the set of possible configurations an agent can see when it enters a node; $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$ is the transition function; and $\phi : \mathcal{S} \to \Lambda$ is the output function. Elements of Σ are triplets (D, x, y) where $D \in \{\texttt{North}, \texttt{South}, \texttt{East}, \texttt{West}\}$ is the direction through which the agent has arrived at the node, x is the number of tokens at that node, and y is the number of tokens carried by the agent. Elements of Λ are triplets (A, s, M) where $A \in \{\texttt{Put}, \texttt{Take}\}$ is the action performed by the agent on the tokens, s is the number of tokens concerned by the action A, and $M \in \{\texttt{North}, \texttt{South}, \texttt{East}, \texttt{West}, \texttt{wait}\}$ is the move performed by the agent.

When two or more agents are on the same node and wish to operate at the same time then the mutual exclusion mechanism guarantees that the agents one by one evaluate functions δ, ϕ. The sequence δ, ϕ is atomic. The order by which the agents evaluate their functions is handled by an adversary.

We assume that the memory of an agent is proportional to the number of bits required to encode its states which we take to be $\Theta(\log(|\mathcal{S}|))$ bits. Note that in our algorithms all computations by the agents are independent of the size $n \times m$ of the torus network. The agents have no knowledge of n, m. There is exactly

one black hole in the network. An agent can start from any node other than the black hole and no two agents are initially co-located (we say that they are *scattered*). The agents' initial locations and the black hole location are decided by an adversary. Once an agent detects a link to the black hole, it marks the link permanently as dangerous (i.e., disables this link). Since the agents do not have enough memory to remember the location of the black hole, we require that at the end of a black hole search scheme, all links incident to the black hole (and only those links) are marked dangerous and that there is at least one surviving agent. Each agent has the following primitives:

1. Walk(x): move to an adjacent node in direction x or through port x and return the incoming port label.
2. Opposite(dir): return the direction opposite to dir.
3. Put(t): leave t tokens at the current node.
4. Read(): return the number of tokens at the current node.
5. Take(t): remove t tokens from the current node.

The mutual exclusion mechanism guarantees that two or more co-located agents execute one by one a sequence containing *Read, Put, Take* actions.

4 Negative Results

In any correct algorithm for solving the Black Hole Search problem each node apart from at most one of the network must be visited by at least one agent in the worst case, since if there are two or more unvisited nodes, the agents cannot decide which one is the black hole.

In [5] it has been proven that no finite team of agents can solve the BHS problem in all oriented torus networks using any constant number of *unmovable* tokens:

Theorem 1. *[5] For any constant numbers k, t, there exists no algorithm that solves BHS in all oriented tori containing one black hole and k scattered agents, where each agent has a constant memory and t unmovable tokens.*

Hence, as in the case of an oriented torus, a correct algorithm for the BHS problem in an unoriented torus should use movable tokens.

4.1 Black Hole Search in a Torus of Type 3

Lemma 1. *There is no universal algorithm solving the BHS problem in all synchronous semi-oriented tori of type* 3*, using less than* 4 *scattered agents carrying any constant number of tokens even when the agents have unlimited memory.*

Proof. To locate a black hole, any BHS algorithm functioning with scattered agents must move all agents by at least one node and must also force agents to traverse multiple rings of the torus. Consider such a BHS algorithm. Suppose (without loss of generality) that the agents agree on the horizontal orientation

of the torus. Then an adversary can choose agents' orientation on the vertical direction and initial locations so that one agent will vanish into the black hole while traveling horizontally and two more agents will fall into the black hole while traveling vertically (without having met any tokens other than their own tokens). Hence, a fourth agent is needed to compute where the black hole is located. ■

In [5] it has been proven that three scattered agents carrying one token each cannot solve the BHS problem in a synchronous *oriented* torus. The basic argument in that proof is the following: either the agents stay 'close' to their tokens and in that case they fail to explore the whole torus, or they go far from their tokens (more than a constant number of steps), they manage to explore the torus and meet the black hole but fail to leave a clear indication at nearby nodes for the remaining agents. We argue similarly here and prove the following lemma.

Lemma 2. *There is no universal algorithm which solves the BHS problem in all synchronous semi-oriented tori of type* 3*, using less than* 5 *scattered agents with one movable token each if the agents have constant memory.*

Proof.(Sketch) Since in view of Theorem 1, solving the BHS problem with unmovable tokens is impossible, a correct BHS algorithm should eventually instruct the agents to leave their tokens down. This decision should be taken after a constant number of steps (independent of the size of the torus) due to the constant number of agents' states. Moreover this decision has to be taken at the same time for all agents since the agents are anonymous and start at the same state.

If agents always move a constant number of steps away from their tokens, then an adversary can always select the size of the torus, the initial locations of the agents and the black hole location, so that any agent will never meet a token of another agent, another agent, or the black hole. Moreover the agents will be eventually trapped visiting the same nodes and there will be nodes in the torus which remain unvisited by any agent. Therefore in a correct algorithm the agents should move more than a constant number of steps away from their tokens. Suppose without loss of generality that the agents disagree on the horizontal orientation. First consider the case in which the agents move more than a constant number of nodes away from their own tokens in the horizontal axis. Due to disagreement in the horizontal orientation, the adversary can force two agents to vanish at the black hole at the same time leaving their tokens more than a constant number of steps away from the black hole. The adversary may also arrange that a third agent enters the black hole without having met the others' tokens and leaving its token somewhere more than a constant distance away from the black hole. The remaining case in which the agents move more than a constant number of nodes away from their own tokens in the vertical axis can be argued analogously. Hence in both cases a fourth agent may now find a token other than its own token but cannot decide the black hole location. ■

The above negative results also hold for tori of type 2, 1 or 0.

4.2 Black Hole Search in a Torus of Type 2

When the agents agree only on which axis of the torus is horizontal and which is vertical but they do not agree on their orientations (torus of type 2), then analogously as in Lemma 1 we can show that:

Lemma 3. *There is no universal algorithm solving the BHS problem in all synchronous semi-oriented tori of type* 2*, using less than* 5 *scattered agents carrying any constant number of tokens even when the agents have unlimited memory.*

Proof. An adversary can choose agents' orientation and initial locations such that two agents will enter the black hole while traveling horizontally and another two agents will vanish into the black hole while traveling vertically (without having met tokens other than their own tokens). Then, a fifth agent is needed to compute where the black hole is located. ■

With a similar reasoning as in the proof of Lemma 2 the following lemma holds:

Lemma 4. *There is no universal algorithm which solves the BHS problem in all synchronous semi-oriented tori of type* 2*, using less than* 6 *scattered agents with one movable token each if the agents have constant memory.*

The above negative results also hold for tori of type 1 or 0.

4.3 Black Hole Search and Exploration in a Torus of Type 0

In a type 0 torus the agents do not agree on anything related to the orientation of the torus. We however assume a local port labeling (i.e., port labels of incident edges of a node are different), otherwise an agent is not capable even to visit all neighbors of its current node. This port labeling is fixed by an adversary and is not globally consistent (i.e., an agent which always exit nodes by port *East* does not necessarily traverses a complete ring of the torus). We note that once a port label is fixed by the adversary, it cannot be changed (i.e., the adversary cannot change previously fixed port labels).

We show below (Theorem 2) that any constant number of scattered agents with constant memory and one token cannot solve the BHS problem in all tori of type 0. The idea of the proof is that the agents with only one token are not able even to explore a safe torus (Lemma 6), leaving many nodes unvisited (in contrast with semi-oriented safe tori, where only one agent with constant memory and one token can visit all nodes of the torus). Hence an adversary can place the black hole in one of the unvisited nodes and the agents are not able to decide its location. We first prove the following lemma:

Lemma 5. *There is no universal algorithm which solves the exploration problem in all synchronous unoriented (safe) tori of type* 0 *using one agent with constant memory and one movable token.*

Proof.(Sketch) Consider such an exploration algorithm for the sake of contradiction and let $\sigma \geq 2$ be the number of states of the agent. An adversary assigns port labels so that the agent can complete its first $2\sigma - 1$ moves (i.e., in which encounters 2σ states). The labels are assigned to ports so that for each edge the pair of the two port labels is either (*East, West*) or (*North, South*).

Suppose the algorithm instructs the agent not to release its token during this sequence p of 2σ states. Due to the σ number of different states, there is at least one state which has been repeated in p. Consider the first state S^* which is repeated and also has the following property: assuming u_1, u_2 to be the nodes where the agent is located when it encounters state S^* at times t_1, t_2 respectively (where $t_1 < t_2$), the entry port label to u_2 is different than the exit port label from u_1. If there is no such state in p, this means that there is a subsequence of p starting and ending at a state S' with the agent locating at the same node (basically in this subsequence the agent moves back and forth traversing the same edge). Suppose that such a state S^* exists. If $u_1 \equiv u_2$, then, since the whole sequence of states and moves will be repeated, the agent visits again and again the same nodes. Suppose that $u_1 \neq u_2$. Then the adversary can arrange the port labels so that nodes u_1, u_2 are on the same horizontal ring. The sequence of moves and states is repeated and if we call u_i the node at which the agent encounters state S^* for the $i-$th time, then the adversary can arrange the port labels so that nodes u_2, u_3 are on the same vertical ring (notice that the distance $d(u_2, u_3)$ will be equal to $d(u_1, u_2)$), nodes u_3, u_4 are on the same horizontal ring, u_4 is at the same vertical ring with u_1 (since $d(u_3, u_4)$ will be equal to $d(u_1, u_2)$), and finally nodes u_4, u_5 are on the same vertical ring and $u_5 \equiv u_1$ (since $d(u_4, u_5)$ will be equal to $d(u_2, u_3)$). The agent has visited a total number of at most $8\sigma - 1$ different nodes and then it keeps visiting the same nodes.

Suppose that the algorithm instructs the agent to release its token within the first $2\sigma - 1$ first moves. In fact this has to be done within the first σ moves otherwise it will never be done. The agent continues moving without a token. If the adversary can assign the port labels so that after another at most 8σ moves the agent does not meet its token, then similarly as above, the agent passes twice from the same node being at the same state and then keeps visiting the same nodes. If the agent meets its token within 8σ moves then after at most $\sigma+1$ times meeting its token will again meet its token being at the same state S_t. Between those two repetitions of state S_t the agent has visited at most $c\sigma^2$ nodes, where c is a constant. After that the agent repeats this orbit but maybe on a different area of the torus. However in an e.g., $n \times n$ torus, after at most n repetitions of this orbit the agent will be at the same state and at the same node (meeting its token) and therefore after that the agent will repeat everything visiting exactly the same nodes. The agent has been visited at most $O(n)$ nodes out of the n^2 nodes of a $n \times n$ torus. ■

Generalizing the previous lemma, it can be shown that:

Lemma 6. *There is no universal algorithm which solves the exploration problem in all synchronous (safe) unoriented tori of type* 0 *using any constant number* k *of scattered agents with constant memory and one movable token each.*

Hence, when there is a black hole in the torus network the adversary can place it in one of the unvisited nodes and therefore the following theorem holds:

Theorem 2. *There is no universal algorithm which solves the BHS problem in all synchronous unoriented tori of type* 0 *using any constant number of scattered agents with one movable token each if the agents have constant memory.*

As we will discuss in the next section, an agent with constant memory and only two tokens can explore (without stop) any unoriented torus of type 0.

In view of Lemma 3 and Theorem 2, any BHS algorithm for any type 0 unoriented torus would need at least five agents with two tokens each.

5 Positive Results

5.1 Exploration of an Unoriented Torus by a Finite Automaton

In the previous section we proved that any constant number of finite automata with one token each cannot solve the black hole search problem in all unoriented tori. The proof relies on Lemma 6 stating that there is no universal algorithm for any team of a constant number of finite automata with one token each that can *explore* all (safe) unoriented tori (type 0). This result is in agreement with the well known result of Rollik which says that an agent with any constant number of tokens cannot explore all cubic planar graphs ([22]). Hence a natural question is whether an agent with two tokens would be able to explore all unoriented tori. The answer is surprisingly positive. As we present in this section (Algorithm `Explore2Tokens`), one agent with constant memory having only two movable tokens can explore (perpetual exploration without stop) any type 0 unoriented torus. This result shows that although a torus is already a non-planar graph with degree 4, its special properties can be exploited by a not very complicated algorithm which solves the exploration problem in such a weak model.

The basic idea of the algorithm is hidden in `Function ExploreRing` which enables the agent to explore a whole ring of the torus, when the torus consists of rings of at least 4 nodes[1]. The idea[2] can be described as follows: The agent located at a node u leaves a token down and selects a port leaving u and entering an adjacent node v. Now in order to discover which node adjacent to v and different

[1] Small tori containing rings of 2 or 3 nodes can be easily explored by traversing all paths of length 2 or 3 as shown in Procedure `SmallRing`.

[2] As it has been brought to our attention by the anonymous referees, the techniques for local orientation of tori we use in the algorithm have some similarities with the techniques presented in [20] for solving the *Leader Election* problem in unlabeled tori using messages.

Procedure SmallRing

```
1: //Take care the case of a torus with a ring of less than 4 nodes
2: Put(2)
3: for i ← 2 to 3 do
4:    Explore all paths of length i and return
5:    if a token is found in more than 2 different paths then
6:       Stop //the torus is i × i and has been already explored
7:    if a token is found in exactly 2 paths starting at ports p1, p2 then
8:       //the torus is i × n
9:       loop
10:         Move the tokens taking a port p ≠ p1, p2
11:         Explore all paths of length i and return
12:         Let p1, p2 be the starting ports of paths with tokens
13: Take(2)
```

Algorithm Explore2Tokens

```
1: SmallRing //Take care the case of a torus with a ring of less than 4 nodes
2: //Any ring has at least 4 nodes
3: V_t ← ∅
4: C ← ∅
5: Choose a port H_s
6: loop
7:    H_f ← ExploreRing(H_s)
8:    if H_f = 0 then
9:       Insert H_s in set C
10:      if there is a new port not in C then
11:         Choose a new port H_s ∉ C
12:         Take(2)
13:      else
14:         Stop
15:   else
16:      Choose a port V_s ≠ H_s, H_f, V_t
17:      V_t ← Walk(V_s)
18:      repeat
19:         Move one step on the direction towards port m ≠ V_t
20:         Let m' be the incoming port
21:         Explore all paths of length 2 not starting with port m' and return
22:      until you found a token at a path ending at port H_s
23:      H_s ← m
24:      Traverse ports m', V_t, Take(1) and traverse port V_s
```

Function ExploreRing(H_s)

```
1: RingCompleted ← FALSE
2: Put(2) //Ring start
3: p ← Walk(H_s)
4: repeat
5:    CorrectDir ← 0
6:    while (There is an untraversed port p' ≠ p) AND (CorrectDir = 0) do
7:       p'' ← Walk(p')
8:       Explore all paths of length 2 not starting with port p'' and return
9:       if no token is found during this exploration then
10:         CorrectDir ← p'
11:      Walk(p'') //Go back taking port p''
12:   RingCompleted ← (CorrectDir = 0)
13:   Walk(p)
14:   if (RingCompleted) AND (you see two tokens) then
15:      //The ring under exploration has 4 nodes
16:      Return 0
17:   else
18:      Take(1) (the second token) and return
19:   if ¬RingCompleted then
20:      Put(1) //Leave the second token
21:      p ← Walk(CorrectDir) //Advance the exploration
22: until RingCompleted
23: repeat
24:    Explore all paths of length 3 starting at ports different than p
25: until until you find a node with a token
26: Return the incoming port label
```

than u lies in the same (horizontal or vertical) ring of the torus with v and u the agent tests all adjacent nodes of v (apart from u) to find a node w for which all paths of length 2 starting at w and not passing from v do not end up at a node with a token (i.e., node u). Then node u, v, w belong in the same ring of the torus. By repeating this procedure, the agent can explore a ring. The exploration of a ring finishes when the agent finds its (homebase) token which had been left at the starting node. Then it can move tokens in the next ring and continue. In that way the agent will eventually explore (without stop) the whole torus.

Theorem 3. *Algorithm* `Explore2Tokens` *enables one agent with constant memory and two movable tokens to explore (without stop) any unoriented torus.*

In view of Lemma 6, Algorithm `Explore2Tokens` is optimal with respect to the number of agents and tokens it uses. The algorithm can be extended for solving exploration *with* stop using three movable tokens.

5.2 BHS in Semi-oriented Tori of Type 1

In this section, we state Algorithm $UBHS_{5,3}$ which enables 5 scattered agents with constant memory and 3 tokens each to locate the black hole in any torus of type 1 (i.e., a torus in which the agents agree only on the orthogonal axes). Intuitively the algorithm works as follows: Each agent leaves 2 of its tokens at its starting node and explores its starting horizontal (perceived) ring using the Procedure `CautiousTest` with its last remaining token (leaves a token, walks to a neighbor at a given direction and returns to pick up its token). Then moves the 2 (homebase) tokens onto the next horizontal ring and repeats the procedure. Eventually, there will be at least one and at most four agents entering the black hole and leaving at an adjacent node 1 or 3 tokens. At least one of the remaining agents will find such a configuration of 1 or 3 tokens (which we call a *bad token* configuration) and will locate the (near by) black hole by calling Procedure `LocalSearch`.

Procedure `LocalSearch`**:** Suppose the agent arrives from a direction *dir* at a node u where it finds a *bad token* configuration. This means that the black-hole is at a node v which is adjacent to u. The agent takes the following actions: If $dir = East$ ($dir = South$) then the agent searches (using `CautiousTest`) the adjacent nodes *South* and *North* (*West* and *East*) of u in this order; either the agent vanishes leaving behind another *bad token* configuration or otherwise decides that the black hole is *East* (*South*) of u respectively.

Algorithm $UBHS_{5,3}$ (each agent, in parallel)

```
1: repeat
2:    Put(2) //Start of "ring scan"
3:    count ← 0
4:    dir ← East
5:    repeat
6:       count ← count + 1
7:       repeat
8:          CautiousTest(1, dir)
9:          Walk(dir)
10:         t ← Read()
11:      until t > 0
12:      Danger ← BHConfig(t)
13:   until ((count ≥ 5)OR(Danger))
14:   if ¬Danger then
15:      dir ← South
16:      CautiousTest(1, dir) //try the next ring
17:      Take(2) //remove the home base
18:      Walk(dir)
19:      Danger ← BHConfig(Read())
20: until (Danger)
21: LocalSearch(dir)
```

Theorem 4. *Algorithm $UBHS_{5,3}$ identifies a black hole in any type 1 torus using 5 scattered agents with constant memory and 3 movable tokens each.*

Algorithm $UBHS_{5,3}$ can also solve the BHS problem in tori of type 2 or 3. For tori of type 1 this algorithm is almost optimal since in view of Lemma 4, in such tori the problem cannot be solved using 5 agents with 1 token.

6 Conclusion

We showed that any constant number of scattered agents with constant memory and one movable token each cannot locate the black hole in all unoriented tori (type 0) mainly due to the fact that it is impossible for the agents with just one token to explore all such tori. However we also proved that one agent with constant memory and just two movable tokens *can* explore all unoriented tori. This result is optimal on the number of agents and tokens and has its own importance since it has been shown ([22]) that an agent with any constant number of tokens cannot explore all cubic planar graphs. It remains unclear whether a small team of agents (at least 5 are needed) with constant memory and two movable tokens each could solve the BHS problem in all unoriented tori. Since one agent with two tokens can explore all tori of type 0, a negative answer to the above question would need a different reasoning than in the one token scenario. We also showed that four (five) scattered agents with constant memory and one token are not able to locate a black hole in all semi-oriented tori of type 3 (2 or 1). We gave an almost optimal algorithm which enables five scattered agents with constant memory and three tokens each to locate the black hole in all semi-oriented tori of type 1, 2 or 3. This algorithm can be transformed to work with five agents and two tokens in all semi-oriented tori of type 3 (it will appear in the full version of the paper). While we conjecture that a small team of scattered agents (at least five) with constant memory equipped with a few tokens (at least two) would be able to locate a black hole in all totally unoriented tori, a tight solution remains an open question.

Aknowledgements. We wish to thank Dr. Shantanu Das for the discussion on the exploration problem of an unoriented torus by a finite automaton. We also wish to thank the anonymous referees for their helpful suggestions.

References

1. Balamohan, B., Dobrev, S., Flocchini, P., Santoro, N.: Asynchronous Exploration of an Unknown Anonymous Dangerous Graph with $O(1)$ Pebbles. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 279–290. Springer, Heidelberg (2012)
2. Bender, M.A., Slonim, D.: The power of team exploration: Two robots can learn unlabeled directed graphs. In: Proc. of 35th Annual Symp. on Foundations of Computer Science, pp. 75–85 (1994)
3. Blum, M., Kozen, D.: On the power of the compass (or, why mazes are easier to search than graphs). In: Proc. of 19th Symp. on Foundations of Computer Science, pp. 132–142 (1978)

4. Budach, L.: Automata and labyrinths. Mathematische Nachrichten 86(1), 195–282 (1978)
5. Chalopin, J., Das, S., Labourel, A., Markou, E.: Black Hole Search with Finite Automata Scattered in a Synchronous Torus. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 432–446. Springer, Heidelberg (2011)
6. Chalopin, J., Das, S., Labourel, A., Markou, E.: Tight Bounds for Scattered Black Hole Search in a Ring. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 186–197. Springer, Heidelberg (2011)
7. Cooper, C., Klasing, R., Radzik, T.: Searching for Black-Hole Faults in a Network Using Multiple Agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 320–332. Springer, Heidelberg (2006)
8. Czyzowicz, J., Kowalski, D., Markou, E., Pelc, A.: Complexity of searching for a black hole. Fundamenta Informaticae 71(2,3), 229–242 (2006)
9. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. J. of Graph Theory 32(3), 265–297 (1999)
10. Dobrev, S., Flocchini, P., Královic, R., Santoro, N.: Exploring an Unknown Graph to Locate a Black Hole Using Tokens. In: Navarro, G., Bertossi, L., Kohayakawa, Y. (eds.) TCS 2006. IFIP, vol. 209, pp. 131–150. Springer, Boston (2006)
11. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Searching for a black hole in arbitrary networks: Optimal mobile agents protocols. Distributed Computing 19(1), 1–19 (2006)
12. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Mobile search for a black hole in an anonymous ring. Algorithmica 48, 67–90 (2007)
13. Dobrev, S., Santoro, N., Shi, W.: Using scattered mobile agents to locate a black hole in an un-oriened ring with tokens. Int. J. of Foundations of Computer Science 19(6), 1355–1372 (2008)
14. Flocchini, P., Ilcinkas, D., Santoro, N.: Ping pong in dangerous graphs: Optimal black hole search with pebbles. Algorithmica 62(3-4), 1006–1033 (2012)
15. Flocchini, P., Kellett, M., Mason, P., Santoro, N.: Map construction and exploration by mobile agents scattered in a dangerous network. In: Proc. of IEEE Int. Symp. on Parallel & Distributed Processing, pp. 1–10 (2009)
16. Fraigniaud, P., Gasieniec, L., Kowalski, D., Pelc, A.: Collective tree exploration. Networks 48, 166–177 (2006)
17. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Hardness and approximation results for black hole search in arbitrary graphs. Theoretical Computer Science 384(2-3), 201–221 (2007)
18. Kosowski, A., Navarra, A., Pinotti, C.M.: Synchronization Helps Robots to Detect Black Holes in Directed Graphs. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS. LNCS, vol. 5923, pp. 86–98. Springer, Heidelberg (2009)
19. Kozen, D.: Automata and planar graphs. In: Proc. of Fundamentals of Computation Theory, pp. 243–254 (1979)
20. Mans, B.: Optimal distributed algorithms in unlabeled tori and chordal rings. Journal of Parallel and Distributed Computing 46(1), 80–90 (1997)
21. Rabin, M.: Maze threading automata. Seminar talk presented at the University of California at Berkeley (October 1967)
22. Rollik, H.: Automaten in planaren graphen. Acta Informatica 13, 287–298 (1980)
23. Shannon, C.E.: Presentation of a maze-solving machine. In: Proc. of 8th Conf. of the Josiah Macy Jr. Found (Cybernetics), pp. 173–180 (1951)
24. Shi, W.: Black Hole Search with Tokens in Interconnected Networks. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 670–682. Springer, Heidelberg (2009)

Algorithms for Partial Gathering of Mobile Agents in Asynchronous Rings*

Masahiro Shibata, Shinji Kawai, Fukuhito Ooshita,
Hirotsugu Kakugawa, and Toshimitsu Masuzawa

Graduate School of Information Science and Technology, Osaka University
{m-sibata,s-kawai,f-oosita,kakugawa,masuzawa}@ist.osaka-u.ac.jp

Abstract. In this paper, we consider the partial gathering problem of mobile agents in asynchronous unidirectional rings equipped with whiteboards on nodes. The partial gathering problem requires, for a given input g, that each agent should move to a node and terminates so that at least g agents should meet at the same node. The requirement for the partial gathering is weaker than that for the ordinary (total) gathering, and thus, we have interests in clarifying the difference on the move complexity between them. We propose two algorithms to solve the partial gathering problem. One algorithm is deterministic and assumes unique ID of each agent. The other is randomized and assumes anonymous agents. The deterministic (resp., randomized) algorithm achieves the partial gathering in $O(gn)$ (resp., expected $O(gn + n \log k)$) total number of moves where n is the ring size and k is the number of agents, while the total gathering requires $\Omega(kn)$ moves. We show that the move complexity of the deterministic algorithm is asymptotically optimal.

Keywords: distributed system, mobile agent, gathering problem, partial gathering.

1 Introduction

1.1 Background and Our Contribution

A *distributed system* is a system that consists of a set of computers (*nodes*) and communication links. In recent years, distributed systems have become large and design of distributed systems has become complicated. As a way to design efficient distributed systems, (mobile) agents have attracted a lot of attention [1–10]. Agents simplify design of distributed systems because they can traverse the system and process tasks on each node.

The gathering problem is a fundamental problem for cooperation of agents [1–11]. The gathering problem requires all agents to meet at a single node in finite time. The gathering problem is useful because, by meeting at a single node, all agents can share information or synchronize behaviors among them.

* This work is supported in part by Grant-in-Aid for Scientific Research ((B)2030012, (B)22300009, (B)23700056, (C)24500039) of JSPS.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 254–268, 2012.

Table 1. Proposed algorithms for the g-partial gathering problem in asynchronous unidirectional rings

Model	Algorithm 1	Algorithm 2
Unique ID	Available	Not available
Deterministic/Randomized	Deterministic	Randomized
Knowledge of k	Not available	Available
The total number of moves	$O(gn)$	$O(n \log k + gn)$

In this paper, we consider a new variant of the gathering problem, called the *partial gathering problem*. The partial gathering problem does not always require all agents to gather at a single node, but requires agents to gather partially at several nodes. More precisely, we consider the problem which requires, for given input g, that each agent should move to a node and terminate so that at least g agents should meet at the same node. We define this problem as the *g-partial gathering problem*. Clearly, if $k/2 < g \leq k$ holds, the g-partial gathering problem is equal to the ordinary gathering problem. If $g \leq k/2$ holds, the requirement for the g-partial gathering problem is weaker than that for the ordinary gathering problem, and thus it seems possible to solve the g-partial gathering problem with a smaller total number of moves. In addition, the g-partial gathering problem is still useful because agents can share information and process tasks cooperatively among at least g agents.

The contribution of this paper is to clarify the difference on the move complexity between the gathering problem and the g-partial gathering problem. We consider the g-partial gathering problem in asynchronous unidirectional rings equipped with whiteboards on nodes. The contribution of this paper is summarized in Table 1. First, we propose a deterministic algorithm to solve the g-partial gathering problem for the case that agents have distinct IDs. This algorithm requires $O(gn)$ total number of moves. Second, we propose a randomized algorithm to solve the g-partial gathering problem for the case that agents have no IDs and agents know the number of agents. This algorithm requires $O(n \log k + gn)$ total number of moves, while the total gathering requires $\Omega(kn)$ moves. The two algorithms imply that the g-partial gathering problem can be solved in a smaller total number of moves compared to the ordinary (total) gathering problem for both cases. In addition, we show that the total number of moves is $\Omega(gn)$ for the g-partial gathering problem if $g \geq 2$. This means the first algorithm is asymptotically optimal in terms of the total number of moves.

1.2 Related Works

Many fundamental problems for cooperation of mobile agents have been studied in literature. For example, the searching problem [7, 8], the gossip problem [9], the election problem [10], and the gathering problem [1–11] have been studied.

In particular, the gathering problem has received a lot of attention and has been extensively studied in many topologies, which include trees [1, 9], tori [1, 5],

and rings [1–4, 6–11]. The gathering problem for rings has been extensively studied because algorithms for such highly symmetric topologies give techniques to treat the essential difficulty of the gathering problem such as breaking symmetry. Actually, to solve the gathering problem, it is necessary to select exactly one gathering node, i.e., a node where all agents meet. There are many ways to select the gathering node. For example, in [1–6], agents leave marks (tokens) on their initial nodes and select the gathering node based on every distance of neighboring tokens. In [7, 8], agents have distinct IDs and select the gathering node based on the IDs. In [11], agents can use random numbers and select the gathering node based on IDs generated randomly. In [1, 9, 10], agents execute the leader agent election and the elected leader decides the gathering node.

2 Preliminaries

2.1 Network Model

A *unidirectional ring network* R is a tuple $R = (V, L)$, where V is a set of nodes and L is a set of communication links. We denote by n $(= |V|)$ the number of nodes. Then, ring R is defined as follows.

- $V = \{v_0, v_1, \ldots, v_{n-1}\}$
- $L = \{v_i, v_{(i+1) \bmod n} \mid 0 \leq i \leq n-1\}$

We define the direction from v_i to v_{i+1} as a *forward* direction, and the direction from v_{i+1} to v_i as a *backward* direction.

In this paper, we assume nodes are anonymous, i.e., each node has no ID. Every node $v_i \in V$ has a whiteboard and agents on node v_i can read from and write to the whiteboard of v_i. We define W as a set of all states of a whiteboard.

2.2 Agent Model

Let $A = \{a_1, a_2, \ldots, a_k\}$ be a set of agents. We consider two model variants.

In the first model, we consider agents that are distinct (i.e., agents have distinct IDs) and execute a deterministic algorithm. We model an agent as an identical finite automaton $(S, \delta, s_{initial}, s_{final})$. The first element S is the set of all states of agents, which includes initial state $s_{initial}$ and final state s_{final}. The second element δ is the state transition function. Since we treat deterministic algorithms, δ is described as δ: $S \times W \to S \times W \times M$, where $M = \{1, 0\}$ represents whether the agent makes a movement or not in the step. The value 1 represents movement to the next node and 0 represents stay at the current node. Since rings are unidirectional, each agent only moves to its forward node. We assume that agents move instantaneously, that is, agents always exist at nodes (do not exist at links). Moreover, we assume that each agent cannot detect the number of agents on its current node.

In the second model, we consider agents that are anonymous (i.e., agents have no IDs) and execute a randomized algorithm. We model an agent similarly to

the first model except for state transition function δ. Since we treat randomized algorithms, δ is described as δ: $S \times W \times R \to S \times W \times M$, where R represents a set of random values. In addition, we assume that each agent knows the number of agents.

2.3 System Configuration

In an agent system, (global) *configuration* c is defined as a product of states of agents, states of nodes (whiteboards), and locations of agents. We define C as a set of all configurations. In initial configuration $c_0 \in C$, we assume that no pair of agents stay at the same node. We assume that each node v_j has variable $v_j.initial$ that indicates existence of agents in the initial configuration. If there exists an agent on node v_j in the initial configuration, the value of $v_j.initial$ is one. Otherwise, the value of $v_j.initial$ is zero.

Let A_i be an arbitrary non-empty set of agents. When configuration c_i changes to c_{i+1} by the action of every agent in A_i, we denote the transition by $c_i \xrightarrow{A_i} c_{i+1}$. In c_i, each $a_j \in A_i$ reads values written on its node's whiteboard, executes local computation, writes values to the node's whiteboard, and decides whether a_j moves to the next node or stay the current node. In this consecutive sequence, we say that a_j takes one step. If multiple agents at the same node are included in A_i, the agents take steps in an arbitrary order. When $A_i = A$ holds for any i, all agents take steps. This model is called the *synchronous model*. Otherwise, the model is called the *asynchronous model*.

If sequence of configurations $E = c_0, c_1, \ldots$ satisfies $c_i \xrightarrow{A_i} c_{i+1}$ $(i \geq 0)$, E is called an *execution* starting from c_0. Execution E is infinite, or ends in final configuration c_{final} where no agent can take a step.

2.4 Partial Gathering Problem

The requirement of the partial gathering problem is that, for a given input g, each agent should move to a node and terminate so that at least g agents should meet at the node. Formally, we define the g-partial gathering problem as follows.

Definition 1. *Execution E solves the g-partial gathering problem when the following conditions hold:*

- *Execution E is finite.*
- *In the final configuration, for any node v_j such that there exist some agents on v_j, there exist at least g agents on v_j.*

For the g-partial gathering problem, we have the following lower bound.

Theorem 1. *The total number of moves required to solve the g-partial gathering problem is $\Omega(gn)$ if $g \geq 2$.*

3 A Deterministic Algorithm for Distinct Agents

In this section, we propose a deterministic algorithm to solve the g-partial gathering problem for distinct agents (i.e., agents have distinct IDs). The basic idea to solve the g-partial gathering is that agents select a leader and then the leader instructs other agents which node they meet at. However, since $\Omega(n \log k)$ total number of moves is required to elect one leader [9], it is impossible to solve the g-partial gathering in asymptotically optimal total number of moves (i.e., $O(gn)$). To overcome this lower bound, we select multiple agents as leaders by executing leader agent election partially. By this behavior, our algorithm solves the g-partial gathering problem in $O(gn)$ total number of moves.

The algorithm consists of two parts. In the first part, agents execute leader agent election partially and elect some leader agents. In the second part, leader agents instruct the other agents which node they meet at, and the other agents move to the node by the instruction.

3.1 The First Part

The aim of the first part is to elect leaders that satisfy the following properties: 1) At least one agent is elected as a leader, 2) At most $\lfloor k/g \rfloor$ agents are elected as leaders, and 3) There exist at least $g-1$ non-leader agents between two leader agents. To attain this goal, we use a traditional leader election algorithm [12]. However the algorithm in [12] is executed by nodes and the goal is to elect exactly one leader. So we modify the algorithm to be executed by agents, and then agents elect at most $\lfloor k/g \rfloor$ leader agents by executing the algorithm partially.

During the execution of leader election, the states of agents are divided into the following three types:

- *active*: The agent is performing the leader agent election as a candidate of leaders.
- *inactive*: The agent has dropped out from the candidate of leaders.
- *leader*: The agent has been elected as a leader.

First, we explain the idea of leader election by assuming that the ring is bidirectional. The algorithm consists of several phases. In each phase, each active agent compares its own ID with IDs of its left and right neighbor active agents. More concretely, each active agent writes its ID on the whiteboard of its current node, and then moves forward and backward to observe IDs of the forward and backward active agents. If its own ID is the smallest among the three agents, the agent remains active as a candidate of leaders. Otherwise, the agent drops out from candidates of leaders and becomes inactive. By doing this, at least half active agents become inactive in each phase. Consequently, after executing $\lceil \log g \rceil$ phases, the number of active agents becomes at most $\lfloor k/g \rfloor$. Then, from [12], the number of inactive agents between two active agents is at least $g-1$. Therefore, all remaining active agents become leaders. Note that, during the execution of the algorithm, the number of active agents may become one. In this case, the active agent immediately becomes a leader.

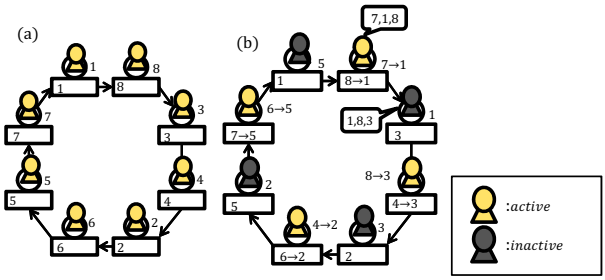


Fig. 1. An example for a g-partial gathering problem($k = 9, g = 3$)

In the following, we implement the above algorithm in asynchronous unidirectional rings. First, we apply a traditional approach [12] to implement the above algorithm in a unidirectional ring. Let us consider the behavior of active agent a_h. In unidirectional rings, a_h cannot move backward and so cannot observe the ID of its backward active agent. Instead, a_h moves forward until it observes IDs of two active agents. Then, a_h observes IDs of three successive active agents. We assume a_h observes id_0, id_1, id_2 in this order. Note that id_0 is the ID of a_h. Here this situation is similar to that the active agent with ID id_1 observes id_0 as its backward active agent and id_2 as its forward active agent in bidirectional rings. For this reason, a_h behaves as if it would be an active agent with ID id_1 in bidirectional rings. That is, if id_1 is the smallest among the three IDs, a_h remains active as a candidate of leaders. Otherwise, a_h drops out from the candidate of leaders and becomes inactive. After the phase, a_h assigns id_1 to its ID if it remains active as a candidate. For example, consider the initial configuration in Fig. 1 (a). In figures, the number near each agent is the ID of the agent and the box of each node represents the whiteboard. First, each agent writes its own ID to the whiteboard on its initial node. Next, each agent moves forward until it observes two IDs, and then the configuration is changed to the one in Fig. 1 (b). In this configuration, each agent compares three IDs. The agent with ID 1 observes IDs (1, 8, 3), and so it drops out from the candidate because the middle ID 8 is not the smallest. The agents with IDs 3, 2, and 5 also drop out from the candidate. The agent with ID 7 observes IDs (7, 1, 8), and so it remains active as a candidate because the middle ID 1 is the smallest. Then, it updates its ID to 1. The agents with IDs 8, 4, and 6 also remain active as candidates and similarly update their IDs.

Next, we explain the way to treat asynchronous agents. To recognize the current phase, each agent manages *phase number*. Initially, the phase number is one, and it is incremented when each phase is completed. Each agent compares IDs with agents that have the same phase number. To realize this, when each agent writes its ID to the whiteboard, it also writes its phase number. That is, at the beginning of each phase, active agent a_h writes a tuple $(phase, id_h)$ to the whiteboard on its current node, where $phase$ is the current phase number and

id_h is the ID of a_h. After that, agent a_h moves until it sees two IDs with the same phase number as that of a_h. Then, a_h decides whether it remains active as a candidate or becomes inactive. If a_h remains active, it updates its own ID. Agents repeat these behaviors until they complete the $\lceil \log g \rceil$-th phase.

Pseudocode. The pseudocode to elect leader agents is given in Algorithm 1. All agents start the algorithm with active states. The pseudocode describes the behavior of active agent a_h, and v_j represents the node where agent a_h currently stays. If agent a_h becomes an inactive state or a leader state, a_h immediately moves to the next part and executes the algorithm for an inactive state or a leader state in section 3.2. Agent a_h uses variables $a_h.id_1$, $a_h.id_2$, and $a_h.id_3$ to store three IDs of three successive active agents. Note that a_h stores its ID on $a_h.id_1$ and initially assigns its initial ID ($a_h.id$) to $a_h.id_1$. Variable $a_h.phase$ stores the phase number of a_h. Agent a_h also has variable $a_h.passed$ that indicates whether it has been passed by an another agant. The initial value of $a_h.passed$ is zero. Each node v_j has variable $(v_j.phase, v_j.id)$, where an active agent writes its phase number and its ID. For any v_j, variable $(v_j.phase, v_j.id)$ is $(0, 0)$ initially. In addition, each node v_j has variable $v_j.inactive$. This variable represents whether there exists an inactive agent on v_j. That is, agents update the variable to keep the following invariant: If there exists an inactive agent on v_j, $v_j.inactive = 1$ holds, and otherwise $v_j.inactive = 0$ holds. Initially $v_j.inactive = 0$ holds for any v_j. In Algorithm 1, a_h uses procedure $BasicAction()$, by which agent a_h moves to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$.

We give the pseudocode of $BasicAction()$ in Algorithm 2. In $BasicAction()$, the main behavior of a_h is to move to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$. To realize this, a_h skip nodes such that no agent initially exists (i.e., $v_j.initial = 0$) or an inactive agent currently exists (i.e., $v_j.inactive = 1$), and continue to move until it reaches a node where some active agents start the same phase (lines 2 to 4). In addition to this behavior, a_h makes some behaviors to treat asynchrony. If a_h finds agent a_x that has not yet started the algorithm on v_j, a_h makes a_x drop out from candidates by setting $v_j.inactive = 1$ (lines 5 to 8). When a_h notices that it has passed some active agents, a_h waits until the agents catch up with a_h (lines 9 to 14). If the agent becomes inactive, a_h continues to move (lines 11 to 13). If a_h has been passed by an another agant, a_h move in the ring until it observes IDs of two active agent and then become inactive (lines 11 to 13 of Algortithm 1 and lines 15 to 17 of Algorithm 2). During the algorithm, it is possible that a_h becomes the only one candidate of leaders. In this case, a_h immediately becomes a leader (lines 19 to 20).

We have the following lemma about Algorithm 1 [12].

Lemma 1. *After executing Algorithm 1, the configuration satisfies the following.*

- *There exists at least one leader agent.*
- *There exist at most* $\lfloor \frac{k}{g} \rfloor$ *leader agents.*
- *There exist at least* $g - 1$ *inactive agents between two leader agents.*

In addition, we have the following lemma [12].

Algorithm 1. The behavior of active agent a_h (v_j is the current node of a_h)

```
1: set a_h.phase = 1 and a_h.id_1 = a_h.id
2: if v_j.inactive = 1 then
3:    // Some agents have passed a_h before a_h starts the algorithm.
4:    become inactive
5: end if
6: set (v_j.phase, v_j.id) = (a_h.phase, a_h.id_1)
7: BasicAction()
8: set a_h.id_2 = v_j.id
9: BasicAction()
10: set a_h.id_3 = v_j.id
11: if a_h.passed = 1 then
12:    set v_j.inactive = 1 and become inactive
13: end if
14: if a_h.id_2 > min(a_h.id_1, a_h.id_3) then
15:    set v_j.inactive = 1 and become inactive
16: else
17:    if a_h.phase = ⌈log g⌉ then
18:       become a leader
19:    else
20:       set a_h.phase = a_h.phase + 1
21:       set a_h.id_1 = a_h.id_2
22:    end if
23:    return to step 6
24: end if
```

Lemma 2. *The total number of moves to execute Algoritm 1 is* $O(n \log g)$.

3.2 The Second Part

In this section, we explain the second part, i.e., an algorithm to achieve g-partial gathering by using leaders elected by the algorithm in section 3.1. Let leader nodes (resp., inactive nodes) be the nodes where an agent becomes a leader (resp., an inactive agent) in the first part. The idea of the algorithm is as follows: First each leader agent a_h writes 0 to the whiteboard on the current node. Then, a_h repeatedly moves and, whenever a_h visits an inactive node, a_h writes y mod g to the whiteboard, where y is the number of inactive nodes a_h has ever visited. That is, a_h writes $0, 1, \ldots, g-1, 0, 1, \ldots$ to the whiteboard on inactive nodes. This number is used to instruct inactive agents where they should move to achieve g-partial gathering. Agent a_h continues this operation until it visits the node where 0 is already written to the whiteboard. Note that this node is a leader node. For example, consider the configuration in Fig. 2 (a). In this configuration, agents a_1 and a_2 are leader agents. First, a_1 and a_2 write 0 to their current whiteboards, and then they move and write numbers to whiteboards until they visit the node where 0 is written on the whiteboard. Then, the system reaches the configuration in Fig. 2 (b).

Algorithm 2. Procedure *BasicAction*() for a_h

```
move to the forward node
while (v_j.initial = 0) ∨ (v_j.inactive = 1) do
    move to the forward node
end while
if v_j.phase = 0 then
    set v_j.inactive = 1
    return to step 2
end if
if a_h.phase > v_j.phase then
    wait until v_j.phase = a_h.phase or v_j.inactive = 1
    if v_j.inactive = 1 then
        return to step 2
    end if
end if
if a_h.phase < v_j.phase then
    set a_h.passed = 1
end if
// a_h reaches v_j s.t. v_j.phase = a_h.phase.
if (v_j.phase, v_j.id) = (a_h.phase, a_h.id_1) then
    become a leader
end if
```

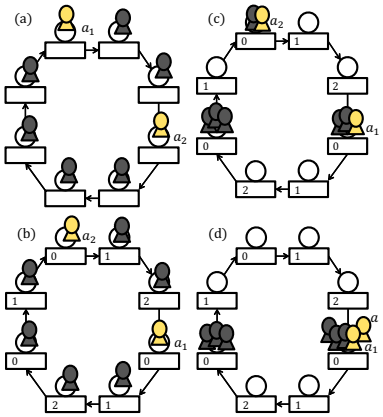


Fig. 2. The realization of partial gathering($g = 3$)

Then, each non-leader agent (i.e., inactive agent) moves based on the leader's instruction, i.e., the number written to the whiteboard. More concretely, each inactive agent moves to the node where 0 is written to the whiteboard. For example, after the configuration in Fig. 2 (b), the system reaches the configuration in Fig. 2 (c). Note that a node where 0 is written is a leader node or an inactive node. If the node is an inactive node, g agents meet at the node. If the node is

a leader node, it is possible that only less than g agents meet at the node. In this case, the agents continue to move until they visit the next node where 0 is written. By executing such operations, agents can solve the g-partial gathering problem. For example, there exist only two agents on the node where a_2 exists in Fig. 2 (c). So the two agents continue to move until they visit the next node where 0 is written (Fig. 2 (d)).

Pseudocode. In the following, we show the pseudocode of the algorithm. In this part, states of agents are divided into the following three states

- *leader*: The agent instructs inactive agents where they should move.
- *inactive*: The agent waits for the leader's instruction.
- *moving*: The agent moves to its gathering node.

In this part agents continue to use $v_j.initial$ and $v_j.inactive$. Remind that $v_j.initial = 1$ if and only if there exists an agent at v_j initially. Algorithm 1 assures $v_j.inactive = 1$ if and only if there exists an inactive agent at v_j. Note that, since each agent becomes inactive or a leader at a node such that there exists an agent initially, agents can ignore and skip every node $v_{j'}$ such that $v_{j'}.initial = 0$.

The pseudocode of leader agents is described in Algorithm 3. Variable $a_h.count$ is used to count the number of inactive nodes a_h visits (The counting is done modulo g). The initial value of $a_h.count$ is 0. Variable $v_j.count$ is used for leader agents to instruct inactive agents. That is, leader agent a_h writes $a_h.count$ to $v_j.count$ when it visits inactive node v_j. For any v_j, the initial value of $v_j.count$ is $\perp$. In asynchronous rings, leader agent a_h may pass agents that still execute Algorithm 1. To avoid this, a_h waits until the agents catch up with a_h. More precisely, when leader agent a_h visits the node v_j such that $v_j.initial = 1$, it passed such agents if $v_j.inactive = 0$ and $v_j.count = \perp$ hold. This is because $v_j.inactive = 1$ should hold if some agent becomes inactive at v_j, and $v_j.count = 0$ holds if some agent becomes leader at v_j. In this case, a_h waits there until either $v_j.inactive = 1$ or $v_j.count = 0$ holds (lines 8 to 10). When leader agent updates $v_j.count$, an inactive agent on node v_j becomes a moving state (line 12). This behavior of inactive agents is given in the pseudocode of inactive agents (See Algorithm 4). After a leader agent reaches the next leader node, it becomes a moving agent to move to the node where at least g agents meet (line 17). Note that variable $a_h.Bcount$ is used in the pseudocode for moving agents, and so we explain the variable later.

The pseudocode of moving agents is described in Algorithm 5. Moving agent a_h continues to move until it visits node v_j such that $v_j.count = 0$. When a_h visits such a node, it is possible that only less than g agents come to the node like Fig. 2 (c). To solve this case, a_h keeps the value of $v_l.count$ for the previous inactive node v_l as variable $a_h.Bcount$. When a_h visits node v_j such that $v_j.count = 0$, if $a_h.Bcount = g - 1$ holds, at least g agents come to v_j. Otherwise, less than g agents come to v_j, and so a_h moves to the next node $v_{j'}$ such that $v_{j'}.count = 0$. Since there exist at least $g - 1$ inactive nodes between two leader nodes, at least g agents meet at $v_{j'}$. Note that such additional moves

Algorithm 3. The behavior of leader agent a_h (v_j is the current node of a_h)

```
1: set a_h.count = 0, v_j.count = a_h.count and a_h.count = a_h.count + 1
2: move to the forward node
3: while v_j.count ≠ 0 do
4:    while v_j.initial = 0 do
5:       move to the forward node
6:    end while
7:    if (v_j.inactive = 0) ∧ (v_j.count =⊥) then
8:       wait until v_j.inactive = 1 or v_j.count = 0
9:    end if
10:   if v_j.inactive = 1 then
11:      set v_j.count = a_h.count
12:      // an inactive agent at v_j becomes a moving state
13:      set a_h.count = (a_h.count + 1) mod g
14:      set a_h.Bcount = v_j.count
15:      move to the forward node
16:   end if
17: end while
18: become a moving state
```

Algorithm 4. The behavior of inactive agent a_h (v_j is the current node of a_h)

```
1: wait until v_j.count ≠⊥
2: set a_h.Bcount =⊥
3: become a moving state
```

Algorithm 5. The behavior of moving agent a_h (v_j is the current node of a_h)

```
1: while v_j.count ≠ 0 do
2:    move to the forward node
3:    if (v_j.initial = 1) ∧ (v_j.count =⊥) then
4:       wait until v_j.count ≠⊥
5:    end if
6:    if v_j.count ≠⊥ then
7:       set a_h.Bcount = v_j.count
8:    end if
9: end while
10: if (a_h.Bcount ≠ g − 1) ∧ (a_h.Bcount ≠⊥) then
11:    set a_h.Bcount = 0
12:    move to the forward node
13:    return to step 1
14: end if
15: terminate
```

are required only for leader nodes. If inactive node v_j satisfies $v_j.count = 0$, at least g agents comes to v_j. This means inactive agent a_h on such v_j does not need to move, which is treated as $a_h.Bcount = \perp$.

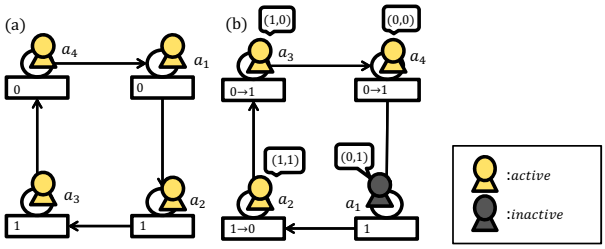


Fig. 3. A randomized leader election for anonymous agents

In asynchronous rings, a moving agent may pass leader agents. To avoid this, the moving agent waits until the leader agent catches up with it. More precisely, if moving agent a_h visits node v_j such that $v_j.initial = 1$ and $v_j.count = \perp$, a_h passed a leader agent. To wait for the leader agent, a_h waits there until the value of $v_j.count$ is updated.

We have the following lemma about the algorithm in section 3.2.

Lemma 3. *After the leader agent election, agents solve the g-partial gathering problem in $O(gn)$ total number of moves.*

From Lemmas 2 and 3, we have the following theorem.

Theorem 2. *When agents have distinct IDs, our deterministic algorithm solves the g-partial gathering problem in $O(gn)$ total number of moves.*

4 A Randomized Algorithm for Anonymous Agents

In this section, we propose a randomized algorithm to solve the g-partial gathering problem for the case of anonymous agents. The idea of the algorithm is the same as that in section 3. Agents elect leader agents in the first part, and the leader agents instruct the other agents where they move in the second part. The difference from the algorithm in section 3 is that agents elect exactly one leader by randomization in the first part. In the second part, we use the same algorithm as that in section 3.

4.1 The First Part

In this subsection, we explain a randomized algorithm to elect one leader agent from anonymous agents. Similarly to section 3, the state of each agent is either active, inactive, or leader. Initially all agents are active. If an agent becomes inactive or a leader, it immediately moves to the second part of the algorithm.

The algorithm consists of several phases. In each phase, each active agent a_h writes a random bit to the whiteboard and moves to the next node where its forward agent a_f writes a random bit. Then, a_h compares the random bit of a_h with that of a_f. If the random bit of a_h is zero and the random bit of a_f is one, a_h drops out from candidates of the leader. Otherwise, a_h remains

Algorithm 6. The behavior of active agent a_h (v_j is the current node of a_h)

1: set $a_h.phase = 1$ and $a_h.num = 0$
2: **if** $v_j.inactive = 1$ **then**
3: // Some agents pass a_h before a_h starts the algorithm.
4: become inactive
5: **end if**
6: set $a_h.r = 0$ with probability 1/2
and $a_h.r = 1$ with probability 1/2
7: set $(v_j.phase, v_j.r) = (a_h.phase, a_h.r)$
8: *BasicAction*2()
9: **if** $a_h.r = 0$ and $v_j.r = 1$ **then**
10: set $v_j.inactive = 1$ and become inactive
11: **end if**
12: set $a_h.phase = a_h.phase + 1$ and $a_h.num = 0$
13: return to step 6

active as a candidate, and moves to the next phase. Since a_h drops out with probability 1/4 and at least one of a_h and a_f remains active as a candidate, eventually exactly one active agent remains active as a candidate by repeating the phase. For example, consider the initial configuration in Fig. 3 (a). Numbers on whiteboards are random bits written by the resident agents. Each agent moves to the next node and then compares random bits. Because the random bit of a_1 is zero and the random bit of its forward agent a_2 is one, a_1 drops out from the candidate of the leader. The other agents remain active as candidates and update random numbers on the whiteboards (Fig. 3).

To execute the above algorithm, we treat asynchronous agents in the same way as the algorithm in section 3.1. Each agent manages *phase number* to recognize the current phase. Each agent writes its random bit together with its phase number, and compares its random bit with an agent that has the same phase number. In addition, since active agent a_h may pass some other active agents, a_h waits in the same way until the agents catch up with a_h.

Pseudocode. The pseudocode of active agents is described in Algorithm 6. Agent a_h stores its phase number in variable $a_h.phase$ and its random bit in variable $a_h.r$. Each node v_j has variable $(v_j.phase, v_j.r)$, where an active agent writes its phase number and its random bit. For any v_j, variable $(v_j.phase, v_j.r)$ is $(0, 0)$ initially. In addition to these variables, a_h manages $a_h.num$ to count the number of inactive agents in each phase. If $a_h.num = k - 1$ holds, a_h is a unique active agent and thus becomes a leader (This behavior is implemented in *BasicAction*2()).

At the beginning of each phase, each a_h generates a random bit and stores it in $a_h.r$. Then, it writes $(a_h.phase, a_h.r)$ to variable $(v_j.phase, v_j.r)$ at the whiteboard of its current node v_j. Since the forward active agent of a_h also writes a random bit to the whiteboard of its current node v_f, agent a_h compares the two random bits when a_h visits v_f. In Algorithm 6, a_h uses procedure *BasicAction*2(), by which a_h moves to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$.

Algorithm 7. Procedure *BasicAction*2()

```
 1: move to the forward node
 2: while (v_j.initial = 0) ∨ (v_j.inactive = 1) do
 3:    if v_j.inactive = 1 then
 4:       set a_h.num = a_h.num + 1
 5:    end if
 6:    move to the forward node
 7: end while
 8: if v_j.phase = 0 then
 9:    set v_j.inactive = 1 and a_h.num = a_h.num + 1
10:    return to step 2
11: end if
12: if v_j.phase ≠ a_h.phase then
13:    wait until v_j.phase = a_h.phase or v_j.inactive = 1
14:    if v_j.inactive = 1 then
15:       set a_h.num = a_h.num + 1
16:       return to step 2
17:    end if
18: end if
19: // a_h reaches v_j s.t. v_j.phase = a_h.phase.
20: if a_h.num = k − 1 then
21:    become a leader
22: end if
```

We give the pseudocode of *BasicAction*2() in Algorithm 7. The implementation is almost the same as that of *BasicAction*() in section 3.2. The difference is that a_h increments $a_h.num$ whenever a_h sees inactive agents. If a_h observes $k-1$ inactive agents, a_h is a unique active agent and thus becomes a leader.

We have the following lemma about Algorithm 6.

Lemma 4. *Algorithm 6 solves the leader agent election with* $O(n \log k)$ *expected total moves.*

4.2 The Second Part

In the second part of this algorithm, we use the same algorithm in section 3.2. Since Algorithm 6 selects exactly one leader agent, the conditions in Lemma 1 hold for Algorithm 6. In addition, Algorithm 6 satisfies the following: 1) Each agent becomes inactive or a leader at node v_j such that $v_j.initial = 1$, and 2) If there exists an inactive agent on v_j, $v_j.inactive = 1$ holds, and otherwise $v_j.inactive = 0$ holds. Since these are sufficient conditions to apply the algorithm in section 3.2, we can execute the algorithm in section 3.2 after the algorithm in section 4.1.

From Lemmas 4 and 3, we have the following theorem.

Theorem 3. *When agents have no IDs, our randomized algorithm solves the g-partial gathering problem in* $O(n \log k + gn)$ *expected total moves.*

5 Conclusion

In this paper, we have proposed two algorithms to solve the g-partial gathering problem in asynchronous unidirectional rings. The first algorithm is deterministic and assumes distinct agents, and the second algorithm is randomized and assumes anonymous agents. In the first algorithm, several agents are elected as leaders by executing the leader agent election partially. On the other hand, in the second algorithm, the unique leader is elected. After the leader election, leader agents instruct the other agents where they meet. We have showed that the first algorithm requires $O(gn)$ total moves, which is asymptotically optimal. One of future works is to propose a randomized algorithm for anonymous agents to solve the g-partial gathering problem in $O(gn)$ expected total moves. Another future work is to consider the solvability of deterministic g-partial gathering, that is, we will clarify what initial configurations are solvable and what complexity is required.

References

1. Kranakis, E., Krozanc, D., Markou, E.: The mobile agent rendezvous problem in the ring. Synthesis Lectures on Distributed Computing Theory (2010)
2. Gąsieniec, L., Kranakis, E., Krizanc, D., Zhang, X.: Optimal Memory Rendezvous of Anonymous Mobile Agents in a Unidirectional Ring. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Štuller, J. (eds.) SOFSEM 2006. LNCS, vol. 3831, pp. 282–292. Springer, Heidelberg (2006)
3. Kranakis, E., Santoro, N., Sawchuk, S.: Mobile agent rendezvous in a ring. In: Distributed Computing Systems (2003)
4. Flocchini, P., Kranakis, E., Krizanc, D., Santoro, N., Sawchuk, C.: Multiple Mobile Agent Rendezvous in a Ring. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 599–608. Springer, Heidelberg (2004)
5. Kranakis, E., Krizanc, D., Markou, E.: Mobile Agent Rendezvous in a Synchronous Torus. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 653–664. Springer, Heidelberg (2006)
6. Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F.L., Santoro, N., Sawchuk, C.: Mobile Agents Rendezvous When Tokens Fail. In: Kralovic, R., Sýkora, O. (eds.) SIROCCO 2004. LNCS, vol. 3104, pp. 161–172. Springer, Heidelberg (2004)
7. Dobrev, S., Flocchini, P., Prencipe, G., Nicola, N.: Mobile search for a black hole in an anonymous ring. Algorithmica 48, 67–90 (2007)
8. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Multiple Agents RendezVous in a Ring in Spite of a Black Hole. In: Papatriantafilou, M., Hunel, P. (eds.) OPODIS 2003. LNCS, vol. 3144, pp. 34–46. Springer, Heidelberg (2004)
9. Suzuki, T., Izumi, T., Ooshita, F., Kakugawa, H., Msuzawa, T.: Move-optimal gossiping among mobile agents. Theoretical Computer Science 393, 90–101 (2008)
10. Barriere, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Rendezvous and election of mobile agents: impact of sense of direction. Theory of Computing System 40, 143–162 (2007)
11. Kawai, S., Ooshita, F., Kakugawa, H., Masuzawa, T.: Randomized Rendezvous of Mobile Agents in Anonymous Unidirectional Ring Networks. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 303–314. Springer, Heidelberg (2012)
12. Peterson, G.L.: An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. TOPLAS 4, 758–762 (1982)

Causality, Influence, and Computation in Possibly Disconnected Synchronous Dynamic Networks*

Othon Michail[1,2], Ioannis Chatzigiannakis[1,2], and Paul G. Spirakis[1,2]

[1] Computer Technology Institute & Press "Diophantus" (CTI), Patras, Greece
[2] Computer Engineering and Informatics Department (CEID), University of Patras
{michailo,ichatz,spirakis}@cti.gr

Abstract. In this work, we study the *propagation of influence and computation in dynamic networks that are possibly disconnected at every instant*. We focus on a *synchronous message passing* communication model with *broadcast* and bidirectional links. To allow for bounded end-to-end communication we propose a set of minimal *temporal connectivity conditions* that bound from the above the time it takes for information to make progress in the network. We show that even in dynamic networks that are disconnected at every instant information may spread as fast as in networks that are connected at every instant. Further, we investigate *termination criteria* when the nodes know some upper bound on each of the temporal connectivity conditions. We exploit our termination criteria to provide efficient protocols (optimal in some cases) that solve the fundamental *counting* and *all-to-all token dissemination* (or *gossip*) problems. Finally, we show that any protocol that is correct in instantaneous connectivity networks can be adapted to work in temporally connected networks.

Keywords: dynamic graph, mobile computing, worst-case dynamicity, adversarial schedule, temporal connectivity, counting, information dissemination.

1 Introduction

Distributed computing systems are more and more becoming dynamic. The static and relatively stable models of computation can no longer represent the plethora of recently established and rapidly emerging information and communication technologies. In recent years, we have seen a tremendous increase in the number of new mobile computing devices. Most of these devices are equipped with some sort of communication, sensing, and mobility capabilities. Even the Internet has become mobile. The design is now focused on complex collections of

* This work has in part been supported by the EU (ESF) and Greek national funds through the Operational Programme "Education and Lifelong Learning" (EdLL), under the title "Foundations of Dynamic Distributed Computing Systems" (FOCUS).

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 269–283, 2012.

heterogeneous devices that should be robust, adaptive, and self-organizing, possibly moving around and serving requests that vary with time. Delay-tolerant networks are highly-dynamic, infrastructure-less networks whose essential characteristic is a possible absence of end-to-end communication routes at any instant. Mobility may be *active*, when the devices control and plan their mobility pattern (e.g. mobile robots), or *passive*, in opportinistic-mobility networks, where mobility stems from the mobility of the carries of the devices (e.g. humans carrying cell phones) or a combination of both (e.g. the devices have partial control over the mobility pattern, like for example when GPS devices provide route instructions to their carriers). Thus, it can vary from being completely predictable to being completely unpredictable. Gossip-based communication mechanisms, e-mail exchanges, peer-to-peer networks, and many other contemporary communication networks all assume or induce some sort of high dynamicity.

The formal study of dynamic communication networks is hardly a new area of research. There is a huge amount of work in distributed computing that deals with causes of dynamicity such as failures and changes in the topology that are rather slow and usually eventually stabilize (like, for example, in self-stabilizing systems [Dol00]). However the low rate of topological changes that is usually assumed there is unsuitable for reasoning about truly dynamic networks. Even graph-theoretic techniques need to be revisited: the suitable graph model is now that of a *dynamic graph* (a.k.a. *temporal graph* or *time-varying graph*) (see e.g. [CFQS11]), in which each edge has an associated set of time-labels indicating availability times. Even fundamental properties of classical graphs do not carry over to their temporal counterparts. For example, Kempe, Kleinberg, and Kumar [KKK00] found out that there is no analogue of Menger's theorem (see e.g. [Bol98] for a definition) for arbitrary temporal networks, which additionally renders the computation of the number of node-disjoint s-t paths **NP**-complete. Even the standard network diameter metric is no more suitable and has to be replaced by a dynamic/temporal version. In a dynamic star graph in which all leaf-nodes but one go to the center one after the other in a modular way, any message from the node that enters last the center to the node that never enters the center needs $n-1$ steps to be delivered, where n is the size (number of nodes) of the network; that is the *dynamic diameter* is $n-1$ while, one the other hand, the classical diameter is just 2 [AKL08] (see also [KO11]).

2 Related Work

Distributed systems with worst-case dynamicity were first studied in [OW05]. Their outstanding novelty was to assume a communication network that may change arbitrarily from time to time subject to the condition that each instance of the network is connected. They studied asynchronous communication and considered nodes that can detect local neighborhood changes; these changes cannot happen faster than it takes for a message to transmit. They studied *flooding* (in which one node wants to disseminate one piece of information to all nodes) and *routing* (in which the information need only reach a particular destination

node t) in this setting. They described a uniform protocol for flooding that terminates in $O(Tn^2)$ rounds using $O(\log n)$ bit storage and message overhead, where T is the maximum time it takes to transmit a message. They conjectured that without identifiers (IDs) flooding is impossible to solve within the above resources. Finally, a uniform routing algorithm was provided that delivers to the destination in $O(Tn)$ rounds using $O(\log n)$ bit storage and message overhead.

Computation under worst-case dynamicity was further and extensively studied in a series of works by Kuhn *et al.* in the synchronous case. In [KLO10], the network was assumed to be *T-interval connected* meaning that any time-window of length T has a static connected spanning subgraph (persisting throughout the window). Among others, *counting* (in which nodes must determine the size of the network) and *all-to-all token dissemination* (in which n different pieces of information, called tokens, are handed out to the n nodes of the network, each node being assigned one token, and all nodes must collect all n tokens) were solved in $O(n^2/T)$ rounds using $O(\log n)$ bits per message, almost-linear-time randomized approximate counting was established for $T = 1$, and two lower bounds on token dissemination were given. Several variants of *coordinated consensus* in 1-interval connected networks were studied in [KOM11]. [Hae11] is a recent work that presents information spreading algorithms in worst-case dynamic networks based on *network coding*. An *open* setting (modeled as high churn) in which nodes constantly join and leave has very recently been considered in [APRU12]. For an excellent introduction to distributed computation under worst-case dynamicity see [KO11]. Two very thorough surveys on dynamic networks are [Sch02, CFQS11].

Another notable model for dynamic distributed computing systems is the *population protocol* model [AAD+06]. In that model, the computational agents are passively mobile, interact in ordered pairs, and the connectivity assumption is a *strong global fairness condition* according to which all events that may always occur, occur infinitely often. These assumptions give rise to some sort of structureless interacting automata model. The usually assumed *anonymity* and *uniformity* (i.e. n is not known) of protocols only allow for commutative computations that eventualy stabilize to a desired configuration. Most computability issues in this area have now been established. Constant-state nodes on a complete interaction network (and several variations) compute the *semilinear predicates* [AAER07]. Semilinearity persists up to $o(\log \log n)$ local space but not more than this [CMN+11]. If constant-state nodes can additionally leave and update fixed-length pairwise marks then the computational power dramatically increases to the commutative subclass of $\mathbf{NSPACE}(n^2)$ [MCS11a]. For a very recent introductory text see [MCS11b].

3 Contribution

In this work, we study worst-case dynamic networks that are *free of any connectivity assumption about their instances*. Our dynamic network model is formally defined in Section 4.1. We only impose some *temporal connectivity* conditions

on the adversary guaranteeing that *another causal influence occurs within every time-window of some given length*, meaning that, in that time, another node first hears of the state that some node u had at some time t (see Section 4.2 for a formal definition of *causal influence*). Note that our temporal connectivity conditions are minimal assumptions that allow for bounded end-to-end communication in any dynamic network including those that have disconnected instances. Based on this basic idea, we define several novel generic metrics for capturing the speed of information spreading in a dynamic network. In particular, we define the *outgoing influence time* (oit) as the maximal time until the state of a node *influences* the state of another node, the *incoming influence time* (iit) as the maximal time until the state of a node *is influenced by* the state of another node, and the *connectivity time* (ct) as the maximal time until the two parts of any cut of the network become connected. These metrics are defined in Section 5, where also several results that correlate these metrics are presented.

In Section 5.1, we present a simple but very fundamental dynamic graph based on alternating matchings that has oit 1 (equal to that of instantaneous connectivity networks) but at the same time is *disconnected in every instance*. In Section 6, we exhibit another dynamic graph additionally guaranteeing that edges take maximal time to reappear. That graph is based on a geometric edge-coloring method due to Soifer for coloring a complete graph of even order n with $n-1$ colors [Soi09]. Similar results have appeared before but to the best of our knowledge only in probabilistic settings [CMM+08, BCF09].

In Section 7, we turn our attention to terminating computations and, in particular, we investigate termination criteria in networks in which an upper bound on the ct or the oit is known. By "termination criterion" we essentially mean any locally verifiable property that can be used to determine whether a node has heard from all other nodes. Note that we do not allow to the nodes any further knowledge on the network; for instance, nodes *do not* know the dynamic diameter of the network. In particular, in Section 7.1, we study the case in which an upper bound T on the ct is known and we present an optimal termination criterion that only needs time linear in the dynamic diameter and in T. Then, in Section 7.2, we study the case in which an upper bound K on the oit is known. We first present a termination criterion that needs time $O(K \cdot n^2)$. Additionally, we establish that even the optimal termination criterion for the ct case does not work in the oit case. These criteria share the fundamental property of hearing from the past. We then develop a new technique that gives an optimal termination criterion (time linear in the dynamic diameter and in K) by hearing from the future (by this we essentially mean that a node is interested for its outgoing influences instead for its incoming ones). Additionally, we exploit throughout the paper our termination criteria to provide protocols that solve the fundamental *counting* and *all-to-all token dissemination* (or *gossip*) problems; in the former nodes must determine the size of the network n and in the latter each node of the network is provided with a unique piece of information, called *token*, and all nodes must collect all n tokens. Then, we show that any protocol that is correct

in 1-interval connected networks can be adapted to work in networks in which an upper bound on the oit, the iit, or the ct is known.

Finally, in Section 8, we conclude and discuss some interesting future research directions.

4 Preliminaries

4.1 The Dynamic Network Model

A *dynamic network* is modeled by a *dynamic graph* $G = (V, E)$, where V is a set of n nodes (or processors) and $E : \mathbb{N} \to \mathcal{P}(E')$ (wherever we use $\mathbb{N}$ we mean $\mathbb{N}_{\geq 1}$) is a function mapping a round number $r \in \mathbb{N}$ to a set $E(r)$ of bidirectional links drawn from $E' = \{\{u, v\} : u, v \in V\}$. [1] Intuitively, a dynamic graph G is an infinite sequence $G(1), G(2), \ldots$ of *instantaneous graphs*, whose edge sets are subsets of E' chosen by a *worst-case adversary*. A *static network* is just a special case of a dynamic network in which $E(i + 1) = E(i)$ for all $i \in \mathbb{N}$. The set V is assumed throughout this work to be *static*, that is it remains the same throughout the execution.

We assume that nodes in V have unique identities (ids) drawn from some namespace $\mathcal{U}$ (we assume that ids are represented using $O(\log n)$ bits) and that they do not know the topology or the size of the network, apart from some minimal necessary knowledge to allow for terminating computations (usually an upper bound on the time it takes for information to make some sort of progress). Any such assumed knowledge will be clearly stated. Moreover, nodes have unlimited local storage (though they usually use a reasonable portion of it).

Communication is *synchronous message passing* [Lyn96, AW04], meaning that it is executed in discrete steps controlled by a global clock that is available to the nodes and that nodes communicate by sending and receiving messages (usually of length that is some reasonable function of n, like e.g. $\log n$). We use the terms *round*, *time*, and *step* interchangeably to refer to the discrete steps of the system. Naturally, real rounds begin to count from 1 (e.g. "first round") and we reserve time 0 to refer to the initial state of the system. We assume that the message transmission model is *anonymous broadcast*, in which, in every round r, each node u generates a single message $m_u(r)$ to be delivered to all its current neighbors in $N_u(r) = \{v : \{u, v\} \in E(r)\}$ without knowing $N_u(r)$.

In every round, the adversary first chooses the edges for the round; for this choice it can see the internal states of the nodes at the beginning of the round. At the same time and independently of the adversary's choice of edges each node generates its message for the current round. Note that a node does not have any information about the internal state of its neighbors when generating its messages. In deterministic algorithms, nodes are only based on their current internal state to generate their messages and this implies that the adversary can infer the messages that will be generated in the current round before choosing

[1] By $\mathcal{P}(S)$ we denote the *powerset* of the set S, that is the set of all subsets of S.

the edges. In this work, we only consider deterministic algorithms. Each message is then delivered to the sender's neighbors, as chosen by the adversary; the nodes transition to new states, and the next round begins.

4.2 Spread of Influence in Dynamic Graphs (Causal Influence)

Probably the most important notion associated with a dynamic network/graph is the *causal influence*, which formalizes the notion of one node "influencing" another through a chain of messages originating at the former node and ending at the latter (possibly going through other nodes in between). We denote by (u,t) the state of node u at time t and usually call it the *t-state of* u. The pair (u,t) is also called a *time-node*. We use $(u,r) \rightsquigarrow (v,r')$ to denote the fact that node u's state in round r influences node v's state in round r'. Formally:

Definition 1 ([Lam78]). *Given a dynamic graph* $G = (V,E)$ *we define an order* $\rightarrow \subseteq (V \times \mathbb{N}_{\geq 0})^2$*, where* $(u,r) \rightarrow (v,r+1)$ *iff* $u = v$ *or* $\{u,v\} \in E(r+1)$*. The* causal order $\rightsquigarrow \subseteq (V \times \mathbb{N}_{\geq 0})^2$ *is defined to be the reflexive and transitive closure of* $\rightarrow$*.*

Obviously, for a dynamic distributed system to operate as a whole there must exist some upper bound on the time needed for information to spread through the network. This is the weakest possible guarantee since without it global computation is impossible. An abstract way to talk about information spreading is via the notion of the *dynamic diameter*. The *dynamic diameter* (also called *flooding time*, e.g., in [CMM+08, BCF09]) of a dynamic graph, is an upper bound on the time required for each node to causally influence (or, equivalently, to be causally influenced by) every other node; formally, the dynamic diameter is the minimum $D \in \mathbb{N}$ s.t. for all times $t \geq 0$ and all $u,v \in V$ it holds that $(u,t) \rightsquigarrow (v,t+D)$. A small dynamic diameter allows for fast dissemination of information. In this work, we do not allow nodes to know the dynamic diameter of the network. We only allow some minimal knowledge (that will be explained in the sequel) based on which nodes may infer bounds on the dynamic diameter.

A class of dynamic graphs with small dynamic diameter is that of T*-interval connected* graphs. Formally, a dynamic graph $G = (V,E)$ is said to be T*-interval connected*, for $T \geq 1$, if, for all $r \in \mathbb{N}$, the static graph $G_{r,T} := (V, \bigcap_{i=r}^{r+T-1} E(r))$ is connected [KLO10]; that is, in every time-window of length T, a connected spanning subgraph is preserved.

Let us also define two very useful sets. We define by $\text{past}_{(u,t')}(t) := \{v \in V : (v,t) \rightsquigarrow (u,t')\}$ [KOM11] the *past set of a time-node* (u,t') *from time* t and by $\text{future}_{(u,t)}(t') := \{v \in V : (u,t) \rightsquigarrow (v,t')\}$ the *future set of a time-node* (u,t) *at time* t', for times $0 \leq t \leq t'$. In words, $\text{past}_{(u,t')}(t)$ is the set of nodes whose t-state (i.e. their state at time t) has causally influenced the t'-state of u and $\text{future}_{(u,t)}(t')$ is the set of nodes whose t'-state has been causally influenced by the t-state of u. If $v \in \text{future}_{(u,t)}(t')$ we say that at time t' node v *has heard of/from* the t-state of node u. If it happens that $t = 0$ we say simply that v has heard of u. Note that $v \in \text{past}_{(u,t')}(t)$ iff $u \in \text{future}_{(v,t)}(t')$.

For a distributed system to be able to perform global computation, nodes need to be able to determine for all times $0 \leq t \leq t'$ whether $\text{past}_{(u,t')}(t) = V$. If nodes know n, then a node can easily determine at time t' whether $\text{past}_{(u,t')}(t) = V$ by counting all different t-states that it has heard of so far. If it has heard the t-states of all nodes then the equality is satisfied. If n is not known then various techniques may be applied (which is the subject of this work). By *termination criterion* we mean any locally verifiable property that can be used to determine whether $\text{past}_{(u,t')}(t) = V$.

Remark 1. Note that any protocol that allows the nodes to determine whether $\text{past}_{(u,t')}(t) = V$ can be used to solve the counting and all-to-all token dissemination problems. The reason is that if a node knows at round r that it has been causally influenced by the initial states of all other nodes then it can solve counting by writing $|\text{past}_{(u,r)}(0)|$ on its output and all-to-all dissemination by writing $\text{past}_{(u,r)}(0)$ (provided that all nodes send their initial states and all nodes constantly broadcast all initial states that they have heard of so far).

5 Our Metrics

As already stated, in this work we aim to deal with dynamic networks that are allowed to have disconnected instances. To this end, we define some novel generic metrics that are particularly suitable for capturing the speed of information propagation in such networks.

5.1 The Influence Time

Recall that the guarantee on propagation of information resulting from instantaneous connectivity ensures that any time-node (u, t) influences another node *in each step* (if an uninfluenced one exists). From this fact, we extract two novel generic influence metrics that capture the maximal time until another influence (outgoing or incoming) of a time-node occurs.

We now formalize our first influence metric. We define the *outgoing influence time* (oit) as the minimum $k \in \mathbb{N}$ s.t. for all $u \in V$ and all times $t, t' \geq 0$ s.t. $t' \geq t$ it holds that

$$|\text{future}_{(u,t)}(t' + k)| \geq \min\{|\text{future}_{(u,t)}(t')| + 1, n\}.$$

Intuitively, the oit is the maximal time until the t-state of a node influences the state of another node (if an uninfluenced one exists) and captures the speed of information spreading.

Our second metric, the *incoming influence time* (iit), is similarly defined as the minimum $k \in \mathbb{N}$ s.t. for all $u \in V$ and all times $t, t' \geq 0$ s.t. $t' \geq t$ it holds that $|\text{past}_{(u,t'+k)}(t)| \geq \min\{|\text{past}_{(u,t')}(t)| + 1, n\}$.

We can now say that the oit of a T-interval connected graph is 1 and that the iit can be up to $n - 2$. However, is it necessary for a dynamic graph to be T-interval connected in order to achieve unit oit? First, let us make a simple but useful observation:

Proposition 1. *If a dynamic graph $G = (V, E)$ has* oit *(or* iit*) 1 then every instance has at least $\lceil n/2 \rceil$ edges.*

Proposition 1 is easily generalized as: if a dynamic graph $G = (V, E)$ has oit (or iit) k then for all times t it holds that $|\bigcup_{i=t}^{t+k-1} E(i)| \geq \lceil n/2 \rceil$. The reason is that now any node must have a neighbor in any k-window of the dynamic graph (and not necessarily in every round).

Now, inspired by Proposition 1, we define a minimal dynamic graph that at the same time satisfies oit 1 and always disconnected instances:

The Alternating Matchings Dynamic Graph. Take a ring of an even number of nodes $n = 2l$, partition the edges into 2 disjoint perfect matchings A and B (each consisting of l edges) and alternate round after round between the edge sets A and B.

Proposition 2. *The Alternating Matchings dynamic graph has* oit *1 and any node needs precisely $n/2$ rounds to influence all other nodes.*

In the alternating matchings construction any edge reappears every second step but not faster than this. We now formalize the notion of the *fastest edge reappearence* (fer) of a dynamic graph.

Definition 2. *The* fastest edge reappearence *(fer) of a dynamic graph $G = (V, E)$ is defined as the minimum $p \in \mathbb{N}$ s.t., $\exists e \in \{\{u, v\} : u, v \in V\}$ and $\exists t \in \mathbb{N}$, $e \in E(t) \cap E(t + p)$.*

Clearly, the fer of the alternating matchings dynamic graph described above is 2, because no edge ever reappears in 1 step and all and always reappear in 2 steps. In Section 6, by invoking a geometric edge-coloring method, we generalize this basic contruction to a more involved dynamic graph with oit 1, always disconnected instances, and fer equal to $n-1$. Note that the fer is always bounded from above by a function of n.

5.2 The Connectivity Time

We now propose another natural and practical metric for capturing the temporal connectivity of a possibly disconnected dynamic network that we call the *connectivity time* (ct).

Definition 3. *We define the* connectivity time *(ct) of a dynamic network $G = (V, E)$ as the minimum $k \in \mathbb{N}$ s.t. for all times $t \in \mathbb{N}$ the static graph $(V, \bigcup_{i=t}^{t+k-1} E(i))$ is connected.*

In words, the ct of a dynamic network is the maximal time of keeping the two parts of any cut of the network disconnected. That is to say, in every ct-window of the network an edge appears in every (V_1, V_2)-cut. Note that, in the extreme case in which the ct is 1, every instance of the dynamic graph is connected and

we thus obtain a 1-interval connected graph. On the other hand, greater ct allows for different cuts to be connected at different times in the ct-round interval and the resulting dynamic graph can very well have disconnected instances. For an illustrating example, consider again the alternating matchings graph from Section 5.1. Draw a line that crosses two edges belonging to matching A partitioning the ring into two parts. Clearly, these two parts communicate every second round (as they only communicate when matching A becomes available), thus the ct is 2 and every instance is disconnected. We now provide a result associating the ct of a dynamic graph with its oit.

Proposition 3. *(i) oit $\leq$ ct but (ii) there is a dynamic graph with oit 1 and ct $= \Omega(n)$.*

Proof. (i) We show that for all $u \in V$ and all times $t, t' \in \mathbb{N}$ s.t. $t' \geq t$ it holds that $|\text{future}_{(u,t)}(t'+\mathsf{ct})| \geq \min\{|\text{future}_{(u,t)}(t')|+1, n\}$. Assume $V \backslash \text{future}_{(u,t)}(t') \neq \emptyset$ (as the other case is trivial). In at most ct rounds at least one edge joins $\text{future}_{(u,t)}(t')$ to $V \backslash \text{future}_{(u,t)}(t')$. Thus, in at most ct rounds $\text{future}_{(u,t)}(t')$ increases by at least one.

(ii) Recall the alternating matchings on a ring dynamic graph from Section 5.1. Now take any set V of a number of nodes that is a multiple of 4 (this is just for simplicity and is not necessary) and partition it into two sets V_1, V_2 s.t. $|V_1| = |V_2| = n/2$. If each part is an alternating matchings graph for $|V_1|/2$ rounds then every u say in V_1 influences 2 new nodes in each round and similarly for V_2. Clearly we can keep V_1 disconnected from V_2 for $n/4$ rounds without violating oit $= 1$. □

6 Fast Propagation of Information under Continuous Disconnectivity

In Section 5.1, we presented a simple example of an always-disconnected dynamic graph, namely, the alternating matchings dynamic graph, with optimal oit (i.e. unit oit). Note that the alternating matchings dynamic graph may be conceived as simple as it has small fer (equal to 2). We pose now, and answer to the positive, an interesting question: Is there an always-disconnected dynamic graph with unit oit and fer as big as $n - 1$?

Let us define a very useful dynamic graph coming from the area of edge-coloring.

Definition 4. *We define the following dynamic graph S based on an edge-coloring method due to Soifer [Soi09]: $V(S) = \{u_1, u_2, \ldots, u_n\}$ where $n = 2l$, $l \geq 2$. Place u_n on the center and $u_1, \ldots, n_{n-1}$ on the vertices of a $(n-1)$-sided polygon. For each time $t \geq 1$ make available only the edges $\{u_n, u_{m(0)}\}$ for $m(j) := (t-1+j \bmod n-1)+1$ and $\{u_{m(-i)}, u_{m(i)}\}$ for $i = 1, \ldots, n/2-1$; that is make available one edge joining the center to a polygon-vertex and all edges perpendicular to it.*

Theorem 1. *For all $n = 2l$, $l \geq 2$, there is a dynamic graph of order n, with* oit *equal to 1,* fer *equal to $n-1$, and in which every instance is a perfect matching. This is Soifer's graph.*

Note that Theorem 1 is optimal w.r.t. fer as *it is impossible to achieve at the same time unit* oit *and* fer *strictly greater than $n-1$.* To see this, notice that if no edge is allowed to reappear in less than n steps then any node must have no neighbors once every n steps.

7 Termination and Computation

We now turn our attention to termination criteria that we exploit to solve the fundamental counting and all-to-all token dissemination problems. Keep in mind that nodes have no *a priori* knowledge of the size of the network.

7.1 Nodes Know an Upper Bound on the ct: An Optimal Termination Criterion

We here assume that all nodes know some upper bound T on the ct. We will give an optimal condition that allows a node to determine whether it has heard from all nodes in the graph. This condition results in an algorithm for counting and all-to-all token dissemination which is optimal, requiring $O(D+T)$ rounds in any dynamic network with dynamic diameter D. The core idea is to have each node keep track of its past sets from time 0 and from time T and terminate as long as these two sets become equal. This technique is inspired from [KOM11], where a comparison between the past sets from time 0 and time 1 was used to obtain an optimal termination criterion in 1-interval connected networks.

Theorem 2 (Repeated Past). *Node u knows at time t that* $\text{past}_{(u,t)}(0) = V$ *iff* $\text{past}_{(u,t)}(0) = \text{past}_{(u,t)}(T)$.

Proof. If $\text{past}_{(u,t)}(0) = \text{past}_{(u,t)}(T)$ then we have that $\text{past}_{(u,t)}(T) = V$. The reason is that $|\text{past}_{(u,t)}(\ 0)| \geq \min\{|\text{past}_{(u,t)}(T)| + 1, n\}$. To see this, assume that $V \backslash \text{past}_{(u,t)}(T) \neq \emptyset$. At most by round T there is some edge joining some $w \in V \backslash \text{past}_{(u,t)}(T)$ to some $v \in \text{past}_{(u,t)}(T)$. Thus, $(w,0) \rightsquigarrow (v,T) \rightsquigarrow (u,t) \Rightarrow w \in \text{past}_{(u,t)}(0)$. In words, all nodes in $\text{past}_{(u,t)}(T)$ belong to $\text{past}_{(u,t)}(0)$ and at least one node not in $\text{past}_{(u,t)}(T)$ (if one exists) must belong to $\text{past}_{(u,t)}(0)$.

For the other direction, assume that there exists $v \in \text{past}_{(u,t)}(0) \backslash \text{past}_{(u,t)}(T)$. This does not imply that $\text{past}_{(u,t)}(0) \neq V$ but it does imply that even if $\text{past}_{(u,t)}(0) = V$ node u cannot know it has heard from everyone. Note that u heard from v at some time $T' < T$ but has not heard from v since then. It can be the case that arbitrarily many nodes were connected to no node until time $T-1$ and from time T onwards were connected only to node v (v in some sense conceals these nodes from u). As u has not heard from the T-state of v it can be the case that it has not heard at all from arbitrarily many nodes, thus it cannot decide on the count. □

We now give a time-optimal $O(D+T)$-round algorithm for counting and all-to-all token dissemination that is based on Theorem 2.

Protocol *A*. All nodes constantly forward all 0-states and T-states of nodes that they have heard of so far (the ids of the nodes accompanied with 0 and T timestamps, respectively) and a node halts as soon as $\text{past}_{(u,t)}(0) = \text{past}_{(u,t)}(T)$ and outputs $|\text{past}_{(u,t)}(0)|$ for counting or $\text{past}_{(u,t)}(0)$ for all-to-all dissemination.

7.2 Known Upper Bound on the oit: Another Optimal Termination Criterion

Now we assume that all nodes know some upper bound K on the oit.

7.2.1 Inefficiency of Hearing the Past

We begin by proving that if a node u has at some point heard of l nodes, then u hears of another node in $O(Kl^2)$ rounds (if an unknown one exists).

Theorem 3. *In any given dynamic graph with* oit *upper bounded by* K*, take a node* u *and a time* t *and denote* $|\text{past}_{(u,t)}(0)|$ *by* l*. It holds that* $|\{v : (v,0) \rightsquigarrow (u, t + Kl(l+1)/2)\}| \geq \min\{l+1, n\}$.

Proof. Consider a node u and a time t and define $A_u(t) := \text{past}_{(u,t)}(0)$ (we only prove it for the initial states of nodes but easily generalizes to any time), $I_u(t') := \{v \in A_u(t) : A_v(t') \backslash A_u(t) \neq \emptyset\}$, $t' \geq t$, that is $I_u(t')$ contains all nodes in $A_u(t)$ whose t'-states have been influence by nodes not in $A_u(t)$ (these nodes know new info for u), $B_u(t') := A_u(t) \backslash I_u(t')$, that is all nodes in $A_u(t)$ that do not know new info, and $l := |A_u(t)|$. The only interesting case is for $V \backslash A_u(t) \neq \emptyset$. Since the oit is at most K we have that at most by round $t + Kl$, (u,t) influences some node in $V \backslash B_u(t)$ say via some $u_2 \in B_u(t)$. By that time, u_2 leaves B_u. Next consider $(u, t + Kl + 1)$. In $K(l-1)$ steps it must influence some node in $V \backslash B_u$ since now u_2 is not in B_u. Thus, at most by round $t + Kl + K(l-1)$ another node, say e.g. u_3, leaves B_u. In general, it holds that $B_u(t' + K|B_u(t')|) \leq \max\{|B_u(t')| - 1, 0\}$. It is not hard to see that at most by round $j = t + K(\sum_{1 \leq i \leq l} i)$, B_u becomes empty, which by definition implies that u has been influenced by the initial state of a new node. In summary, u is influenced by another initial state in at most $K(\sum_{1 \leq i \leq l} i) = kl(l+1)/2$ steps. □

The good thing about the upper bound of Theorem 3 is that it associates the time for a new incoming influence to arrive at a node only with an upper bound on the oit, which is known, and the number of existing incoming influences which is also known, and thus the bound is locally computable at any time. So, there is a straightforward translation of this bound to a termination criterion and further to an $O(Kn^2)$ algorithm for counting and all-to-all dissemination.

Note that the upper bound of Theorem 3 is loose. The reason is that if a dynamic graph has oit upper bounded by K then in $O(Kn)$ rounds all nodes have causally influenced all other nodes and clearly the iit can be at most $O(Kn)$.

In fact, it is not hard to construct a dynamic graph that achieves this worst possible gap between the iit and the oit. On the other hand, the bound of Theorem 3 is optimal in the following sense: a node cannot obtain a better upper bound based solely on K and l.

We now show that even the criterion of Theorem 2, that is optimal if an upper bound on the ct is known, does not work in dynamic graphs with known an upper bound K on the oit. In particular, we show that for any time $t' \in \mathbb{N}$ which can only depend on K (otherwise it is fixed) there is a dynamic graph with oit upper bounded by K, a node u, and a time $t \in \mathbb{N}$ s.t. $\text{past}_{(u,t)}(0) = \text{past}_{(u,t)}(t')$ while $\text{past}_{(u,t)}(0) \neq V$. In words, for any such t' it can be the case that while u has not been yet causally influenced by all initial states its past set from time 0 may become equal to its past set from time t', which violates the termination criterion of Theorem 2.

Theorem 4. *For any time t' (which can only depend on the upper bound K on the oit) there is a dynamic graph with oit upper bounded by K, a node u, and a time $t \in \mathbb{N}$ s.t. $\text{past}_{(u,t)}(0) = \text{past}_{(u,t)}(t')$ while $\text{past}_{(u,t)}(0) \neq V$.*

Proof. Let n be sufficiently large, that is $n \gg t'$, and for simplicity assume that n is a multiple of 4. As in Proposition 3-ii, we can keep two parts V_1, V_2 of the network, of size $n/2$ each, disconnected up to some time $\Omega(n)$. Let $u \in V_1$. At time $t'+1$ the adversary directly connects some node $v \in V_1$ to all $w \in V_1$. Now v knows the t'-states (and of course also the 0-states) of all nodes in V_1. Then at time $t'+2$ the adversary connects v only to u and to some node in V_2. Clearly, at time $t'+2$, u learns the t'-states of all nodes in V_1 (v inclusive) and it holds that $\text{past}_{(u,t'+2)}(0) = \text{past}_{(u,t'+2)}(t')$. Additionally, $|\text{past}_{(u,t)}(0)| = n/2 \Rightarrow \text{past}_{(u,t)}(0) \neq V$. □

7.2.2 Hearing the Future

We now present an optimal protocol for counting and all-to-all dissemination in dynamic networks with known an upper bound K on the oit, that is based on the following termination criterion. By definition of oit, if $\text{future}_{(u,0)}(t) = \text{future}_{(u,0)}(t+K)$ then $\text{future}_{(u,0)}(t) = V$. The reason is that if there exist uninfluenced nodes, then at least one such node must be influenced in at most K rounds, otherwise no such node exists and $(u,0)$ must have already influenced all nodes. So, a fundamental goal is to allow a node to know its future set. Note that this criterion has a very basic difference from all termination criteria that have so far been applied to worst-case dynamic networks: instead of keeping track of its past set(s) and waiting for new incoming influences a node now directly keeps track of its future set and is informed by other nodes of its progress. We assume, for simplicity, a unique leader l in the initial configuration of the system (we later drop this unnecessary assumption).

Protocol *Hear_from_known*. We denote by r the current round. Each node u keeps a list $Infl_u$ in which it keeps track of all nodes that first heard of $(l,0)$ (the initial state of the leader) by u (u was between those nodes that first acquained

$(l,0)$ to nodes in $Infl_u$), a set A_u in which it keeps track of the $Infl_v$ sets that it is aware of initially set to $(u, Infl_u, 1)$, and a variable *timestamp* initially set to 1. Each node u broadcast in every round (u, A_u) and if it has heard of $(l,0)$ also broadcasts $(l,0)$. Upon reception of an id w that is not accompanied with $(l,0)$, a node u that has already heard of $(l,0)$ adds (w,r) to $Infl_u$ to recall that at round r it notified w of $(l,0)$ (note that it is possible that other nodes also notify w of $(l,0)$ at the same time without u being aware of them; all these nodes will write (w,r) in their lists). If it ever holds at a node u that $r > \max_{(v\neq u,r')\in Infl_u}\{r'\}+K$ then u adds (u,r) in $Infl_u$ (replacing any existing $(u,t) \in Infl_u$) to denote the fact that r is the maximum known time until which u has performed no further propagations of $(l,0)$. If at some round r a node u modifies its $Infl_u$ set, it sets $timestamp \leftarrow r$. In every round, a node u updates A_u by storing in it the most recent $(v, Infl_v, timestamp)$ triple of each node v that it has heard of so far (its own $(u, Infl_u, timestamp)$ inclusive), where the "most recent" triple of a node v is the one with the greatest *timestamp* between those whose first component is v. Moreover, u clears multiple (w,r) records from the $Infl_v$ lists of A_u. In particular, it keeps (w,r) only in the $Infl_v$ list of the node v with the smallest id between those that share (w,r). Similarly, the leader collects all $(v, Infl_v, timestamp)$ triples in its own A_l set. Let *tmax* denote the maximum timestamp appearing in A_l, that is the maximum time for which the leader knows that some node was influenced by $(l,0)$ at that time. Moreover denote by I the set of nodes that the leader knows to have been influenced by $(l,0)$ [2]. If at some round r it holds at the leader that for all $u \in I$ there is a $(u, Infl_u, timestamp) \in A_l$ s.t. $timestamp \geq tmax + K$ and $\max_{(w\neq u,r')\in Infl_u}\{r'\} \leq tmax$ then the leader notifies the other nodes about termination for $K \cdot |I|$ rounds and then outputs $|I|$ or I depending on whether counting or all-to-all dissemination needs to be solved and halts.

The above protocol can be easily made to work without the assumption of a unique leader. The idea is to have all nodes begin as leaders and make all nodes prefer the leader with the smallest id that they have heard of so far. In particular, we can have each node keep an $Infl_{(u,v)}$ only for the smallest v that it has heard of so far. Clearly, in $O(D)$ rounds all nodes will have sticked to the node with the smallest id in the network.

Theorem 5. *Protocol* Hear_from_known *solves counting and all-to-all dissemination in $O(D+K)$ rounds by using messages of size $O(n \log Kn)$, in any dynamic network with dynamic diameter D, and with* oit *upper bounded by some K known to the nodes.*

We defer for the full paper a protocol (inspired from a technique from [MCS12]) that solves counting and all-to-all dissemination in $O(Dn^2+K)$ rounds by using messages of size $O(\log D+\log n)$, in any dynamic network with dynamic diameter D, and with oit upper bounded by some K known to the nodes.

[2] Note that I can be extracted from A_l by $I = \{v \in V : \exists u \in V, \exists timestamp, r \in \mathbb{N}$ s.t. $(u, Infl_u, timestamp) \in A_l$ and $(v,r) \in Infl_u\}$.

Finally, it is not hard to prove that protocols that are correct in 1-interval connected networks carry over to networks in which an upper bound on the oit, iit, or ct is known, with only a small delay being introduced in the process.

8 Conclusions

We studied for the first time worst-case dynamic networks that are free of any connectivity assumption about their instances. To enable a quantitative study we proposed some novel generic metrics that capture the speed of information propagation in a dynamic network. We proved that fast dissemination and computation are possible even under continuous disconnectivity. In particular, we presented optimal termination conditions and protocols based on them for the fundamental counting and all-to-all token dissemination problems.

There are many open problems and promising research directions related to this work. An asynchronous communication model in which nodes can broadcast when there are new neighbors would be a very natural extension of the synchronous model that we studied in this work. Note that in our work (and all previous work on the subject) information dissemination is only guaranteed under continuous broadcasting. How can the number of redundant transmissions be reduced in order to improve communication efficiency? Is there a way to exploit visibility to this end? Does predictability help? Finally, randomization will be certainly valuable in constructing fast and symmetry-free protocols. We strongly believe that these and other known open questions and research directions will motivate the further growth of this emerging field.

References

[AAD+06] Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing, 235–253 (March 2006)

[AAER07] Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)

[AKL08] Avin, C., Koucký, M., Lotker, Z.: How to Explore a Fast-Changing World (Cover Time of a Simple Random Walk on Evolving Graphs). In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 121–132. Springer, Heidelberg (2008)

[APRU12] Augustine, J., Pandurangan, G., Robinson, P., Upfal, E.: Towards robust and efficient computation in dynamic peer-to-peer networks. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 551–569. SIAM (2012)

[AW04] Attiya, H., Welch, J.: Distributed computing: fundamentals, simulations, and advanced topics, vol. 19. Wiley-Interscience (2004)

[BCF09] Baumann, H., Crescenzi, P., Fraigniaud, P.: Parsimonious flooding in dynamic graphs. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC 2009, pp. 260–269. ACM (2009)

[Bol98] Bollobás, B.: Modern Graph Theory, corrected edn. Springer (July 1998)
[CFQS11] Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-Varying Graphs and Dynamic Networks. In: Frey, H., Li, X., Ruehrup, S. (eds.) ADHOC-NOW 2011. LNCS, vol. 6811, pp. 346–359. Springer, Heidelberg (2011)
[CMM+08] Clementi, A.E., Macci, C., Monti, A., Pasquale, F., Silvestri, R.: Flooding time in edge-markovian dynamic graphs. In: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, PODC 2008, pp. 213–222. ACM, New York (2008)
[CMN+11] Chatzigiannakis, I., Michail, O., Nikolaou, S., Pavlogiannis, A., Spirakis, P.G.: Passively mobile communicating machines that use restricted space. Theor. Comput. Sci. 412(46), 6469–6483 (2011)
[Dol00] Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
[Hae11] Haeupler, B.: Analyzing network coding gossip made easy. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing, pp. 293–302. ACM (2011)
[KKK00] Kempe, D., Kleinberg, J., Kumar, A.: Connectivity and inference problems for temporal networks. In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, pp. 504–513 (2000)
[KLO10] Kuhn, F., Lynch, N., Oshman, R.: Distributed computation in dynamic networks. In: Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, pp. 513–522. ACM, New York (2010)
[KO11] Kuhn, F., Oshman, R.: Dynamic networks: models and algorithms. SIGACT News 42, 82–96 (2011); Keidar, I. (ed): Distributed Computing Column
[KOM11] Kuhn, F., Oshman, R., Moses, Y.: Coordinated consensus in dynamic networks. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 1–10 (2011)
[Lam78] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
[Lyn96] Lynch, N.A.: Distributed Algorithms, 1st edn. Morgan Kaufmann (1996)
[MCS11a] Michail, O., Chatzigiannakis, I., Spirakis, P.G.: Mediated population protocols. Theor. Comput. Sci. 412(22), 2434–2450 (2011)
[MCS11b] Michail, O., Chatzigiannakis, I., Spirakis, P.G.: New Models for Population Protocols. In: Lynch, N.A. (ed.) Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool (2011)
[MCS12] Michail, O., Chatzigiannakis, I., Spirakis, P.G.: Brief Announcement: Naming and Counting in Anonymous Unknown Dynamic Networks. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 437–438. Springer, Heidelberg (2012)
[OW05] O'Dell, R., Wattenhofer, R.: Information dissemination in highly dynamic graphs. In: Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing, DIALM-POMC 2005, pp. 104–110 (2005)
[Sch02] Scheideler, C.: Models and Techniques for Communication in Dynamic Networks. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 27–49. Springer, Heidelberg (2002)
[Soi09] Soifer, A.: The Mathematical Coloring Book: Mathematics of Coloring and the Colorful Life of its Creators, 1st edn. Springer (2009)

Wait-Free Stabilizing Dining Using Regular Registers*

Srikanth Sastry[1,**], Jennifer L. Welch[2,***], and Josef Widder[3,†]

[1] CSAIL, MIT
Cambridge, MA - 02139, USA
[2] Texas A&M University
College Station, TX - 77843, USA
[3] Technische Universität Wien
Vienna, Austria

Abstract. Dining philosophers is a scheduling paradigm that determines when processes in a distributed system should execute certain sections of their code so that processes do not execute 'conflicting' code sections concurrently, for some application-dependent notion of a 'conflict'. Designing a stabilizing dining algorithm for shared-memory systems subject to process crashes presents an interesting challenge: classic stabilization relies on *all* processes continuing to execute actions forever, an assumption which is violated when crash failures are considered. We present a dining algorithm that is both wait-free (tolerates any number of crashes) and is pseudo-stabilizing. Our algorithm works in an asynchronous system in which processes communicate via shared regular registers and have access to the eventually perfect failure detector $\Diamond\mathcal{P}$. Furthermore, with a stronger failure detector, the solution becomes wait-free and self-stabilizing. To our knowledge, this is the first such algorithm. Prior results show that $\Diamond\mathcal{P}$ is necessary for wait-freedom.

1 Introduction

In shared-memory distributed systems, the code for a distributed application at each process is a sequence of actions, certain sections of which — called *critical sections* — are designated as needing to be executed indivisibly with respect to the critical sections of application modules at certain other processes. Actions at different processes might conflict, for instance, because they access a

* We would like to thank the reviewers for their suggestions in improving the paper.
** This work is supported in part by NSF Award Numbers CCF-0726514, CCF-0937274, and CNS-1035199, and AFOSR Award Number FA9550-08-1-0159. This work is also partially supported by Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.
*** Supported in part by NSF grant 0964696.
† Supported in part by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF), and by the Vienna Science and Technology Fund (WWTF) grant PROSEED.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 284–299, 2012.

shared resource or because the relative order of their execution results in a race condition. Synchronizing such actions of distributed applications, that need to occur without interference, is often delegated to scheduler that is implemented as a solution to the dining philosophers problem (or *dining*, for short).

Dining is a scheduling paradigm in which critical-section actions at each process may be in conflict with a static subset of processes in the system, and a solution to the dining philosophers problem ensures that whenever a process is executing its critical-section actions, no other conflicting process is executing its respective critical-section actions.

Solutions to dining are well understood in 'fault-free' systems, in which all the components behave according to their respective specification. However, in the presence of faults — deviations of any component from its specification — designing dining algorithms becomes challenging. In this paper, we focus on solving dining in the presence of two distinct fault classes: transient faults and crash faults. A *transient fault* occurs when the state of the system is corrupted to an arbitrary state, and a *crash fault* occurs when a process ceases execution without warning and never recovers.

Often, recovery from transient faults is achieved through *stabilization* [13], which is a property that guarantees that from any arbitrary state, the system is guaranteed to converge to a 'safe' state and operate henceforth in accordance with its specification. A classic assumption for correctness in stabilizing systems is that all processes continue executing actions to recover from an arbitrary state. However, a crashed process ceases executing actions. This makes the intersection of the two fault classes an interesting avenue for research.

Contribution. While existing solutions to dining are either stabilizing [1,2,5,23] or wait-free (tolerate an arbitrary number of crashes) [28,30,9], to our knowledge, there are none that are both. In this paper, we propose a distributed dining algorithm that is pseudo-stabilizing and wait-free. The algorithm assumes the existence of multi-reader single-writer regular registers and the eventually perfect failure detector $\Diamond\mathcal{P}$. We remark that the above assumptions are modest or even necessary: wait-free multi-reader single-writer regular register can be constructed from wait-free dual-reader single-writer safe bits [17], and $\Diamond\mathcal{P}$ is necessary for wait-free dining [29] in asynchronous systems. We see that if the algorithm accesses the perfect failure detector, then it becomes wait-free and self-stabilizing.

Organization. Background and related work is presented in Sect. 2. Section 3 describes the system model. Section 4 specifies two problems: mutual exclusion and dining philosophers. Section 5 describes a wait-free pseudo-stabilizing mutual exclusion algorithm, which is then used in the wait-free pseudo-stabilizing dining algorithm described in Sect. 6. We conclude in Sect. 7.

2 Background and Related Work

Designing dining algorithms that are stabilizing or crash-tolerant, especially wait-free, or both, has been an active area of research.

Crash Tolerant and Stabilizing Dining. There has been some prior investigations on constructing stabilizing dining algorithms that tolerate process crashes. For instance, [24] provides a stabilizing dining algorithm that tolerates (malicious) crashes, and in [26], the stabilizing dining algorithm tolerates byzantine processes. However, in these algorithms, some correct processes could stall even if one process crashes. In contrast, we focus on the stronger property of wait-freedom which guarantees that all correct processes continue taking steps despite arbitrary process crashes.

Self-stabilizing Dining. There has been a significant body of work [23,1,5,25] which investigates self-stabilizing dining; however, these algorithms are not wait-free. The algorithm presented in [2], in addition to being self-stabilizing, ensures k-fairness[1]. However, the aforementioned algorithms assume that the low-level shared-memory registers in the system satisfy read/write atomicity. In contrast, we assume that we have regular registers, which are weaker than atomic registers. Furthermore, in all of the above algorithms, if some process crashes, other processes are not guaranteed to continue taking actions; in other words, these implementations are not fault-tolerant.

Crash-Tolerant Dining. There has been a lot of work in designing crash-tolerant dining algorithms. However, in asynchronous systems, it is impossible to design dining algorithms in which neighbors and neighbors' neighbors of a crashed process do not stall [7]; in other words, asynchronous systems cannot guarantee a crash locality of less than 2. Achieving a smaller crash locality requires some recourse to crash detection. Subsequently, [27] showed that with access to sufficient crash detection ability, the crash locality of dining algorithms can be reduced to 1. The results in [27] employed the eventually perfect failure detector ($\Diamond\mathcal{P}$)[2] [6].

Later $\Diamond\mathcal{P}$ is shown in [28,30] to be sufficient to achieve wait-freedom at the expense of a weaker exclusion guarantee: $\Diamond\mathcal{P}$-based dining algorithms may schedule conflicting actions concurrently, but only finitely many times; that is, exclusion is only *eventual*. The results in [29] demonstrate the necessity of the above trade-off.

Failure Detectors. Intuitively, failure detectors [6] are oracles that may provide hints about process crashes. Since, our goal is to implement wait-free and stabi-

[1] A k-fair scheduler guarantees the following. For any process i, between any two consecutive accesses to its respective critical section by i, no other process enters its own respective critical section more than k times.

[2] Briefly, the eventually perfect failure detector, or $\Diamond\mathcal{P}$ for short, may provide arbitrary information to processes for some arbitrary, but finite, duration; however, eventually (after some potentially unknown time), it provides perfect information about process crashes.

lizing schedulers, the failure detectors that we employ must also be stabilizing. Self-stabilizing implementations of failure detectors have been an area of active research. In message passing systems, one of the first self-stabilizing implementations are for the perfect failure detector proposed in [21,20], but only when at most one process crashes. Later, self-stabilizing algorithms for failure detectors in the presence of multiple process crashes was proposed in [3] for systems that have local clocks and have bounds on the relative messages delays. Subsequently, [18] proposed time-free self-stabilizing implementations of failure detectors in a similar system as [3], but where processes do not have local clocks. More recently, in [8], Delporte-Gallet et al. propose a self-stabilizing implementation of the Ω failure detector[3].

For shared memory systems, Dolev et al. [15] propose a self-stabilizing implementation of the Ω failure detector in systems with unknown bounds on relative process speeds. In [16], Fischer et al. implement and use a variant Ω called 'Ω?', which eventually determines whether not there is a leader, to solve self-stabilizing leader election in anonymous systems.

3 System Model

The system consists of a set of processes and a set of shared single-writer multi-reader read/write regular [19] registers.

Processes. The system contains the set Π of n processes where each process is a state machine. Each process has a unique incorruptible ID from the set $\{0, \ldots, n-1\}$ and is known to all the processes in the system. Since a process ID uniquely determines a process, in the remainder of this paper, we refer to a process and its ID interchangeably.

States and Private Variables. The state of the system (or, *system state*) is determined by the state of each process and the state of each shared variable in the system. In turn, the state of each process is determined by the set of *private variables* at that process. A private variable at a process i accessible only to process i and no other process may read or write to that variable. The states of shared variables are discussed later.

Steps. The read and write operations of shared variables and the state changes of processes are modeled by *steps*, which are of four types: invocation, response, transition, and crash. The invocation and response steps are discussed when describing shared variables. Transition steps are discussed when describing actions. Crash steps are discussed when describing faults.

Shared Variables. The system contains a set of *shared variables* that are *regular* single-writer multi-reader registers. Each shared variable is a state machine that interacts with the processes through read and write operations. Each operation

[3] Briefly, the Ω failure detector outputs a process ID at each process infinitely often. Eventually, after some potentially unknown time, and forever thereafter, Ω is guaranteed to output the ID of some unique correct processes.

consists of two steps: *invocation* by the process and *response* by the shared variable. Each shared variable may be read by any processes in the system and is *owned* by some unique fixed process. Only the owner of a shared variable may write to it.

Actions. The state change at each process is determined by a set of *actions*. Actions are guarded commands [12] which are of the form "$\{guard\} \rightarrow command$"; *guard* is a predicate on the state of the process, and *command* is a representation of at most one shared memory operation followed by the new state of the process.

Precisely, each command is an ordered tuple that consists of either a read or write operation and a transition step, or just a transition step. Recall that a read (or write) operation consists of an invocation step followed by a response step. In a *transition step*, a process may change its local state (by modifying the value of private variables) based on the current state and value returned by shared-memory operation in that action (if applicable).

Given an action $A \equiv \{guard\} \rightarrow command$ at a process i, in every state s of the system in which *guard* is *true*, the action a is said to be *enabled* in s at i. The algorithm that a process follows is described as a set of actions called *program actions*.

Tasks are a partitioning of program actions at each process; a task t is simply a set of program actions at a process. Each program action belongs to exactly one task. The notion of tasks is useful in describing fair executions.

Faults. Processes are prone to *crash faults*. When a crash fault occurs at process i, the process ceases taking steps without warning and never recovers. In effect, when a crash fault occurs at a process i, it disables all the program actions at i. A crash fault at each process i is modeled as an explicit action that consists of a single *crash* step, but is assumed to not be a program action, and therefore, it is not in any task. The crash fault action for each process i is continuously enabled until it is executed; however, the action (for each process i) occurs at most once. It is admissible for a process to never crash; that is, it is admissible for a crash fault action to never occur.

Eventually Perfect Failure Detector. We assume that the system is augmented with a self-stabilizing implementation of the *eventually perfect failure detector*, or $\Diamond\mathcal{P}$, for short. Informally, $\Diamond\mathcal{P}$ provides information about crash faults to all the processes in the system in the form of a *suspect list* which contains a set of processes; eventually, $\Diamond\mathcal{P}$ never suspects correct processes and always suspects crashed processes. More precisely, we assume that each process i has a private variable $\Diamond\mathcal{P}_i$ that contains a set of process IDs. $\Diamond\mathcal{P}$ is specified by a set of actions, one action a_i for each process i, where a_i writes a value to $\Diamond\mathcal{P}_i$ continually such that, eventually and permanently, $\Diamond\mathcal{P}_i$ contains exactly the set of IDs of crashed processes.

Although we do not provide an explicit self-stabilizing implementation of $\Diamond\mathcal{P}$ here, we remark that there are many existing self-stabilizing $\Diamond\mathcal{P}$ and Ω implementations (discussed in Sect. 2) that may be modified appropriately and employed

here. The choice of the implementation depends on the partial synchrony satisfied by the underlying distributed system and is beyond the scope of this paper.

Executions. An execution describes the state evolution of a system as a (potentially infinite) sequence $\alpha = S_0, a_1, S_1, a_2, \ldots$ of alternating system states and steps such that the following properties are satisfied. An execution may start from any system state.

1. For each process i, the subsequence α_i of α that consists of all the steps at process i can be partitioned into a sequence of actions $A_1, A_2, A_3, \ldots$. Let $s_{i.0}$ be the state of process i in S_0. There exists a sequence $s_{i.1}, s_{i.2}, \ldots$ of states of process i such that for every positive natural number x, action A_x is enabled in state $s_{i.x-1}$ and applying A_x causes i to transition to state $s_{i.x}$.
2. In α, for each process i, no step at i follows a crash step at i.
3. For every shared register r, the sequence of invocation and response steps in α for r satisfies the regularity property [19]. That is, every read operation returns either the value written by latest preceding write or the value being written by an overlapping write (if applicable).
4. For each correct process i, the sequence of values of $\Diamond\mathcal{P}_i$ during the execution satisfies the properties of $\Diamond\mathcal{P}$: eventually and permanently, $\Diamond\mathcal{P}_i$ contains exactly the set of IDs of faulty processes.

An execution α is said to be a *fair execution* if it satisfies the following properties. (1) If α is a finite execution, then no program action is enabled in the final state of the system after executing α. (2) If α is an infinite execution, then for each task t, either some enabled program action in t occurs infinitely often in α, or in infinitely many states of α, no program action in t is enabled. Note that crash actions are not in any task and, therefore, need not occur 'fairly'.

In any execution α, the set of processes at which a crash fault occurs is said to be *faulty* in α, and all the other processes are said to be *correct* in α. Each process is said to be *live* until it crashes.

4 Dining and Mutual Exclusion

In this section, we describe the two problems which are the primary focus of this article: *dining philosophers* and *mutual exclusion*.

Dining. The *dining philosophers problem* [22] is a scheduling problem that is represented by an undirected graph $G = (\Pi, E)$, called a *conflict graph*, where the set Π of processes (called *diners*) denotes the set of vertices of the conflict graph. The neighbors of each process i in G are denoted $N(i)$. Each process is assumed to know the conflict graph G. The state space of each process is partitioned into four sets: *thinking*, *hungry*, *eating*, and *exiting*. A solution to dining philosophers determines when a diner can transition from a hungry state to an eating state, and from an exiting state to a thinking state. In order to specify the dining philosopher's problem, we assume that the diners satisfy a set of 'well-formedness' properties defined next.

A diner is said to be *well-formed* iff (1) a diner becomes hungry only when thinking, but may remain thinking forever, (2) a diner remains hungry until it starts eating, and (3) a correct eating diner eventually exits.

An execution of the system which satisfies the well-formedness conditions is said to be a *well-formed execution*.

A solution to dining philosophers is an algorithm $\mathcal{A}$ that satisfies the following properties. (1) There exists a non-empty set of states, called *start states* in which all the diners are thinking. (2) Furthermore, in every fair well-formed execution that starts from a start state, the following properties are satisfied.

(a) Mutual exclusion: For each diner i, while i is live and eating, no live process in $N(i)$ is eating.
(b) Eventual Exit: If a correct diner is exiting, then eventually that diner is thinking.
(c) Fairness: If some correct diner is hungry, then eventually that diner is eating.

Fix such an algorithm $\mathcal{A}$. Let $\mathcal{E}_s$ be the set of all fair well-formed executions that start from some start state. Let Q_{safe} be the set of states that occur in any execution in $\mathcal{E}_s$. The set Q_{safe} is said to be the set of *safe states* of $\mathcal{A}$.

A dining philosophers algorithm $\mathcal{A}$ is said to be pseudo-stabilizing [4] and wait-free if it guarantees that for any fair well-formed execution α (starting from an arbitrary state), there exists a suffix of α in which mutual exclusion, eventual exit, and fairness are satisfied regardless of the number of process crashes. $\mathcal{A}$ is said to be self-stabilizing [11] and wait-free, if, in addition to being pseudo-stabilizing and wait-free, it guarantees that every well-formed execution α that starts from a safe state is the suffix of some execution in $\mathcal{E}_s$.

Mutual Exclusion. Mutual exclusion [10] is a degenerate case of the dining philosophers problem in which the conflict graph is a complete graph. In the mutual exclusion parlance, a thinking state is called a *remainder* state, a hungry state is called a *trying* state, an eating state is called a *critical (section)* state, and an exiting state is called an *exiting* state.

5 Wait-Free Pseudo-Stabilizing Mutual Exclusion

Our proposed wait-free pseudo-stabilizing dining algorithm is constructed in three parts. In the first part, we construct a wait-free pseudo-stabilizing ring of processes. In the second part, we deploy a modified version of Dijkstra's mutual-exclusion algorithm on the ring from part one. Finally, in the third part, we construct a wait-free pseudo-stabilizing dining algorithm using a collection of overlapping mutual exclusion instances. This section describes the first two parts

Wait-Free Pseudo-Stabilizing Ring. We use the eventually perfect failure detector $\Diamond\mathcal{P}$ to construct a wait-free pseudo-stabilizing ring over an arbitrary set $\Pi_r \subseteq \Pi$ of processes. The algorithm is straightforward. Each process $i \in \Pi_r$ (locally) determines its predecessor j in the ring as follows. The predecessor of i is a process j such that j is the largest ID smaller than i modulo n such

that j is in Π_r and not suspected by $\Diamond\mathcal{P}$. Precisely, the predecessor of i is determined by the function *pred* as follows: $pred(i) = j$, where $j \in \Pi_r \setminus \Diamond\mathcal{P}_i$ and $\forall k \in \mathbb{N}^+ : 0 < k < i - j \pmod n : i - k \pmod n \notin \Pi_r \setminus \Diamond\mathcal{P}_i$.

The correctness of the above algorithm is also straightforward. Eventually, $\Diamond\mathcal{P}$ suspects exactly the crashed processes, and provides the same output to all the processes. Therefore, eventually all the live processes in Π_r have a consistent and accurate view of the processes that are live in Π_r, and they converge to a unique ring encompassing all the live processes in Π_r.

Wait-Free Pseudo-Stabilizing Mutual Exclusion. We construct a wait-free pseudo-stabilizing mutual exclusion algorithm for an arbitrary set $\Pi_r \subseteq \Pi$ of processes as follows. The algorithm in [14] modifies Dijkstra's self-stabilizing mutual exclusion algorithm [11] for rings with read/write regular registers. Note that Dijkstra's algorithm (in [14] and [11]) requires some process be the 'distinguished' process whose actions are different from other processes. In our case, we require this 'distinguished' process to be correct, and we require no other process to be 'distinguished'. Each process determines whether or not it is distinguished by computing the following local function $leader : \Pi_r \to \{true, false\}$. For each process i, $leader(i)$ is *true* iff $i = \min(\Pi_r \setminus \Diamond\mathcal{P}_i)$.

Eventually, $leader(i)$ is *true* only for the process i with the lowest ID among the correct processes in Π_l, and for all other processes i', $leader(i')$ is $false$. Thus, the function *leader* eventually determines a unique distinguished process in Π_r. Now we simply deploy the algorithm from [14] in the previously described wait-free pseudo-stabilizing ring over Π_r with multi-reader single-writer regular registers with the modification that a process i behaves as the distinguished process when $leader(i)$ is *true*, and it behaves as a non-distinguished process when $leader(i)$ is $false$. Thus, we obtain a wait-free pseudo-stabilizing mutual exclusion algorithm over the set Π_r of processes.

The correctness and stability of the algorithm is straightforward and has been omitted from this version of the paper.

We use multiple instances of the above algorithm in solving dining. For disambiguation, we adopt the following convention: an instance of the above algorithm over a set Π_x of processes is denoted $\mathcal{MX}_x$.

Note that $\mathcal{MX}_r$ interacts with clients at each process in Π_r. We assume that for each process $i \in \Pi_r$, $\mathcal{MX}_r$ contains a variable $mutex_i$. The variable $mutex_i$, at each process i, can have one of four values: *remainder*, *trying*, *critical*, and *exiting*. If $mutex_i$ is *remainder*, then process i is in its *remainder* section; if $mutex_i$ is *trying*, then i is *trying*; if $mutex_i$ is *critical*, then i is in its *critical section*; and if $mutex_i$ is *exiting*, then i has finished accessing its critical section and *exiting* to the remainder section.

We denote the set of safe states for this algorithm as *mutex safe states*[4]. We will use this notion of mutex safe states when arguing for the correctness of the dining algorithm that is described next.

[4] Note that we do not explicitly specify the set of safe states here; it is specified and described in [14]. Here we merely assert its existence, which is sufficient for our purposes.

6 Wait-Free Pseudo-Stabilizing Dining

In this section, we use multiple instances of the mutual exclusion algorithm $\mathcal{MX}$ from Sect. 5 to construct a pseudo-stabilizing wait-free dining algorithm. The algorithm is inspired by the HRA algorithm from [22].

6.1 Algorithm Description

Let $G = (\Pi, E)$ be the conflict graph. Let $\mathcal{R}$ be the set of maximal cliques in G. Let $|\mathcal{R}|$ be k. For convenience, let $\mathcal{R} = \{R_x | x \in \mathbb{N}^+ \wedge 0 < x \leq k\}$. We assume a total order on the cliques such that R_x is ordered before R_y iff $x < y$. For each clique R_x, let Π_x denote the set of processes (diners) in R_x. Each clique $R_x \in \mathcal{R}$ represents a subset of resources to be accessed in isolation by diners in Π_x. Consequently, for each clique R_x, we associate an instance of the wait-free pseudo-stabilizing mutual-exclusion algorithm $\mathcal{MX}_x$, and the participants in $\mathcal{MX}_x$ constitute the set Π_x.

For each diner i, let C_i denote the set of all cliques R_x such that $i \in \Pi_x$; that is, diner i contends for exclusive access to the all the resources associated with cliques in C_i.

Variables. For disambiguation, the variable $mutex_i$ in the mutual exclusion instance $\mathcal{MX}_x$ will be referred to as $\mathcal{MX}_x.mutex_i$. Each diner has access to the private variables $\mathcal{MX}_x.mutex_i$, where $R_x \in C_i$. Finally, we introduce a new private variable $diningState_i$ for each diner i in the system. The variable may contain one of the four values: *thinking*, *hungry*, *eating*, and *exiting*. If $diningState_i$ is *thinking*, then i is thinking, if $diningState_i$ is *hungry*, then i is hungry, and so on.

Three Functions. For each diner i, apart from the actions of algorithm $\mathcal{MX}_x$ for each $R_x \in C_i$, we introduce three additional actions denoted $D.1$–$D.3$ which constitute a new task. Before describing the actions, we introduce three functions $csPrefix$, $currentMutex$, and $badSuffix$ which are used in specifying the guards for the three actions.

Sequence $\mathcal{C}_i$. Let $\mathcal{C}_i$ denote the sequence over all the cliques from C_i such that a clique R_x precedes a clique R_y in $\mathcal{C}_i$ iff $x < y$.

Functions csPrefix and currentMutex. The function $csPrefix(\mathcal{C}_i)$ returns the longest prefix of $\mathcal{C}_i$ such that, for each R_x in $csPrefix(\mathcal{C}_i)$, $\mathcal{MX}_x.mutex_i =$ *critical* (i is in the critical section of $\mathcal{MX}_x$). The function $currentMutex(\mathcal{C}_i)$ returns the first clique following $csPrefix(\mathcal{C}_i)$ in $\mathcal{C}_i$, if such a clique exists; otherwise, it returns $\perp$.

Function badSuffix. The function $badSuffix(\mathcal{C}_i)$ is *true* iff there exists some R_x in the suffix of $\mathcal{C}_i$ following $currentMutex(\mathcal{C}_i)$ such that $\mathcal{MX}_x.mutex_i$ is either *trying* or *critical* (i is either trying or in the critical section of $\mathcal{MX}_x$).

An informal motivation for the foregoing functions follows. Upon becoming hungry, each diner i starts trying in $\mathcal{MX}_c = currentMutex(\mathcal{C}_i)$, and when i enters the critical section of $\mathcal{MX}_c$, $\mathcal{MX}_c$ becomes a part of $csPrefix(\mathcal{C}_i)$. Subsequently, i starts trying in $\mathcal{MX}_{c'} = currentMutex(\mathcal{C}_i)$ which follows $\mathcal{MX}_c$

in $\mathcal{C}_i$, and so on, until i is in the critical section of all $\mathcal{MX}$ instances in $\mathcal{C}_i$. In the absence of faults, while a diner i is hungry, i is in the critical section of all $\mathcal{MX}$ instances in $csPrefix(\mathcal{C}_i)$ and in the remainder or exiting section of all the $\mathcal{MX}$ instances in the suffix following $currentMutex(\mathcal{C}_i)$ in $\mathcal{C}_i$. However, due to a transient fault, it is possible for a diner i to be in a state in which i is either trying or in the critical section of some $\mathcal{MX}$ instance in the suffix; when this occurs, we say that the suffix is "bad". This is captured by the predicate $badSuffix(\mathcal{C}_i)$.

Actions. We introduce three new actions (that constitute a single new task) for each process (diner) i in our proposed wait-free self-stabilizing dining algorithm. The pseudocode is given in Fig. 1 and described next.

The first action, *Action D.1*, is enabled when the diner (say) i is either thinking or exiting, or $badSuffix(\mathcal{C}_i)$ is *true*. When Action D.1 is executed, it sets each *mutex* variable that is not *remainder* to *exiting* and sets $diningState_i$ to *thinking*. That is, diner i is either in the remainder section or in the exit section in all the mutual-exclusion instances. Note that this transition need not satisfy the well-formedness condition of mutual-exclusion clients. For stabilization, it is important to ensure that such well-formedness violations occur only finitely many times in any execution.

The second action, *Action D.2*, is enabled when the diner i is hungry, i is not in the critical section of all the associated mutual-exclusion instances ($csPrefix(\mathcal{C}_i) \neq \mathcal{C}_i$), and $badSuffix(\mathcal{C}_i)$ is $false$. When Action D.2 is executed, i starts trying in the mutual-exclusion instance of $currentMutex(\mathcal{C}_i)$.

The third action, *Action D.3*, is enabled when the diner i is hungry, and i is in the critical section of all the associated $\mathcal{MX}$ instances ($csPrefix(\mathcal{C}_i) = \mathcal{C}_i$). When Action D.3 is executed, diner i starts eating.

Note that the client of the dining service at each process is responsible for transitioning from thinking to hungry and from eating to thinking.

6.2 Pseudo-Stabilization

In order to establish the pseudo-stabilization property, we have to define a set of safe states for each process in the system. *The system is said to be in a safe state iff every mutual-exclusion instance is in a mutex safe state, and every live diner is in a "diner-safe state", as defined next.*

A diner i is said to be in a *diner-safe state* if (1) $badSuffix(\mathcal{C}_i)$ is false, (2) if i is eating, then every $\mathcal{MX}$ instance at i is in its critical section, and (3) if i is thinking, then every $\mathcal{MX}$ instance at i is either in its remainder section or is exiting. Precisely, $\neg badSuffix(\mathcal{C}_i) \wedge ((diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)) \wedge ((diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting)))$ is *true*. The system is said to be in a *diner-safe state* iff each live diner is in a diner-safe state.

Closure. We prove closure with respect to diner states in Lemma 4 using three helper lemmas. Let s be an arbitrary state of the system executing the action system from Fig. 1, where each mutual-exclusion instance is an instance of $\mathcal{MX}$.

private variable $diningState_i$
foreach $R_x \in C_i$:
  variable $\mathcal{MX}_x.mutex_i$ *Variable $mutex_i$ in instance $\mathcal{MX}_x$ described in Sect. 5*
/ Note that the client at process i sets $diningState_i$ to "hungry" upon becoming hungry and to "exiting" upon finishing eating. Client actions are not shown.*
*Also, the client is assumed to be "well-formed". */*
The three actions below constitute a single task

---

1 : $\{(diningState_i = thinking) \vee (diningState_i = exiting)$
  $\vee(badSuffix(\mathcal{C}_i))\} \longrightarrow$ *Action D.1*
2 : **foreach** $R_x \in C_i$:
3 : **if** $(\mathcal{MX}_x.mutex_i \neq remainder)$
4 : $\mathcal{MX}_x.mutex_i \leftarrow exiting$ *// If eating, then exit; if hungry, then abort.*
5 : $diningState_i \leftarrow thinking$ *// Exit in all mutex instances, and start thinking*

---

6 : $\{(diningState_i = hungry) \wedge (csPrefix(\mathcal{C}_i) \neq \mathcal{C}_i)$
  $\wedge(\neg badSuffix(\mathcal{C}_i))\} \longrightarrow$ *Action D.2*
7 : $R_x \leftarrow currentMutex(\mathcal{C}_i)$ *// If hungry and in the critical section of a prefix of*
8 : **if** $(\mathcal{MX}_x.mutex_i = remainder)$ *// cliques, then start trying in the mutex*
9 : $\mathcal{MX}_x.mutex_i \leftarrow trying$ *// associated with the next clique*

---

10 : $\{(diningState_i = hungry) \wedge (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)\} \longrightarrow$ *Action D.3*
11 : $diningState_i \leftarrow eating$ *// Transit from hungry to eating*

Fig. 1. Self-Stabilizing Wait-Free Dining. Action system at process i

Lemma 1. *For any process i, if $s.badSuffix(\mathcal{C}_i)$ is $false$, then for each successor s' of s, $s'.badSuffix(\mathcal{C}_i)$ is $false$.*

Lemma 2. *For any process i, if $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is true in state s, then for any successor s' of s, $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i))$ remains true.*

Lemma 3. *For any process i, if $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ is true in state s, then for any successor s' of s, $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ remains true.*

The proofs of the above three lemmas are straightforward. They consider each possible successor s' of s by considering each enabled action. By case analysis for each such action, we confirm that the lemmas are true.

Closure, with respect to diner-safe states, follows from Lemmas 1, 2, and 3. We have the following lemma.

Lemma 4. *Every fair execution of the action system from Fig. 1, where each mutual-exclusion instance is an instance of $\mathcal{MX}$, satisfies closure (with respect to diner-safe states): if the system eventually reaches a diner-safe state s, then the suffix of the execution following s contains only diner-safe states.*

Convergence. We prove convergence, with respect to diner safe states, in three parts. First, we prove that if the system is in a state where for some correct

diner i, $badSuffix(\mathcal{C}_i)$ is *true*, then eventually, $badSuffix(\mathcal{C}_i)$ becomes $false$. Next, we prove that if the system is in a state where, for some diner i, i is eating, but $csPrefix(\mathcal{C}_i) \neq \mathcal{C}_i$, then eventually we reach a state where, if i is eating, then $csPrefix(\mathcal{C}_i) = \mathcal{C}_i$. Finally, we show that the following. If the system is in a state where, if some diner i is thinking, but in some $\mathcal{MX}$ instance i is neither exiting nor in the remainder section, then we eventually reach a state in which the following is true. If i is thinking, then in all $\mathcal{MX}$ instances (for i), i is either exiting or in the remainder section.

For the following lemmas, fix α to be an arbitrary fair execution of the action system from Fig. 1, where each mutual-exclusion instance is an instance of $\mathcal{MX}$. For any given pair of states s and s' in the system-state sequence associated α, any if s' occurs after s, then s' is said to be a *descendant* of s.

Lemma 5. *In α, for each correct diner i, if $s.badSuffix(\mathcal{C}_i)$ is true for some state s, then there exists a descendant s' of s such that $s'.badSuffix(\mathcal{C}_i)$ is $false$.*

Proof sketch. Note that Action D.1 at i is enabled in state s and remains enabled until executed. Upon executing Action D.1 at i, $badSuffix(\mathcal{C}_i)$ becomes false.

Lemma 6. *In α, for each correct diner i, if the system is in a state s where $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is $false$, then, there exists a descendant s' of s in α, such that $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is true, in s'.*

Proof sketch. Note that i eats for finite durations in α. Since i is eating in state s, there exists a descendant s' of s in which i is not eating, and $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is *true*, in s'. □

Lemma 7. *In α, for each correct diner i, if the system is in a state s where $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ is $false$, then there exists a descendant s' of s in α, such that $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ is true in s'.*

Proof sketch. If i becomes hungry, then the lemma is satisfied. Otherwise, note that Action D.1 at i is enabled in state s and remains enabled until executed. Upon executing Action D.1 at i, we see that for each R_x in C_i Action D.1 sets $\mathcal{MX}_x.mutex_i$ to either *remainder* or *exiting*. □

Lemma 8. *The action system from Fig. 1, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance, satisfies convergence (with respect to diner-safe states): from any arbitrary state, upon executing enabled program actions from Fig. 1 and all the $\mathcal{MX}$ instances, the system eventually reaches a diner-safe state.*

The proof follows from Lemmas 5, 6, and 7.

Thus, we have shown pseudo-stabilization with respect to diner-safe states. In order to complete the proof for pseudo-stabilization, we have to prove that

the action system in Fig. 1 in eventually well-formed. Note that Action D.3 does not change the value of *mutex* variables, and Action D.2 changes a *mutex* variable to *trying* from *remainder*, which satisfies mutual exclusion well-formedness conditions. However, Action D.1 could set the *mutex* variables to *trying* from *exiting* and violate the well-formedness condition. Therefore, it remains to show that Action D.1 violates the well-formedness conditions only finitely many times.

Lemma 9. *In any fair execution of the action system from Fig. 7, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance, at each correct process i, only finitely many occurrences of Action D.1 violate mutual exclusion well-formedness conditions.*

Proof sketch. From the pseudocode, we see that Action D.1 violates mutual exclusion well-formedness conditions if line 4 is executed at a correct diner i when $\mathcal{MX}_x.mutex_i$ is $trying$ for some $r_x \in C_i$. From Lemmas 8 and 4, we know that in any fair execution, eventually forever $badSuffix(\mathcal{C}_i)$ is $false$. Therefore, eventually forever, when i executes Action D.1 $diningState_i$ is either $thinking$ or $exiting$. However, from Lemmas 3 and 7, we know that when $diningState_i$ is thinking, $\mathcal{MX}_x.mutex_i$ is not $trying$. Finally, if i executes Action D.1 when $diningState_i$ is $exiting$, then recall that while i was eating (just prior to exiting) $csPrefix(\mathcal{C}_i)$ is $true$. Consequently, when $diningState_i$ is $exiting$, $csPrefix(\mathcal{C}_i)$ is $true$; that is, $\mathcal{MX}_x.mutex_i$ is not $trying$. Therefore, at each correct process i, only finitely many occurrences of Action D.1 violate mutual exclusion well-formedness conditions. □

Therefore, from Lemmas 4, 8, and 9, we establish pseudo-stabilization.

6.3 Correctness

We demonstrate safety and progress starting from a safe state after $\Diamond\mathcal{P}$ stops falsely suspecting correct processes. The safety condition for wait-free dining is that no two neighboring diners are live and eating concurrently. The progress condition is that every correct hungry diner eventually eats.

Lemma 10. *In any fair execution starting from a safe state, the action system from Fig. 7, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance satisfies safety: no two neighboring diners are live and eating concurrently.*

Proof sketch. Let α be a fair execution starting from a safe state. Let α' be a suffix of α in which $\Diamond\mathcal{P}$ does not suspect live processes. Let i and j be two live neighbors in some safe state s in α' in which i is eating. Since i and j are neighbors there exists clique R_y such that $R_y \in C_i \cap C_j$. Since i is eating, we know that $\mathcal{MX}_y.mutex_i = critical$ in s. From the mutual exclusion property, we know that $\mathcal{MX}_y.mutex_j \neq critical$ in s. In other words, $csPrefix(\mathcal{C}_j) \neq \mathcal{C}_j$ in s. Therefore, j is not eating concurrently with i. □

Lemma 11. *In any fair well-formed execution starting from a safe state, the action system from Fig. 7, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance, satisfies progress: every correct hungry diner eventually eats.*

Proof sketch. For contradiction, we assume that in some fair well-formed execution α of the system starting from a safe state, some hungry correct diner never eats. If i is such a diner, then i must be trying for ever in some $\mathcal{MX}$ instance Since $\mathcal{MX}_x$ is wait-free, this implies that some correct neighbor j of i is in the critical section of $\mathcal{MX}_x$ forever. From Fig. 1, this means that for some other clique R_y, $y > x$, j must be trying in $\mathcal{MX}_y$ forever. Consequently, j is also hungry forever. By the total order in which processes start trying in the $\mathcal{MX}$ instances, x cannot be in the critical section of $\mathcal{MX}_y$. Therefore, some other process k must be in the critical section of $\mathcal{MX}_y$ forever, and consequently, that process k must be trying forever in some $\mathcal{MX}_z$, where $z > y$, (and therefore, k must be hungry forever), and so on. Since there are only finitely many process and finitely many $\mathcal{MX}$ instances, there must exist some process $\hat{i}$ that is (a) hungry forever, (b) in the critical section of some $\mathcal{MX}_{\hat{z}}$, and (c) not trying in any $\mathcal{MX}_{z'}$, where $z' > \hat{z}$. However, from Fig. 1, we see that this is impossible. Thus, we have a contradiction. □

From Lemmas 10 and 11, we establish correctness.

Theorem 1. *The action system in Fig. 1, where each mutual-exclusion instance is an $\mathcal{MX}$ instance, solves wait-free pseudo-stabilizing dining.*

7 Discussion

Wait-Free Self-stabilizing Regular Registers. Our algorithm assumes that the system contains wait-free self-stabilizing regular registers. We remark that existing results from [17] may be used to construct wait-free self-stabilizing regular registers from wait-free self-stabilizing safe registers. Although results in [17] construct atomic registers from regular registers, this construction is expensive and uses unbounded counters.

Achieving Self-stabilization. Our proposed algorithm is wait-free and pseudo-stabilizing, and not self-stabilizing, because starting from a safe state, if $\Diamond\mathcal{P}$ falsely suspects a correct process, it could result in the system transitioning to an unsafe state. Since $\Diamond\mathcal{P}$ eventually ceases such false suspicions, we are guaranteed pseudo-stabilization. If we view a false suspicion by $\Diamond\mathcal{P}$ as a transient fault in the system, then we see that our algorithm is, in fact, self-stabilizing as well. Alternatively, if we replace $\Diamond\mathcal{P}$ with the perfect failure detector $\mathcal{P}$ [6] which never suspects live processes and eventually and permanently suspects crashed processes, our algorithm becomes self-stabilizing (in addition to being wait-free).

On Assuming $\Diamond\mathcal{P}$. Recall that wait-free dining is unsolvable in asynchronous systems. Consequently, we resort to using $\Diamond\mathcal{P}$ to achieve wait-freedom. In fact, $\Diamond\mathcal{P}$ is the weakest failure detector to solve wait-free dining under eventual exclusion [29]. Therefore, assuming $\Diamond\mathcal{P}$ is necessary to solve wait-free self-stabilizing dining in any (partially synchronous) shared-memory system.

Future Work. There are several ways in which our result can be extended. For instance, if we can implement self-stabilizing $\Diamond\mathcal{P}$ in a partially synchronous

system with safe registers, then we can adapt our algorithm to solve wait-free self-stabilizing dining with safe registers instead of regular registers. Another avenue for improvement is in fairness. Our algorithm is weakly fair — every hungry diner eventually eats, but could be overtaken unboundedly many times by hungry neighbors. However, we know that $\Diamond\mathcal{P}$ is sufficient to solve eventually bounded-fair dining [30]. It remains to be seen if we can solve wait-free stabilizing dining with bounded fairness using $\Diamond\mathcal{P}$.

References

1. Antonoiu, G., Srimani, P.K.: Mutual Exclusion Between Neighboring Nodes in an Arbitrary System Graph Tree That Stabilizes Using Read/Write Atomicity. In: Amestoy, P.R., Berger, P., Daydé, M., Duff, I.S., Frayssé, V., Giraud, L., Ruiz, D. (eds.) Euro-Par 1999. LNCS, vol. 1685, pp. 823–830. Springer, Heidelberg (1999)
2. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-stabilizing local mutual exclusion and daemon refinement. Chicago Journal of Theortical Computer Science 2002(1) (2002)
3. Beauquier, J., Kekkonen-Moneta, S.: Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. International Journal of Systems Science 28(11), 1177–1187 (1997)
4. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. Distributed Computing 7(1), 35–42 (1993)
5. Cantarell, S., Datta, A.K., Petit, F.: Self-Stabilizing Atomicity Refinement Allowing Neighborhood Concurrency. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 102–112. Springer, Heidelberg (2003)
6. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
7. Choy, M., Singh, A.K.: Localizing failures in distributed synchronization. IEEE Transactions on Parallel and Distributed Systems 7(7), 705–716 (1996)
8. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust Stabilizing Leader Election. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 219–233. Springer, Heidelberg (2007)
9. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Distributed Computing 65(4), 492–505 (2005)
10. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM 8(9), 569 (1965)
11. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
12. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
13. Dolev, S.: Self-Stabilization. MIT Press (2000)
14. Dolev, S., Herman, T.: Dijkstra's Self-Stabilizing Algorithm in Unsupportive Environments. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 67–81. Springer, Heidelberg (2001)
15. Dolev, S., Kat, R.I., Schiller, E.M.: When Consensus Meets Self-stabilization: Self-stabilizing Failure-Detector, Consensus and Replicated State-Machine (Extended Abstract). In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 45–63. Springer, Heidelberg (2006)

16. Fischer, M., Jiang, H.: Self-stabilizing Leader Election in Networks of Finite-State Anonymous Agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 395–409. Springer, Heidelberg (2006)
17. Hoepman, J.H., Papatriantafilou, M., Tsigas, P.: Self-stabilization of wait-free shared memory objects. Journal of Parallel and Distributed Computing 62(5), 818–842 (2002)
18. Hutle, M., Widder, J.: On the Possibility and the Impossibility of Message-Driven Self-stabilizing Failure Detection. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, pp. 153–170. Springer, Heidelberg (2005)
19. Lamport, L.: On interprocess communication. Part II: Algorithms. Distributed Computing 1(2), 86–101 (1986)
20. Line, J.C., Ghosh, S.: A methodology for constructing a stabilizing crash-tolerant application. In: Proceedings of the 13th Symposium on Reliable Distributed Systems, pp. 12–21 (1994)
21. Line, J.C., Ghosh, S.: Stabilizing algorithms for diagnosing crash failures. In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing, p. 376 (1994)
22. Lynch, N.A.: Upper bounds for static resource allocation in a distributed system. Journal of Computer and System Sciences 23(2), 254–278 (1981)
23. Mizuno, M., Nesterenko, M.: A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. Inf. Process. Lett. 66(6), 285–290 (1998)
24. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: Proceedings of the 22nd International Conference on Distributed Computing Systems, pp. 172–179 (2002)
25. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. Journal of Parallel and Distributed Computing 62(5), 766–791 (2002)
26. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, pp. 22–29 (2002)
27. Pike, S.M., Sivilotti, P.A.: Dining philosophers with crash locality 1. In: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems, pp. 22–29 (2004)
28. Pike, S.M., Song, Y., Sastry, S.: Wait-Free Dining Under Eventual Weak Exclusion. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) ICDCN 2008. LNCS, vol. 4904, pp. 135–146. Springer, Heidelberg (2008)
29. Sastry, S., Pike, S.M., Welch, J.L.: The weakest failure detector for wait-free dining under eventual weak exclusion. In: Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures, pp. 111–120 (2009)
30. Song, Y., Pike, S.M.: Eventually k-bounded wait-free distributed daemons. In: IEEE International Conference on Dependable Systems and Networks, pp. 645–655 (2007)

Node Sampling Using Random Centrifugal Walks

Andrés Sevilla[1], Alberto Mozo[2], and Antonio Fernández Anta[3]

[1] Dpto Informática Aplicada, U. Politécnica de Madrid, Madrid, Spain
[2] Dpto Arquitectura y Tecnología de Computadores, U. Politécnica de Madrid, Madrid, Spain
{asevilla,amozo}@eui.upm.es
[3] Institute IMDEA Networks, Madrid, Spain
antonio.fernandez@imdea.org

Abstract. Sampling a network with a given probability distribution has been identified as a useful operation. In this paper we propose distributed algorithms for sampling networks, so that nodes are selected by a special node, called the *source*, with a given probability distribution. All these algorithms are based on a new class of random walks, that we call *Random Centrifugal Walks* (RCW). A RCW is a random walk that starts at the source and *always* moves *away* from it.

Firstly, an algorithm to sample any connected network using RCW is proposed. The algorithm assumes that each node has a weight, so that the sampling process must select a node with a probability proportional to its weight. This algorithm requires a preprocessing phase before the sampling of nodes. In particular, a minimum diameter spanning tree (MDST) is created in the network, and then nodes' weights are efficiently aggregated using the tree. The good news are that the preprocessing is done only once, regardless of the number of sources and the number of samples taken from the network. After that, every sample is done with a RCW whose length is bounded by the network diameter.

Secondly, RCW algorithms that do not require preprocessing are proposed for grids and networks with regular concentric connectivity, for the case when the probability of selecting a node is a function of its distance to the source.

The key features of the RCW algorithms (unlike previous Markovian approaches) are that (1) they do not need to warm-up (stabilize), (2) the sampling always finishes in a number of hops bounded by the network diameter, and (3) it selects a node with the *exact probability distribution*.

1 Introduction

Sampling a network with a given distribution has been identified as a useful operation in many contexts. For instance, sampling nodes with uniform probability is the building block of epidemic information spreading [13,12]. Similarly, sampling with a probability that depends on the distance to a given node [3,17] is useful to construct small world network topologies [14,7,2]. Other applications that can benefit from distance-based node sampling are landmark-less network positioning systems like NetICE9 [16], which does sampling of nodes

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 300–314, 2012.

with special properties to assign synthetic coordinates to nodes. In a different context, currently there is an increasing interest in obtaining a representative (unbiased) sample from the users of online social networks [9]. In this paper we propose a distributed algorithm for sampling networks with a desired probability distribution.

Related Work. One technique to implement distributed sampling is to use gossiping between the network nodes. Jelasity et al. [12] present a general framework to implement a uniform sampling service using gossip-based epidemic algorithms. Bertier et al. [2] implement uniform sampling and DHT services using gossiping. As a side result, they sample nodes with a distribution that is close to Kleinberg's harmonic distribution (one instance of a distance-dependent distribution). Another gossip-based sampling service that gets close to Kleinberg's harmonic distribution has been proposed by Bonnet et al. [3]. However, when using gossip-based distributed sampling as a service, it has been shown by Busnel et al. [5] that only partial independence (ϵ-independence) between views (the subsets of nodes held at each node) can be guaranteed without re-executing the gossip algorithm. Gurevich and Keidar [11] give an algorithm that achieves ϵ-independence in $O(ns \log n)$ rounds (where n is the network size and s is the view size).

Another popular distributed technique to sample a network is the use of random walks [18]. Most random-walk based sampling algorithms do uniform sampling [1,9], usually having to deal with the irregularities of the network. Sampling with arbitrary probability distributions can be achieved with random walks by re-weighting the hop probabilities to correct the sampling bias caused by the non-uniform stationary distribution of the random walks. Lee et al. [15] propose two new algorithms based on Metropolis-Hastings (MH) random walks for sampling with any probability distribution. These algorithms provide an unbiased graph sampling with a small overhead, and a smaller asymptotic variance of the resulting unbiased estimators than generic MH random walks.

Sevilla et al. [17] have shown how sampling with an arbitrary probability distribution can be done without communication if a uniform sampling service is available. In that work, as in all the previous approaches, the desired probability distribution is reached when the stationary distribution of a Markov process is reached. The number of iterations (or hops of a random walk) required to reach this situation (the warm-up time) depends on the parameters of the network and the desired distribution, but it is not negligible. For instance, Zhong and Sheng [18] found by simulation that, to achieve no more than 1% error, in a torus of 4096 nodes at least 200 hops of a random walk are required for the uniform distribution, and 500 hops are required for a distribution proportional to the inverse of the distance. Similarly, Gjoka et al. [10] show that a MHRW sampler needs about 6K samples (or 1000-3000 iterations) to obtain the convergence to the uniform probability distribution. In the light of these results, Markovian approaches seem to be inefficient to implement a sampling service, specially if multiple samples are desired.

Contributions. In this paper we present efficient distributed algorithms to implement a sampling service. The basic technique used for sampling is a new class of random walks that we call *Random Centrifugal Walks* (RCW). A RCW starts at a special node, called the *source*, and *always* moves *away* from it.

All the algorithms proposed here are instances of a generic algorithm that uses the RCW as basic element. This generic RCW-based algorithm works essentially as follows. A RCW always starts at the source node. When the RCW reaches a node x (the first node reached by a RCW is always the source s), the RCW stops at that node with a *stay probability*. If the RCW stops at node x, then x is the node selected by the sampling. If the RCW does not stop at x, it jumps to a neighbor of x. To do so, the RCW chooses only among neighbors that are farther from the source than the node x. (The probability of jumping to each of these neighbors is not necessarily the same.) In the rest of the paper we will call all the instances of this generic algorithm as *RCW algorithms.*

Firstly, we propose a RCW algorithm that samples *any* connected network with *any* probability distribution (given as weights assigned to the nodes). Before starting the sampling, a preprocessing phase is required. This preprocessing involves building a minimum distance spanning tree (MDST) in the network[1], and using this tree for efficiently aggregating the node's weights. As a result of the weight aggregation, each node has to maintain one numerical value per link, which will be used by the RCW later. Once the preprocessing is completed, any node in the network can be the source of a sampling process, and multiple independent samplings with the exact desired distribution can be efficiently performed. Since the RCW used for sampling follow the MDST, they take at most D hops (where D is the network diameter).

Secondly, when the probability distribution is distance-based and nodes are at integral distances (measured in hops) from the source, RCW algorithms without preprocessing (and only a small amount of state data at the nodes) are proposed. In a *distance-based probability distribution* all the nodes at the same distance from the source node are selected with the same probability. (Observe that the uniform and Kleinberg's harmonic distributions are special cases of distance-based probability distributions.) In these networks, each node at distance $k > 0$ from the source has neighbors (at least) at distance $k - 1$. We can picture nodes at distance k from the source as positioned on a ring at distance k from the source. The center of all the rings is the source, and the radius of each ring is one unit larger than the previous one. Using this graphical image, we refer the networks of this family as *concentric rings networks.* [2]

The first distance-oriented RCW algorithm we propose samples with a distance-based distribution in a network with grid topology. In this network, the source node is at position $(0, 0)$ and the lattice (Manhattan) distance is used. This grid

[1] Using, for instance, the algorithm proposed by Bui et al. [4] whose time complexity is $O(n)$.

[2] Observe that *every* connected network can be seen as a concentric rings network. For instance, by finding the breadth-first search (BFS) tree rooted at the source, and using the number of hops in this tree to the source as distance.

contains all the nodes that are at a distance no more than the radius R from the source (the grid has hence a diamond shape[3]). The algorithm we derive assigns a stay probability to each node, that only depends on its distance from the source. However, the hop probabilities depend on the position (i, j) of the node and the position of the neighbors to which the RCW can jump to. We formally prove that the desired distance-based sampling probability distribution is achieved. Moreover, since every hop of the RCW in the grid moves one unit of distance away from the source, the sampling is completed after at most R hops.

We have proposed a second distance-oriented RCW algorithm that samples with distance-based distributions in concentric rings networks *with uniform connectivity*. These are networks in which all the nodes in each ring k have the same number of neighbors in ring $k-1$ and the same number in ring $k+1$. Like the grid algorithm, this variant is also proved to finish with the desired distribution in at most R hops, where R is the number of rings.

Unfortunately, in general, concentric rings networks have no uniform connectivity. This case is faced by creating, on top of the concentric rings network, an overlay network that has uniform connectivity. In the resulting network, the algorithm for uniform connectivity can be used. We propose a distributed algorithm that, if it completes successfully, builds the desired overlay network. We have found via simulations that this algorithm succeeds in building the overlay network in a large number of cases.

In summary, RCW can be used to implement an efficient sampling service because, unlike previous Markovian (e.g., classical random walks and epidemic) approaches, (1) it always finishes in a number of hops bounded by the network diameter, (2) selects a node with the *exact probability distribution*, and (3) does not need warm-up (stabilization) to converge to the desired distribution. Additionally, in the case that preprocessing is needed, this only has to be executed once, independently on the number of sources and the number of samples taken from the network.

The rest of the paper is structured as follows. In Section 2 we introduce concepts and notation that will be used in the rest of the paper. In Section 3 we present the RCW algorithm for a connected network. In Sections 4 and 5 we describe the RCW algorithm on grids and concentric rings networks with uniform connectivity. In Section 6 we present the simulation based study of the algorithm for concentric rings topologies without uniform connectivity. Finally, we conclude the paper in Section 7.

2 Definitions and Model

Connected Networks. In this paper we only consider connected networks. This family includes most of the potentially interesting networks we can find. In every network, we use N to denote the set of nodes and $n = |N|$ the size of that set. When convenient, we assume that there is a special node in the network, called the *source* and denoted by s. We assume that each node $x \in N$ has

[3] A RCW algorithm for a square grid is easy to derive from the one presented.

an associated weight $w(x) > 0$. Furthermore, each node *knows* its own weight. The weights are used to obtain the desired probability distribution p, so that the probability of selecting a node x is proportional to $w(x)$. Let us denote $\eta = \sum_{j \in N} w(j)$. Then, the probability of selecting $x \in N$ is $p(x) = w(x)/\eta$. (In the simplest case, weights are probabilities, i.e., $w(x) = p(x), \forall x$ and $\eta = 1$.)

RCW in Connected Networks. As mentioned, in order to use RCW to sample connected networks, some preprocessing is done. This involves constructing a spanning tree in the network and performing a weight aggregation process. After the preprocessing, RCW is used for sampling. A RCW starts from the source. When the RCW reaches a node $x \in N$, it selects x as the sampled vertex with probability $q(x)$, which we call the *stay probability*. If x is not selected, a neighbor y of x in the tree is chosen, using for that a collection of *hop probabilities* $h(x, y)$. The values of $q(x)$ and $h(x, y)$ are computed in the preprocessing and stored at x. The probability of reaching a node $x \in N$ in a RCW is called the *visit probability*, denoted $v(x)$.

Concentric Rings Networks. We also consider a subfamily of the connected networks, which we call *concentric rings networks.* These are networks in which the nodes of N are at integral distances from s. In these networks, no node is at a distance from s larger than a radius R. For each $k \in [0, R]$, we use $\mathbb{R}_k \neq \emptyset$ to denote the set of nodes at distance k from s, and $n_k = |\mathbb{R}_k|$. (Observe that $\mathbb{R}_0 = \{s\}$ and $n_0 = 1$.) These networks can be seen as a collection of concentric rings at distances 1 to R from the source, which is the common center of all rings. For that reason, we call the set $\mathbb{R}_k$ the *ring at distance* k. For each $x \in \mathbb{R}_k$ and $k \in [1, R]$, $\gamma_k(x) > 0$ is the number of neighbors of node x at distance $k - 1$ from s (which is only 1 if $k = 1$), and $\delta_k(x)$ is the number of neighbors of node x at distance $k + 1$ from s (which is 0 if $k = R$).

The concentric rings networks considered must satisfy the additional property that the probability distribution is *distance based*. This means that, for all $k \in [0, R]$, every node $x \in \mathbb{R}_k$ has the same probability p_k to be selected. We assume that each node $x \in \mathbb{R}_k$ knows its own p_k. These properties allow, in the subfamilies defined below, to avoid the preprocessing required for connected networks.

Grids. A first subfamily of concentric rings networks considered is the grid with lattice distances. In this network, the source is at position $(0, 0)$ of the grid, and it contains all the nodes (i, j) so that $i, j \in [-R, R]$ and $|i| + |j| \leq R$. For each $k \in [0, R]$, the set of nodes in ring k is $\mathbb{R}_k = \{(i, j) : |i| + |j| = k\}$. The neighbors of a node (i, j) are the nodes $(i - 1, j)$, $(i + 1, j)$, $(i, j - 1)$, and $(i, j + 1)$ (that belong to the grid).

Uniform Connectivity. The second subfamily considered is formed by the concentric rings networks with *uniform connectivity*. These networks satisfy that

$$\forall k \in [1, R], \forall x, y \in \mathbb{R}_k, \delta_k(x) = \delta_k(y) \wedge \gamma_k(x) = \gamma_k(y). \quad (1)$$

In other words, all nodes of ring k have the same number of neighbors δ_k in ring $k + 1$ and the same number of neighbors γ_k in ring $k - 1$.

RCW in Concentric Rings Networks. The behavior of a generic RCW was already described. In the algorithm that we will present in this paper for concentric rings networks we guarantee that, for each k, all the nodes in $\mathbb{R}_k$ have the same visit probability v_k and the same stay probability q_k. A RCW starts from the source. When it reaches a node $x \in \mathbb{R}_k$, it selects x as the sampled vertex with stay probability q_k. If x is not selected, a neighbor $y \in \mathbb{R}_{k+1}$ of x is chosen.

The desired *distance-based probability distribution* is given by the values p_k, $k \in [0, R]$, where it must hold that $\sum_{k=0}^{R} n_k \times p_k = 1$. The problem to be solved is to define the stay and hop probabilities so that the probability of a node $x \in \mathbb{R}_k$ is p_k.

Observation 1. *If for all $k \in [0, R]$ the visit v_k and stay q_k probabilities are the same for all the nodes in $\mathbb{R}_k$, the RCW samples with the desired probability iff $p_k = v_k \cdot q_k$.*

3 Sampling in a Connected Network

In this section, we present a RCW algorithm that can be used to sample any connected network. As mentioned, in addition to connectivity, it is required that each node knows its own weight. A node will be selected with probability proportional to its weight.

Preprocessing for the RCW Algorithm. The RCW algorithm for connected networks requires some preprocessing which will be described now. This preprocessing has to be done only once for the whole network, independently of which nodes act as sources and how many samples are taken.

Building a spanning tree. Initially, the algorithm builds a spanning tree of the network. A feature of the algorithm is that, if several nodes want to act as sources for RCW, they can all share the same spanning tree. Hence only one tree for the whole network has to be built. The algorithm used for the tree construction is not important for the correctness of the RCW algorithm, but the diameter of the tree will be an upper bound on the length of the RCW (and hence possibly the sampling latency). There are several well known distributed algorithms (see, e.g., [6] and the references therein) that can be used to build the spanning tree. In particular, it is interesting to build a minimum diameter spanning tree (MDST) because, as mentioned, the length of the RCW is upper bounded by the tree diameter. There are few algorithms in the literature to build a MDST. One possible candidate to be used in our context is the one proposed by Bui et al. [4]. Additionally, if link failures are expected, the variation of the former algorithm proposed by Gfeller et al. [8] can be used.

Weight aggregation. Once the spanning tree is in place, the nodes compute and store aggregated weights using the algorithm of Figure 1. The algorithm executes at each node $i \in N$, and it computes in a distributed way the aggregated weight of each subtree that can be reached following one of the links of i. In particular, for each node x that is in the set of neighbors of i in the tree, $neighbors(i)$, the

```
1  task Weight_Aggregation(i)
2  if i is a leaf then
3    send WEIGHT(w(i)) to neighbor x
4    receive WEIGHT(p) from neighbor x
5    T_i(x) ← p
6  else
7    repeat
8      receive WEIGHT(p) from x ∈ neighbors(i)
9      T_i(x) ← p
10     foreach y ∈ neighbors(i) \ {x} do
11       if received WEIGHT(·) from neighbors(i) \ {y} then
12         send WEIGHT(w(i) + Σ_{z∈neighbors(i)\{y}} T_i(z)) to y
13     end foreach
14   until received WEIGHT(·) from every x ∈ neighbors(i)
```

Fig. 1. Weight aggregation algorithm. Code for node i.

algorithm computes a value $T_i(x)$ and stores it at i. Let (i, x) be a link of the spanning tree, then by removing the link (i, x) from the spanning tree there are two subtrees. We denote by $stree(x, i)$ the subtree out of them that contains node x.

Theorem 1. *After the completion of the Weight Aggregation algorithm (of Figure 1), each node $i \in N$ will store, for each node $x \in neighbors(i)$, in $T_i(x)$ the value $\sum_{y \in stree(x,i)} w(y)$.*

Proof. Consider $stree(x, i)$ a tree rooted at x. We prove the claim by induction in the depth of this tree. The base case is when the tree has depth 1. In this case x is a leaf and, from the algorithm, it sends to i its weight $w(x)$, which is stored at i as $T_i(x)$. If the depth is $k > 1$, by induction hypothesis x ends up having in $T_x(y)$ the sum of the weight of the subtree $stree(x, y)$, for each $y \in neighbors(x) \setminus \{i\}$. These values plus $w(x)$ are added up and sent to i, which stores the resulting value as $T_i(x)$.

The values $T_i(x)$ computed in this preprocessing phase will later be used by the RCW algorithm to perform the sampling. We can bound now the complexity of this process in terms of messages exchanged and time to complete. We assume that all nodes start running the Weight Aggregation algorithm simultaneously, that the transmission of messages takes one step, and that computation time is negligible. The proof of the following theorem can be found in [19].

Theorem 2. *The Weight Aggregation algorithm (of Figure 1) requires $2(n-1)$ messages to be exchanged, and completes after D steps, where D is the diameter of the tree.*

RCW Sampling Algorithm. In this RCW algorithm (Figure 2) any node can be the source. The spanning tree and the precomputed aggregated weights are

1 **task** $RCW(i)$
2 **when** $RCW\_MSG(s)$ **received from** x
3 $candidates \leftarrow neighbors(i) \setminus \{x\}$
4 **with** *probability* $q(i) = \frac{w(i)}{w(i)+\sum_{z \in candidates} T_i(z)}$ **do**
5 *select node* i *and report to source* s
6 **otherwise**
7 *choose a node* $y \in candidates$ **with** *probability* $h(i,y) = \frac{T_i(y)}{\sum_{z \in candidates} T_i(z)}$
8 **send** $RCW\_MSG(s)$ *to* y

Fig. 2. RCW algorithm for connected networks. Code for node i.

used by any node to perform the samplings (as many as needed). The sampling process in the RCW algorithm works as follows. To start the process, the source s sends a message $RCW\_MSG(s)$ to itself. When the $RCW\_MSG(s)$ message is received by a node i from a node x, it computes a set of candidates for next hop in the RCW, which are all the neighbors of i except x. Then, the RCW stops and selects that node with a *stay probability* $q(i) = \frac{w(i)}{w(i)+\sum_{z \in candidates} T_i(z)}$ (Line 4). If the RCW does not select i, it jumps to a neighbor of i different from x. To do so, the RCW chooses only among nodes y in the set of candidates (that move away from s) using $h(i,y) = \frac{T_i(y)}{\sum_{z \in candidates} T_i(z)}$ as hop probability (Line 7).

Analysis. We show now that the algorithm proposed performs sampling with the desired probability distribution.

Theorem 3. *Each node* $i \in N$ *is selected by the RCW algorithm with probability* $p(i) = \frac{w(i)}{\eta}$.

Proof. If a node i receives the $RCW\_MSG(s)$ from x, let us define $candidates = neighbors(i) \setminus \{x\}$, and $T(i) = w(i) + \sum_{z \in candidates} T_i(z)$. We prove the following stronger claim: Each node $i \in N$ is visited by the RCW with probability $v(i) = \frac{T(i)}{\eta}$ and selected by the RCW algorithm with probability $p(i) = \frac{w(i)}{\eta}$.

We prove this claim by induction on the number of hops from the source s to node i in the spanning tree. The base case is when the node i is the source s. In this case x is also s, $candidates = neighbors(s)$, and $T(s) = \eta$. Hence, $v(s) = \frac{T(s)}{\eta} = 1$ and $q(s) = \frac{w(s)}{\eta}$, yielding $p(s) = \frac{w(s)}{\eta}$.

The induction hypothesis assumes the claim true for a node x at distance k from s, and proves the claim for i which is at distance $k+1$. We have that $\Pr[\text{visit } i] = v(x)\,(1-q(x))\,\frac{T(i)}{T(x)-w(x)}$, where $1-q(x)$ is the probability of not selecting node x when visiting it, and $\frac{T(i)}{T(x)-w(x)}$ is the probability of choosing the node i in the next hop of the RCW. The stay probability of x and i are $q(x) = w(x)/T(x)$ and $q(i) = w(i)/T(i)$, respectively (Line 4). Then, $v(i) = \frac{T(x)}{\eta}\left(1 - \frac{w(x)}{T(x)}\right)\frac{T(i)}{T(x)-w(x)} = \frac{T(x)}{\eta}\left(\frac{T(x)-w(x)}{T(x)}\right)\frac{T(i)}{T(x)-w(x)} = \frac{T(i)}{\eta}$ and $\Pr[\text{select } i] = v(i)q(i) = \frac{T(i)}{\eta}\frac{w(i)}{T(i)} = \frac{w(i)}{\eta}$.

4 Sampling in a Grid

If the algorithm for connected networks is applied to a grid, given its regular structure, the construction of the spanning tree could be done without any communication among nodes, but the weight aggregation process has to be done as before. However, we show in this section that all preprocessing and the state data stored in each node can be avoided if the probability distribution is based on the distance. RCW sampling process was described in Section 2, and we only redefine stay and hop probabilities. From Observation 1, the key for correctness is to assign stay and hop probabilities that guarantee visit and stay probabilities that are homogenous for all the nodes at the same distance from the source.

Stay Probability. For $k \in [0, R]$, the stay probability of every node $(i, j) \in \mathbb{R}_k$ is defined as

$$q_k = \frac{n_k \cdot p_k}{\sum_{j=k}^{R} n_j \cdot p_j} = \frac{n_k \cdot p_k}{1 - \sum_{j=0}^{k-1} n_j \cdot p_j}. \tag{2}$$

As required by Observation 1, all nodes in $\mathbb{R}_k$ have the same q_k. Note that $q_0 = p_0$ and $q_R = 1$, as one may expect. Since the value of p_k is known at $(i, j) \in \mathbb{R}_k$, n_k can be readily computed[4], and the value of $\sum_{j=0}^{k-1} n_j \cdot p_j$ can be piggybacked in the RCW, the value of q_k can be computed and used at (i, j) without requiring precomputation nor state data.

Hop Probability. In the grid, the hops of a RCW increase the distance from the source by one unit. We want to guarantee that the visiting probability is the same for each node at the same distance, to use Observation 1. To do so, we need to observe that nodes (i, j) over the axes (i.e., with $i = 0$ or $j = 0$) have to be treated as a special case, because they can only be reached via a single path, while the others nodes can be reached via several paths. To simplify the presentation, and since the grid is symmetric, we give the hop probabilities for one quadrant only (the one in which nodes have both coordinates non-negative). The hop probabilities in the other three quadrants are similar. The first hop of each RCW chooses one of the four links of the source node with the same probability 1/4. We have three cases when calculating the hop probabilities from a node (i, j) at distance k, $0 < k < R$, to node (i', j').

- *Case A:* The edge from (i, j) to (i', j') is in one axis (i.e., $i = i' = 0$ or $j = j' = 0$). The hop probability of this link is set to $h_k((i, j), (i', j')) = \frac{i+j}{i+j+1} = \frac{k}{k+1}$.
- *Case B:* The edge from (i, j) to (i', j') is not in the axes, $i' = i+1$, and $j' = j$. The hop probability of this link is set to $h_k((i, j), (i + 1, j)) = \frac{2i+1}{2(i+j+1)} = \frac{2i+1}{2(k+1)}$.
- *Case C:* The edge from (i, j) to (i', j') is not in the axes, $i' = i$, and $j' = j+1$. The hop probability of this link is set to $h_k((i, j), (i, j + 1)) = \frac{2j+1}{2(i+j+1)} = \frac{2j+1}{2(k+1)}$.

It is easy to check that the hop probabilities of a node add up to one.

[4] $n_0 = 1$, while $n_k = 4k$ for $k \in [1, R]$.

Analysis. In the following we prove that the RCW that uses the above stay and hop probabilities selects nodes with the desired sample probability.

Lemma 1. *All nodes at the same distance $k \geq 0$ to the source have the same visit probability v_k.*

Proof. The proof uses induction. The base case is $k = 0$, and obviously $v_k = 1$. When $k = 1$, the probability of visiting each of the four nodes at distance 1 from the source s is $v_i = \frac{1-q_0}{4}$, where $1 - q_0$ is the probability of not staying at source node. Assuming that all nodes at distance $k > 0$ have the same visit probability v_k, we prove the case of distance $k + 1$. Recall that the stay probability is the same q_k for all nodes at distance k.

The probability to visit a node $x = (i', j')$ at distance $k + 1$ depends on whether x is on an axis or not. If it is in one axis it can only be reached from its only neighbor (i, j) at distance k. This happens with probability (case A) $\Pr[\text{visit } x] = v_k(1 - q_k)\frac{i+j}{i+j+1} = v_k(1 - q_k)\frac{k}{k+1}$. If x is not on an axis, it can be reached from two nodes, $(i' - 1, j')$ and $(i', j' - 1)$, at distance k (Cases B and C). Hence, the probability of reaching x is then $\Pr[\text{visit } x] = v_k(1 - q_k)\frac{2(i'-1)+1}{2(i'+j')} + v_k(1 - q_k)\frac{2(j'-1)+1}{2(i'+j')} = v_k(1 - q_k)\frac{k}{k+1}$. Hence, in both cases the visit probability of a node x at distance $k+1$ is $v_{k+1} = v_k(1 - q_k)\frac{k}{k+1}$. This proves the induction and the claim.

Theorem 4. *Every node at distance $k \in [0, R]$ from the source is selected with probability p_k.*

Proof. If a node is visited at distance k, it is because no node was selected at distance less than k, since a RCW always moves away from the source. Hence, $\Pr[\exists x \in \mathbb{R}_k \text{ visited}] = 1 - \sum_{j=0}^{k-1} n_j p_j$. Since all the n_k nodes in $\mathbb{R}_k$ have the same probability to be visited (from the previous lemma), we have that $v_k = \frac{1-\sum_{j=0}^{k-1} n_j p_j}{n_k}$. Now, since all the n_k nodes in $\mathbb{R}_k$ have the same stay probability is q_k, the probability of selecting a particular node x at distance k from the source is $\Pr[\text{select } x] = v_k q_k = \frac{1-\sum_{j=0}^{k-1} n_j p_j}{n_k} \frac{n_k p_k}{\sum_{j=k}^{R} n_j p_j} = p_k$, where it has been used that $(1 - \sum_{j=0}^{k-1} n_j p_j) = \sum_{j=k}^{R} n_j p_j$.

5 Sampling in a Concentric Rings Network with Uniform Connectivity

In this section we derive a RCW algorithm to sample a concentric rings network with uniform connectivity, where all preprocessing is avoided, and only a small (and constant) amount of data is stored in each node. Recall that uniform connectivity means that all nodes of ring k have the same number of neighbors δ_k in ring $k + 1$ and the same number of neighbors γ_k in ring $k - 1$.

```
1  task RCW(x, k, δ_k, γ_k, p_k)
2  when RCW_MSG(s, v_{k-1}, p_{k-1}, n_{k-1}, δ_{k-1}) received:
3     n_k ← n_{k-1} δ_{k-1}/γ_k;   v_k ← n_{k-1} (v_{k-1} − p_{k-1})/n_k;   q_k ← p_k/v_k
4     with probability q_k do select node x and report to s
5     otherwise
6        choose a neighbor y in ring k + 1 with uniform probability
7        send RCW_MSG(s, v_k, p_k, n_k, δ_k) to y
```

Fig. 3. RCW algorithm for concentric rings with uniform connectivity (code for node $x \in \mathbb{R}_k$, $k > 0$)

Distributed Algorithm. The general behavior of the RCW algorithm for these networks was described in Section 2. In order to guarantee that the algorithm is fully distributed, and to reduce the amount of data a node must know a priori, a node at distance k that sends the RCW to a node in ring $k+1$ piggybacks some information. More in detail, when a node in ring k receives the RCW from a node of ring $k-1$, it also receives the probability v_{k-1} of the previous step, and the values p_{k-1}, n_{k-1}, and δ_{k-1}. Then, it calculates the values of n_k, v_k, and q_k. After that, the RCW algorithm uses the stay probability q_k to decide whether to select the node or not. If it decides not to select it, it chooses a neighbor in ring $k+1$ with uniform probability. Then, it sends to this node the probability v_k and the values p_k, n_k, and δ_k, piggybacked in the RCW.

The RCW algorithm works as follows. The source s selects itself with probability $q_0 = p_0$. If it does not do so, it chooses one node in ring 1 with uniform probability, and sends it the RCW message with values $v_0 = 1$, $n_0 = 1$, p_0, and δ_0. Figure 3 shows the code of the RCW algorithm for nodes in rings $\mathbb{R}_k$ for $k > 0$. Each node in ring k must only know initially the values δ_k, γ_k and p_k. Observe that n_k (number of nodes in ring k) can be locally calculated as $n_k = n_{k-1}\delta_{k-1}/\gamma_k$. The correctness of this computation follows from the uniform connectivity assumption (Eq. 1).

Analysis. The uniform connectivity property can be used to prove by induction that all nodes in the same ring k have the same probability v_k to be reached. The stay probability q_k is defined as $q_k = p_k/v_k$. Then, from Observation 1, the probability of selecting a node x of ring k is $p_k = v_k q_k$. What is left to prove is that the value v_k computed in Figure 3 is in fact the visit probability of a node in ring k.

Lemma 2. *The values v_k computed in Figure 3 are the correct visit probabilities.*

Proof. Let us use induction. For $k = 1$ the visit probability of a node x in ring $\mathbb{R}_1$ is $\frac{1-q_0}{n_1} = \frac{1-p_0}{n_1}$. On the other hand, when a message RCW_MSG reaches x, it carries $v_0 = 1$, $n_0 = 1$, p_0, and δ_0 (Line 2). Then, v_1 is computed as $v_1 = n_0 \frac{v_0 - p_0}{n_1} = \frac{1-p_0}{n_1}$ (Line 3). For a general $k > 1$, assume the value v_{k-1} is the correct visit probability of a node in ring $k-1$. The visit probability of a node in ring k is $v_{k-1} n_{k-1}(1-q_{k-1})/n_k$, which replacing $q_{k-1} = p_{k-1}/v_{k-1}$ yields the expression used in Figure 3 to compute v_k (Line 3).

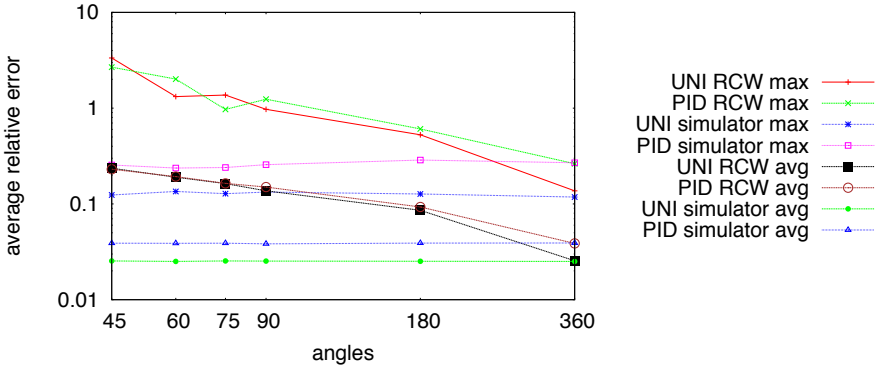


Fig. 4. UNI and PID scenarios without uniform connectivity

The above lemma, together with the previous reasoning, proves the following.

Theorem 5. *Every node at distance k of the source is selected with probability p_k.*

6 Concentric Rings Networks without Uniform Connectivity

Finally, we are interested in evaluating, by means of simulations, the performance of the RCW algorithm for concentric rings with uniform connectivity when it is used on a more realistic topology: a concentric rings network *without uniform connectivity*. The experiment has been done in a concentric rings topology of 100 rings with 100 nodes per ring, and it places the nodes of each ring uniformly *at random* on each ring. This deployment does not guarantee uniform connectivity. Instead, the nodes' degrees follow roughly a normal probability distribution. In order to establish the connectivity of nodes, we do a geometric deployment. A node x in ring k is assigned a position in the ring. This position can be given by an angle α. Then, each network studied will have associated a connectivity angle β, the same for all nodes. This means that x will be connected to all the nodes in rings $k-1$ and $k+1$ whose position (angle) is in the interval $[\alpha - \beta/2, \alpha + \beta/2]$ (see Figure 7 in [19]). Observe that the bigger the angle β is, the more neighbors x has in rings $k-1$ and $k+1$. We compare the relative error of the RCW algorithm when sampling with two distributions: the uniform distribution (UNI) and a distribution proportional to the inverse of the distance (PID). We define the relative error e_i for a node x in a collection C of s samples as $e_i = \frac{|fsim_x - f_x|}{f_x}$, where $fsim_x$ is the number of instances of x in collection C obtained by the simulator, and $f_x = p_x \cdot s$ is the expected number of instances of x with the ideal probability distribution (UNI or PID). We compare the error of the RCW algorithm with the error of a generator of pseudorandom numbers. For each configuration, a collection of 10^7 samples has been done.

Figure 4 presents the results obtained in the UNI and PID scenarios. In both cases, we can see that the RCW algorithm performs much worse than the UNI

```
1  function AssignAttachmentPoints(x, k)
2  ap ← LCM(n_k, n_{k+1}) / n_k
3  C ← N_{k+1}(x) /* neighbors of x in ring k + 1 */
4  A_x ← ∅ /* A_x is a multiset */
5  loop
6    choose c from C
7    send ATTACH_MSG to c
8    receive RESPONSE_MSG from c
9    if RESPONSE_MSG = OK then
10     ap ← ap − 1
11     add c to A_x /* c can be in A_x several times */
12   else C ← C \ {c}
13 until (ap = 0) ∨ (C = ∅)
14 if (ap = 0) then return A_x
15 else return FAILURE
```

Angle	% success
15 °	0%
30 °	0%
45 °	3%
60 °	82%
75 °	99%
90 °	100%
150 °	100%
180 °	100%
360 °	100%

Fig. 5. Assignment Attachment Points (AAP) Function (left side). Success rate of the AAP algorithm as a function of the connectivity angle (right side).

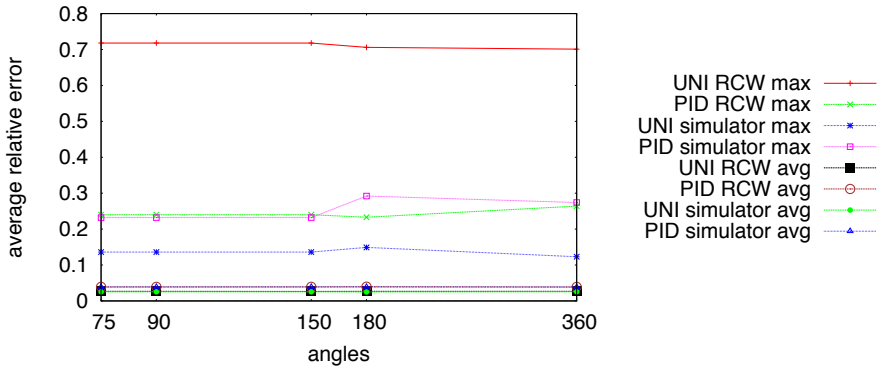


Fig. 6. UNI and PID scenarios without uniform connectivity, using the AAP algorithm

and PID simulators. The simulation results show a biased behavior of RCW algorithm because the condition of Eq. 1 is not fulfilled in this experiment (i.e. a node has no neighbors, or there are two nodes in a ring k that have different number of neighbors in rings $k-1$ or $k+1$).

Assignment Attachment Points (AAP) Algorithm. To eliminate the errors observed when there is no uniform connectivity, we propose a simple algorithm to transform the concentric rings network without uniform connectivity into an overlay network with uniform connectivity.

To preserve the property that the visit probability is the same for all the nodes in a ring, nodes will use different probabilities for different neighbors. Instead of explicitly computing the probability for each neighbor, we will use the following process. Consider rings k and $k+1$. Let $r = LCM(n_k, n_{k+1})$, where LCM is the *least common multiple* function. We assign $\frac{r}{n_k}$ *attachment points* to each node

in ring k, and $\frac{r}{n_{k+1}}$ attachment points to each node in ring $k+1$. Now, the problem is to connect each attachment point in ring k to a different attachment point in ring $k+1$ (not necessarily in different nodes). If this can be done, we can use the algorithm of Figure 3, but when a RCW is sent to the next ring, an attachment point (instead of a neighbor) is chosen uniformly. Since the number of attachments points is the same in all nodes of ring k and in all nodes of ring $k+1$, the impact in the visit probability is that it is again the same for all nodes of a ring.

The connection between attachment points can be done with the simple algorithm presented in Figure 5, in which a node x in ring k contacts its neighbors to request available attachment points. If a neighbor that is contacted has some free attachment point, it replies with a response message *RESPONSE_MSG* with value *OK*, accepting the connection. Otherwise it replies to x notifying that all its attachment points have been connected. The node x continues trying until its $\frac{r}{n_k}$ attachment points have been connected or none of its neighbors has available attachment points. If this latter situation arises, then the process failed. The algorithm finishes in $O(\max_k\{n_k\})$ communication rounds. (Note that $r \leq n_k \cdot n_{k+1}$ and $|C| \leq n_{k+1}$). Combining these results with the analysis of Section 5, we can conclude with the following theorem.

Theorem 6. *Using attachment points instead of links and the distributed RCW-based algorithm of Figure 3, it is possible to sample a concentric rings network without uniform connectivity with any desired distance-based probability distribution p_k, provided that the algorithm of Figure 5 completes (is successful) in all the nodes.*

Figure 6 shows the results when using the AAP algorithm. As we can see, the differences have disappeared. The conclusion is that, when nodes are placed uniformly at random and AAP is used to attach neighbors to each node, RCW performs as good as perfect UNI or PID simulators.

In general, the algorithm of Figure 5 may not be succesful. It is shown in the table of Figure 5 (right side) the success rate of the algorithm for different connectivity angles. It can be observed that the success rate is large as long as the connectivity angles are not very small (at least 60°). (For an angle of 60° the expected number of neighbors in the next ring for each node is less than 17.) For small angles, like 15° and 30°, the AAP algorithm is never successful. For these cases, the algorithm for connected network presented in Section 3 can be used.

7 Conclusions

In this paper we propose distributed algorithms for node sampling in networks. All the proposed algorithms are based on a new class of random walks called centrifugal random walks. These algorithms guarantee that the sampling end after a number of hops upper bounded by the diameter of the network, and it samples with the exact probability distribution. As future works we want to explore

sampling in dynamic networks using random centrifugal walks. Additionally, we will investigate a more general algorithm that would also concern distributions that do not only depend on the distance from the source.

References

1. Awan, A., Ferreira, R.A., Jagannathan, S., Grama, A.: Distributed uniform sampling in unstructured peer-to-peer networks. In: HICSS. IEEE CS (2006)
2. Bertier, M., Bonnet, F., Kermarrec, A.M., Leroy, V., Peri, S., Raynal, M.: D2HT: The best of both worlds, integrating RPS and DHT. In: EDCC. IEEE CS (2010)
3. Bonnet, F., Kermarrec, A.-M., Raynal, M.: Small-World Networks: From Theoretical Bounds to Practical Systems. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 372–385. Springer, Heidelberg (2007)
4. Bui, M., Butelle, F., Lavault, C.: A distributed algorithm for constructing a minimum diameter spanning tree. J. Parallel Distrib. Comput. (May 2004)
5. Busnel, Y., Beraldi, R., Baldoni, R.: On the uniformity of peer sampling based on view shuffling. Journal of Parallel and Distributed Computing (2011)
6. Elkin, M.: A faster distributed protocol for constructing a minimum spanning tree. In: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, Philadelphia, PA, USA (2004)
7. Fraigniaud, P., Giakkoupis, G.: On the searchability of small-world networks with arbitrary underlying structure. In: Schulman, L.J. (ed.) STOC. ACM (2010)
8. Gfeller, B., Santoro, N., Widmayer, P.: A distributed algorithm for finding all best swap edges of a minimum-diameter spanning tree. IEEE Trans. Dependable Secur. Comput. (January 2011)
9. Gjoka, M., Kurant, M., Butts, C.T., Markopoulou, A.: Walking in facebook: A case study of unbiased sampling of osns. In: INFOCOM, pp. 2498–2506. IEEE (2010)
10. Gjoka, M., Kurant, M., Butts, C.T., Markopoulou, A.: Practical recommendations on crawling online social networks. IEEE Journal on Selected Areas in Communications (October 2011)
11. Gurevich, M., Keidar, I.: Correctness of gossip-based membership under message loss. SIAM J. Comput. 39(8) (December 2010)
12. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., van Steen, M.: Gossip-based peer sampling. ACM Trans. Comput. Syst. 25(3) (2007)
13. Kempe, D., Kleinberg, J.M., Demers, A.J.: Spatial gossip and resource location protocols. J. ACM 51(6), 943–967 (2004)
14. Kleinberg, J.M.: Navigation in a small world. Nature 406(6798) (August 2000)
15. Lee, C.-H., Xu, X., Eun, D.Y.: Beyond random walk and metropolis-hastings samplers: why you should not backtrack for unbiased graph sampling. In: SIGMETRICS 2012. ACM (2012)
16. Milić, D., Braun, T.: Netice9: A stable landmark-less network positioning system. In: 2010 IEEE 35th Conference on Local Computer Networks (October 2010)
17. Sevilla, A., Mozo, A., Lorenzo, M.A., López-Presa, J.L., Manzano, P., Fernández Anta, A.: Biased Selection for Building Small-World Networks. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 32–47. Springer, Heidelberg (2010)
18. Zhong, M., Shen, K.: Random walk based node sampling in self-organizing networks. SIGOPS Oper. Syst. Rev. 40, 49–55 (2006)
19. Sevilla, A., Mozo, A., Fernández Anta, A.: Node Sampling using Random Centrifugal Walks. CoRR, abs/1107.1089, version 3 (2012)

Physarum-Inspired Self-biased Walkers for Distributed Clustering

Devan Sohier[1], Giorgos Georgiadis[2], Simon Clavière[1], Marina Papatriantafilou[2], and Alain Bui[1]

[1] Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden
{georgiog,ptrianta}@chalmers.se
[2] Laboratoire PRiSM (UMR CNRS 8144), Université de Versailles St-Quentin-en-Yvelines, 78035 Versailles, France
{devan.sohier,simon.claviere,alain.bui}@prism.uvsq.fr

Abstract. We propose a distributed scheme to compute distance-based clusters. We first present a mechanism based on the flow of distributed tokens called walkers, circulating randomly between a source and a sink to compute a shortest path. Each time a walker takes an edge, it reinforces the probability that subsequent walkers take it. This mechanism is a discrete emulation of the slime mould (*Physarum polycephalum*) dynamics presented in [16]: each node observes the flow of walkers going through each adjacent edge and uses this flow to compute the probabilities with which it sends the walkers through each edge. Then, based on this mechanism, we show how several sources compute a shortest path DAG to a given sink. Finally, given some clusterheads acting like sinks, we show that this process converges to distance-based clusters (i.e. nodes join the clusterhead to which they are closest) with shortest-path DAGs. The algorithm is designed with a special focus on dynamic networks: the flow locally adapts to the appearance and disappearance of links and nodes, including clusterheads.

1 Introduction

We focus on overlay construction over networks, in a way that nodes are clustered around specified clusterheads. Clustering can facilitate locality and scalability properties in a variety of networks – wireless, ad-hoc, sensor networks – for services such as routing, data aggregation, resource finding and sharing in neighborhoods. The latter is instrumental for realizing interest-based groups and facilitating grouping based on gradient proximity measures [27]. When it comes to resource sharing, load balancing is aligned with clustering; it forms an incentive to achieve and a way to enforce fairness and group participation in collaboration.

In particular the overlay we are aiming for here should define for each node efficient ways of reaching the closest clusterhead through shortest paths (using an application related distance metric); moreover, nodes at equal distances from

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 315–329, 2012.

several clusterheads should connect to all of them, thus also forming bridges among clusters. We are interested in methods that do not need global knowledge of the network and have strong locality and self-organizing properties and small, lightweight messages [21].

Towards this goal one of the basic instruments will be based on an adaptation of *random walks* we call self-biased walkers. Random walks have been shown to be effective in the searching, dissemination and construction of overlay networks as a statistical process that associates with important parameters and properties of the corresponding network [18]. We will show that the proposed walkers adaptively develop bias towards corresponding shortest paths and can effectively simulate a process of natural computation, in particular the slime mould *Physarum polycephalum*. We will further show how to use this basic form of computation towards achieving the aforementioned goal of this work.

In their seminal work, Nakagaki et al [25] found that they could use the slime mould *Physarum polycephalum* in order to solve simple mazes, by placing food at the exits, introducing the mould into the maze and letting it reach the food using extensible tube-like protuberances (pseudopodia). The mould not only solved the maze but did so efficiently, by forming shortest paths between the entry and the exit points of the maze. In doing so, the mould changes the diameter of its pseudopodia, depending on whether the particular protuberance is vital in the nutrient circulation. Based on these works and with further experimentations, researchers succeeded in reproducing the transportation networks of the Tokyo metropolitan area [26] and countries such as Brazil [15] by placing food in population centers and having the mould connect them.

The mechanism through which the mould achieves the quality of its solutions remains an open research question. Several authors [25,23,24,26] suggested an electric circuit model, where nodes are connected through resistors with a diameter property that can affect the amount of current that passes through them. This diameter corresponds roughly to the pseudopodia diameter of the mould and tends to follow the electric current flow: if the flow becomes greater the diameter increases, otherwise it decreases. For the exact relation between diameter D_{ij} and current Q_{ij} passing through a resistor (i, j) though, several mechanisms have been proposed, for example

$$\text{[23]}\ \frac{dD_{ij}}{dt} = f\left(|Q_{ij}|\right) - D_{ij} \quad (1) \qquad \text{[24]}\ \frac{dD_{ij}}{dt} = |Q_{ij}| - D_{ij} \quad (3)$$

$$\text{[25]}\ \frac{dD_{ij}}{dt} = |Q_{ij}|^{\mu} - aD_{ij} \quad (2) \qquad \text{[26]}\ \frac{dD_{ij}}{dt} = \frac{|Q_{ij}|^{\gamma}}{1+|Q_{ij}|^{\gamma}} - D_{ij} \quad (4)$$

which lead to different dynamics. It has been conjectured that the usage of these dynamics can lead to efficient algorithms in applications that involve shortest paths and similar transportation problems. To our knowledge, the distributed algorithm presented here is the first such algorithm.

Related work on Physarum. In the Physarum-related literature, a number of different dynamics have been explored to explain the mould's behavior, focusing on the different forms of function $f(\cdot)$ of equation 1 [26,19,23,16]. Specifically Ito et al [19] use the equation 1 for the diameter dynamics and prove that their

model is able to solve transportation problems, while the diameters converge exponentially to their final values. Miyaji and Onishi [24] use equation 1 and prove analytically that the dynamics converge for planar graphs with a unique shortest path between source and sink nodes, to a flow that uses only the shortest path. Bonifaci et al [16] move one step further by proving that dynamics based on equation 1 similarly converge to the unique shortest path for all graphs, provided a unique short path between source and sink exists. Furthermore, they show that in the generic case of multiple sources and sinks, the dynamics are attracted to a set of equilibria that are candidate solutions for the transportation problem. Recently the Physarum dynamics have been proposed to address common computer science problems such as the work of Johannson and Zou [20] on Linear Programming problems and Li et al [22] on routing protocols for sensor networks: the first proposes a method to encode any linear program into the physarum dynamics and solve it efficiently, while the second adapts two Physarum mechanisms, path growth and path evolution, to efficiently route messages in a wireless sensor network.

Related work on clustering. A number of methods have been proposed in the literature to construct clusters in unstructured networks. For example, the algorithms of [2,4,8,9] produce clusters with radius one. Each cluster has a node called a *clusterhead* and all other nodes in that cluster are neighbors of the clusterhead. Clusters can be built using a hierarchical method: the clustering algorithm is iterated on the overlay network obtained by considering clusters as nodes, until a single cluster is obtained [7,10,12]. On the other hand, the solution given in [1] builds k-hop clusters (clusters of radius k) and in [6], a self-stabilizing[1] $O(n)$-time algorithm is given for computing a minimal k-dominating set; this set can then be used as the set of clusterheads for a k-clustering. Other clustering algorithms are based on random walks [3,5]. In [5] the clusters are built around bounded-size connected dominating sets, while in [3] the algorithm recursively breaks the network into two clusters as long as every cluster size satisfies a lower bound.

These solutions are bottom-up processes while we propose a top-down approach [11] by relying on predefined clusterheads. We then compute clusters consisting of the nodes that are closer to a given clusterhead than to any other. Our algorithm has a focus on topological changes: the use of a flow-based process allows the system to reconfigure after a topological change without affecting nodes of other clusters. In particular, a topological change can affect the cluster in which it occurs and possibly neighboring clusters, but not further nodes.

To our knowledge, the present paper is the first work that uses the Physarum dynamics together with a discrete probabilistic tool such as self-biased walks in order to solve a distributed problem, namely node clustering.

Outline. In the algorithm presented here we show how adaptive self-biased walkers can emulate the dynamics of the Physarum. In section 2 we introduce the

[1] A self-stabilizing system is a system that eventually recovers a normal behavior after a transient fault.

necessary notations and definitions. We then proceed by showing the algorithm for the walkers and furthermore how to use the method in order to enable each node in a network, in a distributed way, to determine the direction towards its closest clusterhead (using a distance metric noted by L in the following) (section 3). In section 4 we study experimentally the behavior of the algorithm in terms of resulting cluster computation, reaction to changes in the graph and cost. We complete our presentation with our conclusions and future work in section 5.

2 Definitions and Notation

We consider a distributed system, consisting of nodes with computation capabilities and communication edges between them, in the form of an undirected graph $G = (V, E)$, $|V| = n, |E| = m$. Nodes have no id and the system is anonymous. We treat synchronous systems and c-asynchronous, i.e. *(i)* in a round, all processors receive all pending messages and treat them and *(ii)* in a round for a given node, no node has executed more than c rounds. Each edge (i, j) has also a length $L_{ij} > 0$ that we assume is known to both endnodes and a time-varying diameter $D_{ij}(t) > 0$.

A discrete time biased walk on a graph with a bias function C (here, the conductance) is a process by which a node is chosen at each step according to the following rule: if node i holds the walker at step t, then any neighbor j of i will hold it at step $t + 1$ with probability $P_{ij} = \frac{C_{ij}}{\sum_{k \in \Gamma_i} C_{ik}}$, where Γ_i is the set of immediate neighbors of node i. Such a process can easily be implemented in a distributed system.

In their seminal work, Doyle and Snell [17] connected random walks on weighted undirected graphs and electric circuits, by showing that a random walker behaves in a way similar to the simplistic view of electrical current as movement of electrons, provided every edge (i, j) is treated as a resistor with *conductance* $C_{ij}(t)$[2] equal to the edge's weight. In fact they showed that, when a unit current is injected into the graph, the *electrical current* $Q_{ij}(t)$ flowing over a resistor (i, j) corresponds to the mean number of walker crossings over the respective edge. Recent work showed that it is possible to use this paradigm in order to form and study overlay networks in an efficient way [28,29]. These results imply that the electrical current flow can be emulated by counting the mean number of walker crossings of an edge and be used to form a clustering overlay.

The connection between the physarum dynamics and electrical networks, as well as between electrical networks and random walks, is the basis for the implementation presented in this paper.

In the rest of the paper we define the conductance $C_{ij}(t)$ of an edge (i, j) to be equal to $C_{ij}(t) = \frac{D_{ij}(t)}{L_{ij}}$. The transition probability of the biased walker from node i to node j now becomes $P_{ij}(t) = \frac{C_{ij}(t)}{\sum_{k \in \Gamma_i} C_{ik}(t)}$. The mould dynamics

[2] Note that all electrical measures are necessarily time-varying in order to simulate the Physarum dynamics.

are defined by the evolution of the edge diameters of the network and here we follow a discretization of the definition of Bonifaci et al [16]:

$$\begin{aligned} D_{ij}(t+1) &= D_{ij}(t) + \varepsilon(|Q_{ij}(t)| - D_{ij}(t)) \\ &= (1-\varepsilon)D_{ij}(t) + \varepsilon|Q_{ij}(t)| \end{aligned} \tag{5}$$

3 Distributed Emulation of Physarum Dynamics: PECAN Algorithm

The PECAN (*Physarum walkErs Clustering*) algorithm we present here is based on monitoring the flow of walkers on each edge and then adjusting probabilities accordingly. This flow is used as an estimation of the electrical current flowing on edges and the diameter D is updated accordingly.

The algorithm assumes there is a unit flow of walkers that are created at nodes called *sources* and destroyed at nodes called *sinks*. At each round, every node counts the number of walkers going in and out each of its adjacent edges and updates its flow information. Instant flow, meaning the flow measured during a round, may vary abruptly from round to round, due to the discrete stochastic nature of the system. To avoid sharp changes, we take into account past observations. However, to allow the system to evolve with newer observations, we measure the flow with a discount factor applied to past observations. In practice the flow estimation Q is computed as

$$Q_{ij}(t+1) = \alpha Q_{ij}(t) + (1-\alpha)\#w_{ij}(t) \tag{6}$$

where $\#w_{ij}(t)$ is the observed flow, i.e. the number of walkers that have crossed (i,j) in this direction at round t minus the number that have crossed in the opposite direction and $\alpha \in [0,1]$ a parameter. This formula is such that if $\#w_{ij}$ eventually reaches a probability distribution with an average, Q_{ij} stabilizes to this average.

Equation 5 expresses the fact that $D_{ij}(t+1)$ is a weighted average of $D_{ij}(t)$ and $|Q_{ij}(t)|$, and equation 6 that $Q_{ij}(t+1)$ itself is a weighted average of $Q_{ij}(t)$ and $\#w_{ij}(t)$. Asymptotically, the sign of Q should stabilize or Q tends to 0. Thus, asymptotically, $D_{ij}(t+1)$ will be a weighted average of $D_{ij}(t)$ and $\#w_{ij}(t)$. By taking $\alpha' = (1-\varepsilon)\alpha$, the diameter D_{ij} then follows asymptotically equation 6. In the following we take $D_{ij}(t+1) = |Q_{ij}(t)|$ and the transition probabilities are then:

$$P_{ij}(t+1) = \frac{|Q_{ij}(t)|/L_{ij}}{\sum_{k\in\Gamma_i} |Q_{ik}(t)|/L_{ik}} \tag{7}$$

As we show in the following lemma, the dynamics converge to the same point as the one of Bonifaci et al [16] in the case of single source and single sink, namely by setting the diameter of edges on shortest paths equal to 1, as well as 0 to all the others (cf [30] for the full proof).

Lemma 1. *If there is a unique shortest path from source to sink and the dynamics stabilize, the diameter of edges on the shortest path converge to 1 while those not on the shortest path to 0.*

The α parameter has a strong influence on the behavior of the system: a conservative α (close to 1) slows down the convergence, while a small α may make the system converge to a suboptimal solution (which a slight modification of the system dynamics can solve), or entail sharp changes in the values of Q_{ij} even after the system seems to have converged, thus making the solution *"unstable"*.

Any path between the source and the sink is a fixed point of these dynamics (even if only shortest paths are attractive): if Q_{ij} is set to 1 on all edges of a non-shortest path from the source to the sink and to 0 everywhere else, then the system does not evolve anymore. However, such a solution is an *"unstable equilibrium"*, as shown by [16] in the continuous case. To solve these situations, we add a β parameter, the role of which is to prevent the system from converging fully. The β parameter is a lower bound on the probability that a node sends the walker across an adjacent edge. Thus, if close to a non-optimal (repulsive) solution, the system will be driven away from it. If close to an optimal (attractive) solution, the system will tend to get closer to it, even if β hinders this progression.

The β parameter also allows to initialize the system with a null flow, by setting Q_{ij} to 0 for all edges on all nodes. The algorithm presented below then ensures that Q_{ij} is really a flow:

- for any nodes i and j, $Q_{ij} = -Q_{ji}$
- for any node i that is not a source or a sink, if it does not hold a token in this round, $\sum_j Q_{ij} = 0$
- for a source (*resp.* a sink), $\sum_j Q_{ij}$ tends to 1 (*resp.* minus the number of sources).

In order to build clusters, all clusterheads will be sinks and all nodes willing to join a cluster will be sources.

3.1 Monitoring the Flow and Routing Walkers

A node i has one variable Q_{ij} for each adjacent edge (i, j). This variable is used to monitor the flow. Each time a walker comes in through a given edge (i, j), Q_{ij} is increased and each time a walker comes out through it, it is decreased. At each round, the value of Q_{ij} is scaled down.

The only type of messages we use is a $Walker$ message, with no content, and there are three types of nodes:

- *sinks*, that at each round update their flow information with equation 6 according to the number of received walkers and subsequently delete them;
- *ordinary nodes*, that at each round update their flow information and forward received walkers;
- *sources*, which act as ordinary nodes but send an extra token per round.

Depending on the choice of sources and sinks, this algorithm may be used to compute various related distributed structures: shortest paths, shortest path DAGs and distance-based clusters. We detail those properties in the next section.

Procedure 1. Upon a *Walker* reception from j

$w_{in}[j] \leftarrow w_{in}[j] + 1$

Procedure 2. At each round, on an ordinary node (*resp.* source node)

$w_{in} \leftarrow 0$ (*resp.* 1)
for *any adjacent node* j **do**
  $w_{out}[j] \leftarrow 0$
  $w_{in} \leftarrow w_{in} + w_{in}[j]$
  $Q_{ij} \leftarrow Q_{ij} - \alpha(1-\alpha)w_{in}[j]$
  $Q_{ij} \leftarrow \alpha Q_{ij}$
for $i = 0$ *to* w_{in} **do**
  `Choose a neighbor` k `at random according to` Q/L
  `Send` *Walker* `to` k
  $w_{out}[k] \leftarrow w_{out}[k] + 1$
for *any adjacent node* j **do**
  $Q_{ij} \leftarrow Q_{ij} + (1-\alpha)w_{out}[j]$

Procedure 3. At each round, on a sink node

for *any adjacent node* j **do**
  $Q_{ij} \leftarrow Q_{ij} - \alpha(1-\alpha)w_{in}[j]$
  $Q_{ij} \leftarrow \alpha Q_{ij}$

Procedure 4. Choose a neighbor at random according to Q/L

$sum \leftarrow 0$
for *each adjacent node* j **do**
  if $Q_{ij}/L_{ij} > \beta$ **then**
    $sum \leftarrow sum + Q_{ij}/L_{ij}$
  else
    $sum \leftarrow sum + \beta$
$v \leftarrow$ `random value in` $[0, sum]$
$j \leftarrow$ `first neighbor of` i
$sum \leftarrow Q_{ij}/L_{ij}$
while $sum < v$ **do**
  $j \leftarrow$ `next neighbor of` i
  if $Q_{ij}/L_{ij} > \beta$ **then**
    $sum \leftarrow sum + Q_{ij}/L_{ij}$
  else
    $sum \leftarrow sum + \beta$
return(j)

When receiving a walker, a node updates the weight of the edge through which it has come. Then, if it is not a sink, it chooses a neighbor at random according to Q/L, sends the walker to it and updates the out-edge value (procedures 2, 4 and 1).

Note that choosing a neighbor according to Q/L consists of i choosing a random neighbor j with a probability proportional to $\frac{Q_{ij}}{L_{ij}}$. If this probability $\frac{Q_{ij}}{L_{ij}}$ is smaller than a parameter β we replace it by β, thus a neighbor has always a positive probability of being chosen.

Ordinary nodes, at each round, forward the received walkers and update their variables. The array w_{in} (resp. w_{out}) stores the number of walkers received (resp. sent) on each adjacent edge, so as to update the outgoing flow after the walkers have been sent. Additionally, sources create a new walker at each round and send it: they act as ordinary nodes, except for the computation of w_{in}, to which 1 is added to account for the extra token that the source creates during the round (procedure 2). On the other hand, sinks delete all received tokens but they keep track of the incoming flow (procedure 3).

Note that messages are received one step after they have been sent, so that Q_{ji} has been multiplied by α before i receives the walker sent by j. Thus, to maintain the symmetry $Q_{ij} = -Q_{ji}, \forall (i,j)$ and for the flow to be valid, we are led to subtract $\alpha(1-\alpha)$. On the other hand, note that when a node sends several messages to the same neighbor in a given round, these walkers can be replaced by a single weighted walker, at the expense of introducing a content in the walkers. We ran the algorithm in both settings and called the messages in the latter setting *aggregated messages*.

The above algorithm has two parameters, α and β. The α parameter is used to *"smoothen"* the flow evaluation. The β parameter ensures that all edges will be used, which is necessary for the system not to get *"trapped"* in a suboptimal solution (non shortest paths to the sink with weight 1, which are fixed points of the system dynamics, but repulsive ones). In particular, if topological changes occur, the β parameter allows walkers to visit new and possibly better paths.

3.2 Using Flows to Solve Clustering

When several nodes produce walkers, all of them compute shortest paths to the single sink. The optimal substructure property of the shortest paths problem [13] ensures that the process actually converges to a shortest path DAG. Indeed, for a given sink, starting from the point where they meet, two shortest paths from two different sources to the same sink are the same (or, if shortest paths are not unique, can be the same).

When several sinks are present the flows do not distinguish between them. The system acts as if they were a unique sink, consisting in the merging of all sinks, and the shortest paths computation leads to flow running along the shortest path to the closest sink.

Consider now a system in which some clusterheads behave as sinks and nodes seeking clusters behave like sources. After some time, the algorithm will converge to flows circulating along shortest paths from sources to sinks. Thus, nodes sending flow to a given sink are closer to this sink than to any other and have to join this cluster. In this way, any node will be able to route a message to its sink, by sending the message along its edge with the strongest outgoing flow.

Moreover, the flow brings some extra information: since all sources generate a unit flow, the flow that a sink receives is the number of nodes that are members of its cluster. Similarly, suppose that the shortest paths are unique. Then the flow coming from an edge on a node indicates to this node the size of the corresponding subtree and all nodes have a local knowledge of the shortest path tree of their cluster. If the shortest paths are not unique they define a DAG and the flow indicates the number of descendants on this side of the DAG, possibly counted partially as descendants of several nodes.

The clusterhead can then launch a new phase of the algorithm and use the DAG to inform all nodes of its presence if required, allowing to meet the classical clustering specification.

3.3 *c*-Asynchronous Case

Synchronicity and round-based distributed processing of messages is a strong assumption. We can relax it to c-asynchronicity, meaning that in a round for a given node, no node has executed more than c rounds.

With respect to slower nodes, a fast source may appear as generating up to c walkers per round. This output cannot be distinguished from a situation where the faster node is replaced with a central node connected to $c-1$ other nodes in a star formation. The computations of other nodes will run alike and lead to shortest paths to these nodes. Since the shortest paths to these virtual nodes all go through the central node, the computed shortest paths will be consistent.

A fast ordinary node, or a fast sink, will output the right number of walkers (all received walkers are sent forth and received in the next round by slower neighbors) and thus will not interfere with the dynamics. A slow ordinary node (or a slow sink) will also output the right number of walkers on average, but its output will burst periodically while being null the rest of the time. If α is too low (putting the focus on short-term monitoring of the flow) and/or the processor too slow, this may hinder the convergence of the system. Nevertheless, with a bound on c depending on α and β, the system and dynamics will converge to shortest paths.

Note that a slow node generates a weaker flow, if seen by the perspective of a faster one. This flow is taken into account if it is not lower than β. Indeed, if it is lower than β it will be replaced by β in the routing of walkers and the slow node will not contribute to the dynamics.

4 Simulation Results on Clustering

The scheme presented in the previous section computes shortest paths between sources and sinks. During our experiments we ran simulations with different settings:

- one source, one sink (shortest path);
- all nodes are sources, except for one node which is a sink (shortest paths DAG);
- all nodes are sources, except for two nodes which are sinks (distance-based clustering with shortest path DAGs);
- all nodes are sources and sinks have dynamically changed.

We executed the algorithm on randomly weighted 10×10 grids, on unweighted random graphs (100 nodes, $p = 0.2$) and on randomly weighted grids (weights were integer drawn uniformly at random between 1 and 10). Grids were chosen because the structure of the shortest paths in those graphs is easy to describe and because they are far from being unique. Random graphs are intended to be closer to real-world distributed systems.

The simulations were run using DASOR [14]. Note that all graphical results in this section show arrows proportional to the flow and that the simulation

campaign is at its beginning and we do not claim statistical accuracy of these data. Each presented figure is the average of 100 simulation results, together with its standard deviation: the number of messages, of aggregated messages (i.e. weighted messages representing several walkers, as explained in the previous section), the convergence time or measures of the distance to the solution (detailed for each measure in the relevant subsection) were recorded for each simulation and then averaged for each round of simulations in a given setting.

4.1 Shortest Path (1 Source 1 Sink Setting)

When there is only one source and only one sink in the system, the system dynamics allow the computation of a shortest path (or several of them) between the source and the sink. During our experiments we considered that the system had achieved convergence when the strongest flow ran from the sink to the source on a shortest path and remained so until the end of the simulation (2000 rounds).

In an unweighted 10×10 grid with a source and a sink located at opposite corners, with $\alpha = 0.995$ and $\beta = 0.01$, on average, the system converges in 560 rounds (standard deviation: 142), 147,808 messages (standard deviation: 68,407) and 52,095 messages (standard deviation: 19,306) if we aggregate them.

In a randomly weighted grid (with weights uniformly distributed between 1 and 10) with a source and a sink located at opposite corners, with $\alpha = 0.995$ and $\beta = 0.01$, convergence took 691 round on average (standard deviation: 576), 9,542,305 messages (standard deviation: 9,684,811) and 205,930 aggregated messages (standard deviation: 163,589).

In a random graph with 100 nodes and $p = 0.2$ with random source and sink, with $\alpha = 0.999$ and $\beta = 0.01$, convergence was realized after 31 rounds (standard deviation: 12), 38,905 messages (standard deviation: 22,515), or 18,930 aggregated messages (standard deviation: 9219).

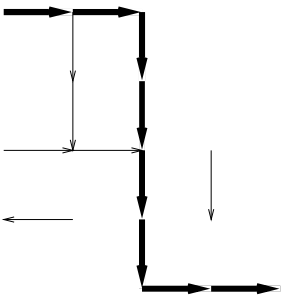


Fig. 1. Flow of walkers on shortest paths (with some remaining extra flow)

Those simulation results confirm the convergence of the discrete dynamics to shortest paths (see figure 1). Time and message complexity are of the order of the ones of existing distributed shortest path algorithm ($O(n)$ and $O(mn)$

respectively) on the two unweighted settings. The results on weighted grids need to be further investigated, with different choices of α and β.

4.2 Shortest Paths DAG of a Cluster: n Sources 1 Sink Setting

In these simulations, we measured convergence by counting the average number of nodes that have the majority of their outgoing flow directed away from the sink, i.e. to a node not closer to the sink than themselves (between parentheses are the standard deviations). All nodes are sources, but for one node (a corner node in the case of the grid) that is a sink.

	25 rounds	50 rounds	100 rounds	200 rounds
convergence	44.5 (5.19)	37.98 (3.19)	1.51 (0.15)	0.48 (0.07)
#msg	26,004 (175)	104,772 (756)	385,795 (3200)	1,238,815 (11,024)
#ag msg	6,339 (43)	14,306 (107)	29,295 (244)	58,384 (606)

Fig. 2. Reconfiguration results on a 10×10 grid with two sinks and $\alpha = 0.975$ and $\beta = 0.02$

	50 rounds	100 rounds	200 rounds	400 rounds
convergence	38.77 (5.77)	12.86 (3.12)	4.28 (2.01)	1.28 (1.25)
#msg	85,408 (4,292)	140,042 (12,289)	191,531 (16,858)	269,050 (23,304)
#ag msg	51,912 (1,529)	88,148 (5,122)	125,615 (7,518)	179,860 (10,515)

Fig. 3. Convergence results on a random graph with 100 nodes and $p = 0.2$, with one sink $\alpha = 0.999$ and $\beta = 0.01$

These results (figures 2 and 3) show that the algorithm converges to a shortest path DAG quite fast for most nodes (see figure 4) and then takes time to reach full convergence. In particular, the very conservative choice of α explains the slow convergence in the second round of simulations. Also note that, in a grid, aggregation of walkers reduces highly the number of exchanged messages. In the experiments presented above, the high number of messages in the first round of simulations make the use of aggregated walkers particularly interesting.

4.3 Clustering (n Sources, Several Sinks Setting)

The simulations were ran on a 10×10 grid, starting from an initial configuration with a unique source at a corner. Following the same methodology as in the previous experiments, we measured convergence by counting the average number of nodes that have the majority of their outgoing flow directed away from the sink, i.e. to a node not closer to the sink than themselves (between parentheses are the standard deviations). All nodes are sources, but for two nodes (opposite corner nodes in the case of the grid) that are sinks.

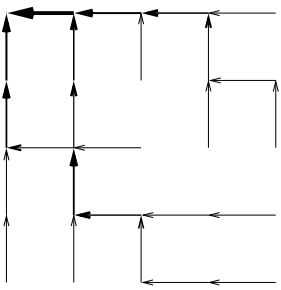


Fig. 4. Flow of the walkers on a shortest paths DAG

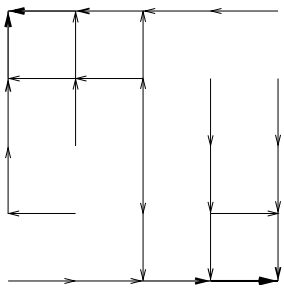


Fig. 5. Computation of two clusters with their shortest paths DAGs

	25 rounds	50 rounds	100 rounds	200 rounds
convergence	19.4 (4.37)	3.89 (2.28)	1.33 (1.24)	0.55 (0.79)
#msg	24,394 (231)	92,060 (1,082)	290,887 (4,414)	710,330 (11,769)
#ag msg	6,192 (51)	13,666 (109)	27,607 (283)	54,230 (687)

Fig. 6. Convergence results on a 10×10 grid with two sinks and $\alpha = 0.975$ and $\beta = 0.02$

	50 rounds	100 rounds	200 rounds	400 rounds
convergence	39.3 (6.5)	12.41 (3.43)	4 (2.05)	1.08 (1.13)
#msg	86,601 (5,148)	147,325 (36,828)	205,162 (105,107)	303,000 (298,876)
#ag msg	51,427 (5,230)	87,955 (9,376)	126,313 (9,942)	180,939 (12,371)

Fig. 7. Convergence results on a random graph with 100 nodes and $p = 0.2$, with two sinks and $\alpha = 0.999$ and $\beta = 0.01$

The results with several sinks (figures 6 and 7) illustrate the reduction in the complexity of the algorithm when several sinks are present. The use of these dynamics allows to compute a clustering, as illustrated by figure 5: nodes can use their incoming flows to reach the closest clusterhead.

4.4 Reconfiguration of Clusters after a Topological Change

The process presented here is tolerant to topological changes: starting from a configuration with flows running to a given sink, the appearance of a new sink or the change in an existing sink, thanks to the β parameter, will lead to changes in the flow in order to adapt to the new situation. The nodes immediately affected by the change in the situation see their flow change; nodes to which they were related are also subject to a change in the flow circulation. However, the dynamics of further nodes are not affected by this topological change.

The simulations were ran on a 10×10 grid, starting from an initial configuration with a unique sink at the bottom right corner and a consistent stable

flow (all nodes send all incoming flow to their bottom neighbor if they have one, or to their right neighbor). Then, two rounds of simulation were ran: one with a new sink appearing at the opposite corner and another with the original sink becoming an ordinary node while a new sink appears at the opposite corner. Same as before, we measured convergence by counting the average number of nodes that have the majority of their outgoing flow directed away from the sink, i.e. to a node not closer to the sink than themselves (between parentheses are the standard deviations). All nodes are sources but for two nodes (opposite corner nodes in the case of the grid) that are sinks, and the parameters used were $\alpha = 0.92$ and $\beta = 0.02$.

	25 rounds	50 rounds	100 rounds	200 rounds
convergence	27.11 (3.2)	6.74 (2.52)	1.27 (1.08)	1.21 (1.05)
#msg	24,703 (205)	94,083 (842)	303,169 (3,159)	739,326 (7810)
#ag msg	5,001 (48)	12,329 (87)	26,618 (205)	54,212 (477)

Fig. 8. Convergence results on a 10×10 grid after the appearance of a second sink, with $\alpha = 0.92$ and $\beta = 0.02$

The reconfiguration of the algorithm after a topological change is carried out rapidly for most nodes. A closer examination of the simulation shows that the nodes closer to the clusterhead that was already in place do not modify their flow structure significantly (see figure 9).

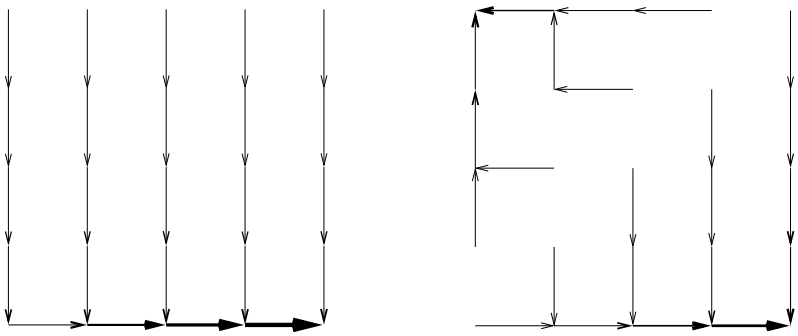


Fig. 9. Convergence after a new sink has appeared (on the left, initial configuration; on the right, configuration after convergence)

5 Conclusions

Based on the analogy between the growth of the mould Physarum and the behavior of electrical flow, we proposed a randomized distributed algorithm in which walkers emulate electrical current (as studied by [17]), while the resistance of the edges are modified by the flow according to a discrete-time version of the equations in [16].

As expected, the simulations show that the dynamics of this algorithm converge to flows running along shortest paths from sources to sinks. This algorithm is local and uses only one type of message with no content. This may compensate partially for the high number of messages needed to reach full convergence. Also, it may be noted that most nodes find a path to their clusterhead in the very first rounds. After a topological change, nodes soon find their new cluster.

This work provides an insight on flows of walkers and on the interest of biasing the walkers in a non-markovian fashion (the future moves of walkers depend on their past moves). Interesting questions following these results are the choice of the algorithm parameters and a comprehensive study of the theoretical foundations of this discrete process.

References

1. Amis, A.D., Prakash, R., Vuong, T.H.P., Huynh, D.T.: Max-min d-cluster formation in wireless ad hoc networks. In: Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE INFOCOM 2000, pp. 32–41 (2000)
2. Basagni, S.: Distributed clustering for ad hoc networks. In: International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN, pp. 310–315 (1999)
3. Bernard, T., Bui, A., Pilard, L., Sohier, D.: Distributed Clustering Algorithm for Large-Scale Dynamic Networks. International Journal of Cluster Computing (2010), doi:10.1007/s10586-011-0153-z
4. Bui, A., Clavière, S., Datta, A.K., Larmore, L.L., Sohier, D.: Self-stabilizing Hierarchical Construction of Bounded Size Clusters. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 54–65. Springer, Heidelberg (2011)
5. Bui, A., Kudireti, A., Sohier, D.: An adaptive random walks based distributed clustering algorithm. International Journal of Foundations of Computer Science 23(4), 802–830 (2012)
6. Datta, A.K., Larmore, L.L., Vemula, P.: A self-stabilizing $O(k)$-time k-clustering algorithm. The Computer Journal 53(3), 342–350 (2010)
7. Dolev, S., Tzachar, N.: Empire of colonies: Self-stabilizing and self-organizing distributing algorithm. Theoretical Computer Science 410, 514–532 (2009)
8. Ephremides, A., Wieselthier, J.E., Baker, D.J.: A design concept for reliable mobile radio networks with frequency hopping signaling. Proceedings of the IEEE, 56–73 (1987)
9. Johnen, C., Nguyen, L.: Robust self-stabilizing weight-based clustering algorithm. Theoretical Computer Science 410(6-7), 581–594 (2009)
10. Sucec, J., Marsic, I.: Location management handoff overhead in hierarchically organized mobile ad hoc networks. In: International Parallel and Distributed Processing Symposium, IPDPS, vol. 2, p. 198, 0194 (2002)
11. Thaler, D.G., Ravishankar, C.V.: Distributed top-down hierarchy construction. In: Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE INFOCOM 1998, vol. 2, pp. 693–701 (1998)
12. Yang, S.-J., Chou, H.-C.: Design Issues and Performance Analysis of Location-Aided Hierarchical Cluster Routing on the MANET. In: Communications and Mobile Computing, CMC, pp. 26–31 (2009)

13. Bellman, R.: Dynamic Programming. Princeton University Press, Dover (1957)
14. Rabat, C.: Dasor, a Discret Events Simulation Library for Grid and Peer-to-peer Simulators. Studia Informatica Universalis 7 (2009)
15. Adamatzky, A., de Oliveira, P.P.B.: Brazilian highways from slime mold's point of view. Kybernetes 40(9), 1373–1394 (2011)
16. Bonifaci, V., Mehlhorn, K., Varma, G.: Physarum can compute shortest paths. In: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 233–240 (2012)
17. Doyle, P.G., Snell, L.J.: Random Walks and Electrical Networks. Mathematical Association of America (December 1984)
18. Gkantsidis, C., Mihail, M., Saberi, A.: Random walks in peer-to-peer networks: Algorithms and evaluation. Performance Evaluation 63(3), 241–263 (2006)
19. Ito, K., Johansson, A., Nakagaki, T., Tero, A.: Convergence properties for the physarum solver. arXiv:1101.5249 (January 2011)
20. Johannson, A., Zou, J.: A Slime Mold Solver for Linear Programming Problems. In: Cooper, S.B., Dawar, A., Löwe, B. (eds.) CiE 2012. LNCS, vol. 7318, pp. 344–354. Springer, Heidelberg (2012)
21. Lenzen, C., Suomela, J., Wattenhofer, R.: Local Algorithms: Self-stabilization on Speed. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 17–34. Springer, Heidelberg (2009)
22. Li, K., Torres, C., Thomas, K., Rossi, L., Shen, C.-C.: Slime mold inspired routing protocols for wireless sensor networks. Swarm Intelligence 5(3), 183–223 (2011)
23. Miyaji, T.: Mathematical analysis to an adaptive network of the plasmodium system. Hokkaido Mathematical Journal 36(2), 445–465 (2007); Mathematical Reviews number (MathSciNet): MR2347434
24. Miyaji, T., Onishi, I.: Physarum can solve the shortest path problem on riemannian surface mathematically rigourously. International Journal of Pure and Applied Mathematics 47(3) (2008)
25. Nakagaki, T., Tero, A., Kobayashi, R., Onishi, I., Miyaji, T.: Computational ability of cells based on cell dynamics and adaptability. New Generation Computing 27(1), 57–81 (2008)
26. Tero, A., Takagi, S., Saigusa, T., Ito, K., Bebber, D.P., Fricker, M.D., Yumiki, K., Kobayashi, R., Nakagaki, T.: Rules for biologically inspired adaptive network design. Science 327(5964), 439–442 (2010)
27. Wagner, D., Wattenhofer, R. (eds.): Algorithms for Sensor and Ad Hoc Networks. LNCS, vol. 4621. Springer, Heidelberg (2007)
28. Georgiadis, G., Papatriantafilou, M.: A Least-Resistance Path in Reasoning about Unstructured Overlay Networks. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 483–497. Springer, Heidelberg (2009)
29. Bui, A., Sohier, D.: How to compute times of random walks based distributed algorithms. Fundamenta Informaticae 80(4), 363–378 (2007)
30. Georgiadis, G., Papatriantafilou, M.: Physarum-inspired self-biased walkers for distributed clustering, Chalmers University of Technology, TR-2012:08 (June 2012)

Wait-Free Linked-Lists*

Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank

Dept. of Computer Science, Technion, Israel
{stimnat,anastas,sakogan,erez}@cs.technion.ac.il

Abstract. *Wait-freedom* is the strongest and most desirable progress guarantee, under which any thread must make progress when given enough CPU steps. Wait-freedom is required for hard real-time, and desirable in many other scenarios. However, because wait-freedom is hard to achieve, we usually settle for the weaker *lock-free* progress guarantee, under which one of the active threads is guaranteed to make progress. With lock-freedom (and unlike wait-freedom), starvation of all threads but one is possible.

The linked-list data structure is fundamental and ubiquitous. Lock-free versions of the linked-list are well known. However, whether it is possible to design a practical wait-free linked-list has remained an open question. In this work we present a practical wait-free linked-list based on the CAS primitive. To improve performance further, we also extend this design using the fast-path-slow-path methodology. The proposed design has been implemented and measurements demonstrate performance competitive with that of Harris's lock-free list, while still providing the desirable wait-free guarantee, required for real-time systems.

1 Introduction

A linked-list is one of the most commonly used data structures. The linked-list seems a good candidate for parallelization, as modifications to different parts of the list may be executed independently and concurrently. Indeed, parallel linked-lists with various progress properties are abundant in the literature. Among these are lock-free linked-lists. A lock-free data structure ensures that when several threads access the data structure concurrently, at least one makes progress within a bounded number of steps. While this property ensures general system progress, it does not prevent starvation of a particular thread, or of several threads. Wait-free data structures ensure that each thread makes progress within a bounded number of steps, regardless of other threads' concurrent execution. Wait-free data structures are crucial for real-time systems, where a deadline may not be missed even in a worst-case scenario. To allow real-time systems and other systems with critical worst-case demands make use of concurrent data structures, we must provide the strong wait-free guarantee. Furthermore, wait-freedom is a desirable progress property for many systems, and in particular operating systems, interactive systems, and systems with service-level guarantees. For all those, the elimination of starvation is highly desirable.

Despite the great practical need for data structures that ensure wait-freedom, almost no practical wait-free data structure is known, because data structures that ensure wait-freedom are notoriously hard to design. Recently, wait-free designs for the simple stack

* This work was supported by the Israeli Science Foundation grant No. 283/10.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 330–344, 2012.

and queue data structures appeared in the literature [7,2]. Wait-free stack and queue structures are not easy to design, but they are considered less challenging as they present limited parallelism, i.e., a limited number of contention points (the head of the stack, and the head and the tail of the queue). We are not aware of any practical wait-free design for any other data structure that allows multiple concurrent operations to occur simultaneously. In particular, to the best of our knowledge, there is no wait-free linked-list algorithm available in the literature except for algorithms of universal constructions, which do not provide practical efficiency.

The main contribution of this work is a practical, linearizable, fast and wait-free linked-list. Our construction builds on the lock-free linked-list of Harris [4], and extends it using a helping mechanism to become wait-free. The main technical difficulty is making sure that helping threads perform each operation correctly, apply each operation exactly once, and return a consistent result (of success or failure) according to whether each of the threads completed the operation successfully. This task is non-trivial and it is what makes wait-free algorithms notoriously hard to design. Our design deals with several races that come up, and a proof of correctness makes sure that no further races exist. Some of our techniques may be useful in future work, especially the *success bit* introduced to determine the owner of a successful operation. Next, we extend our design using the fast-path-slow-path methodology of Kogan and Petrank [8], in order to make it even more efficient, and achieve performance that is almost equivalent to that of the lock-free linked-list of Harris. Here, the idea is to combine both lock-free and wait-free algorithms so that the (lock-free) fast path runs with (almost) no overhead, but is able to switch to the (wait-free) slow path when contention interferes with its progress. It is also important that both paths are able to run concurrently and correctly. Combining the newly obtained wait-free linked-list with the existing lock-free linked-list of Harris is an additional design challenge that is, again, far from trivial.

We have implemented the new wait-free linked-list and compared its efficiency with that of Harris's lock-free linked-list. Our first design(slightly optimized) performs worse by a factor of 1.5 when compared to Harris's lock-free algorithm. This provides a practical, yet not optimal, solution. However, the fast-path-slow-path extension reduces the overhead significantly, bringing it to just 2-15 percents. This seems a reasonable price to pay for obtaining a data structure with the strongest wait-free guarantee, providing non-starvation even in worst-case scenarios, and making it available for use with real-time systems.

We begin in Section 2 with an overview of the algorithm and continue in Section 3 with a detailed description of its most complex operation and crucial parts. Highlights of the correctness proof appear in Section 4. The linearization points of the algorithm are specified in Section 5. We give an overview of the fast-path-slow-path extension of the algorithm in Section 6, and Section 7 presents the performance measurements. In a full version of this work [11] we also provide details about the fast-path-slow-path implementation, the entire pseudo-code, and a full correctness proof for the algorithm.

1.1 Background and Related Work

The first lock-free linked-list was presented by Valois [12]. A simpler and more efficient lock-free algorithm was designed by Harris [4], and Michael [9] added a

hazard-pointers mechanism to allow lock-free memory management for this algorithm. Fomitchev and Rupert achieved better theoretical complexity in [3]. Herlihy and Shavit implemented a variation of Harris's algorithm [6], and we used this implementation both for comparison and as the basis for the Java code we developed.

Wait-free queues were presented in [7,2]. A different approach for building concurrent lock-free or wait-free data structures is the use of universal constructions [5,6,1]. However, universal constructions (at least for the linked-list) are not efficient enough to be applied in practice, and are often non-scalable.

Recently, Kogan and Petrank [8] presented the fast-path-slow-path technique mentioned above. We use the fast-path-slow-path methodology in this work to achieve an efficient and wait-free linked-list.

Our wait-free linked-list design follows the traditional practice, in which concurrent linked-list data structures realize a sorted list, where each key may only appear once in the list [3,4,6,12]. A brief announcement of this work appeared in [10].

2 An Overview of the Algorithm

Before getting into the technical details (in Section 3) we provide an overview of the design. The wait-free linked-list supports three operations: INSERT, DELETE, and CONTAINS. All of them run in a wait-free manner. The underlying structure of the linked-list is depicted in Figure 2. Similarly to Harris's linked-list, our list contains sentinel `head` and `tail` nodes, and the `next` pointer in each node can be marked using a special `mark bit`, to signify that the entry in the node is logically deleted.

To achieve wait-freedom, our list employs a helping mechanism. Before starting to execute an operation, a thread starts by publishing an *Operation Descriptor*, in a special `state` array, allowing all the threads to view the details of the operation it is executing. Once an operation is published, all threads may try to help execute it. When an operation is completed, the result is reported to the state array, using a CAS which replaces the existing operation descriptor with one that contains the result.

A top-level overview of the insert and delete operations is provided in Figure 1. When a thread wishes to INSERT a key k to the list, it first allocates a new node with key k, and then publishes an operation descriptor with a pointer to the new node.

```
1: boolean insert(key)
2:  Allocate a new node (without help)
3:  Publish the operation (without help)
4:  Search for a place to insert the node
5:    If key already exists, return with failure
6:  Direct the new node's next pointer
7:  Insert the node(by modifying its predecessor)
8:  Return with Success
```

```
1: boolean delete(key)
2:  Publish the operation (without help)
3:  Search for the victim node to delete
4:    If key doesn't exist, return with failure
5:  Announce the victim node in the state array
6:  Mark the victim's pointer to logically delete it
7:  Physically remove the victim node
8:  Report that the victim node has been removed
9:  Compete for success (without help)
```

Fig. 1. Insert and delete overview

The rest of the operation can be executed by any of the threads in the system, and may also be run by many threads concurrently. Any thread that executes this operation starts by searching for a place to insert the new node. This is done using the search method, which, given a key k, returns a pair of pointers, *prev* and *curr*. The prev pointer points to the node with the highest key smaller than k, and the curr pointer points to the node with the smallest key larger than or equal to k. If the returned curr node holds a key equal to the key on the node to be inserted, then failure is reported. Otherwise the node should be inserted between prev and curr. This is done by first updating the new node's `next` pointer to point to curr, and then updating prev's `next` field to point to it. Both of these updates are done using a CAS to prevent race conditions, and the failure of any of these CASes will cause the operation to restart from the search method. Finally, after that node has been inserted, success is reported.

While the above description outlines the general process of inserting a node, the actual algorithm is a lot more complex, and requires care to avoid problematic races that can make things go wrong. In addition, there is also a potential ABA problem that requires the use of a version mark on the `next` pointer field[1]. We discuss these and other potential races in Section 3.4.

When a thread wishes to DELETE a key k from the list, it starts by publishing the details of its operation in the `state` array. The next steps can be then executed by any of the threads in the system until the last step, which is executed only by the thread that initiated the operation, denoted the *owner thread*. The DELETE operation is executed (or helped) in two stages. First, the *victim* node to be deleted is chosen. To do this, the search method is invoked. If no node with the key k is found, failure is reported. Otherwise, the victim node is *announced* in the `state` array. This is done by replacing the state descriptor that describes this operation to a state descriptor that has a pointer to the victim node. This announcement helps to ascertain that concurrent helping threads will not delete two different nodes, as the victim node for this operation is determined to be the single node that is announced in the operation descriptor. In the second stage, deletion is executed similarly to Harris's linked-list: the victim node's `next` field is marked, and then it is physically removed from the list. The victim node's removal is then reported back to the `state` array.

However, since multiple threads execute multiple operations, and as it is possible that several operations attempt to DELETE the same node, it is crucial that exactly one operation be declared as successfully deleting the node's key and that the others return failure. An additional (third) stage is required in order to consistently determine which operation can be considered successful. This step is executed only by the owner threads, and is given no help. The threads that initiated the concurrent delete operations compete among themselves for the ownership of the deletion. To this end, an extra `success-bit` designated for this purpose is added to each node in the list. The thread that successfully CASes this bit from false to true is the only one that reports success for this deletion. We believe that using an extra bit to determine an ownership of an operation is a useful mechanism for future wait-free constructions as well. This mechanism is further explained in Section 3.5.

[1] The versioning method provides a simple solution to the ABA problem. A more involved solution that does not require a versioned pointer appears in the full version of this paper [11].

The CONTAINS operation is much simpler than the other two. It starts by publishing the operation. Any helping thread will then search for it in the list, reporting success (on the operation record) if the key was found, or failure if it was not.

3 The Algorithm

In this section we present the details of the algorithm. We fully describe the list structure, the helping mechanism, and the SEARCH and INSERT operations. The INSERT operation is the most complicated part of the algorithm. A detailed description of the DELETE and CONTAINS operations appears in the full version of this paper [11]. We also include in this section a detailed description of the `success-bit` technique used in the DELETE operation, as we believe this mechanism can be useful for future work.

3.1 The Underlying Data Structures

The structure of the linked-list is depicted in Figure 2. A node of the linked list consists of three fields: a key, a `success bit` to be used when deleting this node, and a special pointer field. The special pointer field has its least significant bit used by the algorithm for signaling between threads. In addition, this pointer is versioned, in the sense that there is a counter associated with it (in an adjacent word) and each modification of it (or of its special bit) increments the counter. The modification and counter increment are assumed to be atomic. This can be implemented by squeezing all these fields into a single word, and limiting the size of the counter and pointer, or by using a double-word compare-and-swap when the platform allows. Alternatively, one can allocate a "pointer object" containing all these fields and bits, and then atomically replace the existing pointer object with a new one. The latter approach is commonly used with Java lock-free implementations, and we use it as well.

In addition to the nodes of the list, we also maintain an array with an operation-descriptor for each thread in the system. The OpDesc entry for each thread describes its current state. It consists of a phase field `phase`, the `OpType` field signifying which operation is currently being executed by this thread, a pointer to a node, denoted `node`,

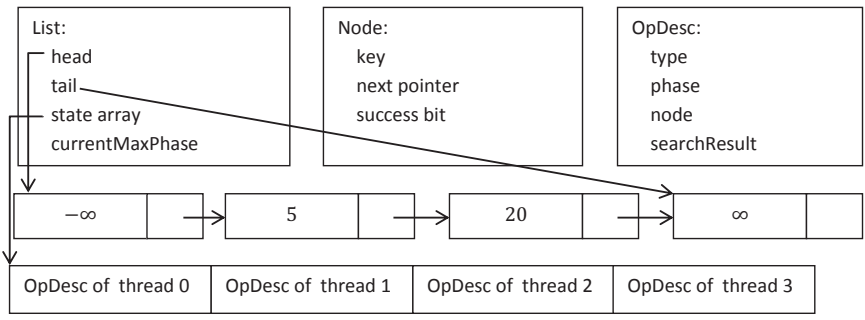


Fig. 2. General structure

which serves the insert and delete operations, and a pair of pointers $(prev, curr)$, for recording the result of a search operation. Recall that the result of a SEARCH operation of a key, k, is a pair of pointers denoted *prev* and *curr*, as explained in Section 2 above.

The possible values for the operation type (OpType) in the operation descriptor state are:

insert	asking for help in inserting a node into the list.
search_delete	asking for help in finding a node with the key we wish to delete.
execute_delete	asking for help in marking a node as deleted (by tagging its next pointer) and unlinking it from the list.
contains	asking for help in finding out if a node with the given key exists.
success	operation was completed successfully.
failure	operation failed (deletion of a non-existing key or insertion of an existing key).
determine_delete	decide if a delete operation completed successfully.

The first four states in the above list are used to request help from other threads. The last three states indicate steps in the executions in which the thread does not require any help. The linked-list also contains an additional long field, `currentMaxPhase`, to support the helping mechanism, as described in Subsection 3.2.

3.2 The Helping Mechanism

Before a thread starts executing an operation, it first selects a phase number larger than all previously chosen phase numbers. The goal of assigning a phase number to each operation is to let new operations make sure that old operations receive help and complete before new operations are executed. This ensures non-starvation. The phase selection mechanism ensures that if operation O_2 arrives strictly later than operation O_1, i.e., O_1 receives a phase number before O_2 starts selecting its own phase number, then O_2 will receive a higher phase number. The phase selection procedure is executed in the MAXPHASE method depicted in Figure 3. Note that although a CAS is used in this method, the success of this CAS is not checked, thus preserving wait-freedom. If the CAS fails, it means that another thread increased the counter concurrently, which is sufficient for the phase numbering. After selecting a phase number, the thread publishes the operation by updating its entry in the `state` array. It then goes through the array, helping all operations with a phase number lower than or equal to its own. This ensures wait-freedom: a delayed operation eventually receives help from all threads and soon completes. See Figure 3 for the pseudo-code.

3.3 The Search Methods

The CONTAINS method, which is part of the data structure interface, is used to check whether a certain key is a part of the list. The SEARCH method is used (internally) by the INSERT, DELETE, and CONTAINS methods to find the location of a key and perform some maintenance during the search. It is actually nearly identical to the original lock-free SEARCH method. The SEARCH method takes a key and returns a pair of pointers

```
1: private long maxPhase() {
2:  long result = currentMaxPhase.get();
3:  currentMaxPhase.compareAndSet
4:    (result, result+1);
5:  return result; }
6:
7: private void help(long phase) {
8:  for (int i = 0; i < state.length(); i++) {
9:   OpDesc desc = state.get(i);
10:  if (desc.phase <= phase) {      ▷ help older op
11:   if (desc.type == OpType.insert) {
12:    helpInsert(i, desc.phase);
13:   } else if
14:   (desc.type == OpType.search_delete
15:   || desc.type == OpType.execute_delete) {
16:    helpDelete(i, desc.phase);
17:   } else if (desc.type == OpType.contains) {
18:    helpContains(i, desc.phase);
19:   } } } }
20:
21: private boolean isSearchStillPending(int tid,
    long ph) {
22:  OpDesc curr = state.get(tid);
23:  return (curr.type == OpType.insert ||
24:    curr.type == OpType.search_delete ||
25:    curr.type == OpType.execute_delete ||
26:    curr.type==OpType.contains) &&
27:   curr.phase == ph; }
28: private Window search(int key, int tid, long
    phase) {
29:  Node pred = null, curr = null, succ = null;
30:  boolean[] marked = {false}; boolean snip;
31:  retry : while (true) {
32:   pred = head;
33:   curr = pred.next.getReference();
34:   while (true) {
35:              ▷ Reading both the ref and the mark:
36:    succ = curr.next.get(marked);
37:    while (marked[0]) {          ▷ logically deleted
38:             ▷ Attempt to physically remove curr:
39:     snip = pred.next.compareAndSet
40:       (curr, succ, false, false);
41:     if (!isSearchStillPending(tid,phase))
42:      return null;          ▷ to ensure wait-freedom.
43:     if (!snip) continue retry;          ▷ list changed
44:     curr = succ;                        ▷ advance curr
45:     succ = curr.next.get(marked);       ▷ and succ
46:    }
47:    if (curr.key >= key)                 ▷ window found
48:     return new Window(pred, curr);
49:    pred = curr; curr = succ;            ▷ advance both
50:   }
51:  }
52: }
53:
54:
```

Fig. 3. The help and search methods

denoted *window*: `pred`, which points to the node containing the highest key less than the input key, and `curr`, which points to the node containing the lowest key higher than or equal to the requested key. When traversing through the list, the SEARCH method attempts to physically remove any node that is logically deleted. If the remove attempt fails, the search is restarted from the head of the list. This endless attempt to fix the list seems to contradict wait-freedom, but the helping mechanism ensures that these attempts eventually succeed. When an operation delays long enough, all threads reach the point at which they are helping it. When that happens, the operation is guaranteed to succeed. The SEARCH operation will not re-iterate if the operation that executes it has completed, which is checked using the ISSEARCHSTILLPENDING method. If the associated operation is complete, then the SEARCH method returns a null. The pseudo-code for the search method is depicted in Figure 3.

3.4 The Insert Operation

Designing operations for a wait-free algorithm requires dealing with multiple threads executing each operation, which is substantially more difficult than designing a lock-free operation. In this section, we present the insert operation and discuss some of the

```
1:  public boolean insert(int tid, int key) {
2:     long phase = maxPhase();                                  ▷ getting the phase for the op
3:     Node newNode = new Node(key);                             ▷ allocating the node
4:     OpDesc op = new OpDesc(phase, OpType.insert, newNode,null);
5:     state.set(tid, op);                                       ▷ publishing the operation
6:     help(phase);                  ▷ when finished - no more pending operation with lower or equal phase
7:     return state.get(tid).type == OpType.success;
8:  }
9:
10: private void helpInsert(int tid, long phase) {
11:    while (true) {
12:       OpDesc op = state.get(tid);
13:       if (!(op.type == OpType.insert && op.phase == phase))
14:          return;                                  ▷ the op is no longer relevant, return
15:       Node node = op.node;                        ▷ getting the node to be inserted
16:       Node node_next = node.next.getReference();
17:       Window window = search(node.key,tid,phase);
18:       if (window == null)                         ▷ operation is no longer pending
19:          return;
20:       if (window.curr.key == node.key) {                    ▷ chance of a failure
21:          if ((window.curr==node)||(node.next.isMarked())){  ▷ success
22:             OpDesc success =
23:                new OpDesc(phase, OpType.success, node, null);
24:             if (state.compareAndSet(tid, op, success))
25:                return;
26:          }
27:          else {                          ▷ the node was not yet inserted - failure
28:             OpDesc fail=new OpDesc(phase,OpType.failure,node,null);
29:                                ▷ the following CAS may fail if search results are obsolete:
30:             if (state.compareAndSet(tid, op, fail))
31:                return;
32:          }
33:       }
34:       else {
35:          if (node.next.isMarked()){                 ▷ already inserted and deleted
36:             OpDesc success =
37:                new OpDesc(phase, OpType.success, node, null);
38:             if (state.compareAndSet(tid, op, success))
39:                return;
40:          }
41:          int version = window.pred.next.getVersion();                ▷ read version.
42:          OpDesc newOp=new OpDesc(phase,OpType.insert,node,null);
43:                                ▷ preventing another thread from reporting a failure:
44:          if (!state.compareAndSet(tid, op, newOp))
45:             continue;                  ▷ operation might have already reported as failure
46:          node.next.compareAndSet(node_next,window.curr,false,false);
47:          if (window.pred.next.compareAndSet
48:                (version, node.next.getReference(), node, false, false)) {
49:             OpDesc success =
50:                new OpDesc(phase, OpType.success, node, null);
51:             if (state.compareAndSet(tid, newOp, success))
52:                return;
53:          }
54:       }
55:    }
56: }
```

Fig. 4. The insert operation

races that occur and how we handle them. The basic idea is to coordinate the execution of all threads using the operation descriptor. But more actions are required, as explained below. Of-course, a proof is required to ensure that all races have been handled. The pseudo-code of the INSERT operation is provided in Figure 4. The thread that initiates the operation is denoted *the operation owner*. The *operation owner* starts the INSERT method by selecting a phase number, allocating a new node with the input key, and installing a link to it in the `state` array.

Next, the thread (or any helping thread) continues by searching the list for a location where the node with the new key can be inserted (Line 17 in the method HELPINSERT). In the original lock-free linked-list, finding a node with the same key is interpreted as failure. However, in the presence of the helping mechanism, it is possible that some other thread that is helping the same operation has already inserted the node but has not yet reported success. It is also possible that the node we are trying to insert was already inserted and then deleted, and then a different node, with the same key, was inserted into the list. To identify these cases, we check the node that was found in the search. If it is the same node that we are trying to insert, then we know that success should be reported. We also check if the (`next` field of the) node that we are trying to insert is *marked* for deletion. This happens if the node was already inserted into the list and then removed. In this case, we also report success. Otherwise, we attempt to report failure. If there is no node found with the same key, then we can try to insert the node between `pred` and `curr`. But first we check to see if the node was already inserted and deleted (line 35), in which case we can simply report success.

The existence of other threads that help execute the same operation creates various races that should be properly handled. One of them, described in the next paragraph, requires the INSERT method to proceed with executing something that may seem redundant at first glance. The INSERT method creates a state descriptor identical to the existing one and atomically replaces the old one with the new one (Lines 42–45). The replacement foils all pending CAS operations by other threads on this state descriptor, and avoids confusion as to whether the operation succeeds or fails. Next, the method executes the actual insertion of the node into the list (Lines 46–48) and it attempts to report success (Lines 49–52). If any of the atomic operations fail, the insertion starts from scratch. The actual insertion into the list (Lines 46–48) is different from the insertion in the original lock-free linked-list. First, the `next` pointer in the new node is not privately set, as it is now accessible by all threads that help the insert operation. It is set by a CAS which verifies that the pointer has not changed since before the search. Namely, the old value is read in Line 16 and used as the expected value in the CAS of Line 46. This verification avoids another race, which is presented below. Moreover, the atomic modification of the `next` pointer in the previous node to point to the inserted node (Lines 47–48) uses the version of that `next` pointer to avoid the ABA problem. This is also justified below.

Let us first present the race that justifies the (seemingly futile) replacement of the state descriptor in Lines 42–45. Suppose Thread T_1 is executing an INSERT operation of a key k. T_1 finds an existing node with the key k and is about to report failure. T_1 then gets stalled for a while, during which the other node with the key k is deleted and a different thread, T_2, helping the same INSERT operation that T_1 is executing, does find

a proper place to insert the key k, and does insert it, but at that point T_1 regains control and changes the descriptor state to erroneously report failure. This sequence of events is bad, because a key has been inserted but failure has been reported. To avoid such a scenario, upon finding a location to insert k, T_2 modifies the operation descriptor to ensure that no stalled thread can wake up and succeed in writing a stale value into the operation descriptor.

Next, we present a race that justifies the setting of the `next` pointer in the new node (Line 46). The INSERT method verifies that this pointer has not been modified since it started the search. This is essential to avoid the following scenario. Suppose Thread T_1 is executing an INSERT of key k and finds a place to insert the new node N in between a node that contains $k-1$ and a node that contains $k+2$. Now T_1 gets stalled for a while and T_2, helping the same INSERT operation, inserts the node N with the key k , after which it also inserts another new node with key $k+1$, while T_1 is stalled. At this point, Thread T_1 resumes without knowing about the insertion of these two nodes. It modifies the next pointer of N to point to the node that contains $k+2$. This modification immediately foils the linked-list because it removes the node that contains $k+1$ from the list. By making T_1 replace the `next` field in N atomically only if this field has not changed since before the search, we know that there could be no node between N and the node that followed it at the time of the search.

Finally, we justify the use of a version for the `next` pointer in Line 47, by showing an ABA problem that could arise when several threads help executing the same insert operation. Suppose Thread T_1 is executing an INSERT of the key k into the list. It searches for a location for the insert, finds one, and gets stalled just before executing Line 47. While T_1 is stalled, T_2 inserts a different k into the list. After succeeding in that insert, T_2 tries to help the same insert of k that T_1 is attempting to perform. T_2 finds that k already exists and reports failure to the state descriptor. This should terminate the insertion that T_1 is executing with a failure report. But suppose further that the other k is then removed from the list, bringing the list back to exactly the same view as T_1 saw before it got stalled. Now T_1 resumes and the CAS of Line 47 actually succeeds. This course of events is bad, because a key is inserted into the list while a failure is reported about this insertion. This is a classical ABA problem, and we solve it using versioning of the next pointer. The version is incremented each time the `next` pointer is modified. Therefore, the insertion and deletion of a different k key while T_1 is stalled cannot go unnoticed.

3.5 The Success Bit Technique

Helping DELETE is different from helping INSERT in the sense that the help method in this case does not execute the entire DELETE operation to its completion. Instead, it stops before determining the success of the operation, and lets the operation owner decide whether its operation was successful. Note that this does not foil wait-freedom, as the operation owner will never get stuck on deciding whether the operation was successful. When the help method returns, there are two possibilities. The simpler possibility is that the requested key was not found in the list. Here it is clear that the operation failed and in that case the state is changed by the helper to a failure and the operation can terminate. The other possibility is that the requested key was found and deleted.

```
 1: public boolean delete(int tid, int key) {
 2:     long phase = maxPhase();                          ▷ getting the phase for the op
 3:     state.set(tid, new OpDesc
 4:     (phase, OpType.search_delete, new Node(key),null));          ▷ publishing
 5:     help (phase);       ▷ when finished - no more pending operation with lower or equal phase
 6:     OpDesc op = state.get(tid);
 7:     if (op.type == OpType.determine_delete)
 8:                                  ▷ Need to compete on the ownership of deleting this node:
 9:       return op.searchResult.curr.success.compareAndSet(false, true);
10:     return false;
11: }
```

Fig. 5. The delete method

In this case, it is possible that several DELETE operations for the same key were run concurrently by several operation owners and by several helping threads. As the delete succeeded, it has to be determined which operation owner succeeded. In such a case there are several operation owners for the deletion of the key k and only one operation owner can return success, because a single DELETE has been executed. The others operation owners must report failure. This decision is made by the operation owners (and not by the helping threads) in Line 9 of the DELETE method itself, depicted in Figure 5. It employs a designated `success bit` in each node. Whoever sets this bit becomes the owner of the deletion for that node in the list and can report success. We believe that this technique for determining the success of a thread in executing an operation in the presence of helping threads can be useful in future constructions of wait-free algorithms.

3.6 Memory Management

The algorithm in this work relies on a garbage collector (GC) for memory management. A wait-free GC does not currently exist. This is a common difficulty for wait-free algorithms. A frequently used solution, which suits this algorithm as well, is Michael's Hazard Pointers technique [9]. Hazard pointers can be used for the reclamation of the operation descriptors as well, and not only for the reclamation of the list nodes themselves.

4 Highlights of the Correctness Proof

We now briefly explain how this algorithm is proven correct. A full proof appears in the full version of this paper [11]. A full proof is crucial for a parallel algorithm as without it, one can never be sure that additional races are not lurking in the algorithm.

Basic Concepts and Definitions. The *mark bit*, is the bit on the next field of each node, and it is used to mark the node as logically deleted. A node can be marked or unmarked according to the value of this bit. We define the nodes that are *logically in the list* to be

the unmarked nodes that are reachable from the list's head. Thus, a *logical change* to the list, is a change to the set of unmarked nodes reachable from the head. We say that a node is an *infant node* if it has never been reachable from the head. These are nodes that have been prepared for insertions but have not been inserted yet.

In the proof we show that at the linearization point of a successful insert, the inserted value becomes logically in the list and that at a linearization point of a successful delete, a node with the given value is logically deleted from the list. To show this, we look at the actual *physical* modifications that may occur to the list.

Proof Structure. One useful invariant is that a *physical change* to the list can only modify the node's next field, as a node's key is final and never changes after the initialization of a node. A second useful invariant is that a marked node is never unmarked, and that it's next field never changes (meaning, it will keep pointing to the same node). This is ascertained by examining all the code lines that change a node's next field, and noting that all of them do it using a CAS which prevents a change from taking effect if the node is marked. We next look at all possible physical changes to a node's next field, and show that each of them falls in one of the following four categories:

* Marking: changing the mark bit of a node that is *logically in the list* to true.
* Snipping: physically removing a *marked* node out of the list.
* Redirection: a modification of an infant node's next pointer (in preparation for its insertion).
* Insertion: a modification of a non-infant node to point to an infant node (making the latter non-infant after the modification).

Proving that every *physical change* to a node's next field falls into one of the four categories listed above, is the most complicated part of the formal proof, and is done by induction, with several intermediate invariants. Finally, it is shown that any operation in the *marking* category matches a successful delete operation and any operation in the *insertion* category matches a successful insert operation. Thus, at the proper linearization points the linked list changes according to its specification. Furthermore, it is shown that physical operations in the *Redirection* and *Snipping* categories cause no *logical* changes to the list, which completes the linearizability proof.

To show wait-freedom, we claim that the helping mechanism ensures that a limited number of concurrent operations can be executed while a given insert or delete execution is pending. At the point when this number is exhausted, all threads will help the pending operation, and then it will terminates within a limited number of steps.

5 Linearization Points

In this section we specify the linearization point for the different operations of the linked-list. The SEARCH method for a key k returns a pair of pointers, denoted *pred* and *curr*. The prev pointer points to the node with the highest key smaller than k, and the curr pointer points to the node with the smallest key larger than or equal to k. The linearization point of the SEARCH method is when the pointer that connects *pred* to *curr* is read. This can be either at Line 36 or 45 of the SEARCH method. Note that *curr*'s

`next` field will be subsequently read, to make sure it is not *marked*. Since it is an invariant of the algorithm that a marked node is never unmarked, it is guaranteed that at the linearization point both *pred* and *curr* nodes were unmarked.

The linearization point for a CONTAINS method is the linearization point of the appropriate SEARCH method. The appropriate SEARCH method is the one called by the thread that subsequently successfully reports the result of the same CONTAINS operation. The linearization point of a *successful insert* is in Lines 47-48 (together they are a single instruction) of the *helpInsert* method. This is the CAS operation that physically links the node into the list. For a *failing insertion*, the linearization point is the linearization point of the SEARCH method executed by the thread that reported the failure.

The linearization point of a *successful delete* is at the point where the node is *logically* deleted, which means successfully marked. Note that it is possible that this is executed by a helping thread and not necessarily by the operation owner. Furthermore, the helping thread might be trying to help a different thread than the one that will eventually own the deletion. The linearization point of an *unsuccessful delete* is more complex. A delete operation may fail when the key is properly deleted, but a different thread is selected as the owner of the delete. In this case, the current thread returns failure, because of the failure of the CAS of the DELETE method (at Line 9). In this case, the linearization point is set to the point when the said node is logically deleted (marked). The linearization point of an *unsuccessful delete*, originating from simply not finding the key, is the linearization point of the SEARCH method executed by the thread that reported the failure.

6 The Fast-Path-Slow-Path Variation

The idea behind the fast-path-slow-path [8] approach is to combine a (fast) lock-free algorithm with a (slower) wait-free one. The lock free algorithm provides a basis for a fast path and we use Harris's lock-free linked-list for this purpose. The execution in the fast path begins by a check whether a help is required for any operation in the slow path. Next, the execution proceeds with running the fast lock-free version of the algorithm while counting the number of contentions that end with a failure (i.e., failed CASes). Typically, few failures occur and help is not required, and so the execution terminates after running the faster lock-free algorithm. If this fast path fails to make progress, the execution moves to the slow path, which runs the slower wait-free algorithm described in Section 3, requesting help (using an operation descriptor in its slot in the state array) and making sure the operation eventually terminates.

The number of CAS failures allowed in the fast path is limited by a parameter called MAX_FAILURES. The help is provided by threads running both the fast and slow path, which ensures wait-freedom: if a thread fails to complete its operation, its request for help is noticed both in the fast and in the slow path. Thus, eventually all other threads help it and its operation completes. However, help is not provided as intensively as described in Section 3. We use the *delayed help* mechanism, by which each thread only offers help to other threads once every several operations, determined by a parameter called HELPING_DELAY.

Combining the fast-path and the slow-path is not trivial, as care is needed to guarantee that both paths properly run concurrently. On top of other changes, it is useful to note

that the DELETE operation must compete on the `success-bit` even in the fast-path, to avoid a situation where two threads running on the two different paths both think they were successful in deleting a node. The full implementation of the fast-path-slow-path variation of the linked-list is described in [11].

7 Performance

Implementation and Platform. We compared four Java implementations of the linked-list. The first is the lock-free linked-list of Harris, denoted *LF*, as implemented by Herlihy and Shavit in [6]. (This implementation was slightly modified to allow nodes with user-selected keys rather than the object's hash-code. We also did not use the *item* field.)

The basic algorithm described in Section is denoted *WF-Orig* in the graphs below. A slightly optimized version of it, denoted *WF-Opt*, was changed to employ a delayed help mechanism, similar to the one used in the fast-path- slow-path extension. This means that a thread helps another thread only once every k operations, where k is a parameter of the algorithm set to 3. The idea is to avoid contention by letting help arrive only after the original thread has a reasonable chance of finishing its operation on its own. This optimization is highly effective, as seen in the results. Note that delaying help is not equivalent to a fast-path-slow-path approach, because all threads always ask for help (there is no fast path). All the operations are still done in the *helpInsert* and *helpDelete* methods.

The fast-path-slow-path algorithm, denoted *FPSP*, was run with the HELPING_DELAY parameter set to 3, and MAX_FAILURES set to 5. This algorithm combines the new wait-free algorithm described in this paper with Harris's lock-free algorithm, to achieve both good performance and the stronger wait-freedom progress guarantee.

We ran the tests in two environments. The first was a SUN's Java SE Runtime, version 1.6.0 on an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors (overall 8 cores). The second was a SUN FIRE machine with an UltraSPARC T1 8 cores each running four hyper-threads.

Workload and Methodology. In the micro-benchmarks tested, we ran each experiment for 2 seconds, and measured the overall number of operations performed by all the threads during that time. Each thread performed 60% CONTAINS, and 20% INSERT and DELETE operations, with keys chosen randomly and uniformly in the range $[1, 1024]$. The number of threads ranges from 1-16 (in the Intel(R) Xeon(R)) or from 1-32 (In the UltraSPARC). We present the results in Figure 6. The graphs show the total number of operations done by all threads in thousands for all four implementations, as a function of the number of threads. In all the tests, we executed each evaluation 8 times, and the averages are reported in the figures.

Results. It can be seen that the fast-path-slow-path algorithm is almost as fast as the lock-free algorithm. On the Intel machine, the two algorithms are barely distinguishable; the difference in performance is 2-3%. On the UltraSPARC the fast-path-slow-path suffers a noticeable (yet, reasonable) overhead of 9-14%. The (slightly optimized) basic wait-free algorithm is slower by a factor of 1.3–1.6, depending on the number of threads. Also, these three algorithms provide an excellent speed up of about 7 when

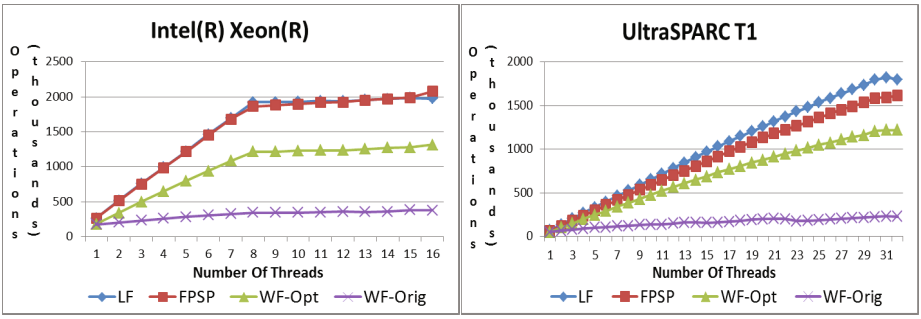


Fig. 6. The number of operations done in two seconds as a function of the number of threads

working with 8 threads (on both machines), and about 24 when working with 32 multi-threads on the UltraSPARC. The basic non-optimized version of the wait-free algorithm doesn't scale as well. There, threads often work together on the same operation, causing a deterioration in performance and scalability. The simple delayed-help optimization enables concurrency without foiling the worst-case wait-freedom guarantee.

References

1. Chuong, P., Ellen, F., Ramachandran, V.: A universal construction for wait-free transaction friendly data structures. In: SPAA, pp. 335–344 (2010)
2. Fatourou, P., Kallimanis, N.D.: A highly-efficient wait-free universal construction. In: SPAA, pp. 325–334 (2011)
3. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC, pp. 50–59. ACM, New York (2004)
4. Harris, T.L.: A Pragmatic Implementation of Non-blocking Linked-Lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)
5. Herlihy, M.: A methodology for implementing highly concurrent objects. ACM Trans. Program. Lang. Syst. 15(5), 745–770 (1993)
6. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
7. Kogan, A., Petrank, E.: Wait-free queues with multiple enqueuers and dequeuers. In: PPOPP, pp. 223–234 (2011)
8. Kogan, A., Petrank, E.: A methodology for creating fast wait-free data structures. In: PPOPP, pp. 141–150 (2012)
9. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. 15(6), 491–504 (2004)
10. Timnat, S., Braginsky, A., Kogan, A., Petrank, E.: Wait-free linked-lists. In: PPOPP, pp. 309–310 (2012)
11. Timnat, S., Braginsky, A., Kogan, A., Petrank, E.: Wait-free linked-lists (2012), http://www.cs.technion.ac.il/%7eerez/%50apers/wfll-full.pdf
12. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: PODC, pp. 214–222. ACM, New York (1995)

Byzantine Chain Replication

Robbert van Renesse, Chi Ho, and Nicolas Schiper

Cornell University*
Ithaca, NY, USA
{rvr,chho,nschiper}@cs.cornell.edu

Abstract. We present a new class of Byzantine-tolerant State Machine Replication protocols for asynchronous environments that we term *Byzantine Chain Replication*. We demonstrate two implementations that present different trade-offs between performance and security, and compare these with related work. Leveraging an external reconfiguration service, these protocols are not based on Byzantine consensus, do not require majority-based quorums during normal operation, and the set of replicas is easy to reconfigure.

One of the implementations is instantiated with $t+1$ replicas to tolerate t failures and is useful in situations where perimeter security makes malicious attacks unlikely. Applied to in-memory BerkeleyDB replication, it supports 20,000 transactions per second while a fully Byzantine implementation supports 12,000 transactions per second—about 70% of the throughput of a non-replicated database.

1 Introduction

Byzantine-tolerant State Machine Replication (BSMR) is the only known generic approach to making applications (servers, routing daemons, and so on) tolerate arbitrary faults beyond crash failures in an asynchronous environment [1]. Various studies in complex systems have shown that crash failures constitute a minority of failures [2, 3], while trends in hardware increase the probability of transient hardware errors such as bit flips [4–6]. Worse yet, most replication protocols deployed in cloud centers provide weak consistency guarantees, meaning that they *introduce* inconsistencies even if there are no faults [7, 8]. Protocols such as Primary-Backup [9] and Chain Replication [10] make strong assumptions about failure detection that are easily violated in datacenter settings while conservative timeouts result in long recovery times.

While BSMR addresses all these problems, to the best of our knowledge there is no deployment of BSMR in today's datacenters. There are good reasons for this:

- Traditional BSMR require $3t + 1$ replicas in order to tolerate t faults [11], whereas primary-backup replication protocols require only $t + 1$ replicas [9].

* This work was supported in part by DARPA grants FA8750-10-2-0238, FA9550-11-1-0137 , FA8750-11-2-0256, NSF grants 1040689, 1047540, 54083, and DOE grant DE-AR0000230.

R. Baldoni, P. Flocchini, and R. Binoy (Eds.): OPODIS 2012, LNCS 7702, pp. 345–359, 2012.

Also, the protocols may require expensive cryptographic operations and significant network bandwidth, further increasing cost;
- Significant progress has been made at reducing the cost of BSMR [12–14]. However, these protocols are complex [15]. Even basic (crash-tolerant) Paxos replication is difficult to implement and debug [16]. Multi-purpose Paxos and BSMR libraries have not proved successful, and few support basic operations such as reconfiguring the locations of the protocol participants, important to practical deployment.

Much of the difficulty stems from replicated services being designed to be *stand-alone*, but in practice the configuration of replicated services is usually managed by an external scalable configuration service. The configuration service is used only in the face of failures. Exploiting this, we can build replication protocols that do not use quorums in the steady-state when there are no failures. This reduces the number of replicas necessary, not only increasing efficiency but also robustness in the face of limited sources of diversity.

Our protocols are based on Chain Replication [10], which is increasingly being used in scalable fault tolerant systems [17–20]. Chain Replication also uses an external configuration service, but cannot tolerate even general crash failures as the protocol depends on accurate failure detection, an overly strong requirement in today's datacenters. In the face of a mistaken failure detection, a chain may split and inconsistent results may be returned to clients. To reduce the likelihood of such problems, timeouts used for failure detection have to be set conservatively (values that exceed one minute are not uncommon in practice), meaning that in the face of an actual crash it takes a long time before remedial action can be taken.

In this paper we make the following contributions:

- We present a new class of highly reconfigurable Chain Replication protocols that are easily reconfigurable, do not require accurate failure detection, and are able to tolerate Byzantine failures.
- We prove the correctness of this class of protocols;
- We present a Byzantine Chain Replication protocol called *Shuttle*;
- We briefly describe two simple implementations of Shuttle: one that can tolerate Byzantine failures in their full generality, and one that tolerates "accidental failures" such as crashes, bit flips, concurrency bugs, and so on;
- We present a performance evaluation of Byzantine Chain Replication applied to the popular BerkeleyDB database service [21];
- We compare the complexity with that of related protocols.

The Shuttle implementation that can tolerate arbitrary failures uses, in steady state, $2t + 1$ replicas, HMAC signatures, and has a message overhead of $2t + 2$ messages per operation. The implementation that is designed to deal with crash failures as well as bit flips and Heisenbugs uses only $t+1$ replicas, and its overhead is essentially the same as Chain Replication for update operations (using CRC checksums and $t + 2$ messages per operation). However, read-only operations

are as expensive as update operations, whereas in Chain Replication read-only operations are only 2 messages.

Section 2 presents *Byzantine Chain Replication*, and Section 3 shows the Shuttle protocol. Implementation details are the topic of Section 4. Section 5 evaluates performance characteristics. A comparison with related work is described in Section 6. Section 7 concludes.

2 Byzantine Chain Replication

This section presents a class of replication protocols, rather than a specific protocol. The class is characterized by the state that correct replicas keep and how they order and persist *operations*. For simplicity, we consider a single object. We model the state of the object as a finite history $\mathcal{H}$ of operations, which is a set of $\langle s, o\rangle$ pairs, where s is a *slot number* and o an operation. It has the following property:

$$\forall s, o, o' : \langle s, o\rangle \in \mathcal{H} \wedge \langle s, o'\rangle \in \mathcal{H} \Rightarrow o = o' \tag{1}$$

That is, each slot can have at most one operation assigned to it.

Initially, $\mathcal{H}$ is empty. The only allowed transition on the object is to add an operation to the history at a particular slot (transitions are atomic):

specification `Object`:
  transition `apply`(s, o):
    precondition: $\nexists o' : \langle s, o'\rangle \in \mathcal{H}$
    action: $\mathcal{H} := \mathcal{H} \cup \{\langle s, o\rangle\}$

Note that the history does not have to be filled sequentially, and thus "holes" in the history are allowed. However, if a running state is maintained, operations should be applied in the order of their slot number in the history.

To make the object highly available, we will replicate it and dynamically change the configuration in order to deal with failures. Similar approaches have been proposed for crash failures only [22, 23]. For now suppose there is an unbounded sequence of configurations $\mathfrak{C} = C_1 :: C_2 :: C_3 :: ....$ The boolean function $succ(C, C')$ evaluates to `true` if and only if C' follows C directly in $\mathfrak{C}$. Each configuration C_i is an ordered *chain* of replicas (see Chain Replication [10]), and we assume that the replicas of any two different configurations are disjoint. Given two replicas ρ and ρ' in the same configuration, we write $\rho \prec \rho'$ if ρ precedes ρ' in the chain.

Any replica ρ can create *statements* of the form $\langle d\rangle_\rho$ where d is arbitrary data. For now, assume that d is data signed by ρ using public key cryptography; more efficient schemes will be described later. If ρ is correct, only ρ can create those statements. We say that ρ *says* d. A global state variable *statements* contains all statements of correct replicas, but may also contain statements of faulty replicas.

specification Chain:

transition `orderCommand`(ρ, C, s, o):

precondition:

$\rho \in C \wedge \rho.mode = \texttt{ACTIVE} \wedge$
$\forall \rho' \in C : \rho' \prec \rho \Rightarrow \langle \texttt{order}, s, o \rangle_{\rho'} \in statements \wedge$
$\nexists o', v', \rho' : \langle s, o', \rho', C, v' \rangle \in \rho.history$

action:

$\rho.history := \rho.history \cup \left\{ \left\langle s, o, \rho, C, \{ \langle \texttt{order}, s, o \rangle_{\rho'} \mid \rho' \in C \wedge \rho' \preceq \rho \} \right\rangle \right\}$
$statements := statements \cup \{ \langle \texttt{order}, s, o \rangle_{\rho} \}$

transition `becomeImmutable`(ρ):

precondition:

$\rho.mode = \texttt{ACTIVE}$

action:

$\rho.mode := \texttt{IMMUTABLE}$
$statements := statements \cup \{ \langle \texttt{wedged}, \rho.history \rangle_{\rho} \}$

transition `switchConfig`(C, C', H):

precondition:

$succ(C, C') \wedge \nexists h : \langle \texttt{inithist}, C', h \rangle \in statements \wedge$
$H \subseteq statements \wedge$
$\exists Q \in \mathcal{Q}_C : |H| = |Q| \wedge$
$\quad \forall \rho \in Q : \exists h : \langle \texttt{wedged}, h \rangle_{\rho} \in H \wedge validHist(h, \rho, C)$

action:

$hist := \{ \left\langle s, o, \max(C'), C', \{ \langle \texttt{order}, s, o \rangle_{\rho'} \mid \rho' \in C' \} \right\rangle \mid$
$\quad \exists h, v : h \in H \wedge \langle s, o, v \rangle \in h \wedge$
$\quad \forall h', v', o' : h' \in H \wedge \langle s, o', v' \rangle \in h' :\Rightarrow |v| \geq |v'| \}$
$statements := statements \cup \{ \langle \texttt{inithist}, C', hist \rangle_{\Omega} \}$

transition `becomeActive`$(\rho, C, hist)$:

precondition:

$\rho \in C \wedge \rho.mode = \texttt{PENDING} \wedge \langle \texttt{inithist}, C, hist \rangle_{\Omega} \in statements$

action:

$\rho.history := hist$
$\rho.mode := \texttt{ACTIVE}$

Fig. 1. State transitions allowed in Chain Replication

A correct replica ρ in configuration C has the following local state:

- $\rho.mode$: `PENDING`, `ACTIVE`, or `IMMUTABLE`. Initially all replicas in C_1 are `ACTIVE`, while replicas in all other configurations are `PENDING`;
- $\rho.history$: a set of *order proofs*. An order proof is a tuple of the following form:

$$\left\langle s, o, \rho, C, \{ \langle \texttt{order}, s, o \rangle_{\rho'} \mid \rho' \in C \wedge \rho' \preceq \rho \} \right\rangle. \tag{2}$$

Here s is a slot number, o an operation, and each $\langle \texttt{order}, s, o \rangle_{\rho'}$ is an ***order*** *statement* said by $\rho' \in C$. An order proof contains `order` statements from all replicas that precede ρ in C and from ρ itself. $\rho.history$ cannot contain

conflicting order proofs, that is, order proofs for the same slot number but different operations.

In addition to replicas, there is an oracle Ω that can sign so-called `inithist` statements. Figure 1 shows what transitions are allowed in Chain Replication by correct replicas and by Ω:

1. `orderCommand`(ρ, C, s, o): Any active replica ρ in configuration C can say $\langle \texttt{order}, s, o \rangle_\rho$ if each preceding replica in C, if any, has done likewise and there is no conflicting operation for s in its history. ρ also adds a new order proof to its history.
2. `becomeImmutable`(ρ): An active replica ρ can suspend updating its history by becoming immutable at any time. Typically it will do this only if one or more of its peer replicas are suspected of being faulty. The replica signs a `wedged` statement to notify Ω that it is immutable and what its history is.
3. `switchConfig`(C, C', H): The oracle Ω waits for a set H of *valid* histories from a quorum of replicas in C. $\mathcal{Q}_C$ is the set of quorums defined for configuration C. A valid history contains at most one order proof per slot. The oracle then issues an `inithist` statement for configuration C' with the order proofs of maximal size for each slot s in the histories in H. $\max(C')$ is the last replica (the "tail") on chain C'. The oracle can issue at most one `inithist` statement per configuration.
4. **transition** `becomeActive`$(\rho, C, hist)$: A pending replica ρ in configuration C can become active if the oracle has issued an `inithist` statement for C.

We require that each quorum $Q \in \mathcal{Q}_C$ contain at least one *honest* replica. An honest replica ρ is defined as follows: if issued, $\langle \texttt{wedged}, h \rangle_\rho$ includes all order proofs corresponding to all $\langle \texttt{order}, s, o \rangle_\rho$ statements that ρ issues. In other words, an honest replica cannot truncate its history, nor can it issue new `order` statements after it has become immutable and issued a `wedged` statement.

Clearly, correct replicas are honest, but replicas that suffer only crash failures are honest as well. Later we will argue that replicas that suffer from bit flips and other types of non-malicious failures can be transformed into honest replicas. Replicas that are not honest are easily identified: the combination of an $\langle \texttt{order}, s, r \rangle_\rho$ statement and a $\langle \texttt{wedged}, h \rangle_\rho$ statement that does not contain an order proof for $\langle s, r \rangle$ constitutes a *proof of misbehavior*.

The transitions specify what actions are safe, but not when or in what order to do them.[1] In other words, the system is *asynchronous*. For liveness, we assume that there is always eventually a configuration in which all replicas are correct and do not become immutable.

[1] The transition specifications are not pseudo-code, and should not be confused with implementation. For example, the code corresponding to the `orderCommand` transition would have to check that the order proofs of preceding replicas are signed correctly.

Safety

We show safety through a sketch of a refinement mapping of Specification Chain to Specification `Object`. The statements that are made by the (correct and faulty) replicas map to the state of the object as follows:

$$\mathcal{H} = \{\langle s, o\rangle \mid \exists C \in \mathcal{C} : \forall \rho \in C : \langle \texttt{order}, s, o\rangle_\rho \in \mathit{statements}\} \tag{3}$$

Thus an $\langle s, o\rangle$ pair is in the object history if all replicas of a configuration have issued `order` statements for that pair. We call the $\langle s, o\rangle$ pairs in $\mathcal{H}$ thus defined *persistent*. For this *refinement mapping* to be well-defined, we need to show that Equation 1 holds. We say that $\langle s, o\rangle$ *persists in* C if $C \in \mathcal{C}$ and $\forall \rho \in C : \langle \texttt{order}, s, o\rangle_\rho \in \mathit{statements}$.

Lemma 1. *If $\langle s, o\rangle$ persists in C and the precondition of* `orderCommand`(ρ', s, o') *holds, where ρ' is a correct replica in configuration C' that follows C in $\mathcal{C}$ (not necessarily directly), then $o = o'$.*

Proof. (Sketch) By definition of *persists*, all replicas in C issue an $\langle \texttt{order}, s, o\rangle$ statement. First assume C' is the configuration that directly follows C. Because the precondition of `orderCommand`(ρ', s, o') holds, ρ' is active, and thus the oracle Ω must have issued an `inithist` statement for C'. Thus a quorum of replicas in C must have issued `wedged` statements and Ω obtained order proofs of at least one honest replica in C. The oracle thus obtained an order proof for $\langle s, o\rangle$ from some honest replica ρ. It is not possible for faulty replicas to create a larger order proof for a different $\langle s, o'\rangle$, as that would require that ρ says $\langle \texttt{order}, s, o'\rangle_\rho$ for the larger order proof. This cannot happen because ρ is honest. So if there is a larger order proof, it must be for $\langle s, o\rangle$.

Iteratively, all correct replicas in all configurations that follow C will have an order proof for $\langle s, o\rangle$. As a correct replica never issues an order statement that conflicts with an order proof in its history, the lemma holds.

Theorem 1. *For all s, o, o', C, and C', if $\langle s, o\rangle$ persists in C and $\langle s, o'\rangle$ persists in C', then $o = o'$.*

Proof. (Sketch) By contradiction, consider some s, o, o', C, and C' such that $o \neq o'$. All correct replicas in C ordered $\langle s, o\rangle$, while all correct replicas in C' ordered $\langle s, o'\rangle$. There are two cases:

- $C = C'$: this means that all correct replicas in C ordered both $\langle s, o\rangle$ and $\langle s, o'\rangle$, which is impossible as a correct replica issues at most one `order` statement per slot number.
- $C \neq C'$: wlog., assume C' follows C in $\mathcal{C}$. Because all correct replicas in C' ordered $\langle s, o'\rangle$, they must each have issued `order` statements for $\langle s, o'\rangle$ and thus underwent `orderCommand` transitions for $\langle s, o'\rangle$. By Lemma 1, $o = o'$.

This demonstrates that Equation 1 holds under any Chain transition.

An important question is how to enforce that each quorum in $\mathcal{Q}_C$ contains at least one honest replica. One way is to have each quorum contain at least $t+1$ replicas and thus at least one correct replica. To guarantee liveness each configuration would have to have at least $2t+1$ replicas, or no sufficient number of replicas might issue a `wedged` statement for the precondition of `switchConfig` to become true.

Another option is to assume that all replicas are honest. Under this assumption, quorums can be singletons and, for liveness, each configuration would have to have at least $t+1$ replicas. Such an assumption could be reasonable if malicious failures are unlikely and each replica maintain a checksum of its history, reporting a failure when it detects that its history is compromised. Alternatively, automated approaches that transform hardware errors into crash failures could be used [5, 6]. There is no way to prevent malicious replicas from issuing truncated histories. However, if they do so they expose themselves as provably faulty.

3 Shuttle: A Byzantine Chain Replication Protocol

Shuttle is a BSMR protocol based on Byzantine Chain Replication. Each replica maintains a running state in addition to the local history of order proofs (we will show later how the history can be truncated periodically). A centralized configuration service called *Olympus* implements the oracle Ω and generates a series of configurations, issuing `inithist` statements (signed by a private key held by Olympus) for each such configuration. Any modern datacenter will contain a configuration service similar to the Olympus and can maintain the configurations of many objects. A configuration statement includes the sequence number of the configuration and an ordered list of replica identifiers.

Olympus generates a new configuration upon receiving a `reconfiguration-request` statement. To reduce the efficacy of trivial Denial-of-Service attacks on the object, Olympus only accepts reconfiguration requests that are sent by replicas in the current configuration or accompanied by a proof of misbehavior (such as the aforementioned conflicting `wedged` and `order` statements from the same replica, or any other set of conflicting statements from the same replica that Olympus recognizes). Olympus does not have to allocate a fresh set of replicas for each configuration—doing so would lead to significant overhead as state is transfered from old replicas to new replicas—however, reused replicas need to be made aware of their new configuration. For liveness, we assume that the Olympus eventually creates a configuration of correct replicas that do not issue reconfiguration requests.

3.1 Proofs

Before describing the Shuttle protocol, we generalize the notion of a proof. A *proof* of d is a tuple:

$$\left\langle d, C, \rho, \{\langle d \rangle_{\rho'} \mid \rho' \in C \wedge \rho' \preceq \rho\} \right\rangle. \tag{4}$$

where $C \in \mathcal{C}$ and $\rho \in C$. Typically, ρ is the issuer of the proof. An `order` proof is an example of a proof, that is, a proof of ρ ordering $\langle s, o \rangle$. We will say that a proof is *complete* if ρ is the tail of the chain in C, that is, all replicas in C have signed d.

3.2 Failure-Free Case

Suppose a client wants to execute an operation o and obtain a result. The client first obtains the current configuration from Olympus, and sends o to the head of the chain. The head orders incoming operations by assigning increasing slot numbers to them, and creating *shuttles* that are sent along the chain.

A shuttle contains two proofs: an order proof for $\langle s, o \rangle$ and a result proof for the result of o. Suppose that the head assigns o to slot s. Each replica ρ, including the head, does the following:

1. checks the validity of the order proof in the shuttle;
2. applies o to its running state and obtains a result r;
3. adds $\langle \texttt{order}, s, o \rangle_\rho$ to the order proof (the replica undergoes an `orderCommand` transition);
4. adds $\langle \texttt{result}, o, \mathcal{S}(r) \rangle_\rho$ to the result proof, where $\mathcal{S}$ is a cryptographic hash function;
5. forwards the shuttle to the next replica if any.

After the tail replica adds its `order` statement to the shuttle, the order proof is complete and $\langle s, o \rangle$ is persistent, and the tail forwards the result proof to the client along with the result r itself. The client believes the result if $\mathcal{S}(r)$ corresponds to all `result` statements in the result proof.

The tail also returns the shuttle with the completed proofs to the head along the chain in the reverse order. We will refer to this as the *result shuttle*. Each replica caches this shuttle (and the result r itself) in order to deal with potential failures, as described next.

3.3 Dealing with Failures

In the case of a replica or a network failure, a client may not receive a valid response. The client uses a timer to try to detect a failure. (The value of the timer does not affect safety and can thus be set aggressively in order to detect failures quickly—however, if set too aggressively the overhead of reconfigurations would negatively affect performance.) When the timer expires, the client retransmits its operation to all replicas in the chain. If a (correct) replica that receives the request has the result shuttle cached, it returns the result along with the result proof to the client. If the replica is immutable, it responds to the client with an `error` statement, in which case the client has to retrieve the latest configuration from Olympus and try again. In all other cases, the replica forwards the request to the head (if it is not the head itself), and starts a timer.

The head, if correct and upon receiving a retransmission (either directly from the client or indirectly through one of the peer replicas), distinguishes three cases:

1. it has cached the result shuttle corresponding to the operation;
2. it has ordered the operation but is still waiting for the result shuttle to come back;
3. it does not recognize the operation.

In the first case, it sends the cached result to the client. In the second case, it starts a timer. In the third case, it allocates a new slot number, starts the protocol from scratch, and starts a timer. In the latter two cases, if receiving the result before its timer expires, the head cancels its timer and responds to the client. Upon receipt of the result each replica does likewise, canceling its timer if it is still outstanding, and sending the result along with the result proof to the client. Should the timer expire at one of the replicas, then the replica sends a `reconfiguration-request` statement to Olympus.

To reconfigure, Olympus executes the following steps:

1. send signed `wedge` requests to each of the replicas in the current configuration. Correct replicas respond with `wedged` statements (`becomeImmutable` transition);
2. await responses from a quorum of replicas, and construct a history h by selecting the longest order proof for each slot;
3. allocate a new configuration C of replicas and seed those with h and a configuration statement for C. The replicas then become active (`becomeActive` transition).

3.4 Clients

After sending an operation, the client is waiting for a response that may never arrive. For liveness, the client checks periodically with Olympus to see if there is a new configuration, and if so retransmits its operation. Consequently, an operation may be executed more than once. Although not currently implemented, duplicate execution can be prevented if the service keeps track of which operations it has already executed (on a per-client basis) and treats duplicates as no-ops. The results of such operations may no longer be available, as caching results for ever would present a significant storage problem. Note that the problem and solution are the same for non-replicated servers [24].

Shuttle can tolerate Byzantine clients in that correct Shuttle replicas do not assume that clients follow a particular specification. In particular, Byzantine clients cannot compromise the integrity of the replicated object, for example, by sending conflicting operations to different replicas. This is a direct result of the head of the object making all ordering decisions within its configuration. However, Byzantine clients can send bogus operations to the replicas[2] and they can mount Denial-of-Service attacks by sending lots of bogus operations. Shuttle has no specific defense for such attacks, except that client operations are signed and thus the source is easily identified. It is assumed that such clients are shut down using external means.

[2] A faulty head of the chain can introduce bogus operations by itself.

4 Implementation

We have implemented two versions of the Shuttle protocol:

1. *CRC Shuttle*: A version that is intended to deal with unintentional failures such as power failures, Heisenbugs, bit flips, and so on, but all replicas are assumed to be honest. It uses $t + 1$ replicas in each configuration, singleton quorums, and CRC checksums for signatures.
2. *HMAC Shuttle*: A version that tolerates arbitrary failures, and uses $2t + 1$ replicas in each configuration, quorums of size $t + 1$, and signatures based on HMACs.

We do not have space to describe the entire implementations, but sketch some aspects below. For a complete description, see [25].

4.1 HMAC Vectors

In the case of HMAC Shuttle, a replica signs a statement with a vector of HMAC signatures, with one entry for each receiver that may need to receive the statement. It is thus possible that a faulty replica provides valid signatures for some but not for all destinations. Worse, a recipient of a statement with a valid HMAC signature in its entry of the vector cannot necessarily convince other processes.

Each replica ρ_i has a secret signing key $k_{ij}^{\mathtt{r}}$ for each other replica ρ_j and a secret signing key $k_{ic}^{\mathtt{c}}$ for each client c. The inter-replica signing keys $k_{ij}^{\mathtt{r}}$ are known to the Olympus as well, while the replica-to-client signing keys $k_{ic}^{\mathtt{c}}$ are created using the Diffie-Hellman [26] protocol. A `wedged` statement from a replica to the Olympus is signed by an HMAC vector with an entry for each other replica. The Olympus can check this signature as well as all order proofs as it is in possession of all the necessary keys.

In the current implementations, Olympus is a trusted server that is not replicated itself, but it could be replicated using something like PBFT [12]. Since Olympus has the inter-replica signing keys, a Byzantine replica of the Olympus service might try to leak these keys to Byzantine object replicas. It is thus important that outputs of the Olympus replicas go through a voting filter; only if $t + 1$ of the Olympus replicas send a copy of particular message, the filter will let it through.

4.2 Checkpointing

It would be infeasible in practice for each replica to maintain the entire history of operations, let alone pass on this history to new replicas during reconfiguration. Thus each replica actually maintains a checkpoint and a suffix of the history of operations that comes after this checkpoint. Periodically the head initiates a checkpoint by sending a shuttle down the chain with a *checkpoint proof* for a hash of its running state. Each replica ρ adds a $\langle \texttt{checkpoint}, \mathcal{S}(\mathit{state}) \rangle_\rho$ to the checkpoint proof. The tail returns the completed checkpoint proof along the

chain so each replica can remove the corresponding prefix from its history. The latest checkpoint proof along with the corresponding state is part of a `wedged` statement of a replica.

In HMAC Shuttle, it is not necessary for all $2t+1$ replicas to maintain running state [13]. The chain is split into two parts. The first $t+1$ (including the head) maintain the running state, but the final t (including the tail) do not. These *witness replicas* have to sign the order proofs in the shuttles, but they do so without knowing the running state. Witnesses do not sign the result proof as they cannot compute it. A client needs only $t+1$ matching copies of the result in order to accept it, knowing that at least one correct replica signed the result.

5 Experimental Evaluation

In this section, we evaluate the performance of HMAC Shuttle experimentally and compare it against Chain Replication [10], a protocol that tolerates only crash failures and assumes perfect failure detection. We expect the performance of Shuttle CRC to be close to Chain Replication: they both arrange $t+1$ replicas in a chain. Computing CRC checksums, an operation only carried out in Shuttle CRC, has negligible overhead. Both HMAC Shuttle and Chain Replication are implemented in Java. To evaluate the overhead of these protocols, we put them side-to-side with a non-replicated server in a failure-free scenario.

The code is deployed on a cluster of Linux machines connected by a Gigabit switch. Replicas and clients run on dual-core 2.8 Ghz AMD Opterons. We use 2 and 3 machines for the replicas of Chain Replication and HMAC Shuttle respectively ($t = 1$); clients run on a separate machine. All replicas maintain running state and HMACs are generated with the SHA-256 algorithm using 256-bit keys.

Each replica runs a copy of a BerkeleyDB, a transactional key-value store. The store consists of a list of accounts with their corresponding balances. We consider an *update-only* workload where transactions deposit a random amount of money into a randomly selected account. We configured BerkeleyDB to run entirely in memory as disk latencies tend to largely dominate and obfuscate the protocol overheads. In the benchmarks, each client submits 10 transactions in parallel and waits to receive all responses before submitting the next ones. Each client submits 30,000 transactions and the experiments comprised between 1 and 24 clients.

In Figure 2, we present the average number of committed transactions per second (throughput) versus the average latency for Chain Replication (CR) and HMAC Shuttle (HMAC). In the left graph of Figure 2, we include the performance of a single BerkeleyDB instance. In both graphs, we report the standard deviation of the latency, except for the single BerkeleyDB instance. The latter was consistently smaller than for the two other protocols.

Chain Replication has a slightly higher latency than the single BerkeleyDB instance for a given load: with Chain Replication the transaction goes through two servers instead of a single one. Surprisingly, Chain Replication supports up to 20,000 transactions per second, a higher load than a single BerkeleyDB

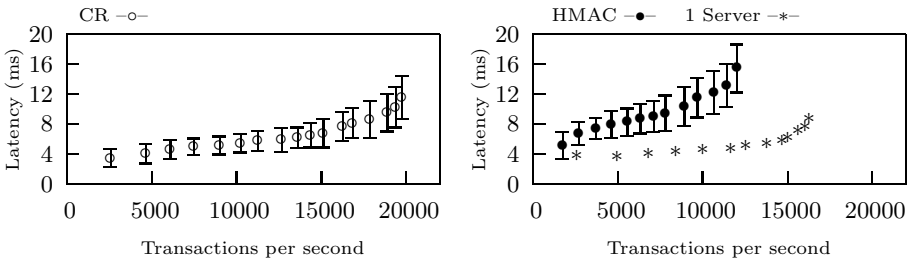


Fig. 2. Throughput vs. Latency of Chain Replication (CR) and HMAC Shuttle (HMAC)

server, which can support about 17,000 transactions per second. It turns out that with Chain Replication the overhead of communication with clients is shared between the head and the tail of the chain—the head receives transactions from clients and the head sends back results. HMAC Shuttle offers good performance for a Byzantine-resilient replication protocol: it supports loads up to 12,000 transactions per second. However, it exhibits higher latencies due to the overhead of verifying and signing statements.

6 Comparison with Prior Work

Many papers have described how to make Byzantine fault tolerance practical. Starting from PBFT [12], various proposals [14, 15, 27–29] aim to reduce latency and increase throughput. Aardvark [30] and Zyzzyvark [31] focus on sustainable performance rather than peak performance. Other proposals focus on reducing the number of full replicas [13, 32, 33].

Table 1 compares various aspects of Shuttle with related work. PBFT is the first Byzantine replication protocol designed for practical use. Zyzzyva [14] is optimized for peak performance. Aliph [15] uses a chain communication pattern, like Shuttle. Zyzzyvark is a recent protocol that tolerates client failures. We consider ZZ [33] the state of the art in reducing replication cost.

We consider the total number of replicas, the number of full replicas, the maximum number of HMAC operations at a replica, the number of message rounds (*aka* message latencies or network latencies), and the effect that Byzantine clients can have on a replicated object. As all of the approaches support batching of multiple requests in order to amortize CPU overhead, we include the effect of batching on the number of crypto operations. In PBFT and ZZ (based on PBFT), a faulty client can trigger reconfigurations in the replicated object. Zyzzyva uses speculative execution, and a faulty client can trigger a rollback that involves multiple rounds and expensive RSA signatures. In Aliph a faulty client can also trigger a rollback.

Table 1. Properties of state-of-the-art BFT replication approaches that tolerate t failures, avoid RSA signatures, and use a batch size b

	PBFT	Zyzzyva	Aliph	Zyzzyvark	ZZ	HMAC Shuttle	CRC Shuttle
total #replicas	$3t+1$	$3t+1$	$3t+1$	$3t+1$	$3t+1$	$2t+1$	$t+1$
#full replicas	$2t+1$	$2t+1$	$3t+1$	$2t+1$	$t+1$	$t+1$	$t+1$
#crypto ops	$2+\frac{8t+1}{b}$	$2+\frac{3t}{b}$	$1+\frac{t+1}{b}$	$2t+\frac{3t}{b}+2$	$2+\frac{10t+3}{b}$	$2+\frac{2t}{b}$	$2+\frac{t}{b}$
#message rounds	4	3	$3t+2$	4	4	$2t+2$	$t+2$
effects of faulty clients	Reconfig.	Rollback	Rollback	None	Reconfig.	None	None

7 Conclusion

Byzantine fault tolerance is becoming increasingly important as we depend more on computer systems, and as those systems have more components that may fail. Byzantine Chain Replication is a new class of replication protocols that needs only few sources of diversity and has modest costs while tolerating a large class of failures. This paper has also presented Shuttle, a simple implementation of a Byzantine Chain Replication protocol, and compared two versions with related work. We find that Byzantine Chain Replication configured with $t = 1$ applied to an in-memory database can support about 70% of the throughput of a non-replicated database, albeit at about twice the latency due to the overhead of checking and verifying HMAC signatures.

References

1. Schneider, F.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
2. Gashi, I., Popov, P., Stankovic, V., Strigini, L.: On Designing Dependable Services with Diverse Off-the-Shelf SQL Servers. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems II. LNCS, vol. 3069, pp. 191–214. Springer, Heidelberg (2004)
3. Vandiver, B., Balakrishnan, H., Liskov, B., Madden, S.: Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In: Proc. of the 21st Symp. on Operating Systems Principles, SOSP 2007, pp. 59–72. ACM (October 2007)
4. Shivakumar, P., Kistler, M., Keckler, S., Burger, D., Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic. In: Dependable Systems and Networks, DSN 2002, pp. 389–398 (2002)
5. Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D.: SWIFT: software implemented fault tolerance. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 243–254 (March 2005)

6. Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 169–182. Springer, Heidelberg (2010)
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proc. of 21st Symposium on Operating Systems Principles (2007)
8. Shafaat, T., Schütt, T., Moser, M., Haridi, S., Ghodsi, A., Reinefeld, A.: Key-based consistency and availability in structured overlay networks. In: Proc. of the 17th Int. Symp. on High-Performance Distributed Computing, HPDC 2008, pp. 235–236. ACM (June 2008)
9. Budhiraja, N., Marzullo, K., Schneider, F., Toueg, S.: The primary-backup approach. In: Mullender, S. (ed.) Distributed Systems, 2nd edn. ACM Press/Addison-Wesley, New York (1993)
10. Van Renesse, R., Schneider, F.: Chain Replication for supporting high throughput and availability. In: 6th Symp. on Operating Systems Design and Implementation, OSDI 2004 (December 2004)
11. Bracha, G., Toueg, S.: Resilient consensus protocols. In: Proc. of the 2nd ACM Symp. on Principles of Distributed Computing, Montreal, Quebec, pp. 12–26. ACM SIGOPS-SIGACT (August 1983)
12. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: Proc. of the 3rd Symposium on Operating Systems Design and Implementation, OSDI 1999, New Orleans, LA. USENIX (February 1999)
13. Yin, J., Martin, J., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating agreement from execution in Byzantine fault-tolerant services. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP 2003, Bolton Landing, NY, pp. 253–268 (October 2003)
14. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative Byzantine fault tolerance. ACM Trans. Comput. Syst. 27(4) (2009)
15. Guerraoui, R., Knezevic, N., Quema, V., Vukolic, M.: The next 700 BFT protocols. In: Proc. of the 5th ACM European Conf. on Computer Systems, EUROSYS 2010, Paris, France (April 2010)
16. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proc. of the 26th ACM Symp. on Principles of Distributed Computing, Portland, OR, pp. 398–407. ACM (May 2007)
17. Andersen, D., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V.: FAWN: A Fast Array of Wimpy Nodes. In: Proc. of the 22nd ACM Symp. on Operating Systems Principles, Big Sky, MT (October 2009)
18. Terrace, J., Freedman, M.: Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In: Proc. of the USENIX Annual Technical Conference, USENIX 2009, San Diego, CA (June 2009)
19. Fritchie, S.: Chain replication in theory and in practice. In: Proceedings of the 9th ACM SIGPLAN Workshop on Erlang (2010)
20. Escriva, R., Wong, B., Sirer, E.: HyperDex: A distributed, searchable key-value store. In: Proceedings of the SIGCOMM Conference, Helsinki, Finland (August 2012)
21. Olson, M., Bostic, K., Seltzer, M.: Berkeley DB. In: Proc. USENIX Annual Technical Conference (1999)
22. Lamport, L., Malkhi, D., Zhou, L.: Brief announcement: Vertical Paxos and Primary-Backup replication. In: Proc. of the 28th ACM Symp. on Principles of Distributed Computing (August 2009)

23. Birman, K., Malkhi, D., Van Renesse, R.: Virtually Synchronous Methodology for Dynamic Service Replication. Technical Report MSR-TR-2010-151, Microsoft Research (2010)
24. Saltzer, J., Reed, D., Clark, D.: End-to-end arguments in system design. Trans. on Computer Systems 2(4), 277–288 (1984)
25. Ho, C.: Reducing costs of Byzantine fault tolerant distributed applications. PhD thesis, Cornell University (May 2011)
26. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory IT-22, 644–654 (1976)
27. Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., Wylie, J.: Fault-scalable Byzantine fault-tolerant services. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles, SOSP 2005, Brighton, UK (October 2005)
28. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In: Proceedings of the Symposium on Operating System Design and Implementation, OSDI 2006. USENIX (2006)
29. Song, Y.J., van Renesse, R.: Bosco: One-Step Byzantine Asynchronous Consensus. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 438–450. Springer, Heidelberg (2008)
30. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making Byzantine fault tolerant systems tolerate Byzantine faults. In: Proceedings of the USENIX Symposium on Network Design and Implementation, NSDI 2009 (2009)
31. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: UpRight cluster services. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP 2009 (October 2009)
32. Li, J., Mazieres, D.: Beyond one-third faulty replicas in Byzantine fault tolerant systems. In: USENIX Symposium on Networked Systems Design and Implementation, NSDI 2007 (2007)
33. Wood, T., Singh, R., Venkataramani, A., Shenoy, P., Cecchet, E.: ZZ and the Art of Practical BFT. In: Proceedings of EuroSys 2011, Salzburg, Austria (2011)

Author Index