Shlomi Dolev
Jorge Cobb
Michael Fischer
Moti Yung (Eds.)

LNCS 6366

Stabilization, Safety, and Security of Distributed Systems

12th International Symposium, SSS 2010
New York, NY, USA, September 2010
Proceedings

NYC 2010

Springer

Shlomi Dolev Jorge Cobb
Michael Fischer Moti Yung (Eds.)

Stabilization, Safety, and Security of Distributed Systems

12th International Symposium, SSS 2010
New York, NY, USA, September 20-22, 2010
Proceedings

Springer

Volume Editors

Shlomi Dolev
Ben-Gurion University of the Negev, Department of Computer Science
Beer-Sheva, Israel 84105
E-mail: dolev@cs.bgu.ac.il

Jorge Cobb
The University of Texas at Dallas, Department of Computer Science
Richardson, TX 75083-0688, USA
E-mail: cobb@utdallas.edu

Michael Fischer
Yale University, Department of Computer Science
51 Prospect Street, New Haven, CT 06511, USA
E-mail: fischer-michael@cs.yale.edu

Moti Yung
Columbia University, Department of Computer Science
New York, NY 10027, USA
E-mail: moti@cs.columbia.edu

Library of Congress Control Number: 2010934646

CR Subject Classification (1998): C.2.4, C.3, F.1, F.2.2, K.6

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

ISSN 0302-9743
ISBN-10 3-642-16022-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16022-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

The papers in this volume were presented at the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), held September 20–22, 2010 at Columbia University, NYC, USA.

The SSS symposium is an international forum for researchers and practitioners in the design and development of distributed systems with self-* properties: (the classical) self-stabilizing, self-configuring, self-organizing, self-managing, self-repairing, self-healing, self-optimizing, self-adaptive, and self-protecting. Research in distributed systems is now at a crucial point in its evolution, marked by the importance of dynamic systems such as peer-to-peer networks, large-scale wireless sensor networks, mobile ad hoc networks, cloud computing, robotic networks, etc. Moreover, new applications such as grid and web services, banking and e-commerce, e-health and robotics, aerospace and avionics, automotive, industrial process control, etc., have joined the traditional applications of distributed systems. SSS started as the Workshop on Self-Stabilizing Systems (WSS), the first two of which were held in Austin in 1989 and in Las Vegas in 1995. Starting in 1995, the workshop began to be held biennially; it was held in Santa Barbara (1997), Austin (1999), and Lisbon (2001). As interest grew and the community expanded, the title of the forum was changed in 2003 to the Symposium on Self-Stabilizing Systems (SSS). SSS was organized in San Francisco in 2003 and in Barcelona in 2005. As SSS broadened its scope and attracted researchers from other communities, a couple of changes were made in 2006. It became an annual event, and the name of the conference was changed to the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS). The last four SSS conferences were held in Dallas (2006), Paris (2007), Detroit (2008), and Lyon (2009).

This year the Program Committee was organized into several tracks reflecting most topics related to self-* systems. The tracks were: (i) Self-Stabilization, (ii) Self-Organization, (iii) Ad-Hoc, Sensor, and Dynamic Networks, (iv) Peer to Peer, (v) Fault-Tolerance and Dependable Systems, (vi) Safety and Verification, (vii) Swarm, Amorphous, Spatial, and Complex Systems, (viii) Security, (ix) Cryptography, and (x) Discrete Distributed Algorithms. The Safety and Verification track is *in Memory of Amir Pnueli.* We received 90 submissions. Each submission was reviewed by at least three Program Committee members with the help of external reviewers. Out of the 90 submitted papers, 39 papers were selected for presentation. The symposium also included brief announcements. Selected papers from the symposium will be published in a special issue of the *Information and Computation* journal.

This year, we were fortunate to have three distinguished invited speakers: Leonid Levin, Mihalis Yannakakis, and Yechiam Yemini.

Among the 39 selected papers, we considered two papers for special awards. The best paper award was given to Anurag Agarwal, Vijay Garg, and Vinit Ogale, for "Modeling and Analyzing Periodic Distributed Computations." The best student paper award was given to Dana Angluin, James Aspnes, Rida A. Bazzi, Jiang Chen, David Eisenstat, and Goran Konjevod for "Storage Capacity of Labeled Graphs."

On behalf of the Program Committee, we would like to thank all the authors who submitted their work to SSS. We thank all the Program Vice-Chairs who managed the various tracks, all the members of the Program Committee, and the external reviewers for their tremendous effort and valuable reviews. We also thank the members of the Steering Committee for their invaluable advice. The process of paper submission, selection, and compilation in the proceedings was greatly simplified due to the strong and friendly interface of the EasyChair system (`http://www.easychair.org`). We are grateful to the Organizing Committee members for their time and invaluable effort, which greatly contributed to the success of this symposium. The support of the EU FRONTS project and the NSF is greatly appreciated.

September 2010

Jorge Cobb
Shlomi Dolev
Michael Fischer
Moti Yung

Conference Organization

General Chair

Shlomi Dolev	Ben-Gurion University of the Negev, Israel

Program Chairs

Jorge Cobb	The University of Texas at Dallas, USA
Michael Fischer	Yale University, USA
Moti Yung	Google and Columbia University, USA

Program Vice-Chairs

Amotz Bar-Noy	City University of New York, USA
Yaneer Bar-Yam	NECSI and Harvard University, USA
Jacob Beal	BBN Technologies and MIT CSAIL, USA
Kirstie Bellman	Aerospace Corporation, USA
Brian Coan	Telcordia Technologies, USA
Thomas Fuhrmann	Technical University Munich, Germany
Jie Gao	Stony Brook University, USA
Benjamin Goldberg	New York University, USA
Mohamed Gouda	National Science Foundation, USA
Gene Itkis	MIT Lincoln Laboratory, USA
Mark Jelasity	University of Szeged, Hungary
Jonathan Katz	University of Maryland, USA
Sandeep Kulkarni	Michigan State University, USA
Franck Petit	Laboratoire d'Informatique de Paris 6, France
Charles Rackoff	Toronto University, Canada
Sergio Rajsbaum	National Autonomous Univ. of Mexico, Mexico
Paul Spirakis	University of Patras, Greece
Cédric Tedeschi	INRIA, France
Oliver Theel	University of Oldenburg, Germany
Philippas Tsigas	Chalmers University, Sweden
Rebecca Wright	Rutgers University, USA
Lenore Zuck	University of Illinois at Chicago, USA

Local Arrangements Chair

Gil Zussman	Columbia University, USA

Publicity Chair

Doina Bein	Pennsylvania State University, USA

Publication Chair

Jorge Cobb	The University of Texas at Dallas, USA

Administration

Sanya Bolbotinovic	Ben-Gurion University of the Negev, Israel
Jessica Rodriguez	Columbia University, USA

Program Committee

Self-Stabilization

James Aspnes
Borzoo Bonakdarpour
Alain Cournier
Ajoy Datta
Murat Demirbas
Stéphane Devismes
Ted Herman
Amos Israeli
Colette Johnen
Toshimisu Masuzawa
Franck Petit (*Chair*)
Elad Michael Schiller
Sébastien Tixeuil
Philippas Tsigas (*Chair*)
Volker Turau
Koichi Wada
Maurice Tchuente

Self-Organization

Dimiter R. Avresky
Kirstie L. Bellman (*Chair*)
Yuriy Brun
Carlos Gershenson
Marie-Pierre Gleizes
Michael Grottke
David Hales
Christian Igel
Mark Jelasity (*Chair*)
Marco Mamei
Alfons Salden
Giovanna Di Marzo Serugendo
Martijn Schut
Praveen Yalagandula

Ad-Hoc, Sensor, and Dynamic Networks

Shigang Chen
Jie Gao (*Chair*)
Anxiao (Andrew) Jiang
Sandeep Kulkarni (*Chair*)
Bhaskar Krishnamachari
Nikola Milosavljevic
Calvin Newport
Ravi Prakash
Guiling Wang
Jennifer Wong

Peer to Peer

Olivier Beaumont
Giuseppe Ciaccio
Davide Frey
Thomas Fuhrmann (*Chair*)
Mark Jelasity
Kendy Kutzner
Erwan Le Merrer
Giancarlo Ruffo
Christian Schindelhauer
Pierre Sens
Georgios Smaragdakis
Cédric Tedeschi (*Chair*)

Fault-Tolerance and Dependable Systems

Yehuda Afek
Marcos K. Aguilera
Rida Bazzi
Brian Coan (*Chair*)
Xavier Défago
Hugues Fauconnier
Christof Fetzer
Felix C. Freiling
Seth Gilbert
Fabíola Greve
Maurice Herlihy
Jonathan Kirsch
Michael Merritt
Lucia Draque Penso
Sergio Rajsbaum (*Chair*)

Safety and Verification (in Memory of Amir Pnueli)

Tevfik Bultan
Marsha Chechik
Benjamin Goldberg (*Chair*)
Holger Hermanns
Warren A. Hunt
Michel Hurfin
Pete Manolios
Achour Mostefaoui
Kedar Namjoshi
Ernst-Rüediger Olderog
Andreas Podelski
Oleg Sokolsky
Neeraj Suri
Oliver E. Theel (*Chair*)
Lenore Zuck (*Chair*)

Swarm, Amorphous, Spatial, and Complex Systems

Yaneer Bar-Yam (*Chair*)
Jacob Beal (*Chair*)
Dan Braha
Rene Doursat
Chris Dwyer
Yuval Elovici
Peter Gacs
Frederic Gruau
Marco Mamei
Olivier Michel
Rami Puzis
Ulrik Schultz
Antoine Spicher
Mirko Viroli
Justin Werfel

Security

Anish Arora	Eunjin Jung
Domagoj Babic	Karl Levitt
Christian Cachin	Alex Liu
Scott Coull	Nasir Memon
Vinod Ganapathy	Rei Safavi-Naini
Mohamed Gouda (*Chair*)	Mukesh Singhal
Xuxian Jiang	Rebecca Wright (*Chair*)

Cryptography

Matthias Fitzi	Jonathan Katz (*Chair*)
Niv Gilboa	Matt Lepinski
Jonathan Herzog	Vladimir Kolesnikov
Gene Itkis (*Chair*)	Moses Liskov
Seny Kamara	Charles Rackoff (*Chair*)

Discrete Distributed Algorithms

Amotz Bar-Noy (*Chair*)	Marios Mavronicolas
Nikhil Bansal	Sriram Pemmaraju
Mordecai J. Golin	Janos Simon
Evangelos Kranakis	Paul Spirakis (*Chair*)
Danny Krizanc	

Steering Committee

Anish Arora	Ohio State University, USA
Ajoy K. Datta	University of Nevada, USA
Shlomi Dolev	Ben-Gurion University of the Negev, Israel
Sukumar Ghosh (*Chair*)	University of Iowa, USA
Mohamed Gouda	The University of Texas at Austin, USA
Ted Herman	University of Iowa, USA
Toshimitsu Masuzawa	Osaka University, Japan
Vincent Villain	Université de Picardie, France

Additional Reviewers

Luca Maria Aiello	Marc Bruenink	Swan Dubois
S. M. Iftekharul Alam	Santosh Chandrasekar	Lionel Eyraud-Dubois
Yaniv Altshuler	Julien Clement	Maria Gradinariu
Nazareno Andrade	Charles Consel	Sascha Grau
Luciana Arantes	Alejandro Cornejo	Taisuke Izumi
Marin Bertier	Alexandre Donzé	Hirotsugu Kakugawa

Table of Contents

Arcane Information, Solving Relations, and Church Censorship

Leonid A. Levin

Boston University
University of Heidelberg
Humboldt Foundation
http://www.cs.bu.edu/fac/lnd

Abstract. Church-Turing Thesis fails for problems that allow multiple answers: many easily solvable problems allow only non-recursive solutions. Its corrected version is: Physical and Mathematical Sequences have Little Common Information. This requires extending Kolmogorov's concept of mutual information to infinite strings. This is tricky; the talk will survey these and other related issues. Related Information can found at: http://arxiv.org/abs/cs.CC/0203029.

Computation of Equilibria and Stable Solutions

Mihalis Yannakakis

Department of Computer Science
Columbia University
455 Computer Science Building
1214 Amsterdam Avenue, Mail Code 0401
New York, NY 10027
mihalis@cs.columbia.edu

Abstract. Many models from a variety of areas involve the computation of an equilibrium or stable solution of some kind. Examples include Nash equilibria in games; price equilibria in markets; optimal strategies and the values of competitive dynamic games (stochastic and other games); stable configurations of neural networks; analysis of stochastic models like branching processes and recursive Markov chains. It is not known whether these problems can be solved in polynomial time. Despite their broad diversity, there are certain common computational principles that underlie different types of equilibria and connect many of these problems to each other. In this talk we will discuss some of these common principles, the corresponding complexity classes that capture them, and their relationship with other open questions in computation.

A Geometry of Networks

Yechiam Yemini

Department of Computer Science
Columbia University
450 Computer Science Building
1214 Amsterdam Avenue, Mailcode: 0401
New York, New York 10027-7003
yemini@cs.columbia.edu

Abstract. This presentation will describe a coordinate geometry of networks and its applications to mobility, security, overlays and traffic management. Given a network graph, one can construct an abstract group and associate elements of the group with graph nodes to provide "coordinates". Much like Cartesian coordinates, a route may be simply computed from the coordinates of the source and destination. Given a metric of link "length" (e.g., delay, utilization), one may easily select shortest-distance routes. Furthermore, this route selection may vary for each source-destination stream, or even for each packet. For example, unlike current networks, one may pursue dispersion-routing where a stream of packets is routed over multiple paths to load-balance traffic. The coordinates geometry admits dynamic topology changes and may thus be used to support various forms of mobility, including mobile adhoc networks, or dynamic deployment of virtual-machines through cloud infrastructures. A given network may admit multiple independent geometry overlays. Each such overlay can serve as a VPN to protect access to respective nodes. Finally, geometry overlays may be flexibly layered over each other, permitting simple scalability, applications-specific-networks and flexible private networks.

Systematic Correct Construction of Self-stabilizing Systems: A Case Study

Ananda Basu[2], Borzoo Bonakdarpour[1,⋆], Marius Bozga[2], and Joseph Sifakis[2]

[1] Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
[2] VERIMAG
2 Avenue de Vignate, 38610, Gières, France

Abstract. Design and implementation of distributed algorithms often involve many subtleties due to their complex structure, non-determinism, and low atomicity as well as occurrence of unanticipated physical events such as faults. Thus, constructing correct distributed systems has always been a challenge and often subject to serious errors. We present a methodology for component-based modeling, verification, and performance evaluation of self-stabilizing systems based on the BIP framework. In BIP, a system is modeled as the composition of a set of atomic components by using two types of operators: interactions describing synchronization constraints between components, and priorities to specify scheduling constraints. The methodology involves three steps illustrated using the distributed reset algorithm due to Arora and Gouda. First, a high-level model of the algorithm is built in BIP from the set of its processes by using powerful primitives for multi-party interactions and scheduling. Then, we use this model for verification of properties of a self-stabilizing algorithm. Finally, a distributed model which is observationally equivalent to the high-level model is generated.

Keywords: Component-based modeling, Verification, Self-stabilization, Distributed algorithms, Reset algorithms.

1 Introduction

Distributed systems are constructed from a set of relatively independent components that form a unified, but geographically and functionally diverse entity. They remain difficult to design, build, and maintain, because of their inherently concurrent, non-deterministic, and non-atomic structure as well as the occurrence of unanticipated physical events such as faults.

We currently lack disciplined methods for rigorous design and correct implementation of distributed systems. These systems are still being constructed in

⋆ For all correspondence, please contact Borzoo Bonakdarpour at borzoo@ecemail.uwaterloo.ca.

an ad-hoc fashion in practice, mainly for two reasons: (1) formal methods are not easy to use by engineers; and (2) there is a wide gap between modeling formalisms and automated verification tools on one side, and practical development and deployment tools on the other side. In fact, it is not clear how existing results can be consistently integrated in design and implementation methodologies. Formalisms such as process algebras [1], I/O automata [13,17], and Unity [9] have been used for modeling and reasoning about the correctness of distributed systems. These methods are either too formal to be used by engineers, or, they require the designer to specify low-level elements of a distributed system such as channels and schedulers [17]. Numerous techniques and algorithms have also been introduced for adding reliability and fault-tolerance to distributed systems. Moreover, an interest has recently emerged in verification of distributed algorithms. While these approaches play an important role in formalizing and achieving correctness of distributed algorithms, we believe that a more practical systematic approach for modeling, verification, and as importantly deployment of distributed systems is still required.

In this paper, we apply a methodology which consistently integrates modeling, verification, and deployment techniques, based on the BIP (Behavior, Interaction, Priority) framework [4,3]. BIP is based on a semantic model encompassing composition of heterogeneous components. In contrast to all other formalisms using a single type interaction (e.g., rendezvous, asynchronous message passing), BIP uses two families of composition operators for expressing coordination between components: *interactions* and *priorities*. Interactions are expressed by combining two protocols: rendezvous and broadcast, which makes BIP more expressive than any formalism based on a single type of interaction [5]. Supporting tools of BIP's theory include techniques for model verification [15] as well as for generating from a high-level model an observationally equivalent multi-threaded or distributed implementation [3,6,7].

To illustrate our methodology, we focus on *self-stabilizing* systems. Pioneered by Dijkstra [10], a self-stabilizing distributed algorithm guarantees that starting from an arbitrary state, it converges to a legitimate state (from where it satisfies its specification) and remains thereafter. As Dijkstra points out in a belated proof of correctness of his token ring algorithm [11], designing and deploying correct self-stabilizing algorithms is not a trivial task at all, although it initially seems straightforward. We describe our methodology to overcome these difficulties using the distributed reset self-stabilizing algorithm [2]. We demonstrate how refinement of a simple algorithm to a less high-level model involves many subtleties that may dramatically affect the correctness of the refined model. We also show how BIP facilitates rigorous modeling, verification, and performance analysis of the distributed reset algorithm. Our methodology involves three steps:

– The starting point is a high-level BIP model of a distributed system obtained as the composition of a set of components. This model represents a system with a global state and atomic transitions. Interactions may lead the system from one global state to another. Modeling a distributed system in such a high-level model confers numerous advantages such as modularity by using abstract behavioral components and faithfulness as coordination is directly expressed by using abstract multi-party interactions instead of low-level primitives. We also show how different functions of a self-stabilizing system (e.g., normal as well as recovery) can be elegantly modeled in BIP in an incremental manner.

- We use this compact high-level model for verification of safety and liveness properties that any self-stabilizing algorithm must satisfy. These properties include *closure*, *deadlock-freedom*, and *finite reachability* of the set of legitimate states. We verify these properties on our BIP model for distributed reset by using model checking techniques.
- Finally, a multi-threaded or distributed executable C++ code is automatically generated from the high-level model for simulations and experiments [3, 6, 7]. This C++ code faithfully represents an actual multi-threaded or distributed implementation of the high-level model. It is obtained by applying two transformations preserving observational equivalence [3,6,7]: (1) multi-party interactions are substituted by protocols based on asynchronous message passing; (2) the state of a component is undefined (due to concurrency) when it performs some internal computation.

Organization of the paper. In Section 2, we review the distributed reset algorithm and basic concepts of the BIP framework. In Section 3, we formally model distributed reset in BIP. Section 4 is dedicated to verification of distributed reset. Finally, we conclude in Section 5.

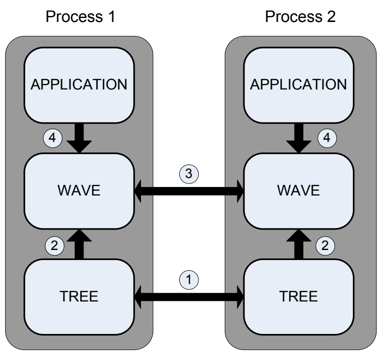
2 Background

2.1 Distributed Reset

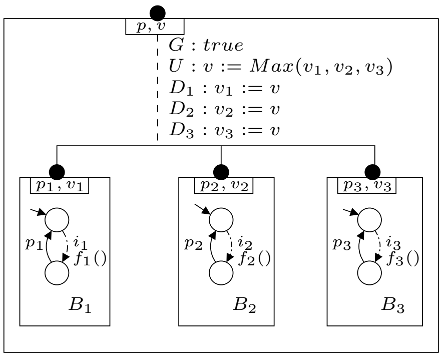
Intuitively, distributed reset [2] augments functionality of a distributed system with a subsystem where each process can initiate a global reset to a predefined global state. Each process is associated with a set of adjacent processes with which it can communicate. At any time instant, an alive process may crash which results in change of the list of adjacent processes. The reset subsystem consists of the following three layers (see Figure 1-a):

In the tree layer, adjacent processes communicate in order to construct and maintain a rooted spanning tree throughout the alive processes. Thus, any changes in the adjacency relationship of processes eventually result in corresponding changes in the structure of the spanning tree. The tree layer is self-stabilizing in that starting from any arbitrary topology and initial structure, construction of a rooted spanning tree within a finite number of steps is guaranteed. Thus, faults such as process failures and local variable corruptions do not result in permanent destruction of the spanning tree. Communication among these processes establish Channel 1 in Figure 1-a.

The application layer may locally choose to initiate a global reset. In this case, the corresponding local component sends a request to the local wave layer described next (see Channel 4 in Figure 1-a).



(a) Two adjacent processes in distributed reset.



(b) A simple BIP model.

Fig. 1. Preliminary concepts

When the wave layer receives a reset request from the application layer it forwards the request to its parent in the current spanning tree until the request reaches the root. Once the root receives a reset request, it initiates a *diffusing computation* as follows. First, the root resets its own state and then initiates a *reset wave*. The reset wave travels towards the leaves of the spanning tree and causes the wave component of each encountered process to reset its state. When the reset wave reaches a leaf process, it bounces as a *completion wave* that travels towards the root process. A process propagates the completion wave to its parent if all its offsprings are complete (see Channel 3 in Figure 1-a). When the completion wave reaches the root, the global reset is complete. Each wave component maintains a *session number* in order to ensure that concurrent resets do not interfere. The wave layer is also self-stabilizing in the sense that starting from any arbitrary configuration of the wave components, the algorithm guarantees an eventual global reset within a finite number of steps. The wave layer always assumes the existence of a sound rooted spanning tree. Thus, the only piece of information that a tree component shares with the corresponding local wave component is the identity of the parent process in the spanning tree (see Channel 2 in Figure 1-a).

2.2 The BIP Framework

In the BIP language [16, 4], an architecture is characterized as a hierarchically structured set of *components* obtained by composition from a set of atomic components. Composition is parameterized by sets of *interactions* between the composed components. The BIP toolset has a compilation chain allowing the generation of different types of C++ code (e.g., monolithic, real-time, multi-threaded, distributed, etc) from BIP models. The generated code is modular and can be executed on a dedicated middleware consisting of one or more

Engines that orchestrate the computation of atomic components by executing their interactions. Hierarchical description allows incremental reasoning and progressive design of complex systems. *Priorities* among interactions allow specifying scheduling policies in BIP.

A BIP component is characterized by its *interface* and its *behavior.* An interface consists of a set of *external* ports used to specify interactions. Each port p is associated with a set v_p of variables which are visible when an interaction involving p is executed. It is assumed that the ports and associated variables of atomic components are disjoint. The behavior of atomic components is described as a finite state automaton extended with data and functions given in C++. A transition of the automaton is labeled by (1) a port p through which an interaction is sought, (2) a function f describing a local computation, and (3) a guard g on local data. For a given control state, a transition can be executed if its guard g is true and an interaction involving p is possible (we precisely define the notion of interactions later in this section). Execution of transitions is atomic: it is initiated by the interaction and followed by the execution of f. A component may have *internal* ports as well. Transitions labeled by internal ports are executed independently and do not require initiation of an interaction.

Composition consists of applying a set of *connectors* to a set of components. A connector is defined by:

1. its *support set of ports* $\{p_1, \ldots, p_n\}$ of the composed components;
2. optionally an *exported port* p by the connector and the associated variables;
3. its set of *interactions*, that are, subsets of the set $\{p_1, \ldots, p_n\}$. Each interaction $\alpha = \{p_{i_1} \ldots p_{i_k}\}$ is annotated by
   (a) a *guard* G, Boolean expression involving variables associated with the ports p_{i_j} involved in the interaction α;
   (b) an *upstream transfer function* U specifying flow of data from variables associated with the support set of ports towards the associated variables of the exported port;
   (c) and *downstream transfer functions* $D_{i_1}, \ldots, D_{i_k}$ specifying flow of data from the variables associated with the exported port towards variables associated with the support set of ports.

When it is clear from the context, we simply denote a connector by only its support set of ports (i.e., $\langle p_1 \ldots p_n \rangle$). The set of interactions associated with a connector is defined using a typing mechanism of ports in its support set of ports. We distinguish two types of ports: *synchron* and *trigger.* Any set of support ports that is either maximal or it contains a trigger denotes a valid interaction. Intuitively, a synchron is a passive port, and needs synchronization with other ports. In other words, such a port cannot initiate an interaction without synchronizing with other ports. However, a special case (such as the one in Figure 1-b) is a connector that only involves synchrons. Such a connector denotes a *rendezvous* and requires all ports to participate. On the other hand, a trigger is an active port, and can initiate an interaction without synchronizing with other ports. The global behavior resulting from the application of a connector to a set of components is defined as follows. An interaction $\alpha = \{p_{i_1} \dots p_{i_k}\}$ of the connector is enabled only if for each one of its ports p_{i_j}, there exists an enabled transition in some component labeled by p_{i_j}. Execution of the interaction involves two steps:

1. a temporary variable v is assigned the value $U(v_{p_{i_1}}, \dots, v_{p_{i_k}})$;
2. the variables v_{i_j} associated with the ports p_{i_j} are assigned values $D_{i_j}(v)$.

The execution of an interaction is followed by the execution of the local computations of the synchronized transitions. A *composite component* is recursively obtained from a set of atomic or sub-components by successive (i.e., acyclic) application of connectors. The support set of any connector contains ports exported either by sub-components or other existing connectors.

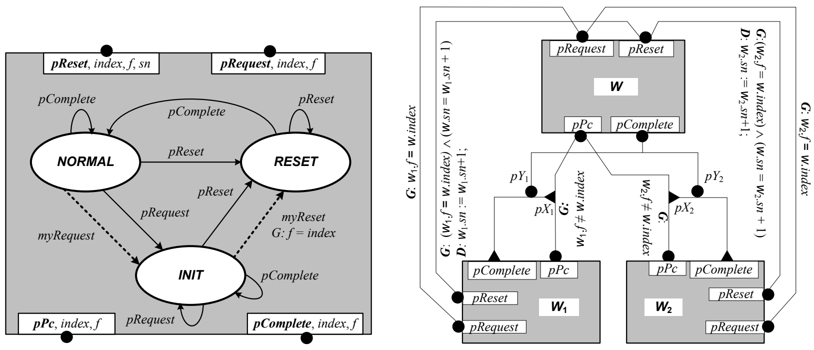
In Figure 1-b, we provide a simple composite component. It is composed of three atomic components B_1, B_2, and B_3. Each atomic component B_k holds an integer variable v_k, exported through an external port p_k. Additionally, the component has an internal port i_k which triggers the execution of an internal computation defined by the function f_k. The ternary connector defines the interaction $\{p_1, p_2, p_3\}$ which is a rendezvous among external ports p_1, p_2, and p_3. As a result of this interaction, following the definition of upstream an downstream transfer functions, each component receives the maximum of the exported values. Notice that the exported port of the connector belongs to the interface of the composite component, that is, it can be used for further interactions.

3 Modeling Distributed Reset in BIP

We model distributed reset according to the BIP system construction methodology: (1) designing the *behavior* of each atomic component (i.e., an automaton extended by variables and ports), (2) applying synchronization mechanisms for ensuring coordination of distributed components through *interactions*, and (3) specifying scheduling constraints by using *priorities*. We apply this methodology to model the wave layer and the tree layer in a modular manner in Subsections 3.1 and 3.2, respectively. Then, we add cross-layer connectors in Subsection 3.3. We also systematically model *normal*, *recovery*, and *faulty* behaviors of distributed reset using independent interactions. From the wave and tree components designed in this section, one can incrementally build a distributed system equipped with the distributed reset functionality according to a topology of interest.

3.1 The Wave Layer

The wave layer is only concerned with achieving a self-stabilizing diffusing computation to accomplish a distributed reset. Each process in the distributed system contains a *wave atomic component*.



(a) Behavior and interface (b) Interactions

Fig. 2. Normal operation of the wave layer

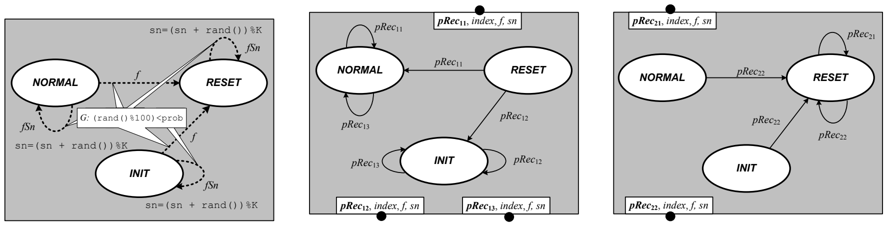
Normal Operation. We start with modeling the normal operation of the wave layer, where each component works correctly in the absence of faults.

Interface and Behavior

– *(Exported Ports)* A wave component has the following four ports: (1) *pRequest* for propagating a reset request from a child to its parent, (2) *pReset* for enforcing a child to reset its state by the parent, (3) *pComplete* for informing a node that its subtree has completed diffusing computation, and (4) *pPc* for identifying adjacent processes that are neither a child nor a parent. As can be seen in Figure 2-a, each port is associated with a subset of variables of the component.
– *(Variables)* Each component maintains the following variables: (1) an integer *index* to represent the unique index of the component, (2) an integer *f* to keep the index of the parent process in the spanning tree, and (3) an integer *sn* for the session number of the current ongoing reset.
– *(Automaton)* A wave component has three *control states*: NORMAL, INIT, and RESET (see Figure 2-a). Initially, all components are in the NORMAL control state. A wave component may move to INIT by either enabling the *myRequest* internal port (e.g., from the application layer of the same process) or when a reset request is received via the *pRequest* port. This move occurs during the *request wave*. Next, the component moves from INIT to RESET and resets its state when the port *pReset* is enabled during the *reset wave*. A component may also move from INIT to RESET on port *pReset*, if it was not involved in the request wave. Finally, a wave component moves back to NORMAL on port *pComplete*, when its subtree has completed the completion wave. A completed wave component is either in NORMAL control state or in INIT if another reset has already been initiated in its subtree. The *pComplete* self-loop at this control state is added for this reason.

Interactions
Notice that each process is associated with a set of adjacent processes according to a topology. The static design of connectors should provide the potential of communication between any two adjacent processes depending upon the topology. Nonetheless, the actual communication in the wave layer should occur only between processes that are allowed to do so by the parent-child relationship determined by the tree layer. Let w be a wave component whose adjacent neighbors are $w_1 \cdots w_n$. We categorize the interactions based on the three waves of the wave layer:

– *(Request Wave)* The first set of connectors is $\{\langle(w.pRequest)(w_i.pRequest)\rangle \mid 1 \leq i \leq n\}$. These connectors allow the component w at *NORMAL* to synchronize with a component w_i, that is already in control state *INIT*: w_i synchronizes with w by taking the *pRequest* self-loop at control state *INIT*. Figure 2-b presents an example, where w has two adjacent processes w_1 and w_2. The connectors between *pRequest* ports are associated with a guard to ensure correct parent-child relationship and bottom-up flow of requests (e.g., $w.index = w_1.f$). Hence, if two processes are adjacent due to the topology, but not in any parent-child relationship, they do not interact to send or receive reset requests. This guard is present in almost all of the connectors in the wave layer. Symmetric conditions in adjacent processes (e.g., w_1 is parent of w) are omitted from the figure for simplicity. Recall that since BIP allows us to associate ports with variables, evaluation of the above guard does not require explicit use of shared memory.
– *(Reset wave)* The second set of connectors is $\{\langle(w.pReset)(w_i.pReset)\rangle \mid 1 \leq i \leq n\}$. Once the root (of the spanning tree) wave component moves to *INIT*, it goes to *RESET* without synchronizing on port *pReset*. This is managed through specifying an internal transition from *INIT* to *RESET* with guard $(w.f = w.index)$. Once a process is in *RESET*, its children can go to *RESET* from either *NORMAL* or *INIT* by synchronizing on port *pReset*. In other words, a child whose parent is in *RESET* can reset its state regardless of its past desire to initiate a global reset. A parent synchronizes with its resetting children through the *pReset* self-loop at control state *RESET*. The guard of these connectors ensures that the session number of a child is one less than the session number of its parent. Finally, when the reset connector gets enabled, it increments *sn* of the child component to mark the session number of the current reset wave.
– *(Completion wave)* A process declares completion only if all its children are complete (which essentially means its entire subtree is complete). The completion mechanism inherently requires a multi-party rendezvous. However, our design should be flexible in that it allows bypassing adjacent processes that are neither a parent nor a child. To this end, we construct a hierarchical connector as follows. First, we include a connector between *pPc* ports of w and w_i, where $1 \leq i \leq n$, which gets enabled when w and w_i are *not* in a parent-child relationship. This connector exports the trigger port pX_i,



(a) Faulty behavior (b) Recovery type 1 (c) Recovery type 2

Fig. 3. Self-stabilization of the wave layer

which gets enabled when the completion of w_i is irrelevant to w. Now, the pair of pX_i and $w_i.pComplete$ constructs another connector, which exports the port pY_i. This port is present in the rendezvous that covers all w_i components. The full interaction can be characterized by the following rendezvous: $\langle (w.pComplete) pY_1 pY_2 \cdots pY_n \rangle$, where $pY_i = \langle (pX_i) + (w_i.pComplete) \rangle$ and $pX_i = \langle (w.pPc)(w_i.pPc) \rangle$. The '+' operator denotes a choice between two enabled ports.

The set of *legitimate states* for two wave components w_1 and w_2 is the following:

$$\begin{aligned}\mathcal{S}_w \equiv \forall w_1, w_2 :: & ((w_1.f = w_2.index \wedge \neg\, w_2.\mathsf{RESET}) \;\Rightarrow \\ & \qquad (\neg w_1.\mathsf{RESET} \wedge w_1.sn = w_2.sn)) \;\wedge \\ & ((w_1.f = w_2.index \wedge \quad w_2.\mathsf{RESET}) \;\Rightarrow \\ & \qquad ((\neg w_1.\mathsf{RESET} \wedge w_2.sn = w_1.sn + 1) \;\vee\; w_2.sn = w_1.sn)).\end{aligned}$$

Faulty Behavior. In distributed reset, faults can lead a process to reach any arbitrary state in $\neg\mathcal{S}_w$ (See Figure 3-a). The transitions labeled by internal port f cause a process to go to *RESET* from either *INIT* or *NORMAL* without synchronizing with its parent. Faults labeled by *fSn* are self-loops that corrupt the session number of a process by executing the C++ instruction `sn = (sn + rand()) % K`, where `K` is the maximum number of processes. To make the occurrence of faults a random event, we associate the guard of fault transitions with a probability `prob`. Notice that the union of transitions in Figures 2-a and 3-a may lead a wave component to reach any arbitrary state. Finally, fault transitions are labeled by internal ports making their occurrence independent of synchronization constraints.

Self-stabilization

Interface and Behavior. We model self-stabilization of the wave layer based on violation of either conjuncts of $\mathcal{S}_w$. Essentially, the recovery mechanism should ensure that starting from any state in $\neg\mathcal{S}_w$, the entire distributed system can reach a state in $\mathcal{S}_w$ within a finite number of steps. For the first conjunct (see Figure 3-b), first, we consider the case where a parent process is not in *RESET*, but one of its children is. To resolve this case, it suffices for the child to (1) move to the control state where its parent is (i.e., either *NORMAL* through synchronization on port $pRec_{11}$ or *INIT* through port $pRec_{12}$), and (2) copy the session number from the parent to ensure consistency. Then, to resolve the case where a parent and its child are in the same control state but their session numbers differ, the processes synchronize on port $pRec_{13}$ and the child copies the parent's session number.

For the second conjunct (see Figure 3-c), if a process and one of its children are in *RESET*, but their session numbers differ, then they synchronize on port $pRec_{21}$ and the child copies the session number. Finally, if a process is in *RESET*, but one of its children is not in *RESET* and the child's session number is not one less than its parent's, then they synchronize on port $pRec_{22}$ and the child copies the session number.

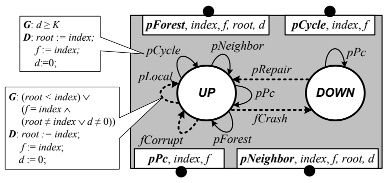
Interactions. Recovery connectors define interactions on corresponding ports between adjacent components. Thus, the set of connectors is $\{\langle (w.pRec_{jk})(w_i.pRec_{jk})\rangle \mid (i = 1..n) \wedge (j = 1..2) \wedge (k = 1..3)\}$, where w_i is adjacent to w.

3.2 The Tree Layer

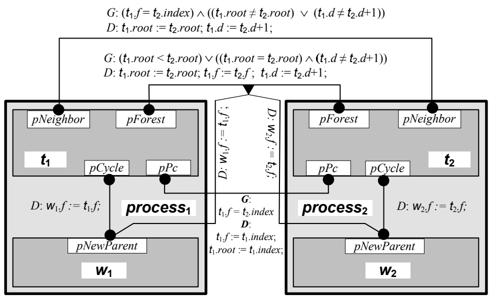
The tree layer is concerned with a self-stabilizing algorithm for constructing a rooted spanning tree (see Figures 4-a and 4-b)

Interface and Behavior

- *(Exported Ports)* Adjacent processes in the tree layer communicate via three ports: (1) *pForest* when two adjacent processes identify two different roots, (2) *pNeighbor* when two a parent and a child identify an inconsistency between them (i.e., existence of multiple roots, incorrect shortest distance to the root, or a root process that is not self-parent), and (3) *pPc* when a parent process crashes. Port *pCycle* is used for cross-layer interactions described in Subsection 3.3.
- *(Variables)* Each tree component maintains the following variables: (1) an integer *index* to represent the unique index of the component, (2) an integer f to keep the index of the parent process in the spanning tree, (3) an integer *root* that contains the index of the root process, and an integer d whose value is the distance of the process to the root. The value of *index* is equal to that of the corresponding wave component and is specified statically. The value of f, however, is determined at runtime across the tree layer. Thus, the tree and wave components of a process need to communicate to maintain consistency. We address this issue in Subsection 3.3. Each component also maintains an array N, which contains the index of all adjacent processes.
- *(Automaton)* Initially, all processes are alive and in the *UP* control state. Faults can alter the value of variables f, *root*, and d arbitrarily through the



(a) Tree component


G: (t1.f = t2.index) ∧ ((t1.root ≠ t2.root) ∨ (t1.d ≠ t2.d+1))
D: t1.root := t2.root; t1.d := t2.d+1;
G: (t1.root < t2.root) ∨ ((t1.root = t2.root) ∧ (t1.d ≠ t2.d+1))
D: t1.root := t2.root; t1.f := t2.f; t1.d := t2.d+1;
pNeighbor
pForest
t1
pCycle
pPc
D: w1.f := t1.f;
process1
pNewParent
w1
D: w1.f := t1.f;
D: w2.f := t2.f;
G:
t1.f = t2.index
D:
t1.f := t1.index;
t1.root := t1.index;
pForest
pNeighbor
pPc
pCycle
t2
process2
D: w2.f := t2.f;
pNewParent
w2

(b) The tree layer and cross-layer interactions

Fig. 4. The tree layer

internal port *fCorrupt*. Also, each process may crash and go to the control state *DOWN* through the internal port *fCrash*. A crashed process may get repaired and return to the *UP* control state through internal port *pRepair*. Thus, faults can potentially break a rooted tree into forests, create cycles, and cause (local or global) inconsistencies. A tree component participates in resolving the above issues when it is in control state *UP*. A local inconsistency is detected in a tree component through the internal port *pLocal* associated with a guard which indicates a discrepancy in the value of either *root* or *d*. A cycle can also be detected locally, if the distance of a process to the root is greater than the maximum number of processes K. A tree component fixes a local inconsistency and breaks a cycle by setting $root = f = index$ and $d = 0$.

Interactions

Let t be a tree component whose adjacent processes are $t_1..t_n$. The interactions between tree components resolve the following issues to construct a rooted spanning tree. Recall that interactions between tree components construct Channel 1 of Figure 1-a:

– *(Process crashes)* The set $\{\langle (t.pPc)(t_i.pPc)\rangle \mid 1 \leq i \leq n\}$ of connectors are used to inform a process that its parent has crashed. As can be seen in Figure 4, this connector is enabled when one participating component is in *UP* and the other process is in *DOWN* control state. The guard of the connector enforces the parent-child relationship. Execution of this interaction invalidates the variables of the child process whose parent is crashed.
– *(Parental inconsistencies)* A connector in the set $\{\langle (t.pNeighbor)(t_i.pNeighbor)\rangle \mid 1 \leq i \leq n\}$ is enabled when a child and its parent either do not agree on the same root, or, the child is not located one step farther of its parent from the root. In either case, the child simply fixes the root index and its distance according to the parent through the data

transfer mechanism of the connector (see the guard G and transfer function D of the connector in Figure 4-b).

– *(Rooted forests)* A connector in the set $\{\langle(t.pForest)(t_i.pForest)\rangle \mid 1 \leq i \leq n\}$ is enabled when multiple roots are detected by a tree component. This situation occurs when there exists an adjacent process whose root has a higher index or the process offers a shorter distance to the root. In this case, the process updates its *root*, f, and d variables via the data transfer mechanism (see the guard G and function D of the connector in Figure 4-b).

Finally, we define the set of legitimate states of the tree layer, where a rooted tree that spans over all alive processes exists, as follows:

$$\begin{aligned}\mathcal{S}_t \equiv\ & (k = \max\{t.index \mid t.\textsf{UP}\}) \wedge \\ & (\forall t_1 \mid t_1.\textsf{UP} :: (t_1.index = k \quad \Rightarrow \\ & \quad (t_1.index = t_1.root \ \wedge\ t_1.index = t_1.f \ \wedge\ t_1.d = 0)) \wedge \\ & \qquad\qquad (t_1.index \neq k \quad \Rightarrow \\ & \quad (\exists t_2 \in t_1.N :: (t_1.f = t_2.index \ \wedge\ t_1.d = t_2.d + 1 \ \wedge \\ & \qquad\qquad\qquad\qquad \forall t_3 \in t_1.N :: t_2.d \leq t_3.d)))).\end{aligned}$$

3.3 Building Distributed Reset

Given the tree layer and wave layer components, one can easily compose them and incrementally build a distributed reset system. To this end, we add cross layer interactions as follows. When a cycle or multiple forests are detected in the tree layer, a tree component may choose a new parent from its neighbors. In this case, the wave component of the same process has to update its parent as well, so the subsequent resets complete maturely (see Channel 2 in Figure 1-a). Thus, we augment each wave component with a *pNewParent* port, which synchronizes with *pCycle* or an exported port by the *pForest* connectors to update its parent (see Figure 4-b).

4 Model Checking Distributed Reset

For a finite instantiation of the algorithm by a grid topology, we start by constructing a finite representation of its overall behavior as a flat labeled transition systems (Lts) using BIP state-space explorer [4]. States correspond to configurations reached by the algorithm, and transitions taken to move from one configuration to another are labeled by the interactions introduced in Section 3. On the Lts model, we have evaluated a set of temporal logic formulas encoding the key properties of distributed reset, using the Evaluator tool of CADP [12,14].

We express the properties using a generic characterization of interactions (i.e., labels). We add a self-loop labeled *steady* to each legitimate state. For the wave layer (respectively, tree layer), all these self-loops participate in a global rendezvous interaction whose guard satisfies expression $\mathcal{S}_w$ (respectively, $\mathcal{S}_t$) introduced in Section 3. We label each internal fault transition introduced in Section 3 by *fault*. This labeling makes the occurrence of a fault an observable event.

We label the remaining interactions by *prog*. This includes recovery as well as interactions that participate in constructing a spanning tree at the tree layer and interactions that contribute in achieving a global reset at the wave layer.

We provide the exact definition of properties in *regular alternation-free μ-calculus* which is the temporal logic formalism handled by the Evaluator tool. This logic is an extension of the alternation-free μ-calculus with action formulas as in ACTL and regular expressions over action sequences as in PDL. The full syntax and semantics can be found in [14]. We consider the following properties that any self-stabilizing system must satisfy:

– *(closure)* legitimate states are preserved by taking non-fault actions (only faults may reach an illegitimate state from a legitimate state):
  $\phi_1 : [any^*]\,(\langle steady\rangle\mathbf{T} \Rightarrow [prog]\langle steady\rangle\mathbf{T})$[1]
– *(deadlock-freedom)* from any reachable state, there exists an outgoing program transition:
  $\phi_2 : [any^*]\langle prog\rangle\mathbf{T}$
– *(reachability)* starting from any state, a legitimate state can be reached by taking only program actions (there always exist a path from any state to a legitimate state):
  $\phi_3 : [any^*]\langle prog^*\rangle\langle steady\rangle\mathbf{T}$
– *(convergence)* starting from any state, a legitimate state is *eventually* reached by taking only program actions (the algorithm never reaches a cycle outside legitimate states):
  $\phi_4 : [any^*]\neg\nu X.\,(\neg\langle steady\rangle\mathbf{T} \wedge \langle prog\rangle X)$

In order to reduce the complexity of verification of distributed reset, we utilize a compositional approach. Specifically, we infer the correctness of the composite distributed reset algorithm by verifying the correctness of the tree layer and wave layer individually. However, such compositional verification needs demonstration of *interference-freedom* between components. Let C_1 and C_2 be two components. We say that C_1 and C_2 do not interfere with each other if whenever C_1 satisfies some property φ and C_2 satisfies some property φ', then their "composition" (e.g., using BIP interactions) satisfies $\varphi \wedge \varphi'$.

Theorem 1. The composition of the tree layer and wave layer in the distributed reset algorithm is interference-free for properties $\phi_1 ... \phi_4$.

The immediate consequence of Theorem 1 is that we can verify the correctness of the layers of distributed reset independently. In order to generate Lts models of manageable size for a reasonably large number of processes in the algorithm we manually applied abstraction, live analysis [8], and we simplified the sequence of occurrence of faults by allowing multiple types of faults occurring at the same

[1] We recall that $q \models \langle a\rangle\varphi$ iff $\exists q \xrightarrow{a} q' : q' \models \varphi$, where q and q' are two states, $\xrightarrow{a}$ is a transition labeled by a, and φ is a formula. Also, $q \models [a]\varphi$ iff $\forall q \xrightarrow{a} q' : q' \models \varphi$. The label *any* denotes any transition label, i.e., $\{steady, prog, fault\}$, $\mathbf{T}$ denotes logical true, and $*$ denotes a sequence. Finally, ν and μ respectively denote the largest and smallest fixpoints in the μ-calculus.

Table 1. Verifying distributed reset using classic model checking

	n	states	transitions	generation time	ϕ_1	ϕ_2	ϕ_3	ϕ_4
tree	4	56	649	< 1	< 1	< 1	< 1	< 1
	6	7022	81390	29	1	1	2	3
	9	2456936	59409357	4000	10	23	19	145
wave	4	996	5840	< 1	< 1	< 1	< 1	< 1
	6	27590	189523	36	2	2	3	5
	9	1539001	7077649	2500	5	7	6	93

time. Table 1 summarizes the results about the size of the models in terms of number of processes in the grid. The LTS generation time as well as the time needed to verify the properties considered are all in seconds. All verification tasks are run on a PC with a 3.2GHz Intel Xeon processor and 4GB RAM.

5 Conclusion

The paper illustrates the application of a methodology consistently integrating high-level modeling with verification of functional properties of a distributed implementation in the BIP framework. BIP allows a natural high-level description of the coordination between atomic components by using structured connectors and multiparty interactions. Consistency is ensured by results guaranteeing preservation of properties of the initial high-level model by its implementation. We demonstrated how one can build-up the self-stabilizing distributed reset algorithm [2] by developing a set of independent atomic components and then wiring them by using connectors by considering functional and recovery tasks independently. We also identified and verified a set of safety and liveness properties that any self-stabilizing algorithm has to satisfy for distributed reset.

Our approach is extremely beneficial for design and implementation of complex concurrency control algorithms. In this context, we are currently working on a generic component-based framework for modeling and analyzing transactional memory algorithms using BIP. We are also working on a wide range of transformations from high-level BIP models into low-level actual implementations such as the Message Passing Interface (MPI), multi-core, and fully distributed platforms. Another interesting research direction is to automate the procedure presented in this paper by transforming algorithms in (shared memory) guarded commands into BIP models.

References

1. Alexander, M., Gardner, W.: Process Algebra for Parallel and Distributed Processing. Chapman & Hall/CRC, Boca Raton (2008)
2. Arora, A., Gouda, M.: Distributed reset. IEEE Transactions on Computers 43, 316–331 (1994)

3. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed semantics and implementation for systems with interaction and priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
4. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Software Engineering and Formal Methods (SEFM), pp. 3–12 (2006)
5. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
6. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: Automated conflict-free distributed implementation of component-based models. In: IEEE Symposium on Industrial Embedded Systems, SIES (to appear 2010)
7. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: ACM International Conference on Embedded Software, EMSOFT (to appear 2010)
8. Bozga, M., Fernandez, J.-C., Ghirvu, L.: State-space reduction based on live variable analysis. Journal of Science of Computer Programming 47(2-3), 203–220 (2003)
9. Chandy, K.M., Misra, J.: Parallel program design: a foundation. Addison-Wesley Longman Publishing Co., Inc., Boston (1988)
10. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
11. Dijkstra, E.W.: A belated proof of self-stabilization. Distributed Computing 1(1), 5–6 (1986)
12. Garavel, H., Lang, F., Mateescu, R., Serve, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
13. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo (1996)
14. Mateescu, R., Sighireanu, M.: Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. Science of Computer Programming 46(3), 255–281 (2003)
15. Bensalem, T.N.S., Bozga, M., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
16. Sifakis, J.: A framework for component-based construction extended abstract. In: Software Engineering and Formal Methods (SEFM), pp. 293–300 (2005)
17. Tauber, J.A., Lynch, N.A., Tsai, M.J.: Compiling IOA without global synchronization. In: Symposium on Network Computing and Applications (NCA), pp. 121–130 (2004)

A Fault-Resistant Asynchronous Clock Function

Ezra N. Hoch, Michael Ben-Or, and Danny Dolev

The Hebrew University of Jerusalem
{ezraho,benor,dolev}@cs.huji.ac.il

Abstract. Consider an asynchronous network in a shared-memory environment consisting of n nodes. Assume that up to f of the nodes might be *Byzantine* ($n > 12f$), where the adversary is full-information and dynamic (sometimes called adaptive). In addition, the non-*Byzantine* nodes may undergo transient failures. Nodes advance in atomic steps, which consist of reading all registers, performing some calculation and writing to all registers.

The three main contributions of the paper are: first, the clock-function problem is defined, which is a generalization of the clock synchronization problem. This generalization encapsulates previous clock synchronization problem definitions while extending them to the current paper's model. Second, a randomized asynchronous self-stabilizing *Byzantine* tolerant clock synchronization algorithm is presented.

In the construction of the clock synchronization algorithm, a building block that ensures different nodes advance at similar rates is developed. This feature is the third contribution of the paper. It is self-stabilizing and *Byzantine* tolerant and can be used as a building block for different algorithms that operate in an asynchronous self-stabilizing *Byzantine* model.

The convergence time of the presented algorithm is exponential. Observe that in the asynchronous setting the best known full-information dynamic *Byzantine* agreement also has an expected exponential convergence time.

1 Introduction

When tackling problems in distributed systems, there are many previously developed building blocks that assist in solving the problem. Some of these building blocks allow one to design a solution under "easy" assumptions, then automatically transform them to a more realistic environment. For example, it is easier to construct an algorithm in the synchronous model, then add an underlying synchronizer (see [4]) to adapt the solution to an asynchronous model. Similarly, developing a self-stabilizing algorithm can be challenging; instead, one can develop a non-self-stabilizing algorithm, and use a stabilizer ([1]) to address transient errors.

Among the different models of distributed systems, specific models received more attention than others; and therefore the availability and versatility of building blocks differ from one model to another. For example, the synchronous no-failures model can automatically be extended in many different directions: asynchronous no-failures, synchronous self-stabilizing, asynchronous self-stabilizing, etc. Zooming-in to the world of self-stabilizing, there are various model-convertors: between shared-memory and message-passing, from an id-based to uniform system, etc. (see [11]).

However, when moving away from the commonly researched models, the availability of such model-converters diminishes. In the current paper we are interested in an asynchronous network with *Byzantine* nodes and transient failures. That is, we aim at solving a problem in a way that is *Byzantine* tolerant, self-stabilizing and operates in an asynchronous network. The *Byzantine* adversary is assumed to be full-information and dynamic (sometimes called adaptive). There are few previous works that operate in similar models [20,21,19]. In these works non-faulty neighbors of *Byzantine* nodes may reach undesired states. However, as far as we know, this is the first work operating in such a setting in which non-*Byzantine* nodes reach their desired state even if they have *Byzantine* neighbors.

The problem we solve in the current work is clock-synchronization. Our solution assumes two simplifying assumptions: a) a "centralized daemon", *i.e.,* each node can run the entire algorithm as an atomic step; b) an "en masse scheduler" that adheres to the following: if p gets scheduled twice, then $n-2f$ other non-faulty nodes get scheduled in between (formally defined in Definition 2). Under these assumptions we define and solve the clock synchronization problem in an asynchronous network while tolerating both *Byzantine* and transient faults. The solution is a randomized algorithm with expected convergence time of $O(3^{n-2f})$.

Both assumptions can be seen as "building blocks that do not yet exist". When constructing a self-stabilizing asynchronous algorithm (without *Byzantine* nodes), it is reasonable to assume a centralized daemon due to the mutual exclusion algorithm of Dijkstra (see [8]). Thus, once an equivalent algorithm can be devised for this paper's model, the first assumption can be removed. In Section 6 we provide an algorithm that implements the second assumption, thus allowing its usage without reducing the generality of a solution that uses it.

Due to our dependence on the first assumption, we consider this work as a step towards a full solution of the clock synchronization in an asynchronous network that is self-stabilizing and *Byzantine* tolerant. We hope it leads to further research of this model, one which will produce an equivalent of Dijkstra's algorithm operating in the current work's model.

Related Work. Being able to introduce consistent "time" in a distributed system is an important task, however difficult it may be in some models. For many distributed tasks the crux of the problem is to synchronize the operations of the different nodes. One method of doing this is using some sort of "time-awareness" at each node, ensuring that different "clocks" advance in a relatively synchronized manner. Therefore, it is interesting to devise such algorithms that are highly robust.

In the past, various models were considered. Ranging from synchronous systems (see [10,12,16]), in which all nodes receive a common signal simultaneously at regular intervals; through bounded-delay systems (see [9,18]), in which only a bound on the message delivery time is given; to completely asynchronous systems (see [7,14]), in which only an eventual (but not bounded) delivery of messages is assumed. Independent of the timing model, different fault tolerance assumptions are considered: the self-stabilizing fault paradigm, in which all nodes follow their protocol but may start with arbitrary values of their variables and program counter (see [11]). Another commonly assumed faults are the Fail-stop faults, in which some of the nodes may crash and cease to participate in the protocol. Lastly, *Byzantine* faults are considered to be the most severe fault model, as they assume that the faulty nodes can behave arbitrarily and even collude in trying to keep the system from reaching its designated goals (see [3,17]).

"Knowing what time it is" acquires different flavors in different models. In systems without any faults, it is usually assumed that each node has a physical clock, and these clocks differ from node to node. The main issue is to synchronize the different clocks as close as possible. In a synchronous, self-stabilizing and *Byzantine* tolerant model, this problem was termed "digital clock synchronization", and consisted of having all nodes agree on some bounded integer and increase it every round (see [2,10,12,16]).

The traditional concept of "clock synchronization" does not hold in an asynchronous environment. Therefore, previous work has defined "phase clocks" or "unison" (see [7,14]), which states that each node has an integer valued clock, and neighboring nodes should be at most ± 1 from each other. It is shown (for example, see [14]) how such "synchronization" is sufficient in solving many problems.

Most previous works in the asynchronous model considered self-stabilizing or *Byzantine* faults, but not both. In the current work, we consider both fault models. However, defining what "telling the time" means in an asynchronous, self-stabilizing and *Byzantine* tolerant manner is a bit tricky. To address that, a new notion of "knowing what time it is" is introduced: *a clock function*.

All previous clock-synchronization (or phase-clock, or unison, etc.) algorithms can be viewed in the following way: each time a non-faulty node is running, it executes some piece of code ("function") that returns a value ("the clock value") and there are constraints on the range of different non-faulty nodes' values. In the synchronous digital clock synchronization problem, the function returns an integer value, and we require that all nodes executing the function at the same round receive exactly the same value and a node executing the function in consecutive rounds receives consecutive values. In an asynchronous network (*i.e.,* in [14]), different nodes may execute their clock-functions at different times and at different rates. The constraint on the returned values can be described as follows: given a configuration of the system, if p would execute its clock-function and receive a value v, then any neighbor of p that would execute its clock-function at the same configuration, would receive a value that differs by at most ± 1 from v.

In the current work it is assumed that the network is fully connected, which means that every node is connected to every other node. Therefore, the constraint of the clock-function is simplified, informally requiring any two non-faulty nodes that execute the clock-function to receive values that are at most one apart.

In synchronous networks the problem of self-stabilizing *Byzantine* tolerant clock synchronization is equivalent to the problem of *Byzantine* agreement, in the sense that any solution to the self-stabilizing *Byzantine* tolerant clock synchronization problem is also a solution to the (non-self-stabilizing) *Byzantine* agreement problem. In asynchronous networks the best known full-information dynamic *Byzantine* agreement has expected exponential convergence time (see [5]). While the synchronous equivalence between clock synchronization and *Byzantine* agreement does not transfer to the asynchronous setting (as it strongly uses the fact that all nodes agree on the exact same clock value), it raises the possibility that improving the result of this paper will require usage of new techniques. That is, it is not known yet if the self-stabilizing clock synchronization of the current work can be used to solve *Byzantine* agreement. However, if it can be used, then improving the exponential convergence of the current work would lead to an improvement of the best known asynchronous *Byzantine* agreement against a dynamic full-information adversary.

Contribution. Our contribution is three-fold. First, we define the clock-function problem, which is a generalization of the clock synchronization problem. This definition provides a meaningful extension of the clock synchronization problem to the asynchronous self-stabilizing *Byzantine* tolerant model.

Second, we provide an algorithm that solves the clock-function problem in the above model. Using shared memory, it has an expected $O(3^{n-2f})$ convergence time, independent of the wraparound value of the clock. Notice that for synchronous networks, the first two contributions were already presented in [12]. Our contribution is with respect to asynchronous networks.

Lastly, in Section 6 we construct a building block that bounds the relative rates at which different non-faulty nodes progress with respect to other non-faulty nodes. More specifically, between any two atomic steps of a non-faulty node p, there are guaranteed to be atomic steps of $n - 2f$ other non-faulty nodes. (See the "en masse scheduler" assumption described in the introduction). We postulate that this building block can be used in other asynchronous self-stabilizing *Byzantine* tolerant settings.

Overview. We start by defining the model (see Section 2). A subset of all possible runs is defined and denoted *"en masse"* (see Definition 2). Section 3 discusses different aspects of defining clock synchronization in an asynchronous, self-stabilizing, *Byzantine* tolerant environment; and defines a clock function, which is a generalization of the clock synchronization problem.

Section 4 introduces Async-Clock, an algorithm that solves the problem at hand. Section 5 contains the correctness proof for Async-Clock. Both Section 4 and Section 5 are correct only for en masse runs, for which Async-Clock requires fault redundancy of $n > 6f$.

In Section 6 the algorithm EnMasse is presented, which transforms any run into an en masse run. Leading to the correctness of Async-Clock for any run.

However, the transformation done by EnMasse increases the fault redundancy of Async-Clock to $n > 12f$. Lastly, Section 7 concludes with a discussion of the results.

2 Distributed Model

The system is composed of a set of n nodes denoted by $\mathcal{P}$. Every pair of nodes $p, q \in \mathcal{P}$ communicates via shared memory (*i.e.,* a fully-connected communication graph), in an asynchronous manner. That is, p and q share two registers: $R_{p,q}, R_{q,p}$.[1] Register $R_{p,q}$ is written by p and read by q.[2] A configuration $\mathcal{C}$ describes the global state of the system and consists of the states of each node and the state of each register. A run of the system is an infinite sequence of configurations $\mathcal{C}_0 \to \mathcal{C}_1 \to \cdots \to \mathcal{C}_r \to \cdots$, such that the configuration $\mathcal{C}_{r+1}$ is reachable from configuration $\mathcal{C}_r$ by a single node's atomic step. In the context of the current paper, an atomic step consists of reading all registers, performing some calculation and then writing to all registers.

The system is assumed to start from an arbitrary initial configuration $\mathcal{C}_0$. We show that eventually - in the presence of continuous *Byzantine* behavior - the system becomes synchronized.

In addition to transient faults, up to f of the nodes may be *Byzantine*. The *Byzantine* adversary has full information, *i.e.,* it can read the values in every node's memory[3] and in the shared registers between any two nodes. There are no private channels and the adversary is computationally unbounded. Moreover, the adversary is dynamic, which means it may choose to "capture" a non-faulty node at any stage of the algorithm. However, once the adversary has "captured" f nodes in some run, it cannot affect other nodes and in a sense becomes static. The results of this paper can be extended to the setting in which the adversary continues to be dynamic throughout the run, as long as the adversary is limited by the rate at which it can release and capture non-faulty nodes. We do not present this extension in the current paper for the sake of clarity. However, one can easily be convinced that it applies, once the main points of the work are explained.

The adversary also has full control of the scheduling of atomic steps and can use its full information knowledge in this scheduling. However, for a clock synchronization algorithm to be meaningful, runs in which some of the non-faulty nodes never get to perform atomic steps should be excluded. Thus, throughout the paper only fair runs are considered:

Definition 1. *A run is* **fair** *if every non-faulty node performs infinitely many atomic steps.*

[1] Pair-wise communication is used to allow *Byzantine* nodes to present different values to different nodes; as opposed to assuming a single register per-node that can be read by all other nodes.

[2] For simpler presentation we assume that p writes and reads $R_{p,p}$.

[3] Actually, the presented algorithm stores all its state in the shared registers.

A subset of all fair runs is defined:

Definition 2. *A run $\mathcal{T}$ is* **en masse with respect** *to node p if for any 2 atomic steps p performs during $\mathcal{T}$ (say at configurations $\mathcal{C}$ and $\mathcal{C}'$, respectively) there are at least $n - 2f$ non-faulty nodes that perform atomic steps between $\mathcal{C}$ and $\mathcal{C}'$.*

A run $\mathcal{T}$ is **en masse** *if it is fair and it is en masse with respect to all non-faulty nodes.*

As stated in the overview, en masse runs are needed for Async-Clock to operate correctly. Assuming all runs are en masse runs, the fault tolerance redundancy required is $n > 6f$. However, in Section 6 we show how to remove the requirement of en masse runs, at the cost of increasing the fault tolerance redundancy to $n > 12f$.

3 Problem Definition

Before formally defining the problem at hand, consider the properties a distributed clock synchronization algorithm should have in an asynchronous setting:

1. *(clock-value)* a means of locally computing the current clock value at any non-faulty node;
2. *(agreement)* if different non-faulty nodes compute the clock value close (in time) to each other, they should obtain similar values;
3. *(liveness)* if non-faulty nodes continuously recompute clock values, then they should obtain increasing values.

For example, in a synchronous network, the clock synchronization problem is usually formulated as: *(clock-value)* each node p has a bounded integer counter $Clock_p$; *(agreement)* for any two non-faulty nodes p, q it holds that $Clock_p = Clock_q$; *(liveness)* if $Clock_p = z$ at round r then $Clock_p = z + 1$ at round $r + 1$. Since the clock is bounded, the previous sentence is slightly modified: "if $Clock_p = z$ at round r, then $Clock_p = z + 1 (\text{mod } k)$ at round $r + 1$"; where k represents the wrap-around value of the clock.

Notation 31 *Denote by $a \oplus_k b$ the value $(a + b \mod k)$.*

In an asynchronous setting, it is impossible to ensure that all nodes update their clocks simultaneously. Thus, the "agreement" property requires a relaxed version as opposed to the synchronous setting's stricter version. In addition, the "liveness" property is somewhat tricky to define, due to the *Byzantine* presence. To illustrate the difficulty, consider a set of f *Byzantine* nodes that "behave as if" they were non-faulty, and they repeatedly recompute the clock value. According to the definition above, the clock value will increase continuously, even though non-faulty nodes did not perform a single step. Therefore, such a clock synchronization algorithm is useless, as the *Byzantine* nodes can make it reach any clock value; in other words, the *Byzantine* nodes "control" the clock value.

It is not immediately clear how these "benign" *Byzantine* nodes can be differentiated from the non-faulty nodes. The following definitions address such difficulties, and present a formalization of the clock synchronization problem in this paper's model.

Definition 3. *A value v' is at most d* **ahead of** *v if there exists j, $0 \le j \le d$, such that $v \oplus_k j = v'$. Denote "v' is at most d ahead of v" by $v \preceq_d v'$.*

Definition 4 addresses the "clock-value" property:

Definition 4. *A* **clock-function** *$\mathcal{F}$ is an algorithm that when executed during an atomic step returns a value in the range $\{0, ..., k-1\}$. Denote by $\mathcal{F}_p(\mathcal{C})$ the value returned when p executes $\mathcal{F}$ during an atomic step at configuration $\mathcal{C}$.*

Consider the "agreement" property: it requires that different non-faulty nodes that compute the clock value simultaneously, receive similar values. What does "simultaneously" mean in an asynchronous setting? It can be captured by requirements on the clock values computed in different runs. In addition, the interference caused by *Byzantine* nodes in different runs needs to be captured.

Informally, "agreement" requires the following from $\mathcal{F}$: given a configuration $\mathcal{C}$, no matter what the adversary does, if different non-faulty nodes execute $\mathcal{F}$ they receive values that are close to each other. Definition 5, Definition 6 and Definition 7 formally state the "agreement" requirement. First, "no matter what the adversary does" is formally defined:

Definition 5. *An* **adversarial move** *from a configuration $\mathcal{C}$ is any configuration reachable by an arbitrary sequence of atomic steps of faulty nodes only.*

Second, "different non-faulty nodes execute $\mathcal{F}$" is divided into two cases. Let p, q be non-faulty nodes. The first case considers the computed value of p (when calculating $\mathcal{F}$ on $\mathcal{C}$) as opposed to the computed value of q (see Definition 6). The second case considers the computed value of q after p has computed its value (see Definition 7). Both cases require the computed values of p and q to be close to each other.

Definition 6. *A configuration $\mathcal{C}$ is ℓ-***well-defined** *(with respect to some clock-function $\mathcal{F}$) if there is a value v s.t. for any non-faulty node p and every adversarial move $\mathcal{C}'$ from $\mathcal{C}$ it holds that $v \preceq_\ell \mathcal{F}_p(\mathcal{C}')$. v is called "a defined value" at $\mathcal{C}$. (There may be more than one such v).*

Informally, Definition 6 says that $\mathcal{C}$ is ℓ-well-defined if there is an intrinsic value v such that any adversarial move cannot increase the clock-value by more than ℓ. Thus, any two non-faulty nodes p, q (in different run extensions from $\mathcal{C}$) that execute $\mathcal{F}$ on $\mathcal{C}$ (no matter what the adversary has done) will receive values in the range $\{v, \ldots, v+\ell\}$; *i.e.,* p and q's values are at most ℓ apart.

Suppose $\mathcal{C}_0$ is ℓ-well-defined with value v, and that a non-faulty node p performs an atomic step at $\mathcal{C}_0$ resulting in $\mathcal{C}_1$ and then a non-faulty node q performs an atomic step at $\mathcal{C}_1$. Definition 6 does not imply any constraint on the value of $\mathcal{F}_p(\mathcal{C}_0)$ with respect to $\mathcal{F}_q(\mathcal{C}_1)$, therefore the following definition is required:

Definition 7. *A run is ℓ-**well-defined** (w.r.t. a clock-function $\mathcal{F}$) if: a) every configuration $\mathcal{C}$ in the run is ℓ-well-defined; b) for two consecutive configurations $\mathcal{C}, \mathcal{C}'$, if v is a defined value of $\mathcal{C}$ and v' is a defined value of $\mathcal{C}'$ then $v \preceq_\ell v'$.*

Definition 7 states that the values of a clock-function $\mathcal{F}$ on consecutive configurations cannot be arbitrary. That is, they must be at most ℓ apart from the previous configuration. However, there is no requirement that they actually increase; *i.e.,* "liveness" is not captured by the previous definitions.

Definition 8. *A run is ℓ-**clock-synchronized** (w.r.t. some clock-function $\mathcal{F}$), if it is ℓ-well-defined (w.r.t. $\mathcal{F}$) and the defined values of consecutive configurations change infinitely many times. (*I.e., *for infinitely many consecutive configurations $\mathcal{C}, \mathcal{C}'$ the defined values of $\mathcal{C}$ differ from the defined values of $\mathcal{C}'$).*

Notice that Definition 7 already requires that defined values of consecutive configurations are non-decreasing (assuming that ℓ is sufficiently small with respect to k). Thus, combined with Definition 8, it implies that in an ℓ-clock-synchronized run, infinitely many configurations are configurations with increasing defined values (informally, "increasing" means that one defined value is achieved by adding less than $\frac{k}{2}$ to a previous defined value).

Remark 1. Definition 7 and Definition 8 impose requirements on the *defined* values of consecutive configurations. However, a specific node p might compute a clock value that is decreasing between consecutive configuration. *i.e.,* p's clock might "go backward". For example, let $\mathcal{C}, \mathcal{C}'$ be two consecutive configurations, and let the defined value of both configurations be v. It is possible that p will compute the clock value to be $v + 1$ for $\mathcal{C}$, while computing the clock value to be v for $\mathcal{C}'$.
However, this possibility is immanent to an asynchronous *Byzantine* tolerant clock synchronization that has a wraparound value k. Consider a setting in which all nodes but one are advanced in a synchronous manner, while a single node p performs atomic steps only once every $k - 1$ rounds. In such a setting, p should update its clock value to be slightly below its previous value (alternatively, it can be seen as increasing the value by $k - 1$).

Definition 9. *An algorithm $\mathcal{A}$ solves the ℓ-**clock-synchronization problem** if there is a clock-function $\mathcal{F}$ s.t. any fair run starting from any arbitrary configuration has a suffix that is ℓ-clock-synchronized with respect to $\mathcal{F}$.*

An ideal protocol would solve the 0-clock-synchronization problem. However, due to the asynchronous nature of the discussed model, the best that can be expected is to solve the 1-clock-synchronization problem. We aim at solving the ℓ-clock-synchronization problem for as many values of $\ell \geq 1$ as possible. Clearly, if $\mathcal{A}$ solves the ℓ_1-clock-synchronization problem, then $\mathcal{A}$ also solves the ℓ_2-clock-synchronization problem for any $\frac{k}{2} > \ell_2 \geq \ell_1$.

Therefore, the rest the paper concentrates on solving the 5-clock-synchronization problem; thus, solving the ℓ-clock-synchronization problem for all $\frac{k}{2} > \ell \geq 5$. In Section 7.1 we show how to use any $\frac{k-1}{2}$-clock-synchronization problem to solve the

```
Algorithm ASYNC-CLOCK                                  /* executed on node q */
01: do forever:

        /* read all registers */
02:     for $i := 1$ to $n$
03:         set $val_i :=$ read $R_{p_i,q} \mod k$;

        /* some internal definitions */
04:     let $\#v$ denote the number of times $v$ appears in $\{val_i\}_{i=1}^{n}$;
05:     let $count(v,l)$ denote $\sum_{j=0}^{l} \#(v \oplus_k j)$;
06:     let $pass(l,a)$ denote $\{v | count(v,l) \geq a\}$;

        /* update my_val */
07:     if $pass(0, n-f) \neq \emptyset$ then
08:         set $my\_val_q := 1 \oplus_k \max\{pass(0, n-f)\}$;
09:     else if $pass(1, n-f) \neq \emptyset$ then
10:         set $my\_val_q := 1 \oplus_k \max\{pass(1, n-f)\}$;
11:     else if $pass(1, n-2f) \neq \emptyset$ then
12:         let $low \notin pass(1, n-2f)$ be such that $low \oplus_k 1 \in pass(1, n-2f)$;
13:         let $relative\_median = min\{l | l \geq 0 \,\&\, count(low, l) > \frac{n}{2}\}$;
14:         set $my\_val_q := low \oplus_k relative\_median$;
15:     else set $my\_val_q :=$ randomly select a value from $pass(1, n-3f) \bigcup \{0\}$;

        /* write my_val to registers */
16:     for $i := 1$ to $n$
17:         write $my\_val_q$ into $R_{q,p_i}$;
18: od;
```

Fig. 1. A self-stabilizing *Byzantine* tolerant algorithm solving the 5-Clock-Synchronization problem

1-clock-synchronization problem, thus solving the ℓ-clock-synchronization problem for all $\frac{k}{2} > \ell \geq 1$.

4 Solving the 5-Clock-Synchronization Problem

An atomic step consists of reading all registers, performing some calculations and writing to all registers. Thus, an atomic step consists of executing once an entire "loop" of ASYNC-CLOCK (see Figure 1).

Each non-faulty node p has a bounded integer variable, my_val_p, which represents the current clock value of p. When p performs an atomic step, it reads all of its registers, thus getting an impression of the clock values of the other nodes. It then computes its own new clock value (which is saved in my_val_p) and writes my_val_p to all registers.

ASYNC-CLOCK operates in a similar fashion to many other *Byzantine* tolerant algorithms. It first gathers information regarding the clock value of the other nodes in the system. Then it uses various thresholds to decide on the clock value for the next step. If no threshold works (*i.e.,* no clear majority is found), it chooses a random value from a small set of options.

To ensure all values read during Line `02-03` are in the range $[0, \ldots, k-1]$, the algorithm applies "$\bmod k$" to the values read. This is a standard way of dealing with uninitialized values.

The crux of Async-Clock is in the exact thresholds and their application (Lines `07-15`). In these lines, node p considers different possibilities. Either it sees a decisive majority towards some clock value (Line `07` and Line `09`) in which case p updates its local clock value to coincide with the majority clock value it has seen. Alternatively, if no clear majority exists (Line `15`), p randomly selects a new clock value. The interesting case is when p sees a "partial" majority (Line `11`), in which case p takes the relative median of the clock values it has seen. We call this a "relative median" since the clock values are "$\bmod k$" and thus the median in not well defined.

The full Async-Clock algorithm appears in Figure 1. Async-Clock solves the ℓ-Clock-Synchronization problem for $\ell = 5$; combined with the discussion at the end of Section 3, it shows how to solve the ℓ-Clock-Synchronization problem for any $\frac{k}{2} > \ell \geq 5$.

5 Correctness Proof

In the following discussion we consider the system only after all transient faults ended and each non-faulty node has taken at least one atomic step. We consider only runs of the system that begin after that initial sequence of atomic steps.

Informally, a round is a portion of a run such that each node that is non-faulty throughout the round performs an atomic step at least once. The first round (of a run $\mathcal{T}$) is the minimal prefix $\mathcal{R}$ of the run $\mathcal{T}$ such that each node that is non-faulty throughout $\mathcal{R}$ performs an atomic step at least once. Consider the suffix $\mathcal{T}'$ of $\mathcal{T}$ after the first round was removed. The second round of $\mathcal{T}$ is the first round of $\mathcal{T}'$; the definition continues so recursively.

Consider any fair run of the system $\mathcal{C}_0 \to \mathcal{C}_1 \to \cdots \to \mathcal{C}_r \to \cdots$, and consider the transition from configuration $\mathcal{C}_r$ to configuration $\mathcal{C}_{r+1}$, due to some (possibly faulty) node p's atomic step. Since we consider only runs after each non-faulty node q has taken at least one atomic step past the end of the transient faults events, the value of my_val_q reflects the latest value written to all of q's write-registers. This property is true for all configurations that we consider. Thus, regarding a non-faulty p that performs an atomic step, for all non-faulty q it holds that $R_{q,p} = my\_val_q$.

Due to space limitations, only an overview of the proof is given. The full proof can be found at [15].

The proof outline is as follows. First, define a tight configuration:

Definition 10. *$H(\mathcal{C}_r, v, d)$ is the set containing every non-faulty node q, such that* my_val$_q$ *(in $\mathcal{C}_r$) is at most d ahead of v.*

A configuration $\mathcal{C}_r$ is **tight** *around value v if $|H(\mathcal{C}_r, v, 1)| \geq n - 2f$; a configuration is tight if it is tight around some value.*

Second, we show that if a configuration $\mathcal{C}_r$ is tight then so is $\mathcal{C}_{r+1}$. Third, if $\mathcal{C}_r$ is not tight, then we show that with probability $\frac{1}{3^{n-2f}}$ some configuration within 2 rounds from $\mathcal{C}_r$ will be tight. Concluding that after an expected $O(3^{n-2f})$ rounds the system reaches a tight configuration; and all following configurations are tight as well. At this stage, we need to show that the value v that a configuration is tight around continuously increases.

To do so, we show that given that all configurations are tight, different non-faulty nodes that perform atomic steps can have values from a set containing (at most) 3 consecutive values. Moreover, for consecutive configurations, the minimal value among these 3 values can increase by at most 3. Lastly, by closely analyzing the behavior of Async-Clock, we conclude that within 4 rounds the minimal value above increases. That is, the clock function value changes, and changes again within at most 4 rounds, *i.e.,* the clock value changes infinitely many times.

The reason behind the increase of the aforementioned minimal value lies in the following claim: one of two things can happen, either the minimal value increases, or all the non-faulty nodes' clock values become at most 1 apart. In the second scenario, after one round, the minimal value will increase. Concluding that the clock value changes infinitely many times, as required.

Remark 2. The en masse property is used in the proof that if $\mathcal{C}_r$ is not tight, then with probability $\frac{1}{3^{n-2f}}$ a configuration within 2 rounds from $\mathcal{C}_r$ will be tight. Since in an en masse run some set of $n - 2f$ different non-faulty nodes are required to take atomic steps in a consecutive manner. Together with a claim stating that each such step has probability of $\frac{1}{3}$ to flip a coin "in the right direction", we get that with probability $\frac{1}{3^{n-2f}}$ a tight configuration is reached.

Following is the main result of the paper, which is shown to be true assuming that the runs are en masse. In the following section en masse runs are constructed from fair runs. Thus, the theorem can be updated to only require that the run is fair.

Theorem 1. Async-Clock *solves the* 5*-clock-synchronization problem within expected $O(3^{n-2f})$ rounds, for any en masse run and wrap-around value greater than 6 (*i.e., $k > 6$*).*

Remark 3. The requirement that $k > 6$ stems from the analysis of the relative median and the different updates performed in Async-Clock. Due to lack of space, we do not go into details. Full details are available in [15].

6 Ensuring En Masse Runs

Our goal is to ensure that if a non-faulty node p performs a step, at least $n - 2f$ non-faulty nodes have performed a step since p's last step. That is, given an algorithm $\mathcal{A}$ we want to ensure that if some non-faulty node performs two steps of $\mathcal{A}$ then there are at least $n-2f$ different non-faulty nodes that also perform steps of $\mathcal{A}$. To ensure this, we present an algorithm EnMasse that ensures that a specific action, denoted "act", is executed twice by the same non-faulty node p only if there are at least $n-4f$ other non-faulty nodes that have also executed "act". By setting "act" to execute an atomic step of $\mathcal{A}$, we achieve the required goal. *I.e.,* Async-Clock will be executed entirely every time "act" appears in EnMasse.

As the algorithm we present ensures only $n-4f$ nodes execute "act" in between two "acts" of every non-faulty node, we must reduce the *Byzantine* tolerance by half ($n > 12f$) to use EnMasse as a subcomponent of Async-Clock. That is, Async-Clock requires a threshold of $\frac{2}{3}n$ non-faulty nodes ($n-2f$ threshold for $n > 6f$); EnMasse ensures a threshold of $n-4f$. Therefore, by reducing the fault tolerance to $n > 12f$ we ensure that $n-4f > \frac{2}{3}n$, as required by Async-Clock.

Our solution borrows many ideas from [13]. Due to our model's atomicity assumptions, each node can read all registers and write to all registers in a single atomic step. Thus, the problems that [13] encounters do not exist in the current paper at all. However, in the current model there are additional faults (*Byzantine* and self-stabilizing) which do not exists in [13]. Interestingly, the same ideas used in [13] can be adapted to the self-stabilizing *Byzantine* tolerant setting.

For each node p, there is a set of labels $Labels_p$ associated with p. In addition, each node p has a variable $label_p$ from the set $Labels_p$; Also, p has an ordering vector $order_p$, of length $|Labels_p|$, which induces an order on the labels in $Labels_p$. Lastly, each node p has a time-stamp $time_p$, which is a vector of n entries, consisting of a single label $time_p[q] \in Labels_q$ for each node q.

Definition 11. *A label b is of* **type** *p if $b \in Label_p$.*

Definition 12. *Two labels b, c of type p are compared according to $order_p$, where $b <_p c$ if b appears before c in the vector $order_p$. The inequalities $\leq_p$, $>_p, \geq_p, =_p$ are similarly defined.*

Definition 13. *Given two time-stamps $time_p, time_q$, and a set of nodes I, we say that $time_p >_I time_q$ if $p, q \in I$ and for every entry $i \in I$, $time_p[i] \geq_i time_q[i]$, $time_p[q] =_q time_q[q]$ and $time_p[p] >_p time_q[p]$.*

To simplify notations, when it is clear from the context, we write $p >_I q$ instead of $time_p >_I time_q$. That is, when comparing nodes (according to $>_I$), we actually compare the nodes' time stamps.

Definition 14. *A set I of nodes is* **comparable** *if for any $p, q \in I$ either $p >_I q$ or $q >_I p$.*

Lemma 1. *If I is a comparable set, and $p, q, w \in I$, and $p >_I q$, $q >_I w$ then $p >_I w$.*

```
Algorithm EnMasse                                          /* executed on node q */
01: do forever:

        /* read all registers and initialize structures */
02:     for each node $p$, read $time_p$ and $order_p$;
03:     set $\mathcal{I} := \emptyset$;
04:     for each set $W \subseteq \mathcal{P}$ s.t. $|W| \geq n - f$ and $q \in W$:
05:         construct $I := \{time_p \mid p \in W\}$;
06:         if $W$ is comparable then $\mathcal{I} := \mathcal{I} \cup \{I\}$;

        /* decide whether to execute "update" and whether to execute "act" */
07:     if for some $I \in \mathcal{I}$, it holds that $I_{\#}(q) \geq n - 3f$ then
08:         update $time_q, order_q$ and "act";
09:     if $\mathcal{I} = \emptyset$ then update $time_q, order_q$;
10:     write $time_q$ and $order_q$;
11: od;
```

Updating $time_q$ is done by setting $time_q[p] = label_p$, for every $p \in \mathcal{P}$.
Updating $order_q$ consists of changing the order induced by $order_q$ such that $label_q$ is first and for other labels the order is preserved.

Fig. 2. A self-stabilizing *Byzantine* tolerant algorithm ensuring en masse runs

Notice that a comparable set I induces a total order among the elements in I, therefore we can refer to the index of an element in I.

Definition 15. *A node $p \in I$ is said to be the kth highest (in I) if $|\{q \in I | q >_I p\}| = k - 1$. Let $I_{\#}(p) = k$ if $p \in I$ is the kth highest in I.*

The 1st highest in I is the node that is larger than all other nodes. The 2nd highest node in I is the node that has only one node larger than it; (and so on).

6.1 Algorithm EnMasse

This section proves general properties of comparable sets. It discusses "static" sets, that do not change over time. The following algorithm considers comparable sets that change from step to step. However, during each atomic step, the comparable sets that are considered do not change, and the claims from the previous section hold. That is, when reasoning about the progress of the algorithm, the comparable sets that are considered are all "static".

In the following algorithm, instead of storing both $label_p$ and $time_p$, each node stores just $time_p$ and the value of $label_p$ is the entry $time_p[p]$. In addition, during each atomic step, the entire algorithm is executed, *i.e.,* a node reads all time stamps and all order vectors of other nodes, and can update its own time stamp during an atomic step.

When a node q performs an update, it changes the value of $time_q$ and $order_q$ in the following way: a) $order_q$ is updated such that $time_q[q]$ is larger than any other label in $Labels_q$. b) $time_q[p]$ is set to be $time_p[p]$, for all p. Notice that the new $order_q$ does not affect the relative order of labels in $Labels_q$ that are not $time_q[q]$. That is, if $l_1, l_2 \neq time_q[q]$ and $l_1 \leq_q l_2$ before the change of $order_q$, it holds that $l_1 \leq_q l_2$ also after updating the $order_q$.

Intuitively, the idea of EnMasse is to increase the time stamp of a node q only if q sees that most of the other nodes are ahead of q. When the time stamp is increased, q also performs "act". This leads to the following dynamics: a) If q has performed an "act" twice, *i.e.,* updated its time stamp twice, then after the first update, q is ahead of all other nodes. b) However, since q is ahead of all non-faulty nodes, if q updates its time stamp again it must mean that many nodes have updated their time stamps after q's first update. *i.e.,* between two "act" of q many other nodes have performed "act" as well.

In a similar manner to Section 5, the lemmas and proofs are available in the full version [15].

We continue with an overview of the proof. First, consider the set of non-faulty nodes, and consider the set of time stamps of these nodes. The proof shows that if this set is comparable for some configuration $\mathcal{C}_r$ then it is comparable for any configuration $\mathcal{C}_{r'}$ where $r' > r$. Second, we consider an arbitrary starting state, and consider the set Y_r containing non-faulty nodes that have updated their time stamp by the end of round r. It is shown that if $|Y_r| \geq n-2f$ then $|Y_{r+1}| \geq n-f$. Moreover, if $|Y_r| < n-2f$ then $|Y_{r+1}| \geq |Y_r| + 1$. Thus, we conclude that within $O(n)$ rounds all non-faulty nodes have performed an update.

Once all non-faulty nodes have performed an update since the starting state, it holds that the set of all non-faulty nodes' time stamps is comparable. Thus, during every round at least $2f$ nodes perform an update (as they see themselves in the lower $3f$ part of the comparable set). This ensures that within $\frac{n}{2f}$ rounds some node will perform "act" twice. That is, there is no deadlock in the EnMasse algorithm. To conclude the proof, it is shown that when the set of all non-faulty nodes values is comparable and some non-faulty node performs "act" twice, it must be that another $n-4f$ non-faulty nodes have performed "act" in between.

Theorem 2. *Starting from round $n-2f+3$, between any non-faulty node's two consecutive "act"s, there are $n-4f$ non-faulty nodes that perform "act". Moreover, every non-faulty node performs an "act" at least once every $\frac{n}{2f}$ rounds.*

Theorem 2 states that using EnMasse one can ensure that nodes executing Async-Clock will have the following properties: 1) every non-faulty node p executes an atomic step of Async-Clock once every $\frac{n}{2f}$ rounds; 2) if non-faulty p executes 2 atomic steps of Async-Clock, then at least $n-4f$ non-faulty nodes execute atomic steps of Async-Clock in between. By setting $n > 12f$, these properties ensure that a fair run $\mathcal{T}$ is an en-masse run $\mathcal{T}'$ with respect to Async-Clock, s.t. each round of $\mathcal{T}'$ consists of at most $\frac{n}{2f}$ rounds of $\mathcal{T}$.

7 Discussion

7.1 Solving the 1-Clock-Synchronization Problem

First, the 5-clock-synchronization problem was solved using Async-Clock while assuming en masse runs. Second, the assumption of en masse runs was removed in Section 6. In this subsection we complete the paper's result by showing how to transform a 5-clock-synchronization algorithm to a 1-clock-synchronization algorithm.

Given any algorithm $\mathcal{A}$ that solves the ℓ-clock-synchronization problem, one can construct an algorithm $\mathcal{A}'$ that solves the 1-clock-synchronization problem. Denote by $k_{\mathcal{A}'}$ the desired wraparound value of $\mathcal{A}'$, and let $k_{\mathcal{A}} = k_{\mathcal{A}'} \cdot \ell$ be the wraparound value for $\mathcal{A}$.

The construction is simple: each time $\mathcal{A}'$ is executed, it runs $\mathcal{A}$ and returns the clock value of $\mathcal{A}$ divided by ℓ (that is, $\lfloor \frac{\mathcal{F}_{\mathcal{A}}}{\ell} \rfloor$). The intuition behind this construction is straightforward: $\mathcal{A}$ solves the ℓ-clock-synchronization problem, thus, the values it returns are at most ℓ apart. Therefore, the values that $\mathcal{A}'$ returns are at most 1 apart from each other.

7.2 Future Work

The current paper has a few drawbacks, each of which is interesting to resolve.

First, is it possible to reduce the atomicity requirements; that is, can an atomic step be defined as a single read or a single write (and not as "read all registers and write all registers")?

Second, can the current algorithm be transported into a message passing model?

Third, can different coin-flipping algorithms that operate in the asynchronous setting (*i.e.,* [6]) be used to reduce the exponential convergence time to something more reasonable? Perhaps even expected constant time?

Fourth, can the ratio between *Byzantine* and non-*Byzantine* nodes be reduced? *I.e.,* can $n > 3f$ be achieved?

Fifth, can the problem of asynchronous *Byzantine* agreement be reduced to the problem of clock synchronization presented in the current work? (This will show that the expected exponential convergence time is as good as is currently known).

Lastly, the building block EnMasse is interesting by itself. It would be interesting to find a polynomial solution to EnMasse.

Acknowledgements

Michael Ben-Or is the incumbent of the Jean and Helena Alfassa Chair in Computer Science, and he was supported in part by the Israeli Science Foundation (ISF) research grant. Danny Dolev is incumbent of the Berthold Badler Chair in Computer Science. Danny Dolev was supported in part by the Israeli Science Foundation (ISF) Grant number 0397373.

References

1. Afek, Y., Dolev, S.: Local stabilizer. In: Proc. of the 5th Israeli Symposium on Theory of Computing Systems (ISTCS 1997), Bar-Ilan, Israel (June 1997)
2. Arora, A., Dolev, S., Gouda, M.G.: Maintaining digital clocks in step. Parallel Processing Letters 1, 11–18 (1991)
3. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons, Chichester (2004)
4. Awerbuch, B.: Complexity of network synchronization. J. ACM 32(4), 804–823 (1985)
5. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: PODC 1983, New York, NY, USA, pp. 27–30 (1983)
6. Canetti, R., Rabin, T.: Fast asynchronous byzantine agreement with optimal resilience. In: STOC 1993, pp. 42–51. ACM, New York (1993)
7. Couvreur, J.M., Francez, N., Gouda, M.: Asynchronous unison. In: Proceedings of the 12th International Conference on Distributed Computing Systems, pp. 486–493 (1992)
8. Dijkstra, W.: Self-stabilization in spite of distributed control. Commun. of the ACM 17, 643–644 (1974)
9. Dolev, D., Hoch, E.N.: Byzantine self-stabilizing pulse in a bounded-delay model. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 234–252. Springer, Heidelberg (2007)
10. Dolev, D., Hoch, E.N.: On self-stabilizing synchronous actions despite byzantine attacks. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 193–207. Springer, Heidelberg (2007)
11. Dolev, S.: Self-Stabilization. The MIT Press, Cambridge (2000)
12. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. Journal of the ACM 51(5), 780–799 (2004)
13. Dwork, C., Waarts, O.: Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible. In: STOC 1992 (1992)
14. Herman, T.: Phase clocks for transient fault repair. IEEE Transactions on Parallel and Distributed Systems 11(10), 1048–1057 (2000)
15. Hoch, E.N., Ben-Or, M., Dolev, D.: A fault-resistant asynchronous clock function. CoRR, abs/1007.1709 (2010)
16. Hoch, E.N., Dolev, D., Daliot, A.: Self-stabilizing byzantine digital clock synchronization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 350–362. Springer, Heidelberg (2006)
17. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
18. Malekpour, M.R.: A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 411–427. Springer, Heidelberg (2006)
19. Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 440–453. Springer, Heidelberg (2006)
20. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: SRDS 2002, Washington, DC, USA, p. 22. IEEE Computer Society, Los Alamitos (2002)
21. Sakurai, Y., Ooshita, F., Masuzawa, T.: A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 283–298. Springer, Heidelberg (2005)

Self-stabilizing Leader Election in Dynamic Networks

Ajoy K. Datta, Lawrence L. Larmore, and Hema Piniganti

School of Computer Science, University of Nevada Las Vegas

Abstract. Three silent self-stabilizing asynchronous distributed algorithms are given for the leader election problem in a dynamic network with unique IDs, using the composite model of computation. A leader is elected for each connected component of the network. A BFS tree is also constructed in each component, rooted at the leader. This election takes $O(Diam)$ rounds, where $Diam$ is the maximum diameter of any component. Links and processes can be added or deleted, and data can be corrupted. After each such topological change or data corruption, the leader and BFS tree are recomputed if necessary. All three algorithms work under the unfair daemon.

The three algorithms differ in their *leadership stability*. The first algorithm, which is the fastest in the worst case, chooses an arbitrary process as the leader. The second algorithm chooses the process of highest priority in each component, where priority can be defined in a variety of ways. The third algorithm has the strictest leadership stability. If the configuration is legitimate, and then any number of topological faults occur at the same time but no variables are corrupted, the third algorithm will converge to a new legitimate state in such a manner that no process changes its choice of leader more than once, and each component will elect a process which was a leader before the fault, provided there is at least one former leader in that component.

Keywords: dynamic network, leader election, self-stabilization, silent algorithm, unfair daemon.

1 Introduction

The *leader election* problem is one of the fundamental problems in distributed computing. In static networks, this problem is to select a process among all the processes in the network to be the *leader*. In this paper, we deal with leader election in *dynamic networks*, where a fault could occur, *i.e.,* data could be corrupted, or the topology could change, by insertion or deletion of processes or links, possibly even causing the network to become disconnected, or causing previously distinct components of the network to become connected. In a dynamic network, the problem is modified slightly in the following manner: The goal is elect a leader for each component of the network after any number of concurrent faults.

An algorithm $\mathcal{A}$ is *self-stabilizing* if, starting from a completely arbitrary configuration, the network will eventually reach a legitimate configuration. Note that any self-stabilizing leader election algorithm for the static network is also a solution for the dynamic leader election problem, since when a fault occurs, we can consider that the algorithm is starting over again from an arbitrary state. There are a number of such algorithms in the literature which require large memory in each process, or which take $O(n)$ time to converge, where n is size of the network. Given the need to conserve time and possibly space, these algorithms may not be practical for the dynamic leader election problem.

Related Work. There are several leader election algorithms for dynamic networks. However, the only self-stabilizing solutions we are aware of are presented in [5,9]. The algorithm of [9] is simpler (both the pseudo-code and proof of correctness) and more efficient in terms of messages and message size than the solution in [5]. However, both solutions suffer from the same drawback, which is the use of a global clock or the assumption of perfectly synchronized clocks. The following is quoted from [9]: "The algorithm relies on the nodes having perfectly synchronized clocks; an interesting open question is to quantify the effect on the algorithm of approximately synchronized clocks." One goal of this paper is to solve the above open problem.

Furthermore, in the execution of the algorithm of [9], a process could change its choice of leader many times. Our third algorithm, DLEND, has the property that if a topological change, however great, occurs, but if no variables are corrupted, no process changes its choice of leader more than once.

There are number of stabilizing leader election algorithms for static networks in the literature. Arora and Gouda [2] present a silent leader election algorithm in the shared memory model. Their algorithm requires $O(N)$ rounds and $O(\log N)$ space, where N is a given upper bound on n, the size of the network. Dolev and Herman [8] give a non-silent leader election algorithm in the shared memory model. This algorithm takes $O(Diam)$ rounds and uses $O(N \log N)$ space. Awerbuch *et al.*[3] solve the leader election problem in the message passing model. Their algorithm takes $O(Diam)$ rounds and uses $O(\log D \log N)$ space, where D is a given upper bound on the diameter. Afek and Bremler [1] also give an algorithm for the leader election problem in the message passing model. Their algorithm takes $O(n)$ rounds and uses $O(\log n)$ bits per process, where n is the size of the network. They do not claim that their algorithm works under the unfair daemon. In [4], we gave a uniform self-stabilizing leader election algorithm. This algorithm works under an arbitrary, *i.e.,* unfair scheduler (daemon). The algorithm has an optimal space complexity of $O(\log n)$ bits per process. From an arbitrary initial configuration, the algorithm elects the leader and builds a BFS tree rooted at the leader within $O(n)$ rounds, and is silent within $O(Diam)$ additional rounds, where $Diam$ is the diameter of the network. The algorithm does not require knowledge of any upper bound on either n or $Diam$.

Our Contributions. Our algorithms have the following combination of features: they are asynchronous, self-stabilizing, and silent, converge in $O(Diam)$ time, and use no global clock. They also use only $O(1)$ variables per process; however, using a technique from [9], one of the variables is an unbounded integer, meaning that if the algorithm runs forever, that integer grows without bound. (The algorithm of [9] also contains an unbounded integer.) We claim that, as a practical matter, this is of little importance, since the size of that unbounded integer increases by at most one per step, and thus should not overflow a modest size memory, even if the algorithm runs for years.

All three of our algorithms elect a leader for each component of the network, and also build a BFS tree rooted at that leader. In each case, after a fault, each component of the network elects a leader within $O(Diam)$ rounds, where $Diam$ is the maximum diameter of any component, provided the variables are not corrupted.

The space and time complexities of our first algorithm, DLE, are smaller than those of [9], as DLE requires fewer variables, and converges in only $Diam + 1$ rounds from an arbitrary configuration, approximately one third the time as the algorithm of [9]. As in [9], DLE picks an arbitrary process to be the leader of each component.

Our second algorithm, DLEP, picks the highest priority process of each component to be its leader. Priority of a process can be defined as a function of its ID, its local topology, or any data the process obtains from the application layer. Since the choice of leader is not arbitrary, DLEP should have greater leadership stability than DLE, *i.e.,* there should be a tendency for leaders to remain the same after small topological changes.

Our third algorithm, DLEND, ensures even greater leadership stability than DLEP. If the network has reached a stable configuration, and if a topological change, however great, occurs in a given step, and no variables are corrupted, and then no fault occurs thereafter, every component (under the new topology) will elect an *incumbent*, *i.e.,* a leader that was a leader before the topological change, if possible. In cases where a component contains no incumbent, or more than one incumbent, DLEND makes a choice based on priority in the same manner as DLEP.

DLEND has an additional stability feature, which we call *no dithering.* If the network has reached a stable configuration, and if a topological change, however great, occurs in a given step, and then no fault thereafter, then no process will change its choice of leader more than once.

Model. A *self-stabilizing* [6,7] system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the system. In particular, a self-stabilizing distributed algorithm will eventually reach a *legitimate state* within finite time, regardless of its initial configuration, and will remain in a legitimate state forever. A distributed algorithm is called *silent* if eventually all execution halts.

We use the composite atomicity model of computation. Each process can read the variables of its neighbors, but can write only to its own variables. A process is *enabled* if it can execute an action. At each step, the *scheduler*, or *daemon*, selects a non-empty set of enabled processes, if there are any, and each of the selected processes executes an action, which consists of changing its own variables. We assume that all selected processes perform their actions instantly.

The daemon is *unfair*, *i.e.,* it is not required to ever select an enabled process unless it is the only enabled process [7].

2 Algorithm DLE

Our first algorithm, DLE, is somewhat similar to the algorithm of [9], although it is asynchronous and works under the unfair daemon. The basic idea is that every process that detects that it cannot possibly be part of what will become a correct BFS tree declares itself to be a leader. When several processes in a component declare themselves to be leaders, one of them will capture the component.

Every process x has a *leadership pair*, $(x.nlp, x.leader)$, indicating that x has chosen the process whose ID is $x.leader$ as its leader. The number, $x.nlp$, is called a *negative leader priority.*

When a process ℓ declares itself to be a leader, it chooses a priority number that is higher than the priority number of its previous leader; but it stores the negative of this priority in the leadership pair. As in [9], we express this priority as a negative number because we want the lexically smallest leadership pair to have priority.

In a legitimate (final) configuration, all processes in any one component C of the network have the same leadership pair, (nlp, ℓ_C), where ℓ_C is some process in the component, the *leader* of C. In addition, there is a BFS tree of the component rooted at ℓ_C. Each process x has a pointer to its parent in the BFS tree, as well as a *level* variable, whose value is the distance from x to ℓ_C.

The basic technique of the algorithm is *flooding*. Under certain conditions, a process declares itself to be a leader by executing Action A1 (as listed in Table 1), creating a new leadership pair with a higher priority than the priority of its previous leader, and setting its level to zero. Each self-declared leader then attempts to capture the entire component by flooding its leadership pair. The smallest (using lexical ordering), *i.e.,* highest priority, leadership pair captures the entire component, and the algorithm halts.

The reason a new leader picks a higher priority than its old leader is that, because of deletion of links, it is possible that the old leader is no longer in the same component. Giving priority to the "youngest" leader guarantees that the ID part of the highest priority leadership pair is the actual ID of some process in the component, provided at least one round has elapsed since any fault.

Variables of DLE*.* For any process x, we have variables:

1. $x.id$, the ID of x. We assume that IDs are unique, and that they form an ordered set. That is, if x, y, and z are distinct processes, then either $x.id < y.id$ or $y.id < x.id$; and if $x.id < y.id$ and $y.id < z.id$, then $x.id < z.id$.
   By an abuse of notation, we will use the same notation to refer to both a process and its ID.
2. $x.leader$, the process that x has selected to be its leader, which we call the *leader* of x.

3. $x.level$, a non-negative integer which, in a legitimate configuration, is the distance from x to $x.leader$.
4. $x.nlp$, a non-positive integer called the *negative leader priority* of x. The value of $x.nlp$ is the negative of the priority that $x.leader$ assigned to itself when it declared itself to be a leader. As in [9], the value of $x.nlp$ is not bounded.
5. $x.vector = (x.nlp, x.leader, x.level)$, the *vector* of x. Vectors are ordered lexically.

   Although we list $x.vector$ as a variable, it is actually an ordered triple of other variables, and hence requires no extra space.
6. $x.parent$, the *parent* of x. In a legitimate configuration, if x is not the leader of its component, $x.parent$ is that neighbor of x which is the parent of x in the BFS tree rooted at the leader. If x is the leader, then $x.parent = x$.

   Because of a prior fault, $x.parent$ might not be the ID of x or of any neighbor of x. In this case, we say that $x.parent$ is *unlawful*.

Functions of DLE

1. Let $N(x)$ be the neighbors of x, and $U(x) = N(x) \cup \{x\}$.
2. If $v = (nlp, \ell, d)$ is a vector, we define $successor(v) = (nlp, \ell, d + 1)$, the smallest vector that is larger than v.
3. $Min\_Nbr\_Vector(x) = \min\{y.vector : y \in U(x)\}$, the *minimum neighborhood vector* of x.
4. $Local\_Minimum(x)$, Boolean, meaning that x is a local minimum, *i.e.,* $x.vector \le Min\_Nbr\_Vector(x)$.
5. $Good\_Root(x)$, Boolean, meaning that x is a local minimum and its own leader, and also a local root, *i.e.,* $x.leader = x.id$, $x.level = 0$, and $x.leader = x.parent = x$.
6. $Good\_Child(x)$, x is a *good child*, *i.e.,* $x.parent.vector = Min\_Nbr\_Vector(x)$ and $x.vector = successor(Min\_Nbr\_Vector(x))$.
7. $Parent(x) = p \in N(x)$ such that $x.vector = successor(p.vector)$. If there is more than one such neighbor of x, choose the one with the smallest ID. If there is no such neighbor of x, define $Parent(x) = x$.

Table 1. Program of DLE **for Process** x

```
A1  Reset    $Local\_Minimum(x)$       $\longrightarrow$  $x.nlp \leftarrow x.nlp - 1$
             $\neg Good\_Root(x)$                         $x.leader \leftarrow x.id$
                                                          $x.level \leftarrow 0$
                                                          $x.parent \leftarrow x$

A2  Attach   $\neg Local\_Minimum(x)$  $\longrightarrow$  $vector(x) \leftarrow$
             $\neg Good\_Child(x)$                            $successor(Min\_Nbr\_Vector(x))$
                                                          $x.parent \leftarrow Parent(x)$
```

Legitimate State of the Algorithm DLE*.* A configuration is *legitimate* if

1. For any component C of the network, there is exactly one process, $\ell_C \in C$ which is a good root, and every other process in C is a good child.
2. For any component C of the network, $x.vector = (\ell_C.nlp, \ell_C, d(x, \ell_C))$ for all $x \in C$, where d is (hop) distance between processes. That is, all processes in C have the same leadership pair, and level equal to the distance to the leader of the component.

3 Dynamic Leader Election with Priority

Algorithm DLE, given in Section 2, selects an arbitrary member of each component to be a leader of that component. In this section, we introduce the requirement that the elected leader of each component be the *best* process in the component, where "best" can be defined any number of ways, depending on the application. Our method is to define some kind of priority measure on all processes, and then make sure that the elected leader is the process which has the highest priority in the component. For example, the process of highest priority could be the process of least ID, or of greatest ID, or of greatest degree, *i.e.,* number of neighbors.

If no fault occurs for $O(Diam)$ rounds, DLE always chooses a leader for each component, but this leader could be any process of the component. This leader is likely not to have been a leader before the fault; it is even possible that the loss of one link of a component could cause the component to elect a new leader, even if no processes of the component were lost and no new processes were added. This behavior could be undesirable in practice. If we define priority of processes in such a way that it is largely unaffected by small faults, we will decrease the frequency of leadership changes in practice.

We assume an abstract function $Priority(x)$, the *priority* of a process x, which must depend only on the topology of the network, the ID of x, and data obtained by x from the application layer. In other words, $Priority(x)$ is not affected by any change of the variables of our algorithm. We also assume that $Priority(x)$ can be computed by x in $O(1)$ time.

Define $Max\_Priority(S) = \max\{Priority(x) : x \in S\}$, if S is any non-empty set of processes, and let $Best(S)$ be that process in S whose priority is equal to $Max\_Priority(S)$. Without loss of generality, the choice of $Best(S)$ is unique, since we can use ID as a tie-breaker. The output condition of DLEP is that, for any component C of the network, $Best(C)$ will be elected leader of C. DLEP consists of four *phases*. The first phase, which elects a *preliminary leader* ℓ_C for each component C and builds a *preliminary BFS tree* of C rooted at ℓ_C, is exactly an emulation of the algorithm DLE given in Section 2. The second and third phases of DLEP make use of the preliminary BFS tree in each component to compute the final leader and the final BFS tree of that component. The second phase of DLEP consists of a convergecast wave in the preliminary BFS tree. Let T_x be the subtree of the preliminary BFS tree rooted at a given process x. During the second phase, the *intermediate leader* of each x is computed to be

$Best(T_x)$. Thus, the intermediate leader of ℓ_C is $Best(C)$, which will also be the final leader of C. The third phase of DLEP consists of a broadcast wave, during which every process is told the identity of $Best(C)$, and selects that process to be its final leader. The fourth phase of DLEP consists of a flooding wave from $Best(C)$ which builds the final BFS tree in C.

Variables of DLEP. For any process x, we have variables, as listed below.

1. These variables are used for the first phase of DLEP, which emulates DLE.
   (a) $x.p\_leader$, the *preliminary leader* of x, which corresponds to $x.leader$ in DLE.
   (b) $x.nplp$, a non-positive integer called the *negative preliminary leader priority* of x, which corresponds to $x.nlp$ in DLE. The value of $x.nplp$ is the negative of the priority that $x.p\_leader$ assigned to itself when it declared itself to be a preliminary leader,
   (c) $x.p\_level$, the *preliminary level* of x, the distance from x to $x.p\_leader$, which corresponds to $x.level$ in DLE.
   (d) $x.p\_vector = (x.nplp, x.p\_leader, x.p\_level)$, the *preliminary vector* of x, which corresponds to $x.vector$ in DLE. Preliminary vectors are ordered lexically.
   (e) $x.p\_parent$, the parent of x in the preliminary BFS tree, which corresponds to $x.parent$ in DLE.
2. $x.i\_leader$, the *intermediate leader* of x, whose value in a stable configuration is $Best(T_x)$.
3. $x.ilp$, the *intermediate leader priority* of x, whose value in a stable configuration is $Priority(Best(T_x))$.
4. $x.i\_vector = (x.ilp, x.i\_leader)$, the *intermediate vector* of x. Intermediate vectors are ordered lexically.
5. $x.f\_leader$, the *final leader* of x, whose value in a stable configuration is $Best(C)$, the elected leader of the component C that x belongs to.
6. $x.f\_level$, the *final level* of x, whose value in a stable configuration is the distance from x to $x.f\_leader$.
7. $x.f\_parent$, whose value in a stable configuration is the parent of x in the final BFS tree.
   Although we list $x.p\_vector$ and $x.i\_vector$ as variables, they are actually ordered triples of other variables, and hence require no extra space.

Functions of DLEP. As in DLE, some of the functions of DLEP are given names which are capitalized versions of the names of variables. Is such cases, the value of the function is what x believes the value of the variable should be.

1. If $(nplp, \ell, d)$ is a preliminary vector, let $successor(nplp, \ell, d) = (nplp, \ell, d{+}1)$, the smallest vector that is larger than $(nplp, \ell, d)$.
2. $Min\_Nbr\_P\_Vector(x) = \min\{y.p\_vector : y \in N(x) \cup \{x\}\}$, the *minimum neighbor preliminary vector* of x.
3. $Local\_Minimum(x) \equiv Min\_Nbr\_P\_Vector(x) \geq x.p\_vector$, Boolean.

4. $Good\_Root(x) \equiv Local\_Minimum(x) \wedge (x.p\_leader = x) \wedge (x.p\_level = 0)$, Boolean.
5. $Good\_Child(x) \equiv x.p\_vector = successor(x.p\_parent.p\_vector)$, Boolean.
6. $P\_Parent(x) = y \in N(x)$ such that $y.p\_vector = Min\_Nbr\_P\_Vector(x)$. If there is more than one such neighbor of x, choose the one with the smallest ID. If there is no such neighbor of x, define $P\_Parent(x) = x$.
7. $P\_Chldrn(x) = \{y : Good\_Child(y)$ and $y.p\_parent = x\}$.
8. We define the Boolean function $Local\_P\_Tree\_Ok(x)$ on a process x to mean that, as far as x can tell by looking at its variables and those of its neighbors, the preliminary leader and the preliminary BFS tree have been constructed. More formally, $Local\_P\_Tree\_Ok(x)$ is true if the following conditions hold for x:

   – x is either a good root or a good child.
   – $x.p\_level = 0$ if and only if x is a good root.
   – $x.p\_leader = x$ if and only if x is a good root.
   – $y.p\_leader = x.p\_leader$ for all $y \in N(x)$.
   – $|y.p\_level - x.p\_level| \leq 1$ for all $y \in N(x)$.

9. $I\_Vector(x) = \max \begin{cases} (Priority(x), x) \\ \max\{y.i\_vector : y \in P\_Chldrn(x)\} \end{cases}$.
10. $F\_Leader(x) = \begin{cases} x.i\_leader & \text{if } Good\_Root(x) \\ x.p\_parent.f\_leader & \text{otherwise} \end{cases}$
11. $F\_Level(x) = \begin{cases} 0 & \text{if } x.f\_leader = x \\ 1 + \min\{y.f\_level : y \in N(x)\} & \text{otherwise} \end{cases}$
12. $F\_Parent(x) = p \in N(x)$ such that $1 + f\_level(p) = f\_level(x)$. If there is more than one such neighbor of x, choose the one with the smallest ID. If there is no such neighbor of x, define $F\_Parent(x) = x$.

Legitimate Configurations for DLEP*.* We define a configuration of the network to be *pre-legitimate* if

1. For any component C of the network, there is exactly one process, $\ell_C \in C$, which is a good root; and all other processes of C are good children.
2. For any component C of the network, $x.p\_vector = (\ell_C.nplp, \ell_C, d(x, \ell_C))$ for all $x \in C$, where $d(x, \ell_C)$ is the distance from ℓ_C to x. That is, all processes in C have the same preliminary leadership pair, and preliminary level equal to the distance to the preliminary leader of the component.

A configuration of the network is *legitimate* if it is pre-legitimate, and if, for each component C and for all $x \in C$:

1. $x.i\_vector = (Priority(y), y)$, where $y = Best(T_x)$.
2. $x.f\_leader = Best(C)$.
3. $x.f\_level = d(x, Best(C))$, the distance from x to $Best(C)$.
4. $x.f\_parent = F\_Parent(x)$.

Table 2. Program of DLEP

A1 priority 1	Reset	$Local\_Minimum(x)$ $\neg Good\_Root(x)$	$\longrightarrow$	$x.nplp \leftarrow x.nplp - 1$ $x.p\_leader \leftarrow x.id$ $x.p\_level \leftarrow 0$ $x.p\_parent \leftarrow x$
A2 priority 1	Preliminary BFS Tree	$\neg Local\_Minimum(x)$ $\neg Good\_Child(x)$	$\longrightarrow$	$p\_vector(x) \leftarrow successor(Min\_Nbr\_Vector(x))$ $x.p\_parent \leftarrow P\_Parent(x)$
A3 priority 2	Intermediate Vector	$x.i\_vector \neq I\_Vector(x)$ $Local\_P\_Tree\_Ok(x)$	$\longrightarrow$	$x.i\_vector \leftarrow I\_Vector(x)$
A4 priority 3	Final Leader	$x.f\_leader \neq F\_Leader(x)$ $Local\_P\_Tree\_Ok(x)$	$\longrightarrow$	$x.f\_leader \leftarrow F\_Leader(x)$
A5 priority 4	Final Level	$x.f\_level \neq F\_Level(x)$ $Local\_P\_Tree\_Ok(x)$ $\forall y \in N(x)$: $y.f\_leader = x.f\_leader$	$\longrightarrow$	$x.f\_level \leftarrow F\_Level(x)$
A6 priority 5	Final Parent	$x.f\_parent \neq F\_Parent(x)$ $Local\_P\_Tree\_Ok(x)$ $\forall y \in N(x)$: $y.f\_leader = x.f\_leader$	$\longrightarrow$	$x.f\_parent \leftarrow F\_Parent(x)$

Explanation of Actions. Action A1 corresponds to Action A1 of DLE, while Action A2 corresponds to Action A2 of DLE. Together, these two actions cause the preliminary leader ℓ_C of each component C to be chosen, and the preliminary BFS tree to be constructed.

Action A3 is the action of the convergecast wave that chooses the intermediate vector for each process after the preliminary BFS tree has been constructed. It is possible for some processes to execute A3 prematurely because they believe, based on local information, that the preliminary BFS tree is finished; in these cases, these processes will recompute their intermediate vectors later.

The final leader of the component C, namely, FL_C, will be the intermediate leader of ℓ_C. Action A4 is the action of the broadcast wave, starting at ℓ_C, that informs every process of the choice of final leader.

After every process knows the final leader, Actions A5 and A6 construct the BFS tree, assigning to each process its final level and final parent, in a broadcast wave starting at FL_C.

4 Algorithm DLEND

Post–legitimate Configurations. Suppose that γ is a legitimate configuration for a distributed algorithm $\mathcal{A}$ on a given network G. Suppose G' is a new network that is obtained from G by an arbitrary topological change; *i.e.,* the processes of G' are the same as those of G, and no variables of any process have been changed, but the links may be different. This change defines a configuration γ' on G', where each process has the same values of its variables as at γ. If a process

x contains a variable which is a pointer to a process y which is a neighbor of x in G, and if y is no longer a neighbor of x in γ', the pointer does not change, but it has nothing to point to. In this case, we say that that pointer is *unlawful* at γ'. We say that γ' is *post–legitimate* configuration.

We now present Algorithm DLEND for the dynamic leadership election problem. DLEND has the following properties:

1. Self Stabilization and Silence: Starting from an arbitrary configuration, within $O(\mathit{Diam})$ rounds, a legitimate configuration is reached and there are no further actions.
2. Incumbent Priority: Starting from a post–legitimate configuration, if a component C contains at least one process which was a leader at the previous legitimate configuration, one of those processes will be elected leader of that component.
3. No Dithering: Starting from a post–legitimate configuration, no process will change its choice of leader more than once.

DLEND shares the first property with Algorithms DLE and DLEP. The second property is an extension of the priority property of DLEP. To achieve the incumbent and no dithering properties, we introduce *colors* to guide the order of computation.

DLEND is very much like DLEP, except that, to achieve the no dithering property, each process is given a *color*, which is in integer in the range $[0 \dots 5]$. The color of a process is related to its current role in the computation. The purpose of the colors is to ensure that the final leader of a process is not computed too early. In a computation that starts from a post–legitimate configuration, the processes pass through the following sequence of colors.

0. In a legitimate configuration, $x.\mathit{color} = 0$ for each process x.
1. Each color changes to 1 when the preliminary BFS tree is being constructed.
2. All processes change color to 2 in a convergecast wave when the preliminary BFS tree is completed.
3. All processes change color to 3 in a broadcast wave after the preliminary leader has color 2.
4. All processes change color to 4 in a convergecast wave that computes the intermediate vector of each process. Each process x chooses as its intermediate leader a process in the subtree T_x which was a leader in the previous legitimate configuration, if any such previous leader exists in T_x.
   It is possible for a process to change color to 2, 3, or 4 prematurely, and then go back to color 1. This can occur when some of the preliminary calculations of the preliminary BFS tree are incorrect, and need to be redone. However, when a good root changes its color to 4, the preliminary BFS tree has been correctly calculated.
5. All processes change color to 5 in a broadcast wave, during which each process chooses a new value of the final leader. All processes in a component choose the same final leader, which is the intermediate leader of the preliminary leader.

Finally, in a flooding wave starting from the final leader, all processes change their color to 0. They then construct the final BFS tree, and eventually the configuration is legitimate and silent.

We define the actions of DLEND in Table 3. DLEND uses variables of DLEP, as well as

- $x.\mathit{former\_leader\_in\_subtree}$, Boolean, meaning that T_x contains a process which was a final leader in the last legitimate configuration.
- $x.\mathit{color} \in \{0, 1, 2, 3, 4, 5\}$.
- $x.\mathit{i\_vector} = (x.\mathit{former\_leader\_in\_subtree}, \mathit{Priority}(x.\mathit{i\_leader}), x.\mathit{i\_leader})$

DLEND uses the functions of DLEP: $\mathit{successor}(\mathit{nplp}, \ell, d) = (\mathit{nplp}, \ell, d + 1)$, where (nplp, ℓ, d) is a preliminary vector, $\mathit{Min\_Nbr\_P\_Vector}(x)$, $\mathit{Good\_Root}(x)$, $\mathit{Local\_Minimum}(x)$, $\mathit{Good\_Child}(x)$, $\mathit{P\_Parent}(x)$, $\mathit{P\_Chldrn}(x)$, $\mathit{Priority}(x)$, and $\mathit{Local\_P\_Tree\_Ok}(x)$. One function of DLEP is redefined for DLEND, namely

$$\mathit{I\_Vector}(x) = \max \begin{cases} (\mathit{Is\_Leader}(x), \mathit{Priority}(x), x) \\ \max \{y.\mathit{i\_vector} : y \in \mathit{P\_Chldrn}(x)\} \end{cases}$$

DLEND uses a number of additional functions, as well.

- $\mathit{Local\_I\_Vector\_Ok}(x)$, Boolean, which is true if $x.\mathit{i\_vector} = \mathit{I\_Vector}(x)$.
- $\mathit{Local\_I\_Leader\_Ok}(x)$, Boolean, which is true if either $x.\mathit{i\_leader} = x$, or $x.\mathit{i\_leader} = y.\mathit{i\_leader}$ for some $y \in \mathit{P\_Chldrn}(x)$. If $\mathit{Local\_I\_Vector\_Ok}(x)$ then $\mathit{Local\_I\_Leader\_Ok}(x)$, but the converse does not hold.
- $\mathit{Local\_F\_Leader\_Ok}(x)$, Boolean, which is true if x is either a good root and $x.\mathit{f\_leader} = x.\mathit{i\_leader}$, or a good child and $x.\mathit{f\_leader} = x.\mathit{p\_parent}.\mathit{f\_leader}$.
- $\mathit{Is\_Leader}(x)$, Boolean, meaning that $x.\mathit{f\_leader} = x$.
- $\mathit{F\_Level}(x) = \begin{cases} 0 & \text{if } x.\mathit{f\_leader} = x \\ 1 + \min \{y.\mathit{f\_level} : y \in N(x)\} & \text{otherwise} \end{cases}$
- $\mathit{F\_Parent}(x) = p \in N(x)$ such that $p.\mathit{color} = 0$ and $1 + \mathit{f\_level}(p) = \mathit{f\_level}(x)$. If there is more than one such neighbor of x, choose the one with the smallest ID. If there is no such neighbor of x, define $\mathit{F\_Parent}(x) = x$.
- $\mathit{Error}(x)$, $\mathit{Normal\_Start}(x)$, and $\mathit{Can\_Start}(x)$, Boolean, defined below:

If x is a good child, we say that x is *color compatible with its parent* if $x.\mathit{color}, x.\mathit{parent}.\mathit{color} \in \{1, 2, 3, 4, 5\}$ and $x.\mathit{color} = x.\mathit{parent}.\mathit{color}$ is odd, $x.\mathit{color}$ is even and $|x.\mathit{parent}.\mathit{color} - x.\mathit{color}| \le 1$, $x.\mathit{color}, x.\mathit{parent}.\mathit{color} \in \{0, 5\}$, or $x.\mathit{color}, x.\mathit{parent}.\mathit{color} \in \{0, 1\}$.

Define the Boolean function $\mathit{Color\_Error}(x)$ to be true if either x is a true root which is color incompatible with some $y \in \mathit{P\_Chldrn}(x)$, or x is a true child which is color incompatible with $x.\mathit{p\_parent}$.

Define the Boolean function $\mathit{P\_Error}(x)$ to be true if $x.\mathit{color} \notin \{0, 1\}$ and there is some $y \in N(x)$ such that $y.\mathit{p\_vector} > \mathit{successor}(x.\mathit{p\_vector})$, *i.e.,* x perceives that y is enabled to execute Action A3. Define the Boolean function $\mathit{I\_Error}(x)$ to be true if either $x.\mathit{color} = 4$ and $\neg\mathit{Local\_I\_Vector\_Ok}(x)$ or $x.\mathit{color} = 5$ and $\neg\mathit{Local\_I\_Leader\_Ok}(x)$.

Let $\mathit{Error}(x)$ be the disjunction of the previous functions, $\mathit{Color\_Error}(x)$, $\mathit{P\_Error}(x)$, $\mathit{I\_Error}(x)$.

Let $\mathit{Normal\_Start}(x)$ be the Boolean function which is true if $x.\mathit{color} = 0$; and if either $\neg\mathit{Local\_I\_Leader\_Ok}(x)$, $\neg\mathit{Local\_F\_Leader\_Ok}(x)$, $y.\mathit{color} = 1$ for some $y \in \mathit{P\_Chldrn}(x)$, and x is a good root; or a good child and $x.\mathit{p\_parent}.\mathit{color} = 1$.

Let $\mathit{Can\_Start}(x)$ be the Boolean function which is true if $x.\mathit{color} \neq 1$ and $\mathit{Error}(x)$, or $\mathit{Normal\_Start}(x)$.

Explanation of Actions. Action A1 changes the color of process to 1, indicating that computation of the preliminary BFS tree is to start, or restart. A process x is enabled to execute A1 when it decides, based on the values of its neighbors, that it must start the computation of the preliminary BFS tree, or that there has been a fault that cannot be corrected without restarting that computation. Color 1 is "contagious," *i.e.,* if $x.\mathit{color} = 0$ and a neighbor process has color 1, x can execute A1.

Action A2 corresponds to Action A1 of DLE, while Action A3 corresponds to Action A2 of DLE. Together, these two actions cause the preliminary leader, ℓ_C of each component C to be chosen, and the preliminary BFS tree to be constructed. While a process is executing those actions, its color remains 1.

Actions A4 and A5 have no analog in Algorithms DLE and DLEP. All processes execute A4 from the bottom of the preliminary BFS tree, changing their colors to 2, and then top-down in the same tree, changing their colors to 3. No other variables are changed, so these actions do not contribute to computation of the preliminary, intermediate, or final leaders. This apparently pointless "waste" of $2\mathit{Diam}$ rounds is needed to ensure the no dithering property of DLEND.

Action A6 is the action of the convergecast wave that chooses the intermediate vector for each process after the preliminary BFS tree has been constructed. As each process executes A6, its color changes to 4.

It is possible for some processes to execute A6 prematurely because they believe, based on local information, that the preliminary BFS tree is finished; in these cases, these processes will recompute their intermediate vectors later. However, the no dithering property is guaranteed by the fact that, if the computation started from a post–legitimate state, no good root will ever execute Action A6 unless it is the actual preliminary leader and the preliminary BFS tree has been correctly constructed.

The final leader of the component C, namely, FL_C, will be the intermediate leader of ℓ_C. Action A7 is the action of the broadcast wave, starting at ℓ_C, that informs every process of the choice of final leader. Each process changes its color to 5 when it executes A7.

When FL_C executes A7, it then executes Action A8, changing its color to 0, starting construction of the final BFS tree. All processes change their color to 0 in a flooding wave starting from FL_C, as they execute Action A9.

Actions A10 and A11 complete the construction of the final BFS tree, assigning to each process its final level and final parent.

Table 3: Program of DLEND

A1 priority 1	Start	$Can\_Start(x)$	$\longrightarrow$	$x.color \leftarrow 1$
A2 priority 2	Declare P–Leader	$Local\_Minimum(x)$ $\neg Good\_Root(x)$	$\longrightarrow$	$x.nplp \leftarrow x.nplp - 1$ $x.p\_leader \leftarrow x.id$ $x.p\_level \leftarrow 0$ $x.color \leftarrow 1$ $x.p\_parent \leftarrow x$
A3 priority 3	P–Attach	$\neg Local\_Minimum(x)$ $\neg Good\_Child(x)$	$\longrightarrow$	$p\_vector(x) \leftarrow$ $successor(Min\_Nbr\_Vector(x))$ $x.color \leftarrow 1$ $x.p\_parent \leftarrow P\_Parent(x)$
A4 priority 4	Wave 2	$x.color = 1$ $\forall y \in N(x) : y.color \in \{1, 2\}$ $\forall y \in P\_Chldrn(x) : y.color = 2$ $Local\_P\_Tree\_Ok(x)$ $\neg Error(x)$	$\longrightarrow$	$x.color \leftarrow 2$
A5 priority 5	Wave 3	$x.color = 2$ $Good\_Root(x) \vee$ $(x.p\_parent.color = 3)$ $\forall y \in N(x) : y.color \in \{2, 3\}$ $Local\_P\_Tree\_Ok(x)$ $\neg Error(x)$	$\longrightarrow$	$x.f\_leader \leftarrow F\_Leader(x)$ $x.color \leftarrow 3$
A6 priority 6	Convergecast Intermediate Leader	$x.color = 3$ $\forall y \in N(x) : y.color \in \{3, 4\}$ $Local\_P\_Tree\_Ok(x)$ $\neg Error(x)$	$\longrightarrow$	$x.i\_vector \leftarrow I\_Vector(x)$ $x.color \leftarrow 4$
A7 priority 7	Broadcast Final Leader	$x.color = 4$ $Good\_Root(x) \vee$ $(x.p\_parent.color = 5)$ $\forall y \in N(x) : y.color \in \{4, 5\}$ $Local\_P\_Tree\_Ok(x)$ $\neg Error(x)$	$\longrightarrow$	$x.f\_leader \leftarrow F\_Leader(x)$ $x.color \leftarrow 5$
A8 priority 8	Become Final Leader	$x.color = 5$ $\forall y \in N(x) : y.color = 5$ $x.f\_leader = x$ $Local\_P\_Tree\_Ok(x)$	$\longrightarrow$	$x.f\_level \leftarrow 0$ $x.f\_parent \leftarrow x$ $x.color \leftarrow 0$
A9 priority 8	F–Attach	$x.color = 5$ $\forall y \in N(x) : y.color \in \{5, 0\}$ $\exists y \in N(x) : y.color = 0$ $Local\_P\_Tree\_Ok(x)$ $\neg Error(x)$	$\longrightarrow$	$x.f\_level \leftarrow 0$ $x.f\_parent \leftarrow F\_Parent(x)$ $x.color \leftarrow 0$
A10 priority 8	F–Level	$x.color = 0$ $\forall y \in N(x) : y.color = 0$ $x.f\_level \neq F\_Level(x)$ $Local\_P\_Tree\_Ok(x)$ $\neg Error(x)$	$\longrightarrow$	$x.f\_level \leftarrow F\_Level(x)$ $x.f\_parent \leftarrow F\_Parent(x)$
A11 priority 9	F–Parent	$x.color = 0$ $\forall y \in N(x) : y.color = 0$ $x.f\_parent \neq F\_Parent(x)$ $Local\_P\_Tree\_Ok(x)$ $\neg Error(x)$	$\longrightarrow$	$x.f\_parent \leftarrow F\_Parent(x)$

5 Sketches of Proofs

After one round of a computation of DLE has elapsed, every process is either a true child or a true root, and, in each component C, the leadership pair of some true root ℓ_C is the minimum leadership pair in that component. Within *Diam* additional rounds, a BFS tree is constructed in C rooted at ℓ_C. Thus, DLE converges within $O(\mathit{Diam})$ rounds, and is silent upon reaching a legitimate configuration.

A distributed algorithm might be proved to converge in a finite number of rounds, but still possibly never converge under the unfair daemon, since that daemon might never select a specific enabled process. We prove that DLE works under the unfair daemon by proving that every computation of DLE is finite.

For each process x, the value of $x.\mathit{vector}$ cannot decrease, and in fact, at every step of the computation, $x.\mathit{vector}$ increases for some process x. Furthermore, from the initial configuration, we can compute an upper bound on the number of possible future values of $x.\mathit{vector}$. Thus, x can execute only finitely many times during the computation, and the computation thus cannot be infinite.

The proof of DLEP uses the *convergence stair* method. We give a nested sequence of benchmarks, each a closed predicate, and the last one legitimacy.

There is a morphism from DLEP to DLE, meaning that every configuration of DLEP maps to a configuration of DLE, and that this mapping is consistent with the actions of the algorithms. From the fact that DLE is correct and silent under the unfair daemon, we can thus conclude that every computation of DLEP eventually reaches a configuration where first benchmark holds. *i.e.,* the preliminary BFS tree is complete.

We define the next benchmark to mean that all values of *i_vector* are correct, the next benchmark to mean that all values of *f_leader* are correct, and the fourth and last benchmark to mean that the configuration is legitimate.

We can show that it takes $O(\mathit{Diam})$ rounds to achieve the first benchmark, and $O(\mathit{Diam})$ additional rounds to achieve each subsequent benchmark. Thus, DLEP converges in $O(\mathit{Diam})$ rounds.

To prove that DLEP works under the unfair daemon, we prove that every computation of DLEP is finite. Given a computation Γ of DLEP, we have, from the properties of DLE and the morphism of DLEP to DLE, that Γ contains only finitely many instances of Actions A1 and A2. We then prove that, after the last instance of one of those actions, Γ contains only finitely many instances of Action A3. We then prove that, after the last instance Action A3, there are only finitely many instances of Action A4. Proceeding in this fashion, we eventually prove that Γ has only finitely many actions, and thus ends at a configuration where no process is enabled. We then prove that if no process is enabled, the configuration is legitimate.

DLEND elects both a preliminary leader and a final leader in each component, but uses colors to control the order of actions, in order to enforce the incumbent and no dithering properties.

Consider an action Γ which begins at post–legitimate configuration, which implies that every process has color 0. If the configuration of a component C

differs only slightly from legitimate, it could happen that DLEND converges without changing the color of any process and every process has the same final leader as initially. Otherwise, we can prove that no process has color 5 until after the preliminary BFS tree is constructed. After that, each process changes its color to 5 exactly once, at which time, and no other, it can change its choice of final leader. All processes in C will choose the same final leader. Each process of C will then, just once, change its color to 0, after which the final BFS tree is constructed rooted at the elected final leader.

We can also prove that, starting at any configuration, DLEND is eventually silent. Suppose Γ is a computation of DLEND. By the result we proved for DLE, Γ can contain only finitely many instances of a structural action. We then prove that $Error(x)$ can be true for some process x only finitely many times, after which there can be only finitely many instances of a color action. Then, all colors are 0, and the only actions that can be enabled are A10 and A11. After finitely many steps, the configuration is legitimate, and silent.

6 Conclusion

We have presented three distributed leader election algorithm for dynamic networks which work under the unfair daemon, and which use no global clock. For Algorithm DLEND, if purely topological changes occur after a legitimate state, an incumbent leader is re-elected, and no process changes leader more than once.

References

1. Afek, Y., Bremler, A.: Self-stabilizing unidirectional network algorithms by power-supply (extended abstract). In: SODA, pp. 111–120 (1997)
2. Arora, A., Gouda, M.G.: Distributed reset. IEEE Transactions on Computers 43, 1026–1038 (1994)
3. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC 1993), pp. 652–661 (1993)
4. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing leader election in optimal space. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 109–123. Springer, Heidelberg (2008); Also to appear in Theoretical Computer Science
5. Derhab, A., Badache, N.: A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. IEEE Transactions on Parallel and Distributed Systems 19(7), 926–939 (2008)
6. Dijkstra, E.W.: Self stabilizing systems in spite of distributed control. Communications of the Association of Computing Machinery 17, 643–644 (1974)
7. Dolev, S.: Self-Stabilization. The MIT Press, Cambridge (2000)
8. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. (1997)
9. Ingram, R., Shields, P., Walter, J.E., Welch, J.L.: An asynchronous leader election algorithm for dynamic networks. In: IPDPS, pp. 1–12 (2009)

Loop-Free Super-Stabilizing Spanning Tree Construction*

Lélia Blin[1,3], Maria Gradinariu Potop-Butucaru[2,3,4], Stephane Rovedakis[1,5], and Sébastien Tixeuil[2,3]

[1] Université d'Evry-Val d'Essonne, 91000 Evry, France
[2] Université Pierre & Marie Curie - Paris 6, 75005 Paris, France
[3] LIP6-CNRS UMR 7606, France
{lelia.blin,maria.gradinariu,sebastien.tixeuil}@lip6.fr
[4] INRIA REGAL, France
[5] Laboratoire IBISC-EA 4526, 91000 Evry, France
stephane.rovedakis@ibisc.fr

Abstract. We propose an univesal scheme to design loop-free and super-stabilizing protocols for constructing spanning trees optimizing any tree metrics (not only those that are isomorphic to a shortest path tree).

Our scheme combines a novel super-stabilizing loop-free BFS with an existing self-stabilizing spanning tree that optimizes a given metric. The composition result preserves the best properties of both worlds: super-stabilization, loop-freedom, and optimization of the original metric without any stabilization time penalty. As case study we apply our composition mechanism to two well known metric-dependent spanning trees: the maximum-flow tree and the minimum degree spanning tree.

1 Introduction

New distributed emergent networks such as P2P or sensor networks face high churn (nodes and links creation or destruction) and various privacy and security attacks that are not easily encapsulated in the existing distributed models. One of the most versatile techniques to ensure forward recovery of distributed systems is that of *self-stabilization* [1,2,3]. A distributed algorithm is self-stabilizing if after faults and attacks hit the system and place it in some arbitrary global state, the system recovers from this catastrophic situation without external (*e.g.* human) intervention in finite time. A recent trend in self-stabilizing research is to complement the self-stabilizing abilities of a distributed algorithm with some additional *safety* properties that are guaranteed when the permanent and intermittent failures that hit the system satisfy some conditions. In addition to being self-stabilizing, a protocol could thus also tolerate crash faults [4,5], nap faults [6,7], Byzantine faults [8,9,10,11], a limited number of topology changes [12,13,14] and sustained edge cost changes [15,16].

The last two properties are especially relevant when building optimized spanning trees in dynamic networks, since the cost of a particular edge and the

* This work was partially founded by ANR projects SHAMAN and SPADES.

network topology are likely to evolve through time. If a spanning tree protocol is *only* self-stabilizing, it may adjust to the new costs or network topology in such a way that a previously constructed spanning tree evolves into a disconnected or a looping structure (of course, in the absence of network modifications, the self-stabilization property guarantees that *eventually* a new spanning tree is constructed). Now, a packet routing algorithm is *loop free* [17,18] if at any point in time the routing tables are free of loops, despite possible modification of the edge-weights in the graph (*i.e.*, for any two nodes u and v, the actual routing tables determines a simple path from u to v, at any time). The *loop-free* property [15,16] in self-stabilization gives the following guarantee. A spanning tree being constructed (not necessarily a "minimal" spanning tree for some metric), then the self-stabilizing convergence to a "minimal" spanning tree maintains a spanning tree at all times. Obviously, this spanning tree is not "minimal" at all times. The consequence of this safety property in addition to that of self-stabilization is that the spanning tree structure can still be used (*e.g.* for routing) while the protocol is adjusting, and makes it suitable for networks that undergo such very frequent dynamic changes. In order to deal with the network churn, *super-stabilization* captures the quality of services a tree stucture can offer during and after a localized topological change. Super-stabilization [19] is an extension of self-stabilization for dynamic settings. The idea is to provide some minimal guarantees (a *passage* predicate) while the system repairs after a topology change. In the case of optimized spanning trees algorithms while converging to a correct configuration (*i.e.* an optimized tree) after some topological change, the system keeps offering the tree service during the stabilization time to all members that have not been affected by this modification.

Related works. Relatively few works investigate merging self-stabilization and loop free routing, with the notable exception of [15,16,20]. In [15], Cobb and Gouda propose a self-stabilizing algorithm which constructs spanning trees with loop-free property. This algorithm allows to optimize general tree metrics from a considered root, such as bandwidth, delay, distance, etc ... To this end, each node maintains a value which reflects its cost from the root for the optimized metric, for example the maximum amount of bandwith on its path to reach the root. The basic idea is to allows a node to select a neighbor as its parent if this one offers a better cost. To avoid loop creation, when the cost of its parent or the edge-cost to its parent changed a propagation of information is started to propagate the new value. A node can safely change its parent if its propagation of information is ended. Thus, a node can not select one of its descendant as its parent. This algorithm requires a upper bound on the network diameter known to every participant to detect the presence of a cycle and to reset the states of the nodes. Each node maintains its distance from the root and a cycle is detected when the distance of a node is higher than the diameter upper bound.

Johnen and Tixeuil [16] propose another loop-free self-stabilizing algorithm constructing spanning trees, which makes no assumption on the network. This algorithm follows the same approach used in [15], that is using propagation of information in the tree. As in [15], this second algorithm constructs trees optimizing several metrics from a root, *e.g.*, depth first search tree, breadth first search tree, shortest path tree, etc. Since no upper bound on the network diameter is used, when a cycle is present in the initial network state the protocol continues the initiate propagation of information to grow the value of the nodes in the cycle. The values of these nodes grow until the value of a node reaches a threshold which is the value of a node out of the cycle. Thus, the node reaching this threshold discover a neighbor which offers a better value and can select it to break the cycle. When no cycle is present in the network, the system converges to a correct state.

Also, both protocols use only a reasonable amount of memory ($O(\log n)$ bits per node) and consider networks with static topology and dynamic edge costs. However, the metrics that are considered in [15,16] are derivative of the shortest path (distance graph) metric. It is considered a much easier task in a distributed setting than that of tree metrics not based on distances, *e.g.*, minimum spanning tree, minimum degree spanning tree, maximum leaf spanning tree, etc. Indeed, the associated metric is *locally optimizable* [21], allowing essentially locally greedy approaches to perform well. By contrast, some sort of *global optimization* is needed for tree metrics not based on distances, which often drives higher complexity costs and thus less flexibility in dynamic networks.

Recently, [20] proposed a loop-free self-stabilizing algorithm to solve the minimum spanning tree problem for networks, assuming a static topology but dynamic edge costs. None of the previously mentioned works can cope with both dynamic edge changes (loop-freedom) and dynamic local topology changes (super-stabilization). Also, previous works are generic only for local tree metrics, while global tree metrics require *ad hoc* solutions.

Our contributions. We propose a distributed generic scheme to transform existing self-stabilizing protocols that construct spanning tree optimizing an arbitrary tree metric (local or global), adding loop-free and super-stabilizing properties to the input protocol. Contrary to existing generic protocols [15,16], our approach provides the loop-free property for *any* tree metric (global or local, rather than only local). Our technique also adds super-stabilization, which the previous works do not guarantee. Our scheme consists in composing a distributed self-stabilizing spanning tree algorithm (established and proved to be correct for a given metric) with a novel BFS construction protocol that is both loop-free and super-stabilizing. The output of our scheme is a loop-free super-stabilizing spanning tree optimizing the tree metric of the input protocol. Moreover, we provide complexity analysis for the BFS construction in both static and dynamic settings. We examplify our scheme with two case study: the maximum flow tree and the minimum degree spanning tree. In both cases, the existing self-stabilizing algorithms can be enhanced via our method with both loop-free and super-stabilizing properties. Interestingly enough, the stabilization time complexity of the original protocols is not worsen by the transformation.

2 Model and Notations

We consider an undirected weighted connected network $G = (V, E, w)$ where V is the set of nodes, E is the set of edges and $w : E \rightarrow \mathbb{R}^+$ is a positive cost function. Nodes represent processors and edges represent bidirectional communication links. Additionally, we consider that $G = (V, E, w)$ is a dynamic network in which the weight of the communication links and the sets of nodes and edges may change. We consider anonymous networks (i.e., processors have no IDs), with one distinguished node, called the *root*[1]. Throughout the paper, the root is denoted r. We denote by $\deg(v)$ the number of v's neighbors in G. The $\deg(v)$ edges incident to any node v are labeled from 1 to $\deg(v)$, so that a processor can distinguish the different edges incident to a node.

The processors asynchronously execute their programs consisting of a set of variables and a finite set of rules. The variables are part of the shared register which is used to communicate with the neighbors. A processor can read and write its own registers and can only read the shared registers of its neighbors. Each processor executes a program consisting of a sequence of guarded rules. Each *rule* contains a *guard* (boolean expression over the variables of a node and its neighborhood) and an *action* (update of the node variables only). Any rule whose guard is *true* is said to be *enabled.* A node with one or more enabled rules is said to be *privileged* and may make a *move* executing the action corresponding to the chosen enabled rule.

A *local state* of a node is the value of the local variables of the node and the state of its program counter. A *configuration* of the system $G = (V, E)$ is the cross product of the local states of all nodes in the system. The transition from a configuration to the next one is produced by the execution of an action of at least one node. We assume a distributed weakly fair scheduler. A *computation* of the system is defined as a *weakly fair, maximal* sequence of configurations, $e = (c_0, c_1, \ldots c_i, \ldots)$, where each configuration c_{i+1} follows from c_i by the execution of a single action of at least one node. During an execution step, one or more processors execute an action and a processor may take at most one action. *Weak fairness* of the sequence means that if any action in G is continuously enabled along the sequence, it is eventually chosen for execution. *Maximality* means that the sequence is either infinite, or it is finite and no action of G is enabled in the final global state.

In the sequel we consider the system can start in any configuration. That is, the local state of a node can be corrupted. Note that we don't make any assumption on the bound of corrupted nodes. In the worst case all the nodes in

[1] Observe that the two self-stabilizing MST algorithms mentioned in the Previous Work section assume that the nodes have distinct IDs with no distinguished nodes. Nevertheless, if the nodes have distinct IDs then it is possible to elect one node as a leader in a self-stabilizing manner. Conversely, if there exists one distinguished node in an anonymous network, then it is possible to assign distinct IDs to the nodes in a self-stabilizing manner [2]. Note that it is not possible to compute deterministically a MST in a fully anonymous network (i.e., without any distinguished node), as proved in [22].

the system may start in a corrupted configuration. In order to tackle these faults we use self-stabilization techniques.

Definition 1 (self-stabilization). *Let $\mathcal{L}_\mathcal{A}$ be a non-empty* legitimacy predicate[2] *of an algorithm $\mathcal{A}$ with respect to a specification predicate Spec such that every configuration satisfying $\mathcal{L}_\mathcal{A}$ satisfies Spec. Algorithm $\mathcal{A}$ is* self-stabilizing *with respect to Spec iff the following two conditions hold:*
*(i) Every computation of $\mathcal{A}$ starting from a configuration satisfying $\mathcal{L}_\mathcal{A}$ preserves $\mathcal{L}_\mathcal{A}$ (*closure*).*
*(ii) Every computation of $\mathcal{A}$ starting from an arbitrary configuration contains a configuration that satisfies $\mathcal{L}_\mathcal{A}$ (*convergence*).*

We define bellow a *loop-free* configuration of a system as a configuration which contains paths with no cycle between any couple of nodes in the system.

Definition 2 (Loop-Free Configuration). *In a configuration C, each node $v \in V$ has a parent noted p_v (except the node designed as root which has no parent). Let $P(u, v) =< e_0, \dots, e_k >$ be the set of edges on the path from u to v, such that $e_0 = (u, p_u)$ and $e_k = (x, p_x)$ with $p_x = v$ in C. Let $Cycle(u, v)$ be the following predicate defined for two nodes u, v on configuration C:*

$$Cycle(u, v) \equiv \exists P(u, v), P(v, u) : P(u, v) \cap P(v, u) = \emptyset.$$

A loop-free configuration is a configuration of the system which satisfies $\forall u, v : Cycle(u, v) = false$.

We use the definition of a loop-free configuration to define a *loop-free stabilizing* system.

Definition 3 (Loop-Free Stabilization). *An algorithm is called loop-free stabilizing if and only if it is self-stabilizing and there exists a non-empty set of configurations such that the following conditions hold: (i) Every computation starting from a loop-free configuration remains loop-free (closure). (ii) Every computation starting from an arbitrary configuration contains a loop-free configuration (convergence).*

Definition 4 (Super-stabilization [19]). *Algorithm $\mathcal{A}$ is super-stabilizing with respect to a class of topology change events Λ iff the following two conditions hold:*
(i) $\mathcal{A}$ is self-stabilizing and (ii) for every computation beginning at a legitimate configuration and containing a single topology change events of type Λ, a passage predicate holds.

In the sequel we study the problem of constructing a spanning tree optimizing a desired metric in self-stabilizing manner, while guaranteeing the loop-free and super-stabilizing properties.

[2] A legitimacy predicate is defined over the configurations of a system and is an indicator of its correct behavior.

3 Super-Stabilizing Loop-Free BFS

In this section, we describe the extension of the self-stabilizing loop-free algorithm proposed in [16] to dynamic networks. Furthermore, we disscuss the super-stabilization of new algorithm. Interestingly, our algorithm preserves the loop-free property without any degradation of the time complexity of the original solution.

3.1 Algorithm Description

Algorithm **Dynamic-LoopFree-BFS** constructs a BFS tree and guarantees the loop-free property for dynamic networks. That is, when topological changes arise in the network (addition or deletion of nodes or edges) the algorithm maintains a BFS tree without creating a cycle in the spanning tree. To this end, each node has two states: *Neutral*, noted N, and *Propagate*, noted P. A node in state N can safely select as parent its neighbor with the smallest distance (in hops) from the root without creating a cycle. A node in state P has an incoherent state according to its parent in the spanning tree. In this case, the node must not select a new parent otherwise a cycle can be created. So, this node has to inform first its descendants in the tree that an incoherency in the BFS tree was detected. Then, it corrects when all its subtrees have recovered a coherent state. Therefore, a node v in state P initiates a propagation of information with feedback in its subtree. When the propagation is finished the nodes in the subtree of v (including v) recover a correct distance and the state N.

We consider a particular node r which acts as the root of the BFS tree in the network. Every node executes the same algorithm, except the root which uses only Rule $\mathsf{R}_{\mathsf{InitRoot}}$ to correct its state. In a correct state, the root r of the BFS tree has no parent, a zero level and the state N. Otherwise, Rule $\mathsf{R}_{\mathsf{InitRoot}}$ is executed by r to correct its state.

The other five rules are executed by the other nodes of the network.

Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ is used by a node v with the state N if it detects a better parent, i.e., a neighbor node with a lower level than the level of its actual parent. In this case, v can execute this rule to update its state in order to select a new parent without creating a cycle in the tree.

If a node v has the best parent in its neighborhood but an incoherent level according to its parent, then v executes Rule $\mathsf{R}_{\mathsf{Level}++}$ to change its status to P and to initiate a propagation of information with feedback which aims to inform its descendants of its new correct level. A descendant x of node v with state N with a parent in state P executes Rule $\mathsf{R}_{\mathsf{Level}++}$ to continue the propagation and to take into account its new level.

When a leaf node x, descendant of v in Status P is reached, x stops the propagation by executing Rule $\mathsf{R}_{\mathsf{EndPropag}}$ to change its state to N and to obtain its correct level. The end of propagation is pull up in the tree using Rule $\mathsf{R}_{\mathsf{EndPropag}}$.

Rule $\mathsf{R}_{\mathsf{LevelCorrect}}$ corrects at node v the variable used to propagate the new level in the tree (variable $\mathsf{NewLevel}_v$) if this variable is lower than the actual level of v.

Rule $\mathsf{R}_{\mathsf{Dynamic}}$ deals with the dynamism of the network. This rule is executed by a node v when it detects that its parent is no more in the network and it cannot select with Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ a new parent because of its level (otherwise it may create a cycle). The aim of this rule is to increase the level of node v using propagations of information as with Rule $\mathsf{R}_{\mathsf{Level++}}$, until v's level allows v to select a neighbor as its new parent without creating a cycle.

Figure 2 illustrates the mechanic of Rule $\mathsf{R}_{\mathsf{Dynamic}}$. In Figure 2(a) is depicted a part of the constructed BFS tree before the deletion of the node of level 2. After the deletion of this node, the node v with level 3 executes Rule $\mathsf{R}_{\mathsf{Dynamic}}$ to increase its level (equal to the lowest neighbor level plus one) in order to recover a new parent. Figure 2(b) shows the new level of v and the new levels v's descendants when the first propagation is ended. However, a level of 5 is not sufficient to allow v to select a new parent, so a second propagation is started by v which affects the levels given by Figure 2(c). Note that a descendant of v can leave v's subtree to obtain a better level if possible, this can be observed in Figure 2(c). Finally, v reaches a state with a level which allows v to execute Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ to select its new parent, and v's descendants execute Rule $\mathsf{R}_{\mathsf{Level++}}$ to correct their levels according to v's level. Figure 2(d) shows the new levels computed by the nodes.

Detailed level description. In the following, we describe the variables, the predicates and the rules used by Algorithm **Dynamic-LoopFree-BFS**.

Variables: For any node $v \in V(G)$, we denote by $N(v)$ the set of all neighbors of v in G and by $\mathcal{D}_v$ the set of sons of v in the tree. We use the following notations:

- p_v: the parent of node v in the current spanning tree;
- status_v: the status of node v, P when v is in a propagation phase, N otherwise;
- level_v: the number of edges from v to the root r in the current spanning tree;
- $\mathsf{NewLevel}_v$: the new level in the current spanning tree (used to propagate the new level).

$$\widehat{level_v} \equiv \begin{cases} \min\{\mathsf{level}_u + 1 : u \in N(v)\} & \text{if } v \neq r \\ 0 & \text{otherwise} \end{cases}$$

$$Min_v \equiv \min\{u : u \in N(v) \wedge \mathsf{level}_u = \widehat{level_v} - 1 \wedge \mathsf{status}_u = N\}$$

$$\widehat{parent_v} \equiv \begin{cases} Min_u & \text{if } \exists u \in N(v), \mathsf{level}_u = \widehat{level_v} - 1 \wedge \mathsf{status}_u = N \\ \perp & \text{otherwise} \end{cases}$$

$$\mathcal{D}_v \equiv \{u : u \in N(v) \wedge \mathsf{p}_u = v \wedge \mathsf{level}_u > \mathsf{level}_v\}$$

$$ubl_v \equiv \begin{cases} \min\{\mathsf{level}_u - 1 : u \in \mathcal{D}_v\} & \text{if } \mathcal{D}_v \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

$$\mathbf{Propag_{End}}(v) \equiv (\forall u \in \mathcal{D}_v, \mathsf{status}_u = N)$$

$$\mathbf{P_{Change}}(v) \equiv (\widehat{level_v} < \mathsf{level}_v \vee (\mathsf{level}_v = \widehat{level_v} \wedge \mathsf{p}_v \neq \widehat{parent_v})) \wedge \widehat{parent_v} \neq \perp$$

$$\mathbf{Level_{up}}(v) \equiv \mathsf{level}_v \neq \mathsf{level}\mathsf{p}_v + 1 \vee (\mathsf{status}\mathsf{p}_v = P \wedge \mathsf{level}_v \neq \mathsf{NewLevel}\mathsf{p}_v + 1)$$

Fig. 1. Predicates used by the algorithm

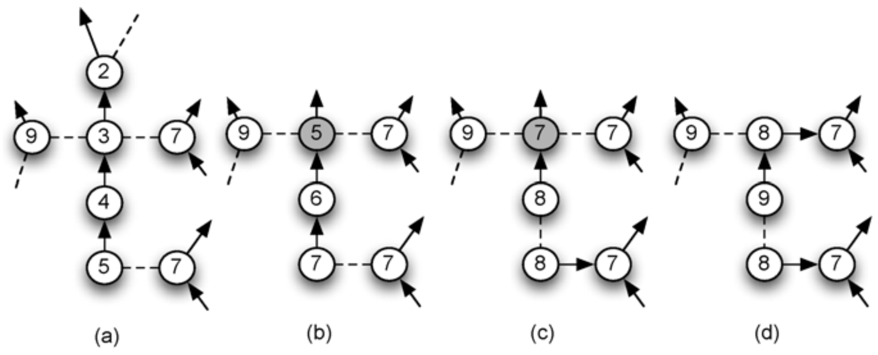


Fig. 2. Correction of the BFS tree after a node deletion

The root of the tree executes only the first rule, named $\mathsf{R}_{\mathsf{InitRoot}}$, while the other nodes execute the five last rules.

$\mathsf{R}_{\mathsf{InitRoot}}$: **(Root Rule)**
  if $v = r \wedge (\mathsf{p}_v \neq \perp \vee \mathsf{level}_v \neq 0 \vee \mathsf{NewLevel}_v \neq 0 \vee \mathsf{status}_v \neq N)$
  then $\mathsf{p}_v := \perp; \mathsf{level}_v := 0; \mathsf{NewLevel}_v := 0; \mathsf{status}_v := N;$

$\mathsf{R}_{\mathsf{SafeChangeP}}$: **(Safe parent change Rule)**
  if $v \neq r \wedge \mathsf{status}_v = N \wedge \mathbf{P_{Change}}(v)$
  then $\mathsf{level}_v := \widehat{level_v}; \mathsf{NewLevel}_v := \mathsf{level}_v; \mathsf{p}_v := \widehat{parent_v};$

$\mathsf{R}_{\mathsf{Level++}}$: **(Increment level Rule)**
  if $v \neq r \wedge \mathsf{status}_v = N \wedge \mathsf{p}_v \in N(v) \wedge \neg\mathbf{P_{Change}}(v) \wedge \mathbf{Level_{up}}(v)$
  then $\mathsf{status}_v := P; \mathsf{NewLevel}_v := \mathsf{NewLevel}_{\mathsf{p}_v} + 1;$

$\mathsf{R}_{\mathsf{EndPropag}}$: **(End of propagation Rule)**
  if $v \neq r \wedge \mathsf{status}_v = P \wedge \mathbf{Propag_{End}}(v) \wedge ubl_v \geq \mathsf{NewLevel}_v$
  then $\mathsf{status}_v := N; \mathsf{level}_v := \mathsf{NewLevel}_v;$

$\mathsf{R}_{\mathsf{LevelCorrect}}$: **(Level correction Rule)**
  if $v \neq r \wedge \mathsf{NewLevel}_v < \mathsf{level}_v$ **then** $\mathsf{NewLevel}_v := \mathsf{level}_v;$

$\mathsf{R}_{\mathsf{Dynamic}}$: **(Increment level Rule for dynamic networks)**
  if $v \neq r \wedge \mathsf{status}_v = N \wedge \mathsf{p}_v \notin N(v) \wedge \neg\mathbf{P_{Change}}(v)$
  then $\mathsf{status}_v := P; \mathsf{NewLevel}_v := \widehat{level_v};$

3.2 Correctness Proof

The algorithm proposed in the precedent subsection extends the algorithm of [16] to dynamic network topologies. When the system is static the correctness of the algorithm directly follows from the results proven in [23]. In the following, we focus only on the case of dynamic topologies, i.e., when nodes/edges of the tree are deleted or nodes/edges are added in the network. Note that in the following, we only study the case of an edge failure. A node failure produces the same consequences, i.e., the spanning tree is splitted and some nodes have no parent. Moreover, we do not consider edges out of the tree because this does not lead the system in an illegitimate configuration. After each failure of node or edge in the tree, we assume the underlying network is always connected.

In [23], a legitimate configuration for the algorithm is defined by the following predicate satisfied by every node $v \in V$: $\mathrm{Pr}_v^{\mathcal{LP}} \equiv [(v = r) \wedge (\mathsf{level}_v = 0) \wedge (\mathsf{status}_v = N)] \vee [(v \neq r) \wedge (\mathsf{level}_v = \overline{\mathsf{level}}_v) \wedge (\mathsf{status}_v = N) \wedge (\mathsf{level}_v = \mathsf{level}_{\mathsf{p}_v} + 1)]$, with $\forall v \neq r, \overline{\mathsf{level}}_v = \min\{\overline{\mathsf{level}}_u + 1 : u \in N(v)\}$ defines the optimal level of node v.

Note that after a failure of an edge of the tree T, Predicate $\mathrm{Pr}_v^{\mathcal{LP}}$ is not satisfied anymore. The tree T splits in a forest F which contains the subtrees of T. Let *Orph* be the set of nodes v such that $\mathsf{p}_v \notin N(v)$, note that $r \notin Orph$. The following predicate is satisfied by every node $v \in V, v \notin Orph$

$$\mathrm{Pr}_v^{\mathcal{LC}} \equiv \begin{cases} \mathsf{level}_{\mathsf{p}_v} + 1 \leq \mathsf{level}_v \wedge \mathsf{NewLevel}_v \geq \mathsf{level}_v & \text{if } v \neq r \\ \mathsf{level}_r = 0 \wedge \mathsf{status}_r = N & \text{otherwise} \end{cases}$$

We show below that each node with no parent in F starts a propagation of information in its subtree.

Lemma 1. *Let a node $v \in V, v \in Orph$. If $\mathsf{status}_v = N$ and $\mathbf{P_{Change}}(v) = false$ then status v eventually moves to P.*

Proof. Let $v \in V, v \in Orph$ be a node such that $\mathsf{status}_v = N$ and $\mathbf{P_{Change}}(v) = false$. v can only execute Rules $\mathsf{R_{LevelCorrect}}$ or $\mathsf{R_{Dynamic}}$, because v can not execute Rules $\mathsf{R_{SafeChangeP}}$, $\mathsf{R_{Level{+}{+}}}$ and $\mathsf{R_{EndPropag}}$ since $\mathsf{status}_v = N$, $\mathbf{P_{Change}}(v) = false$ and $\mathsf{p}_v \notin N(v)$. To change its status from N to P, a node $v \in Orph$ must execute Rule $\mathsf{R_{Dynamic}}$. Suppose that v does not execute Rule $\mathsf{R_{Dynamic}}$. So v can only execute Rule $\mathsf{R_{LevelCorrect}}$. However, after execution of Rule $\mathsf{R_{LevelCorrect}}$ we have $\mathsf{NewLevel}_v := \mathsf{level}_v$ and the guard of Rule $\mathsf{R_{LevelCorrect}}$ is no more satisfied. Thus, only the guard of Rule $\mathsf{R_{Dynamic}}$ is satisfied and v remains enabled until it performs Rule $\mathsf{R_{Dynamic}}$. Therefore, the scheduler eventually selects v to perform Rule $\mathsf{R_{Dynamic}}$. □

According to Lemma 9 in [23], a node v such that $\mathsf{status}_v = P$ eventually performs Rule $\mathsf{R_{EndPropag}}$ to change its status to N. In the following, we show that a node in *Orph* (i.e., without a parent in its neighborhood) eventually leaves the set *Orph*.

Lemma 2. *Let $v \in V, v \in Orph$. Eventually, v is not anymore in the set Orph and selects a parent without creating a cycle.*

Proof. We show the lemma by induction on the height of the subtree of v. Consider the case where a node $v \in Orph$ has a neighbor $u \in N(v)$ such that $\mathsf{level}_u < \mathsf{level}_v$. We assume that for every node x in F, $x \notin Orph$, we have

$\mathsf{level}_{p_x} + 1 \leq \mathsf{level}_x$. So, u can not be a descendant of v. Thus, v performs Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ to choose u as its parent without creating any cycle in F. Otherwise, every node $u \in N(v)$ is a child of v. According to Lemma 9 in [23] and Lemma 1 (above), the level of every node in the subtree of v increases. Since we assume the network is always connected, there exists a leaf node x in the subtree of v such that $\mathsf{level}_x > \overline{\mathsf{level}}_x = \mathsf{level}_y$, with $y \in N(x)$. Thus, x can execute Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ to choose y as its parent and x leaves the subtree of v. Since the height of the subtree of v is finite, eventually v can choose a neighbor u as its parent because u is no more in the subtree of v. Therefore, in a finite time a node $v \in \mathit{Orph}$ leaves the set Orph by selecting a parent in its neighborhood without creating a cycle. □

According to Lemma 2, each node has a parent and no cycle is created. Thus, the system reaches a configuration where a spanning tree is constructed. So the analysis given in [23] can be used to show that the system reaches a configuration in which for each node $v \in V$ we have $\mathsf{level}_v = \overline{\mathsf{level}}_v$. Since the initial configuration contains a spanning tree, the algorithm stabilizes to a breadth first search tree and during the stabilization of the algorithm the loop-free property is maintained, as showed in [23].

Above we consider only the failure of nodes/edges of the tree, now we discuss the addition of nodes and edges in the network. In a legitimate configuration, after the addition of an edge every node $v \in V$ always satisfies $\mathsf{level}_v \geq \overline{\mathsf{level}}_v$. According to Lemma 12 and Corollary 1 in [23], in a finite time eventually for every node $v \in V$ we have $\mathsf{level}_v = \overline{\mathsf{level}}_v$. In a legitimate configuration, after the addition of a node v Rule $\mathsf{R}_{\mathsf{SafeChangeP}}$ is executed by v to select a neighbor $u \in N(v)$ as its parent, there exists such a node u because we assume that the network is always connected. Therefore, the system is in an arbitrary configuration where a spanning tree is constructed. Therefore, the analysis given in [23] can be used to show that in a finite time for every node $v \in V$ we have $\mathsf{level}_v = \overline{\mathsf{level}}_v$. Moreover, in the case of node/edge additions if the initial configuration contains a spanning tree, thus the loop-free property is maintained by the algorithm.

In the following, we prove that the presented algorithm has a superstabilizing property for a particular class of topology change events. We show that a passage predicate is satisfied during the restabilizing execution of our algorithm. We define the considered topology change events, noted ε:

– an addition (resp. a removal) of an edge (u, v) in the network noted $\mathtt{recov}_{uv}$ (resp. $\mathtt{crash}_{uv}$);
– an addition (resp. a removal) of a neighbor node u of v in the network noted $\mathtt{recov}_u$ (resp. $\mathtt{crash}_u$).

In the sequel, we suppose that after every topology change event the network remains connected. We provide below definitions of the topology change events class Λ and passage predicate.

Definition 5 (Class Λ of topology change events). $\mathtt{crash}_{uv}$ *and* $\mathtt{crash}_v$ *compose the class Λ of topology change events.*

Definition 6 (Passage predicate). *Let $T = (V, E_T)$ be the constructed spanning tree in a legitimate configuration C. After the removal of a node $x \in V$ or an edge $e = (y, x) \in E_T$ (y is the parent of x), the parent of each node $v \in V$ is not modified iff v does not belong to the subtree rooted at x in T.*

Lemma 3. *The proposed protocol is superstabilizing for the class Λ of topology change events, and the passage predicate (Definition 6) continues to be satisfied while a legitimate configuration is reached.*

Proof. Consider a legitimate configuration Δ. Suppose ε is a removal of edge (u, v) from the network. If (u, v) is not a tree edge then the levels of u and v are not modified and neither u nor v changes its parent, thus no parent variable is modified. Otherwise, let $\mathsf{p}_v = u$, u's level and u's parent are not modified, it is true for any other node x not contained in the subtree of v since the distance between x and the root r in the graph is not modified (i.e., Predicate $\mathbf{P_{Change}}(x)$ is not satisfied). However, u is no more a neighbor of v so according to Lemma 1 v executes Rule $\mathsf{R_{Dynamic}}$ and starts a propagation phase. Moreover, according to Lemma 2 v selects a new parent without creating a cycle. Therefore, only a node in the subtree connected by the edge (u, v) may change its parent.

Suppose ε is a removal of node u from the network. Any node x not contained in the subtree of u do not change its parent relation because the distance between x and the root node r is not modified (i.e., Predicate $\mathbf{P_{Change}}(x)$ is not satisfied). Consider each edge (u, v) between u and its child v, we can apply the same argument described above for an edge removal. So only any node contained in the subtree connected by u may change its parent. □

3.3 Complexity Analysis

In the following we focus on the complexity analysis of our algorithm in both static and dynamic networks. Note that the original algorithm proposed in [16] had no complexity analysis. Interestingly, we prove that our extension has a zero time extra-cost with respect to the original solution.

Lemma 4. *Starting from an arbitrary configuration, in at most $O(n^2)$ rounds a breadth first search tree is constructed by the algorithm in a static network under a distributed scheduler assuming weak fairness.*

Proof. To construct a spanning tree, the algorithm must remove all the cycles present in the starting configuration. So, we first analyze the number of rounds needed to remove a cycle.

To remove a cycle, a node of the cycle must change its parent to select a node out of the cycle, such a node is named a *break node*. A node can change its parent using Rule $\mathsf{R_{SafeChangeP}}$, but a break node executes Rule $\mathsf{R_{SafeChangeP}}$ if the level of the new parent (out of the cycle) is lower than the level of the break node. Consider a break node x and the neighbor y of x which must be selected as the new parent of x. We note L_x and L_y the level of x and y respectively. To select y as its new parent and to break the cycle, x must have its level L_x such that $L_y < L_x$. In the cycle, a node corrects its level according to its parent by initiating a propagation of information with Rules $\mathsf{R}_{\mathsf{Level}++}$ and $\mathsf{R}_{\mathsf{EndPropag}}$. Thus the number of increments until we have $L_y < L_x$ is equal to $\lceil \frac{(L_x+1)-L_y}{|C|} \rceil$, with $|C|$ the size of the cycle C to break. The propagation of information is in order of the size of C. Thus, $O((L_x + 1) - L_y)$ rounds are needed to have $L_y < L_x$. Since we want to construct a breadth first search tree the level of a node cannot exceed n, with n the size of the network. Thus, we consider that the level of a node is encoded using $\log n$ bits. The biggest value for $(L_x + 1) - L_y$ is obtained when $L_y = 1$ and therefore we have $(L_x + 1) - L_y \leq n$.

Since the maximum number of possible cycles of a network is no more than $n/2$, obtained with cycles of size 2, we have that in $O(n^2)$ all cycles are removed in the network and a spanning tree is constructed. In at most $O(D)$ additional rounds a breadth first tree is constructed, with D the diameter of the network. Indeed, no cycle is created by the algorithm until reaching a legitimate configuration, since the algorithm guarantee the loop-free property. □

Lemma 5. *Starting from a legitimate configuration followed by a topology change event, in at most $O(n^2)$ rounds a breadth first search tree is constructed by the algorithm under a distributed scheduler assuming weak fairness.*

Note that in [24] the authors proposed a BFS super-stabilizing algorithm with a better complexity time. However, their solution has not the loop-free property.

4 Super-Stabilizing Loop-Free Transformation Scheme

Our objective is to design a generic scheme for the construction of a spanning trees considering any metric (not only metrics based on distances in the graph) with loop-free and super-stabilizing properties. The idea is to extend an existing self-stabilizing spanning tree optimized for a given metric (e.g. MST, minimum degree spanning tree, max-flow tree etc) with super-stabilizing and loop-free properties via the composition with a spanning tree construction that already satisfies these properties. Assume $\mathcal{M}$ be the predicate that captures the properties of the metric to be optimized. Consider $\mathcal{A}$ the algorithm that outputs a self-stabilizing spanning tree and verifies $\mathcal{M}$. That is, given a graph, $\mathcal{A}$ computes the set of edges $\mathcal{S}_{\mathcal{A}}$ that satisfies $\mathcal{M}$ and is a spanning tree. Consider Algorithm $\mathcal{B}$ an algorithm that outputs a super-stabilizing and loop-free spanning tree $\mathcal{S}_{\mathcal{B}}$. Ideally, if all edges in $\mathcal{S}_{\mathcal{A}}$ are included in $\mathcal{S}_{\mathcal{B}}$ then there is no need for further transformations. However, in most of the cases the two trees are not identical. Therefore, the idea of our methodology is very simple. Algorithms $\mathcal{A}$ and $\mathcal{B}$ run such that the output of $\mathcal{A}$ defines the graph input for $\mathcal{B}$. That is, the neighborhood relation used by $\mathcal{B}$ is the initial graph filtered by $\mathcal{A}$ to satisfy the predicate $\mathcal{M}$. The principal of this composition is already known in the literature as fair composition [25]. In our case the "slave" protocol is protocol $\mathcal{A}$ that outputs the set of edges input for the "master" protocol $\mathcal{B}$.

The following lemma, direct consequence of the results proven in [25], guaranties the correctness of the composition.

Lemma 6. *Let $\mathcal{M}$ be the predicate that captures the properties of the metric to be optimized. Let $\mathcal{A}$ be an algorithm that outputs a self-stabilizing spanning tree that satisfies $\mathcal{M}$, $\mathcal{S}_{\mathcal{A}}$. Let $\mathcal{B}$ be a loop-free algorithm that computes a spanning tree on the topology defined by $\mathcal{S}_{\mathcal{A}}$ and super-stabilizing for a class of topology changes Λ. The fair composition of $\mathcal{A}$ and $\mathcal{B}$ is an algorithm that outputs a loop-free spanning tree that satisfies $\mathcal{M}$ and is super-stabilizing for Λ.*

Note that our super-stabilizing loop-free BFS can be used as Algorithm $\mathcal{B}$ in the above composition. The interesting property of the composition is that the time complexity will be the sum between $O(n^2)$ and the complexity time of the candidate to be transformed. Note that so far, the best time complexity of a spanning tree optimized for a given metric is $O(n^2)$ which leads to the conclusion that the composition does not alterate the time complexity of the candidate.

In the following, we specify the predicate $\mathcal{M}$ for two well known problems: max-flow trees and minimum degree spanning trees.

Case study 1: Maximum-flow tree The problem of constructing a maximum-flow tree from a given root node r can be stated as follows. Given a weighted undirected graph $G = (V, E, w)$, the goal is to construct a spanning tree $T = (V, E_T)$ rooted at r, such that r has a flow equal to ∞ and for every node $v \in V$ the path between r and v has the maximum flow. Formally, let $fw(v) = \min(fw(\mathsf{p}_v), w(\mathsf{p}_v, v))$ the flow for every node $v \in V$ $(v \neq r)$ in tree $\mathcal{T}$ and mfw_v the maximum flow value of v among all spanning trees of G rooted at r. The maximum-flow tree problem is to compute a spanning tree T, such that $\forall v \in V, v \neq r, fw(v) = mfw_v$. The max flow tree problem has been studied *e.g.* in [21]. In this case, the graph $G_{\mathcal{S}_{\mathcal{A}}} = (V_{\mathcal{S}_{\mathcal{A}}}, \mathcal{S}_{\mathcal{A}})$ for the maximum-flow tree problem must satisfies the following predicate:

$$\mathcal{M} \equiv (|\mathcal{S}_{\mathcal{A}}| = n - 1) \wedge (V = V_{\mathcal{S}_{\mathcal{A}}}) \wedge$$

$$(\forall v \in V, fw(v) = \max\{\min(fw(u), w(u, v)) : u \in N(v)\}).$$

Case study 2: Minimum degree spanning tree Given an undirected graph $G = (V, E)$ with $|V| = n$, the minimum degree spanning tree problem is to construct a spanning tree $T = (V, E_T)$, such that the maximum degree of T is minimum among all spanning trees of G. This is a NP-hard problem. Formally, let $deg_{\mathcal{T}}(v)$ the degree of node $v \in V$ in the subgraph $\mathcal{T}$ and $deg(\mathcal{T})$ the maximum degree of subgraph $\mathcal{T}$ (i.e., $deg(\mathcal{T}) = \max\{deg_{\mathcal{T}}(v) : v \in V\}$). The minimum spanning tree problem is to compute a spanning tree T, such that $deg(T) = \min\{deg(T') : T'$ is a spanning tree of $G\}$. A self-stabilizing approximated solution for this problem has been proposed in [26]. If this solution plays the slave role in our transformation scheme then the graph $G_{\mathcal{S}_{\mathcal{A}}} = (V_{\mathcal{S}_{\mathcal{A}}}, \mathcal{S}_{\mathcal{A}})$ input for the BFS algorithm satisfies the following predicate:

$$\mathcal{M} \equiv (|\mathcal{S}_{\mathcal{A}}| = n - 1) \wedge (V = V_{\mathcal{S}_{\mathcal{A}}}) \wedge$$

$$deg(G_{\mathcal{S}_{\mathcal{A}}}) \leq \min\{deg(T') : T' \text{ a spanning tree of } G\} + 1.$$

5 Concluding Remarks

We presented a scheme for constructing loop-free and super-stabilizing protocol for universal tree metrics, without significant impact on the performance. There are several open questions raised by our work:

1. Decoupling various added properties (such as loop-freedom or super-stabilization) seems desirable. As a particular network setting may not need both properties or temporarily run in conditions where the network is essentially static, some costs could be saved by removing some properties. Of course, stripping our scheme can trivially result in a generic loop-free transformer or to a generic super-stabilizing transformer. Yet, modular design of features, as well as further enhancements (such as safe convergence [27,28]), seems an interesting path for future research.
2. The implementation of self-stabilizing protocols recently was helped by compilers that take as input guarded commands and provide as output actual source code for existing devices [29]. Transformers such as this one would typically benefit programmers' toolboxes as they ease the reasoning by keeping the source code intricacies at a very high level. Actual implementation of our transformer into a programmer's toolbox is a challenging ingeneering task.

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
2. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
3. Tixeuil, S.: Self-stabilizing Algorithms, pp. 26.1–26.45 (November)
4. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance (preliminary version). In: PODC, pp. 195–206 (1993)
5. 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)
6. Dolev, S., Welch, J.L.: Wait-free clock synchronization. Algorithmica 18(4), 486–511 (1997)
7. Papatriantafilou, M., Tsigas, P.: On self-stabilizing wait-free clock synchronization. Parallel Processing Letters 7(3), 321–328 (1997)
8. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. J. ACM 51(5), 780–799 (2004)
9. Ben-Or, M., Dolev, D., Hoch, E.N.: Fast self-stabilizing byzantine tolerant digital clock synchronization. In: PODC, pp. 385–394 (2008)
10. Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 440–453. Springer, Heidelberg (2006)
11. Masuzawa, T., Tixeuil, S.: Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. International Journal of Principles and Applications of Information Science and Technology (PAIST) 1(1), 1–13 (2007)

12. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. 1997 (1997)
13. Herman, T.: Superstabilizing mutual exclusion. Distributed Computing 13(1), 1–17 (2000)
14. Katayama, Y., Ueda, E., Fujiwara, H., Masuzawa, T.: A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. J. Parallel Distrib. Comput. 62(5), 865–884 (2002)
15. Cobb, J.A., Gouda, M.G.: Stabilization of general loop-free routing. J. Parallel Distrib. Comput. 62(5), 922–944 (2002)
16. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Self-Stabilizing Systems, pp. 184–198 (2003)
17. Garcia-Luna-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Trans. Netw. 1(1), 130–141 (1993)
18. Gafni, E.M., Bertsekas, P.: Distributed algorithms for generating loop-free routes in networks with frequently changing topology. IEEE Transactions on Communications 29, 11–18 (1981)
19. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci.1997 (1997)
20. Blin, L., Potop-Butucaru, M., Rovedakis, S., Tixeuil, S.: A new self-stabilizing minimum spanning tree construction with loop-free property. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 407–422. Springer, Heidelberg (2009)
21. Gouda, M.G., Schneider, M.: Stabilization of maximal metric trees. In: WSS, pp. 10–17 (1999)
22. Gupta, S.K.S., Srimani, P.K.: Self-stabilizing multicast protocols for ad hoc networks. J. Parallel Distrib. Comput. 63(1), 87–96 (2003)
23. Johnen, C., Tixeuil, S.: Route preserving stabilization. Technical Report 1353, LRI, Université Paris-Sud XI (2003)
24. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. 1997 (1997)
25. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing 7(1), 3–16 (1993)
26. Blin, L., Potop-Butucaru, M.G., Rovedakis, S.: Self-stabilizing minimum-degree spanning tree within one from the optimal degree. In: IPDPS, pp. 1–11 (2009)
27. Kakugawa, H., Masuzawa, T.: A self-stabilizing minimal dominating set algorithm with safe convergence. In: IPDPS (2006)
28. Kamei, S., Kakugawa, H.: A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 496–511. Springer, Heidelberg (2008)
29. Dalton, A.R., McCartney, W.P., Dastidar, K.G., Hallstrom, J.O., Sridhar, N., Herman, T., Leal, W., Arora, A., Gouda, M.G.: Desal alpha: An implementation of the dynamic embedded sensor-actuator language. In: ICCCN, pp. 541–547 (2008)

A New Technique for Proving Self-stabilizing under the Distributed Scheduler[⋆]

Sven Köhler and Volker Turau

Institute of Telematics
Hamburg University of Technology
Hamburg, Germany
{sven.koehler,turau}@tu-harburg.de

Abstract. Proving stabilization of a complex algorithm under the distributed scheduler is a non-trivial task. This paper introduces a new method which allows to extend proofs for the central scheduler to the distributed scheduler. The practicability of the method is shown by applying it to two existing algorithms, for which stabilization under the distributed scheduler was an open problem.

1 Introduction

The notion of self-stabilization was coined by E. W. Dijkstra [2]. Self-stabilizing distributed systems are guaranteed to converge to a desired state or behaviour in finite time, regardless of the initial state. Convergence is also guaranteed after the system is affected by transient faults, no matter their scale or nature. This makes self-stabilization an elegant and formal approach for non-masking fault-tolerance.

Self-stabilizing algorithms can be designed with different schedulers in mind. Possible schedulers include the central scheduler (only one node can make a move in each step), the distributed scheduler (any number of nodes make a move in each step), and the synchronous scheduler (all nodes make a move in each step). In an early manuscript [3] (written 1973), Dijkstra discusses his choice of the scheduler. Unsure, whether non-trivial algorithms for the distributed scheduler exist, he decides to design his self-stabilizing algorithms for a "rather powerful daemon, that may be awkward to implement": the central scheduler. He points out that his choice avoids difficulties like oscillation which may easily occur when using the distributed scheduler.

The model of the central scheduler provides rather strong assumptions. They make it easy to develop and prove correctness of self-stabilizing algorithms. On the other hand, the distributed scheduler is the more realistic model. Even though some algorithms that are designed for the central scheduler also stabilize under the distributed scheduler, the majority of algorithms does not have this property. In these cases new algorithms have to be devised or existing algorithms

⋆ This research was funded by the German Research Foundation (DFG), contract number TU 221/3-1.

have to be extended. For the latter case, generic methods have been invented: transformers, for example Mutual Exclusion [1] and Conflict Managers [6]. It is considered worthwhile (for example in [1]) to design algorithms for the central scheduler and then transform them to the desired model. But these transformations come with a time overhead of at least $\mathcal{O}(\Delta)$.

In case no transformer is applied, the stabilization proofs are usually very problem-specific and do not allow for generalization. Generic proof techniques such as potential functions and convergence stairs, that work well for the central scheduler, are very hard to apply in case of the distributed scheduler. Section 4 provides a generic technique for proving stabilization under the distributed scheduler by extending existing proofs that assume the central scheduler. The contribution of this paper is completed by Section 5 in which the technique is applied to two complex algorithms for which stabilization under the distributed scheduler has not been proven and it has been an open question, whether this is feasible at all. Section 6 gives a case study of a protocol to which the proof technique cannot be applied.

2 Related Work

Common techniques for proving stabilization include variant functions (also called potential functions) and convergence stairs [4]. In principle, all these do apply to the distributed scheduler, but these techniques rely on properties of transitions from one system configuration to another. In case of the distributed scheduler, transitions are hard to analyse due to the large of number of possible selections of simultaneously executed moves.

Two transformers that allow algorithms written for the central scheduler to stabilize under the distributed scheduler are widely known. The first by Beauquier et al. [1] solves the mutual exclusion problem: (a) at least one process is privileged (b) only non-neighboring nodes are privileged (c) any node is privileged infinitely often. Mutual exclusion allows algorithms designed for a fair central scheduler to stabilize under an unfair distributed scheduler. The second by Gradinariu et al. [6] implements the concept of conflict managers: (a) at least one node is privileged (b) only non-conflicting nodes are privileged. Whether two nodes are conflicting is defined by the symmetric relation $\mathcal{R} \subseteq V \times V$ (the so-called conflict relation). If $\mathcal{R}$ is chosen such that neighboring nodes are conflicting, then this allows algorithms designed for the central scheduler to stabilize under the distributed scheduler.

Both transformers are partly based on the idea that the locally central scheduler (any number of non-neighboring nodes is privileged in each step) is virtually identical to the central scheduler, since the behaviour of a node only depends on the information stored in the node's neighborhood. Hence, moves of non-neighboring nodes can make their moves in an arbitrary order, or even simultaneously. The resulting configuration is always the same. It is rather restrictive to demand that any order of moves is equivalent to their simultaneous execution. For a given set of moves, it suffices to show that one particular sequence is equivalent. Section 4 extends this idea to a technique, that allows to prove that such a sequence exists for any set of enabled moves in any configuration. The new proof-technique is directly applied to algorithms themselves, without any transformer.

Proving probabilistic self-stabilization under the distributed scheduler of protocols designed for the central scheduler is surprisingly easy. In the fashion of the scheduler-luck-game [4], one can show that steps in which only non-neighboring nodes simultaneously make a move exist with positive probability, if each move depends on the outcome of an independent random experiment. In conclusion, executions in which this is the case for every step exist with positive probability. This has facilitated the construction of the probabilistic conflict manager by Gradinariu et al. [6] and the transformations by Turau et al. [10] and Herman [7].

3 Model of Computation

A distributed system is represented as an undirected graph (V, E) where V is the set of *nodes* and $E \subseteq V \times V$ is the set of *edges*. Let $n = |V|$ and Δ denote the maximal degree of the graph. If two nodes are connected by an edge, then they are called *neighbors*. The set of neighbors of node v is denoted by $N(v) \subseteq V$ and $N[v] = N(v) \cup \{v\}$. Each node stores a set of variables. The values of all variables constitute the *local state* of a node. The *configuration* of a system is the n-tuple of all local states in the system and Σ denotes the set of all configurations.

Nodes communicate via locally shared memory, that is every node can read the variables of all its neighbors. Nodes are only allowed to modify their own variables. Each node $v \in V$ executes a protocol consisting of a list of rules. Each rule consists of a *guard* and a *statement*. A guard is a Boolean expression over the variables of node v and its neighbors. A rule is called *enabled* if its guard evaluates to true. A node is called *enabled* if one of the rules is enabled.

Execution of the statements is controlled by a scheduler which operates in *steps*. At the beginning of step i, the scheduler first non-deterministically selects a non-empty subset $S_i \subseteq V$ of enabled nodes. Each node in S_i then executes the statement of its enabled rule. It is said, that the nodes make a *move*. A step is finished, if all nodes have completed their moves. Changes made during the moves become visible to other nodes at the end of the step and not earlier (composite atomicity).

An *execution* $\langle c_0, c_1, c_2, \ldots \rangle$, $c_i \in \Sigma$ is a sequence of configurations c_i where c_0 is the *initial configuration* and c_i is the configuration of the system after the i-th step. In other words, if the current configuration is c_i and all nodes in S_{i+1} make a move, then this yields c_{i+1}.

Time is measured in *rounds*. Let x be an execution and $x_0 = x$. Then x is partitioned into rounds by induction on $i = 0, 1, 2, \ldots$: round r_i is defined to be the minimal prefix of x_i, such that each node $v \in V$ has either made a move or has been disabled at least once within r_i. Execution x_{i+1} is obtained by removing prefix r_i from x_i. The intuition is that within a round, each node that is enabled at the beginning of the round, gets the chance to make a move if it has not become disabled by a move of its neighbors.

Let P be a protocol and let $Legit_P$ denote a Boolean predicate over Σ. If $Legit_P(c)$ is true, then configuration c is called *legitimate*. P is said to be *self-stabilizing* with respect to $Legit_P$, if both the following properties are satisfied. *Convergence*: for any execution of P, a legitimate configuration is reached within a finite number of steps. *Closure*: for any execution of P it holds that once a legitimate configuration is reached, all subsequent configurations are also legitimate. A self-stabilizing protocol is *silent* if all nodes are disabled after a finite number of steps.

3.1 Nonstandard Extensions

The standard model as defined above is extended to a *multi-protocol* model. An *algorithm* is denoted by a set $\mathcal{A}$ of protocols. Each node designates a separate set of variables to each $p \in \mathcal{A}$. An *instance* is a tuple (v, p), $v \in V$ and $p \in \mathcal{A}$. $\mathcal{M} = V \times \mathcal{A}$ is the set of instances. Instance (v, p) is called *enabled*, if p is enabled on node v. Node v is called *enabled*, if any $p \in \mathcal{A}$ is enabled on v. Two instances (v_1, p_1) and (v_2, p_2) are called *neighboring*, if $v_1 \in N[v_2]$ or vice versa.

During the i-th step, a scheduler selects a subset $S_i \subseteq \mathcal{M}$ of enabled instances. In case of the central scheduler it holds $|S_i| = 1$. The distributed scheduler may chose any non-empty subset S_i of enabled instances, even containing several instances of the same node but distinct protocols. If node v executes a rule by protocol $p \in \mathcal{A}$, then it is said that v has made the move (v, p). No assumptions on the fairness of the scheduler are made.

An instance (v, p) cannot modify any other variables than the ones designated to protocol p by node v. Read access is permitted to all variables of all $v \in N[v]$, no matter which protocol they belong to. For any pair of instances (v_1, p_1) and (v_2, p_2) that are being selected during a single step, changes made by (v_1, p_1) don't become visible to (v_2, p_2) until the end of the step (composite atomicity), even if $v_1 = v_2$. Due to these constraints, the result of a step does not depend on the order in which the individual moves are executed. This model defines a natural extension of the notion of rounds. A *round* is a prefix of an execution, such that every instance $m \in \mathcal{M}$ has been executed or has been disabled at least once. This model is identical to the standard model, if $|\mathcal{A}| = 1$. How algorithms designed for this model can be transformed to the standard single-protocol model is discussed in Section 5.3.

Without loss of generality, it is assumed that per instance only one guard can be enabled at a time and that all rules are deterministic. How to widen the techniques to a non-deterministic or randomized model is discussed in the concluding remarks. Using the assumption of deterministic protocols, the following notations are defined: $(c : m)$ denotes the configuration after the execution of $m \in \mathcal{M}$ in c. Similarly, $(c : S)$ denotes the configuration after the simultaneous execution of all instances $S \subseteq \mathcal{M}$ in a single step. The execution of a sequence of instances is denoted by $(c : m_1 : m_2 : \ldots : m_x) = ((c : m_1 : m_2 : \ldots : m_{x-1}) : m_x)$, $m_i \in \mathcal{M}$, $x > 1$. These notations are used to describe executions by the central scheduler or the distributed scheduler. Note that $(c : m)$ and $(c : S)$ are undefined, if m is not enabled in c or if S contains instances that are not enabled in c respectively.

Furthermore, let $c|_m$ denote the part of configuration c which reflects the values of all variables dedicated to instance m. The expression $c \vdash e$ denotes the value of expression e in case that the current configuration equals c. Note, that e can be a variable, function or Boolean predicate.

4 Serialization

To proof stabilization under the distributed scheduler, we first define the notion of a serialization. A serialization of a set of enabled instances is a sequence of instances, that can be executed under the central scheduler and yields the same configuration as executing the set of instances during a single step of the distributed scheduler.

Definition 1. *Let c be a configuration and $S \subseteq \mathcal{M}$ be a set of instances enabled in c. A sequence $s = \langle m_1, m_2, \ldots, m_k \rangle$, $m_i \in \mathcal{M}$ is called a **serialization** of S in c if it satisfies*

$$(c : S) = (c : m_1 : m_2 : \ldots : m_k) \tag{1}$$

*S is called **serializable** in c, if a serialization in c exists.*

With respect to S, the serialization contains each instance $m_i \in S$ at least once. The simple reason is, that the serialization is required to modify $c|_{m_i}$, which no instance other than m_i is capable of. Apart from that, instances may occur multiple times within the sequence. Even additional instances that are not in S may be included, but their effect has to be compensated such that Equation (1) holds again in the end.

Observation 1. The sequence $\langle m_1, m_2, \ldots, m_k \rangle$, $m_i \in \mathcal{M}$ is a serialization of $S \subseteq \mathcal{M}$ in c if and only if

$$(c : S)|_{m_i} = (c : m_1 : m_2 : \ldots : m_k)|_{m_i} \qquad \forall i = 1, 2, \ldots, k$$

If $m_1, m_2, \ldots, m_k$ are distinct, then this is equivalent to

$$(c : S)|_{m_i} = (c : m_1 : \ldots : m_i)|_{m_i} \qquad \forall i = 1, 2, \ldots, k$$

Furthermore, it is clear that $(c : S)|_{m_i} = (c : m_i)|_{m_i}$.

The rest of this section lays the groundwork for a technique that facilitates the construction of serializations. First, the notion of a ranking is defined. It assigns a natural number (the rank) to each enabled instance. The rank describes the behaviour of an instance (i.e. how it changes the variables) depending on the current configuration. The goal is to obtain a serialization by sorting instances by their rank.

Definition 2. *The mapping $r : \mathcal{M} \to \mathbb{N} \cup \{\perp\}$ is called a* ***ranking****, if the following conditions hold for any configuration:*

$$r(m) = \perp \text{ if instance } m \text{ is disabled}$$
$$r(m) \in \mathbb{N} \text{ otherwise}$$

For a given ranking it remains to show that sorting a set of instances by their rank actually yields a serialization. As a step towards this, an invariancy relation on instance/rank-tuples is defined. In general, this relation is not symmetric.

Definition 3. *Let r denote a ranking. A tuple $(m_2, r_2) \in \mathcal{M} \times \mathbb{N}$ is called* ***invariant*** *under the tuple $(m_1, r_1) \in \mathcal{M} \times \mathbb{N}$, if the following two conditions hold for all $c \in \Sigma$ that satisfy $r_1 = c \vdash r(m_1)$ and $r_2 = c \vdash r(m_2)$:*

$$(c : m_1) \vdash r(m_2) = c \vdash r(m_2)$$
$$(c : m_1 : m_2)|_{m_2} = (c : m_2)|_{m_2}$$

If the tuple (m_2, r_2) is invariant under (m_1, r_1), then the rank of m_2 as well as the result of the execution of m_2 remains the same, no matter whether m_1 has been executed prior to m_2, or not. Note, that this invariancy holds for all configurations, in which m_2 and m_1 have the given ranks r_2 and r_1 respectively. The following proposition illustrates, why this is useful.

Proposition 1. *Let c be a configuration, r a ranking, and m_1, m_2, m_3 three distinct enabled instances. If $(m_2, c \vdash r(m_2))$ is invariant under $(m_1, c \vdash r(m_1))$ and $(m_3, c \vdash r(m_3))$ is invariant under both $(m_1, c \vdash r(m_1))$ and $(m_2, c \vdash r(m_2))$, then $\langle m_1, m_2, m_3 \rangle$ is a serialization of $\{m_1, m_2, m_3\}$ in c.*

Proof. By Observation 1 it suffices to prove the following three equations:

$$(c : m_1)|_{m_1} = (c : m_1)|_{m_1} \tag{2}$$
$$(c : m_2)|_{m_2} = (c : m_1 : m_2)|_{m_2} \tag{3}$$
$$(c : m_3)|_{m_3} = (c : m_1 : m_2 : m_3)|_{m_3} \tag{4}$$

Equation (2) is clear. Equation (3) holds, because $(m_2, c \vdash r(m_2))$ is invariant under $(m_1, c \vdash r(m_1))$. In order to understand the validity of Equation (4) it is necessary to take a closer look at the sequential execution of m_1, m_2, and m_3. Consider the intermediate configuration $c' = (c : m_1)$. Because $(m_3, c \vdash r(m_3))$ is invariant under $(m_1, c \vdash r(m_1))$, it holds that $(c' : m_3)|_{m_3} = (c : m_3)|_{m_3}$. In order for Equation (4) to be satisfied, it must be the case that $(c' : m_2 : m_3)$ equals $(c' : m_3)$. This is true, if $(m_3, c' \vdash r(m_3))$ is invariant under $(m_2, c' \vdash r(m_2))$. This becomes clear, if one considers that $c' \vdash r(m_3) = c \vdash r(m_3)$ as well as $c' \vdash r(m_2) = c \vdash r(m_2)$ and that the invariancy of m_3 under m_2 holds no matter whether the current configuration is c or c'. □

Definition 4. *A ranking r is called an* ***invariancy-ranking****, if*

$$r_2 \geq r_1 \Rightarrow (m_2, r_2) \text{ is invariant under } (m_1, r_1)$$

holds with respect to r for all $m_2 \neq m_1$, $m_2, m_1 \in \mathcal{M}$, $r_2, r_1 \in \mathbb{N}$.

Theorem 1. *For an algorithm with an invariancy-ranking, every set of enabled instances is serializable in any configuration.*

Proof. Let r be an invariancy-ranking, c a configuration, $S \subseteq \mathcal{M}$ a set of instances enabled in c, and $s = \langle m_1, m_2, \ldots, m_k \rangle$ a sequence of all instances of S sorted in ascending order by their rank with respect to c. Denote by c_x the configuration $(c : m_1 : m_2 : \ldots : m_x)$ and $c_0 = c$. By Observation 1 it suffices to prove $c_j|_{m_j} = (c_0 : m_j)|_{m_j}$ for $j = 1, 2, \ldots, k$.

In the following, it is shown by induction on i that $(c_{i-1} : m_j)|_{m_j} = (c_0 : m_j)|_{m_j}$ and $c_{i-1} \vdash r(m_j) = c_0 \vdash r(m_j)$ hold for all $i = 1, 2, \ldots, k$ and $j = i, i+1, \ldots, k$. This is obviously true for $i = 1$. Assume the following for $i < k$:

$$\begin{aligned} c_{i-1} \vdash r(m_j) &= c_0 \vdash r(m_j) && \forall j = i, i+1, \ldots, k \\ (c_{i-1} : m_j)|_{m_j} &= (c_o : m_j)|_{m_j} && \forall j = i, i+1, \ldots, k \end{aligned}$$

By assumption, $(m_j, c_{i-1} \vdash r(m_j))$ is invariant under $(m_i, c_{i-1} \vdash r(m_i))$ for all $j = i+1, i+2, \ldots, k$. Hence, the following is satisfied in c_i:

$$\begin{aligned} c_i \vdash r(m_j) = c_{i-1} \vdash r(m_j) &= c_0 \vdash r(m_j) && \forall j = i+1, i+2, \ldots, k \\ (c_i : m_j)|_{m_j} = (c_{i-1} : m_j)|_{m_j} &= (c_0 : m_j)|_{m_j} && \forall j = i+1, i+2, \ldots, k \end{aligned}$$

In particular, it follows that $c_j|_{m_j} = (c_{j-1} : m_j)|_{m_j} = (c_0 : m_j)|_{m_j}$ for all $j = 1, 2, \ldots, k$. □

Corollary 1. *For any execution e under the distributed scheduler of an algorithm with an invariancy-ranking, there exists an execution e' under the central scheduler such that e is a subsequence of e'.*

Theorem 2. *For an algorithm with an invariancy-ranking, move- and round-complexity under the central scheduler are upper bounds for the move- and round-complexity under the distributed scheduler.*

Proof. Omitted due to space restrictions. □

5 Practicability

In the following, the new proof technique is applied to two algorithms, one from [5] and the other from [8]. Both algorithms are transformers that add fault-containing properties to any silent self-stabilizing protocol P. Prior to this paper, it has been an open problem whether these algorithms work under the distributed scheduler.

The overall procedure to first design a ranking of which it is shown that it is an invariancy-ranking. This is done by inspecting all pairs (r_2, r_1) of ranks that satisfy $r_2 \geq r_1$. For each pair, it is shown that all (m_2, r_2) are invariant under any (m_1, r_1). More m_2 is called *invariant* under m_1, if (m_2, r_2) is invariant under (m_1, r_1) for all ranks $r_2, r_1 \in \mathbb{N}$. The following observation justifies, that the proofs only consider the case that m_2 and m_1 are neighboring.

Observation 2. Let r be a ranking, and let m_2 and m_1 be two non-neighboring moves. If $r(m_2)$ solely depends on variables in the neighborhood of m_2, then m_2 is invariant under m_1.

5.1 Algorithm $\mathcal{A}_1$

First, algorithm $\mathcal{A}_1 = \{Q\}$ by Ghosh et al. [5] is analysed. The algorithm is actually a transformer, that is protocol Q internally calls a given silent self-stabilizing protocol P and extends P by fault-containment properties. The goal of fault-containment is to minimize the time that a protocol needs to recover from small scale faults. This property is combined with self-stabilization which guarantees recovery from large scale faults. Protocol Q basically is a silent phase clock, with a limited clock range of $[0, M]$. During stabilization, the timestamps are decremented towards 0 in an evenly fashion until every node has reached 0. Along with this decrementation, three protocols are executed: C, P, and B. Protocol C is executed for upper range timestamps and repairs corrupted P-variables using backups stored on each neighbor. It consists of three phases that are executed by Q in a synchronous fashion. For mid-range timestamps, P is executed by Q and is given enough time to stabilize before B, which is executed for lower range timestamps, creates the backups. Timestamps are globally reset to M if an inconsistency or a fault is detected. For details refer to [5].

Protocol Q is implemented in form of two rules. Each node $v \in V$ stores its timestamp in the variable $v.t$. Rule S_1 resets $v.t$ to M, if $PorC\_inconsistent(v)$ or $raise(v)$ is satisfied.

$$\begin{aligned}
&raise(v) \equiv raise_1(v) \vee raise_2(v)\\
&raise_1(v) \equiv v.t \neq M \wedge \exists u \in N(v) : v.t - u.t > 1 \wedge u.t < M - n\\
&raise_2(v) \equiv v.t < M - n \wedge \exists u \in N(v) : u.t = M\\
&PorC\_inconsistent(v) \equiv v.t = 0 \wedge (\forall u \in N(v) : u.t = 0)\\
&\qquad\qquad\qquad\qquad\qquad \wedge (G_P(v) \vee \neg Legit_C(v))
\end{aligned}$$

The predicate $G_P(v)$ is true if and only if P is enabled on v. The predicate $Legit_C(v)$ is true if and only if any backup differs from the current values of P-variables. If $decrement(v)$ is true, rule S_2 first executes a move of C if $v.t \in [M-2, M]$, a move of P if $v.t \in [3, M - \max\{n, 3\}]$, or a move of B if $v.t = 2$ and then decrements $v.t$ by one. The local state of protocol P for node v is held in the variable $v.x$ which is called *primary state*.

$$\begin{aligned}
&decrement(v) \equiv decrement_1(v) \vee decrement_2(v)\\
&decrement_1(v) \equiv v.t > 0 \wedge \forall u \in N(v) : 0 \leq v.t - u.t \leq 1\\
&decrement_2(v) \equiv \forall u \in N(v) : v.t \geq u.t \wedge u.t \geq M - n
\end{aligned}$$

To obtain serializations, the following ranking is used:

$$r(v,p) := \begin{cases} 0 & \text{if } decrement(v)\\ M - v.t & \text{if } raise(v) \vee PorC\_inconsistent(v)\\ \perp & \text{otherwise} \end{cases}$$

Note, that $raise(v) \vee PorC\_inconsistent(v)$ implies $v.t < M$ and thereby $r(v,Q) > 0$. So all instances of rank 0 decrement $v.t$ and all others reset it to M.

The ranking r is designed in such a way that decrementations occur first within a serialization. Their order is not significant since a decrement move does not disable any neighboring instances. This is due to the pseudo-consistence criteria as defined in [5]. Next, all raise moves occur within the serialization, sorted by their timestamp in descending order. The following example illustrates why this is necessary: Consider a node v which is surrounded by nodes with a timestamp equal to 0 while $v.t = M - n$. In this configuration, node v is enabled by $raise_1(v)$. It is possible, that all neighbors of v are enabled by $raise_2(v)$. If the neighbors of v make a raise move before v does, then $v.t$ becomes pseudo-consistent, and hence v becomes disabled.

Observation 3. $decrement_1(v)$ as well as $decrement_2(v)$ imply $v.t \geq u.t$ for all $u \in N(v)$. So if $decrement(v)$ and $decrement(u)$ hold for two neighboring nodes v and u, then $v.t = u.t$ is true.

Lemma 1. *Assume that there exists an invariancy-ranking for each of the protocols P, C and B. Spreading rank* 0 *of r according to these invariancy-rankings (and shifting the higher ranks accordingly) yields an invariancy-ranking for algorithm $\mathcal{A}_1$.*

Proof. In the following, m_2 and m_1 denote moves by nodes v_2 and v_1 respectively. c denotes a configuration such that $r_2 = c \vdash r(m_2)$ and $r_1 = c \vdash r(m_1)$. Furthermore, c' denotes the configuration $(c : m_1)$. Instances m_2 and m_1 are assumed to be neighboring. This is justified by Observation 2.

Case a) $r_2 = r_1 = 0$: The assumption yields $c \vdash decrement(v_2)$ and $c \vdash decrement(v_1)$. From that and Observation 3 it follows that $c \vdash v_1.t = v_2.t$ and $c' \vdash v_1.t = v_2.t - 1$. If $c \vdash decrement_1(v_2)$, then $c' \vdash decrement_1(v_2)$. Otherwise $c \vdash (decrement_2(v_2) \wedge \neg decrement_1(v_2))$ from which $c \vdash v_2.t > M - n$ follows and thereby $c' \vdash v_1.t \geq M - n$ and $c' \vdash decrement_2(v_2)$.

Case b) $1 \leq r_2 \leq M \wedge r_1 = 0$: Because of $r_1 = 0$, $c \vdash decrement(v_1)$ and thus $c \vdash v_1.t > 0$ must hold. $c \vdash PorC\_inconsistent(v_2)$ cannot be satisfied, since it requires $c \vdash v_1.t = 0$. If $c \vdash raise_1(v_2)$, then there exists some node $w \in N(v_2)$ with $c \vdash (v_2.t - w.t > 1 \wedge w.t < M - n)$. $c \vdash w.t < N - n$ implies $c \vdash \neg decrement_2(w)$ and $c \vdash w.t - v_2.t < -1$ implies $c \vdash \neg decrement_1(w)$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash raise_1(v_2)$. If $c \vdash raise_2(v_2)$, then $c \vdash v_2.t < M - n$ and there exists some node $w \in N(v_2)$ with $c \vdash w.t = M$. $c \vdash w.t - v_2.t > 1$ implies $c \vdash \neg decrement_1(w)$ and $v_2.t < M - n$ implies $v \vdash \neg decrement_2(w)$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash raise_2(v_2)$.

Case c) $1 \leq r_2 \leq M \wedge 1 \leq r_1 \leq r_2$: If $c \vdash v_2.t < M - n$, then $c' \vdash raise_2(v_2)$ since $c' \vdash v_1.t = M$. Otherwise $c \vdash v_2.t \geq M - n$ which implies $c \vdash \neg raise_2(v_2)$ and $c \vdash \neg PorC\_inconsistent(v_2)$. and thus $c \vdash raise_1(v_2)$. Hence there exists some node $w \in N(v_2)$ with $c \vdash (w.t < v_2.t \wedge w.t < M - n)$. From $r_1 \leq r_2$ it follows that $c \vdash v_1.t \geq v_2.t$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash raise_1(v_2)$.

In all cases $c' \vdash r(m_2) = c \vdash r(m_2)$. Furthermore, it is assumed that the moves of rank 0 are sorted in such a way that the order of m_2 and m_1 matches a serialization of either protocol P, C or B, depending on $v_1.t$ and $v_2.t$ which are equal by Observation 3. Hence $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$ in all cases. □

Theorem 3. *If there exists an invariancy-ranking for each of the protocols P, C and B, then for any execution e of $\mathcal{A}_1$ under the distributed scheduler there exists an execution e' under the central scheduler such that e is a subsequence of e'.*

5.2 Algorithm $\mathcal{A}_2$

In this section, algorithm $\mathcal{A}_2 = \{Q, R_1, R_2, \dots, R_\Delta\}$ by Köhler et al. [8] is discussed. It implements a different approach to add fault-containment to any given silent self-stabilizing protocol P. It offers several improvements over algorithm $\mathcal{A}_1$: a constant fault-gap (that is the minimal time between two containable faults), strictly local fault repair without any global effects, and a stabilization time similar to the original protocol P (besides a constant slow-down factor).

Every node $v \in V$ designates one of the Δ protocols R_i to each of its neighbors. For notational convenience, the algorithm is defined to be a mapping $\mathcal{A}_2 : v \mapsto \{Q\} \cup \{R_u \mid u \in N(v)\}$ that assigns a set of protocols to each $v \in V$. In spite of this notation, algorithm $\mathcal{A}_2$ is still uniform. The behaviour of the algorithm is best described based on the notion of a *cell*. For $v \in V$, cell v consists of the instance (v, Q) and the instances (u, R_v), $u \in N(v)$. Cells execute cycles of a simple finite state-machine. During a cycle, cells first repair corrupted variables. The cycle is always guaranteed to start with the repair, even after a fault. Cells then check, whether there are any corruptions in their neighboring cells and if so, they wait for them to become repaired. Only after that, a single move of P is executed and as a final step backups of the variables of protocol P are created. To achieve this behaviour, there is a constant *dialog* between (v, Q) and all instances of R_v. For details refer to [8].

The instance (v, Q) maintains three variables: $v.s, v.q \in \mathbb{Z}_4$ and $v.p$, which is also called *primary state* and stores the local state of protocol P for node v. Each of the variables $v.s$ and $v.q$ can assume one of the four states $0 = \text{Paused}$, $1 = \text{Repaired}$, $2 = \text{Executed}$, and $3 = \text{Copied}$. If $v.s \neq v.q$, then $(v.s, v.q)$ is called a *query* for a transition from state $v.s$ to $v.q$. An instance (u, R_v) maintains the following variables: $u.r_v \in \mathbb{Z}_4$, $u.d_v \in \mathbb{Z}_3$, and $u.c_v$. One of the four state values is assigned $u.r_v$. The so-called *decision-variable* $u.d_v$ assumes one of the values Keep, Update, and Single. The *copy-variable* $u.c_v$ is used for storing backups of $v.p$.

Protocol Q consists of three rules. If $\neg \mathit{dialogConsistent}(v)$, then instance (v, Q) resets $v.s$ and $v.q$ to Paused if they do not already have that value (Rule 1). If $\mathit{dialogPaused}(v) \wedge \mathit{startCond}_Q(v)$, then (v, Q) sets $v.q$ to Repaired (Rule 2). If $\mathit{dialogAcknowledged}(v)$, then (v, Q) calls procedure $\mathit{action}_Q(v)$, sets $v.s$ to $v.q$, and if $v.q \neq \text{Paused}$, then $v.q$ is incremented (Rule 3). Note, that only one guard of (v, Q) can be true at a time. Protocol R_v consists two rules. If $v.s$ and $v.q$ equal Paused, then instance (u, R_v) sets $u.r_v$ to Paused if it doesn't have that value already (Rule 1). If $\mathit{validQuery}(v)$ is true and predicate $\mathit{waitCond}_{R_v}(u)$ is false, then (u, R_v) sets $u.r_v$ to $v.q$ and calls procedure $\mathit{action}_{R_v}(u)$ (Rule 2). Again, only one guard of (u, R_v) can be true at a time.

$$\begin{aligned}
&\mathit{validQuery}(v) \equiv v.q = (v.s+1) \bmod 4\\
&\mathit{dialogConsistent}(v) \equiv (v.q = v.s = \textsc{Paused} \vee \mathit{validQuery}(v))\\
&\qquad\qquad\qquad\qquad\qquad \wedge \forall u \in N(v) : u.r_v \in \{v.s, v.q\}\\
&\mathit{dialogAcknowledged}(v) \equiv \mathit{validQuery}(v) \wedge \forall u \in N(v) : u.r_v = v.q\\
&\mathit{dialogPaused}(v) \equiv v.q = v.s = \textsc{Paused} \wedge \forall u \in N(v) : u.r_v = \textsc{Paused}
\end{aligned}$$

$$\begin{aligned}
&\mathit{copyConsistent}(v) \equiv \forall u \in N(v) : u.c_v = v.p\\
&\mathit{repaired}(v) \equiv \mathit{copyConsistent}(v) \vee (\mathit{dialogConsistent}(v)\\
&\qquad\qquad \wedge (v.s = \textsc{Repaired} \vee v.s = \textsc{Executed})\\
&\qquad\qquad \wedge \forall u \in N(v) : (u.r_v = \textsc{Copied} \Rightarrow u.c_v = v.p))\\
&\mathit{startCond}_Q(v) \equiv \neg\mathit{copyConsistent}(v) \vee G_P(v)\\
&\mathit{waitCond}_{R_v}(u) \equiv v.q = \textsc{Executed} \wedge \neg\mathit{repaired}(u)
\end{aligned}$$

Procedure $\mathit{action}_Q(v)$ performs the following actions: If $v.q = \textsc{Executed}$, then a move of protocol P is executed. If $v.q = \textsc{Repaired}$, then $v.p$ is checked for corruptions by using the backups provided by protocol R_v. If all backups have the same value and $v.p$ differs, then $v.p$ is updated. If there is only one neighbor $u \in N(v)$ and hence only one backup, then $v.p$ is only updated if $u.d_v = \textsc{Update}$. Procedure $\mathit{action}_{R_v}(u)$ performs the following: If $v.q = \textsc{Copied}$, then $u.c_v$ is updated with the value of $v.p$. If $v.q = \textsc{Repaired}$, then $u.d_v$ is set to either $\textsc{Keep}$ or $\textsc{Update}$ depending on whether $v.p := u.c_v$ would disable P on both u and v.

The case that the network contains a single edge only is not covered by the above description of action_Q and action_{R_v}. In case of the distributed scheduler, it needs special treatment like symmetry breaking which is not included in the original version as given in [8]. The following describes a possible solution: Let u and v be neighbors and the only nodes of the system. The special value $\textsc{Single}$, which is assigned to $v.d_u$ and $u.d_v$ in this case, allows the detection of this case. The version of protocol action_Q as given [8] would execute both of the assignments $v.p := u.p$ and $u.p := v.p$ if both v and u are selected in the same step by the distributed scheduler. Either one of the two assignments leads to a legitimate configuration, but in most cases the execution of both does not. Let v be the node with the lower Id. action_Q can be altered in such a way that $v.p := u.p$ is not executed if $u.p := v.p$ leads to a successful repair.

The following ranking r is used to obtain serializations. For convenience, the predicate $R(x)$ is used in the definition of $r(v,p)$. It is satisfied if and only if rule x of instance (v,p) is enabled.

$$r(v,p) := \begin{cases} 1 & \text{if } \exists u \in N(v) : p = R_u \wedge R(2) \wedge u.q = \textsc{Repaired}\\ 2 & \text{if } \exists u \in N(v) : p = R_u \wedge ((R(2) \wedge u.q \neq \textsc{Repaired}) \vee R(1))\\ 3 & \text{if } p = Q \wedge R(2)\\ 4 & \text{if } p = Q \wedge R(3) \wedge v.q = \textsc{Executed}\\ 5 & \text{if } p = Q \wedge ((R(3) \wedge v.q \neq \textsc{Executed}) \vee R(1))\\ \perp & \text{otherwise} \end{cases}$$

With this ranking, any serialization will first execute all moves, that set decision variables. Their value is determined by the evaluation of guards of protocol P which references the primary states of various cells. Hence it is important, that the primary states have not been changed yet (which is only done by instances of rank 4, or 5). Next, all all other moves by instances of protocol R_v occur within the serialization. These do not reference any of the variables that have been changed previously. All instances of Q follow. They are categorized into three different ranks. Instances that are enabled due to rule 2 of Q are executed first to avoid the danger that this rule becomes disabled which is due to $startCond_Q(v)$ that checks whether protocol P is enabled for node v. Instances of rank 4 follow. They execute a single move of P and hence may change primary states. The fact that all moves of P fall into the same rank has the advantage, that invariancy-rankings for protocol P can be used to extent r to an invariancy ranking for algorithm $\mathcal{A}_2$. Last, all instances of rank 5 occur within the serialization. These moves do not read any primary states and their behaviour is hence not influenced by any of the changes done by previous instances of Q.

Lemma 2. *If cell v is not dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Lemma 3. *If cell v is dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Lemma 4. *Let v be a cell. (v, Q) is invariant under any (u, R_w) with $u \in N[v]$ and $w \neq v$.*

Lemma 5. *Assume, that an invariancy-ranking for protocol P exists. Spreading rank 4 of r according to this invariancy-ranking (and shifting the higher ranks accordingly) yields an invariancy-ranking for algorithm $\mathcal{A}_2$.*

Proof. The proof of case $r_2 \in \{3, 4, 5\} \wedge r_1 \in \{1, 2\}$ is based on Lemmas 2, 3, and 4. A detailed proof of those lemmas and a proof covering all other cases of r_2 and r_1 has been omitted due to space restrictions. □

Theorem 4. *If there exists an invariancy-ranking for protocols P, then for any execution e of $\mathcal{A}_2$ under the distributed scheduler there exists an execution e' under the central scheduler such that e is a subsequence of e'.*

5.3 Proof Refinement

The results of Theorems 3 and 4 do not only guarantee stabilization under the distributed scheduler. They guarantee, that the algorithms behave exactly as under the central scheduler, in all aspects – for example with respect to time-complexity and fault-containment properties. Yet, Theorems 3 and 4 require that an invariancy-rankings for protocols P, C, and B exist. This requirements can be relaxed.

Protocol C must show correct behaviour only if the initial configuration is 1-faulty. Such configurations are derived from a legitimate configuration by perturbing variables of a single node. Indeed, an invariancy-ranking can be found under these assumptions. For any other initial configuration, C may show an arbitrary behaviour. The invariancy ranking for B is simply $r_B(v, p) := 0$. Protocol B never reads the variables that it writes.

With respect to P, both $\mathcal{A}_1$ and $\mathcal{A}_2$ solely rely on the property that P terminates after a finite number of steps of the given scheduler. There is no other requirement concerning the behaviour of P. For each single step $S_i \subseteq \mathcal{M}$ of the distributed scheduler, the sequence obtained by sorting S_i by the rankings given above yields a configuration that differs from the execution of S_i only in the primary states of those nodes that make a P-move during the execution of S_i. In addition, it can be shown that P successfully progresses towards its termination during the execution of S_i.

Furthermore, it is easy to transform a given algorithm $\mathcal{A} = \{p_1, p_2, \ldots, p_k\}$ for the multi-protocol model into an algorithm $\mathcal{A}_s = \{q\}$ for the ordinary single-protocol model. In [8], a composition is used for the case of the central scheduler. The idea is to sequentially execute the individual p_i within a single move of q. Due to the nature of the central scheduler as defined in the multi-protocol model it is not a problem that the changes made by p_i become visible to p_{i+1} immediately. In case of the distributed scheduler, a different transformation is needed. Again, all of the p_i are executed sequentially during a single move of q. But to emulate composite atomicity, the changes made by any p_i are hidden from any p_j, $j \neq i$ until the end of the move of q. This may cost some memory overhead which can be avoided by finding a special invariancy-ranking only for serializing instances on a single node. In that style, the protocols of algorithm $\mathcal{A}_2$ can be executed in the order $\langle R_1, R_2, \ldots, R_\Delta, Q\rangle$. We would like to emphasize, that one round of $\mathcal{A}_s$ is equivalent to one round of $\mathcal{A}$.

6 Impossibility

Unfortunately, serializations do not always exist. This is the case for the MIS-protocol proposed in [9] which has been especially designed for the distributed scheduler. The proof of stabilization for the distributed scheduler is not straight forward [9]. In the following, the basic obstacles that prevent the application of the new proof technique are explained.

The protocol assigns one of the three states OUT, WAIT, and IN to each node. If a node is in state OUT and does not have a neighbor in state IN, then its state is changed to IN (Rule 1). A node in state WAIT changes its state to OUT, if it has a neighbor in state IN (Rule 2). A node in state WAIT changes its state to IN, if it does not have a neighbor in state IN and all neighbors in state WAIT have a higher Id (Rule 3). A node in state IN switches to state OUT, if it has a neighbor in state IN (Rule 4).

As a first example, consider two neighboring nodes v and u, both in state IN, while all their neighbors are in state OUT. If the distributed scheduler selects both u and v during a single step, then both simultaneously switch to state to

OUT. Note, that the state of u is the only reason that v is enabled and vice versa. Under the central scheduler, one of the two nodes becomes disabled after the first move and remains in state IN. Hence no serialization exists.

Now imagine that v is in state WAIT and u is in state OUT, while all their neighbors are in state OUT again. Furthermore, assume that the Id of node u is higher then the Id of v. If the distributed scheduler selects both v and u during a single step, then v sets its state to IN by Rule 3 and u switches to state WAIT by Rule 1. Again, there is no serialization because a move by v disables u and vice versa.

For the sake of optimization, [9] proposes a modified version of Rule 4. Rule $4'$ only sets the state to OUT, if there is an IN-neighbor with a lower Id. This allows serializations of the first example by sorting the moves in descending order by the Ids. The node with the lowest Id serves as a "final cause" of the moves by the other nodes. Furthermore, it is possible to modify Rule 1 in such a way that nodes only switch from OUT to WAIT, if there is no WAIT-neighbor with a lower Id. Then the second example becomes serializable as well. Obviously, in order for serializations to exist, situations in which moves disable moves of neighboring nodes must be avoided or at least, it must be possible to resolve these conflicts by sorting.

7 Concluding Remarks

This paper has described a new technique for proving self-stabilization under the distributed scheduler. The task of proving self-stabilization is reduced to the task of finding an invariancy-ranking. The proof that a given ranking is indeed an invariancy-ranking is solely based on properties of sequential executions of pairs of moves under the central scheduler. The new technique has been successfully applied to two algorithms. Even more, Corollary 1 guarantees, that all properties of the algorithms are preserved. In particular, by Theorems 3 and 4, the two algorithms are the first transformers for adding fault-containment to self-stabilizing protocols that are known to work under the distributed scheduler. It has also been discussed that algorithms exist which stabilize under the distributed scheduler, but for which it is impossible to find serializations. Furthermore, rankings may exist that yield serializations but are not invariancy-rankings. We're not aware of any examples for the latter.

It remains to be investigated, how serializations can be found other than simply by sorting moves. Instances may occur multiple times or additional instances may be included in the serialization. Randomized protocols or non-deterministic choices by the scheduler are not a problem. Both issues can be solved by extending the instance tuple with information about the scheduler's choice (the rule number) and the outcome of the random experiment during the move. This way, the information becomes part of the ranking. Another possibility is to generalize the notation $(c : m)$ to $\{c : m\}$ which denotes the set of configurations reachable from $c \in \Sigma$ by the execution of an instance $m \in \mathcal{M}$.

References

1. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-stabilizing local mutual exclusion and daemon refinement. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 223–237. Springer, Heidelberg (2000)
2. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
3. Dijkstra, E.W.: EWD 391, self-stabilization in spite of distributed control. In: Selected Writings on Computing: a Personal Perspective, pp. 41–46. Springer, Berlin (1982); Originally Written in 1973
4. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
5. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing 20(1), 53–73 (2007)
6. Gradinariu, M., Tixeuil, S.: Conflict managers for self-stabilization without fairness assumption. In: Proceedings of the 27th IEEE International Conference on Distributed Computing Systems, p. 45. IEEE Computer Society, Los Alamitos (2007)
7. Herman, T.: Models of self-stabilization and sensor networks. In: Das, S.R., Das, S.K. (eds.) IWDC 2003. LNCS, vol. 2918, pp. 205–214. Springer, Heidelberg (2003)
8. Köhler, S., Turau, V.: Fault-containing self-stabilization in asynchronous systems with constant fault-gap. In: Proceedings of the 30th IEEE International Conference on Distributed Computing Systems, pp. 418–427. IEEE Computer Society, Los Alamitos (2010)
9. Turau, V.: Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. Information Processing Letters 103(3), 88–93 (2007)
10. Turau, V., Weyer, C.: Fault tolerance in wireless sensor networks through self-stabilization. International Journal of Communication Networks and Distributed Systems 2(1), 78–98 (2009)

A Tranformational Approach for Designing Scheduler-Oblivious Self-stabilizing Algorithms[*]

Abhishek Dhama and Oliver Theel

Department of Computer Science,
University of Oldenburg, Germany
{abhishek.dhama,theel}@informatik.uni-oldenburg.de

Abstract. The complexity of designing self-stabilizing systems is often compounded by the assumptions about the underlying schedulers. This paper presents a method to transform a self-stabilizing algorithm working under a given arbitrary, but potentially very restrictive, scheduler to a self-stabilizing algorithm under any weakly fair scheduler. The method presented here implements a *progress monitor* by exploiting the knowledge of a ranking function –used for proving convergence under the original scheduler– to carry out the transformation.

1 Introduction

Self-stabilization is an elegant approach for designing distributed algorithms which can cope with arbitrary transient faults without any auxiliary components. A self-stabilizing distributed algorithm fulfills its specification within a finite number of execution steps irrespective of its starting state and thus can operate correctly despite intermittent occurrences of transient faults. This property has motivated protocol engineers to design network protocols that exhibit self-stabilization as in [1].

A critical part of designing a self-stabilizing algorithm is the proof that the algorithm indeed converges to the behavior outlined in its specification. Such proofs are, however, not easy to draw and automatic methods to do so do not scale well enough [2]. A proof of self-stabilization also depends on the underlying scheduler and the fairness assumption [3]; the increased generality of schedulers – embodying scheduling strategies and fairness assumptions – makes convergence proofs progressively complicated.

In order to get around the complexity of proofs due to the underlying schedulers, Gouda and Haddix [4] suggested the use of a so-called “alternator” to preserve the self-stabilization property under a distributed scheduler and, in turn, spurred investigation into such transformers. These transformers, however, require that the original algorithm must be self-stabilizing under all weakly fair schedulers. While the transformation of the self-stabilization property from weakly fair sequential schedulers to distributed schedulers is well-studied, methods required for self-stabilizing algorithms which exhibit convergence under very

[*] This work was supported by the German Research Foundation (DFG) under grant SFB/TR 14/2 “AVACS.”

restrictive schedulers, such as weakly-stabilizing algorithms [5], have not been investigated extensively. As Devismes *et al.* [6] showed, a weakly-stabilizing algorithm can at best exhibit probabilistic convergence under a distributed randomized scheduler. Nonetheless, there is a need for a method to transform *probable* convergence to *guaranteed* convergence.

The crux of the challenge is to identify and enable –in every state of a system executing the algorithm– processes whose actions are "beneficial" to the overall convergence of the algorithm. Recently, it has been shown that during the design phase of a distributed algorithm, the results of verification can be used to transform the algorithm such that the amount of knowledge a process has, determines whether its actions are enabled or not [7]. This raises the question whether a similar approach can be used to design a transformer for self-stabilizing algorithms under very restrictive schedulers.

Contributions. We suggest the usage of a ranking function, returned as a by-product of a convergence proof of a distributed algorithm under a specific (and possibly restrictive) scheduler, to transform a self-stabilizing algorithm. We present a method to transfer the self-stabilization property of a distributed algorithm proven under a specific scheduler to any *weakly fair scheduler*. The transformation embodies a *progress monitor* [8] which tracks the progress of a self-stabilizing algorithm towards its correct behavior.

Outline. The paper is structured as follows. We introduce the definitions and concepts required for presenting the transformation method together with the system model in Section 2. The transformation is detailed in Section 3 together with the proofs and an optimization technique for the transformer. Section 4 provides an overview of related work. The paper concludes with a perspective on future work in Section 5.

2 System Model

We now present the system model used in this paper along with the necessary definitions.

Distributed System. A distributed system consists of a set of n processes $\Pi := \{P_1, \cdots, P_n\}$ which communicate with each other with the help of shared memory registers. Two processes P_i and P_j are said to be neighbors of each other if they can communicate with each other directly.

Shared Memory Model. A process communicates with other processors via a set of communication registers. This set of registers is divided into two classes: read and write registers. *Write registers* are *owned* by a process (i.e., they form part of this process' local state space) and are used to inform other processors about its local state. Read registers are used to gauge the local states of the neighbors.

Fault Model. A process can be affected by a burst of *transient faults.* A transient fault is temporary and changes local variables and write registers owned by a process arbitrarily, while leaving program code unaffected. It is assumed that temporal separation between any two bursts of transient faults is long enough to allow a self-stabilizing algorithm to converge.

Process Model. A *process* has a system-wide unique identifier –which belongs to a totally ordered set– and is comprised of write registers, read registers, local variables and the sub-algorithm it executes. Memory registers used by a process for internal computation constitute its local variables. They cannot be accessed by any other process in the system. The communication structure of a distributed system can be abstracted as a graph such that each process in Π corresponds to a node in the graph. Each pair of nodes representing processes which can communicate with each other directly, is connected by a bidirectional edge in the graph.

Distributed Algorithm. A sub-algorithm A_{sub_x} executed by a process P_x comprises of *guarded commands* of the format

$$\langle\text{label}\rangle :: \mathcal{G}_{ix} \rightarrow \text{act}_{xi}$$

where "$\langle$label$\rangle$" is a process-wide *unique identifier* of a guarded command. *Guard* "$\mathcal{G}_{ix}$" is a Boolean expression over local variables and communication registers of P_x. "act_{xi}" is an *assignment function* that potentially assigns values to local variables and write registers of P_x. A guarded command is *enabled* if the Boolean expression in its guard holds true. A guarded command is *executed* if and only if the local variables and/or the write registers of a process are modified via the assignment function. A process is enabled if it has *at least one enabled* guarded command. A distributed algorithm is the union of sub-algorithms executed by all the processes in Π, i.e. $A := \bigcup_{x=1}^{n} A_{\text{sub}_x}$.

Global System State. The *local state* ϑ_x of a process P_x consists of the valuation of its local variables and write registers. The *global system state* of a distributed system is a vector $\sigma = [\vartheta_1, \cdots, \vartheta_n]$ whose elements are the local states of all the processes in the system. The set of all possible global system states σ of a distributed algorithm is called *global system state space* Σ.

Execution. An *execution* Ξ of a distributed algorithm A is a sequence of global system states σ_i, i.e., $\Xi := \langle\sigma_1, \sigma_2, \cdots\rangle$ such that σ_1 is the initial state and state σ_{i+1} is reachable from state σ_i via execution of a subset of enabled guarded commands. Such an execution of guarded commands is called an *execution step.* An execution $\widehat{\Xi}$ is a *maximal execution* if 1) $\widehat{\Xi}$ is finite and no guard is enabled in the last global state or 2) $\widehat{\Xi}$ is infinite.

Scheduler. An implementation of a distributed algorithm might have multiple processes with enabled guarded commands at any instant of time. This leads to a potentially non-deterministic system model. A *scheduler* is thus employed to produce executions of the system and thereby resolves non-determinism inherent in such a model. A scheduler $\mathbb{D}$ potentially resolves the non-determinism by selecting a subset of enabled processes for each execution step. An enabled process that is selected for execution by the scheduler is termed as *activated.* Hence, a scheduler can be represented by a countable infinite set $\mathbb{D} = \{\varrho_1, \varrho_2, \cdots\}$ such that an element ϱ_i is a (infinite) sequence $\varrho_i = \langle\rho_1, \rho_2, \cdots\rangle$ where $\rho_m \subseteq 2^{\Pi}$. ρ_j is a subset of enabled processes that is selected for execution by the scheduler in an execution step j. ϱ_i is a sequence in which enabled processes are activated by the scheduler. ϱ_i is synonymously called a *(scheduling) strategy* of the scheduler. Since we are only interested in *interleaved execution* of a system in the scope of this work, we consider schedulers which select *exactly one process in each step* if one is enabled, zero otherwise, i.e., $\rho_i = \{P_x\} | \emptyset, \; P_x \in \Pi$.

Fairness. While proving that a distributed algorithm satisfies a certain *liveness* property, a specific subset of executions is discarded by terming it as "not feasible." Feasibility is defined with respect to a scheduler and the algorithm – and thereby often implicitly tied to a certain notion of *fairness*. A fairness notion, in general, ensures that actions that have remained enabled sufficiently long are selected for execution frequently enough by the scheduler [9]. *Weak fairness* implies that a continuously enabled process is activated infinitely often while *strong fairness* ensures that an infinitely often enabled process is activated infinitely often by the scheduler.

Self-Stabilizing Algorithm. A distributed algorithm is self-stabilizing with respect to a state predicate $\mathcal{P}$ under a scheduler $\mathbb{D}$ if and only if it satisfies the following two conditions [10]: (1) The algorithm is guaranteed to reach a global state satisfying predicate $\mathcal{P}$ *irrespective* of the initial state. This property is called *convergence*. (2) Any execution starting in a state satisfying predicate $\mathcal{P}$ does not contain any state not satisfying predicate $\mathcal{P}$ in the absence of failures. This property is known as *closure*. Predicate $\mathcal{P}$ partitions the state space of a self-stabilizing algorithm into *safe* and *unsafe* states. The set of states satisfying the predicate $\mathcal{P}$ are called safe states and all other states are referred to as unsafe states. We abuse the terminology in rest of the paper and refer to $\mathcal{P}$ as *safety predicate* of a self-stabilizing algorithm.

Ranking Function. A *ranking function* (or *variant function*) $\Delta : \Sigma \rightarrow \Theta$ is a function that maps the global system state space of a distributed algorithm Σ to a *well-founded* set Θ such that $\Delta(\sigma_j) < \Delta(\sigma_i)$ for any two states σ_i and σ_j if σ_j is reachable from σ_i via a single execution step. Let $\delta := \Delta(\sigma_j) - \Delta(\sigma_i)$ be the difference[1] between the values of the ranking function of two such states σ_i and σ_j. The usefulness of a ranking function lies in the fact that it helps to show that a distributed algorithm satisfies the convergence property. Indeed, the existence of such a function Δ, such that 1) $\delta < 0$ for any execution step in a state σ_i with $\sigma_i \not\models \mathcal{P}$, and 2) $\Delta(\sigma_i) = \inf(\Theta)$ if $\sigma_i \models \mathcal{P}$, can be used to prove convergence of a self-stabilizing algorithm with respect to a predicate $\mathcal{P}$ [11,12].

3 Transformation of Self-stabilizing Algorithms

We now present a method to transform an algorithm that is self-stabilizing under a given scheduler to an algorithm that is self-stabilizing under any weakly fair scheduler.

3.1 Transformation Method

Let distributed algorithm A be self-stabilizing with respect to a predicate $\mathcal{P}_A$ under a specific scheduler $\mathbb{D}_A$. The convergence property of algorithm A under the scheduler $\mathbb{D}_A$ has been proven with the help of a ranking function Δ_A. Algorithm A consists of m_x guarded commands $\mathcal{G}_{A_{ix}}, 1 \leq i \leq m_x$ in each process $P_x \in \Pi$.

[1] We assume that Θ has a minus operator "$-$" and an ordering relation "$<$" defined over it.

We further assume that each process $P_x \in \Pi$ has *at most* one enabled guarded command in every state. Note that, as shown in [4], this assumption does not restrict the class of algorithms. Under these assumptions, the "scheduler-oblivious transformation" of algorithm A –synonymously referred to as *use algorithm*– is defined as follows.

Basic Idea. The transformation essentially implements a "progress monitor" of use algorithm A which tracks the progress of algorithm A with respect to its convergence towards the safe states. The transformation ensures that the actions of the use algorithm are enabled only if they guarantee progress towards the set of safe states. The progress of a use algorithm is tracked with help of a ranking function. The ranking function is typically provided by a proof of convergence of the use algorithm under a given scheduler (see *e.g.* [11,12]). Such continuous online calculation of a ranking function value, more often than not, requires global knowledge at each process. This necessitates a global coordination mechanism to support the functioning of such a ranking function-based progress monitor. A self-stabilizing mutual exclusion algorithm –as available in the literature– can be adapted to support the progress monitor for this purpose. However, self-stabilizing mutual exclusion algorithms require a rooted spanning tree to work correctly [13,14]. As we neither assume the presence of such a distinguished process serving as a root nor a spanning tree, a self-stabilizing spanning tree algorithm (thereby identifying a unique root) is required for correct execution of the self-stabilizing mutual exclusion algorithm. The transformation is carried out in two steps.

In the first step, the use algorithm A is modified by strengthening (i.e., modifying) its guards in a systematic fashion. In the second step, this modified algorithm –now called– A' is composed with the self-stabilizing mutual exclusion algorithm of [10, p. 24–27] and the self-stabilizing spanning tree algorithm of [13]. The self-stabilizing mutex algorithm of [10] is also slightly modified in order to make the transformation work. Figure 1 shows the components and the layered architecture of the transformation along with the flow of information between the constituent algorithms. The mutual exclusion layer uses the variable $parent_i$ belonging to spanning tree layer to control the execution of the modified use algorithm. The modified use algorithm requires variables representing a token and a global snapshot to evaluate and execute its guarded commands. Each component and transformation step is explained in requisite detail in the following.

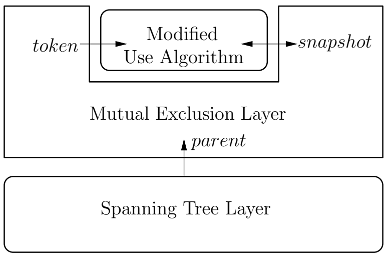


Fig. 1. Layered view of the transformation

Spanning Tree Layer. Each sub-algorithm of the self-stabilizing spanning tree algorithm uses three variables: dis_i, $parent_i$, and $root_i$ to construct a rooted spanning tree. The root_i variable of process P_i contains the identifier of the root node of the spanning tree to which P_i belongs. The shortest distance between the root node and a process P_i is stored in a variable dis_i. The variable parent_i points to the parent of process P_i in the spanning tree. Every process P_i repeatedly compares the three variables defining its spanning tree with that of its neighbors. It requests permission to join the spanning tree of a neighbor if the identifier of the root of the spanning tree of the neighbor is higher than that of its own. Such a request to join a spanning tree is routed through the members of the spanning tree to the root. The requesting node joins the tree when it receives a "grant" to do so from the neighbor through which it initiated the request. A process marks itself as "root" if it has the highest identifier in its neighborhood. The algorithm ensures that multiple spanning trees in a forest merge such that 1) the resultant spanning tree contains all the processes in the system and 2) the process with the highest identifier becomes the root of the resultant spanning tree. The algorithm removes non-existent root identifiers and cycles by running local consistency checks while processing requests and grants for joining a spanning tree in the intermediate nodes. The advantage of this algorithm lies in the fact that it does not require a distinguished process to build a spanning tree. The algorithm produces a spanning tree under any weakly fair scheduler.

Mutual Exclusion Layer. The mutual exclusion layer is implemented with the help of the self-stabilizing mutual exclusion algorithm of [10, p. 24–27]. The self-stabilizing mutual exclusion algorithm ensures that, eventually, in any global state only one process can access its critical section under any weakly fair scheduler. The access to the critical section is regulated with the help of a permission "token." A process can enter its critical section if and only if it has this token. In order to ensure that no process waits indefinitely for access to its critical section, the token is circulated among all the processes. The self-stabilizing algorithm circulates the token over the spanning tree constructed by the above described spanning tree layer. The structure of the communication registers used by the token circulation algorithm is modified for the proposed transformation to work; the token is used not only for mutual exclusion but also for gathering a global snapshot in order to the make the modified use algorithm work correctly. As shown in Figure 2, a communication register $\mathbf{r}_{ij}$ between two neighbor processes P_i and P_j is modified by *adding two parts*: a token part (left) and a global snapshot part (middle). The right part represents the original information communicated through the register due to use algorithm A. The token part of a communication register is an integer which is used by each process to decide whether it has the token and can therefore access the critical section or not. The global snapshot part is a vector of all those variables of use algorithm A which are required to calculate a concrete value of the ranking function Δ_A. For instance, if all those local variables and write registers of P_i which are used in the calculation of Δ_A are represented by a vector x_i, then a copy of x_i, $i = 1, \cdots, n$, would be an element of the global snapshot vector.

$$\boxed{\text{token}_i \vdots \underbrace{x_1^i, x_2^i, \cdots, x_n^i}_{\text{snapshot}_i} \vdots \cdots}$$

Fig. 2. Communication register $\mathbf{r}_{ij}$ between two processes P_i and P_j

One of the processes in the system –with communication infrastructure modified as described above– is labeled as "root" and all other processes are labeled as "non-root" as the result of the stabilization of the spanning tree algorithm executed by the lower layer. A *non-root process* P_i continuously compares the value of its local variable token$_i$ with that of its parent process: if it is not equal to that of its parents, then P_i can access its critical section. Let $\lambda_i = \langle \mathbf{r}_{iu}, \mathbf{r}_{iv}, \cdots, \mathbf{r}_{iz} \rangle$ be a total ordering of communication registers of a non-root process P_i that induces a total ordering of its children. In order to pass the token to its descendants, process P_i writes the token value of its parent process to the communication register $\mathbf{r}_{iu}$ meant for its first child P_u in the spanning tree. A process returns the token by copying the token value written in $\mathbf{r}_{iz}$ to the communication register meant for its parent. A non-root process P_i passes the token amongst its children by copying the value of the token written by a child P_u to write register $\mathbf{r}_{iv}$ meant for the next child P_v. The ordering λ_i of communication registers of a process defines the sequence in which its children get the token. Let $\lambda_x = \langle \mathbf{r}_{xi}, \cdots, \mathbf{r}_{xm} \rangle$ be the ordering of communication registers of a process P_x *designated as root.* Process P_x accesses its critical section if the value of its token is equal to that of its last child P_m. A root process updates its token by incrementing it modulo $4n - 5$ where n is the total number of processes in the system. It passes on the token to other processes in the tree in exactly the same way as a non-root process does. Figure 3 depicts the traversal of the token in an example spanning tree (left) along with the sequence of access to the critical section (right). The thick lines in the left figure indicate communication registers also used in the spanning tree whereas the thin lines indicate communication registers only required by the use algorithm A's logic.

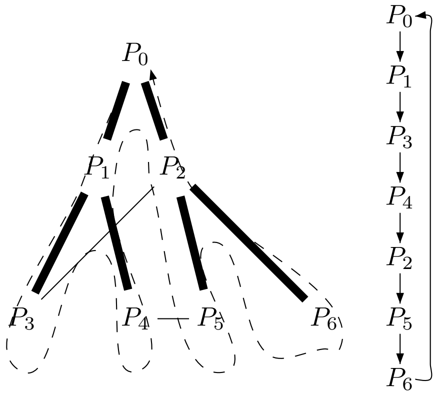


Fig. 3. Token circulation in a system

Modification of the Use Algorithm. The actions of the modified use algorithm are "embedded" in the mutual exclusion algorithm. The guarded commands of the modified use algorithm A' in a process P_i can be executed only if process P_i possesses the token. Thus, in essence, the algorithm A' constitutes the "critical section" guarded by the mutual exclusion algorithm. Furthermore, the set of guarded commands of the use algorithm is also modified during the transformation (see Figure 4).

A guarded command $\mathcal{G}_{ij}$ of use algorithm A in a process P_i, being selected by the scheduler, is executed only (1) if it is enabled, (2) P_i has the token and (3) the execution of $\mathcal{G}_{ij}$ leads to a decrease in Δ_A in case the safety predicate $\mathcal{P}_A$ does not hold (guarded commands of type 1). The term "decrease(Δ_A)" functions as a "look-ahead" operator of algorithm A. It is obtained by computing the sign of the difference between Δ_A and $\Delta_{A_{ij}}$. $\Delta_{A_{ij}}$ is obtained from Δ_A by replacing variables in Δ_A by their respective assignment expressions in act$_{ij}$.

```
process $P_i$
local var $x_i$;
have_token $\equiv$ $((\text{token}_i \neq \text{token}_{\text{parent}_i}) \wedge (\text{parent}_i \neq i) \wedge (\text{root}_i \neq i))$
                $\vee((\text{token}_i = \text{token}_{\text{rightsibling}_i}) \wedge (\text{parent}_i = \text{root}_i = i))$;
release_token $\equiv$ if$(\text{parent}_i = \text{root}_i = i)$
                $\text{token}_i := (\text{token}_{\text{rightsibling}_i} + 1)\,\textbf{mod}(4n-5)$
               else
                $\text{token}_i := \text{token}_{\text{parent}}$
paint_token $\equiv$ $\text{snapshot}_i.x_i^i := x_i$;
if(have_token)
  $[]\grave{\mathcal{G}}_{ij} ::\ \mathcal{G}_{ij} \wedge \neg\mathcal{G}_{i1} \wedge \cdots \wedge \neg\mathcal{G}_{in} \wedge \text{decrease}(\Delta_A) \wedge \neg\mathcal{P}_{\mathcal{A}} \rightarrow \text{act}_{ij}$; paint_token; (1)
  ⋮
  $[]\grave{\mathcal{G}}_{ip} ::\ \mathcal{G}_{ij} \wedge \neg\mathcal{G}_{i1} \wedge \cdots \wedge \neg\mathcal{G}_{in} \wedge \mathcal{P}_{\mathcal{A}} \rightarrow \text{act}_{ij}$; paint_token;        (2)
  ⋮
  $[]\grave{\mathcal{G}}_{iq} ::\ \neg\mathcal{G}_{i1} \wedge \cdots \wedge \neg\mathcal{G}_{in} \rightarrow$ skip; paint_token;        (3)

endif
release_token;
```

Fig. 4. Modified sub-algorithms of A' with guarded commands of type 1, 2, & 3

The actions act_{ij} on the assignment side of guarded commands of algorithm A remain unchanged. However, before a process P_i passes on the token after having executed a guarded command $\mathcal{G}_{ij}$, it writes the updated values of its local variables (required for the calculation of Δ_A) in the snapshot part of the token. This is achieved through the "*paint_token*" statement. In case the safety predicate $\mathcal{P}_A$ holds (guarded commands of type 2), then the truth value of guarded commands and the location of the token in the spanning tree determines the guarded command to be executed. A process P_i must write the latest values of its local variables to the snapshot part before passing on the token if none of its original guarded commands are enabled (guarded commands of type 3).

3.2 Preservation of the Self-stabilization Property

We now show that this transformation of use algorithm A preserves its self-stabilization property with respect to a predicate $\mathcal{P}_A$ under any weakly fair scheduler.

The transformed algorithm $\mathscr{T}(A)$ refers to the algorithm resulting from the composition of the modified use algorithm with the auxiliary algorithms as described in Section 3.1. We give some definitions before we proceed with the proof.

Definition 1 (Projection over Use Algorithm A)
Let $\widehat{\Xi} = \langle \cdots, \sigma_i, \sigma_k, \cdots, \sigma_j, \cdots \rangle$ be a maximal execution of a transformed algorithm $\mathscr{T}(A)$ and let σ_i and σ_j be the global states where the modified use algorithm executes one of its enabled guarded commands of type 1 or 2. The projection $\tilde{\Xi}_A$ of a maximal execution of $\mathscr{T}(A)$ over use algorithm A is obtained

by removing the variables not belonging to use algorithm A from every global state σ_i appearing in $\widehat{\Xi}$ in which use algorithm A executed an enabled guarded command, that is, $\tilde{\Xi}_A := \langle \cdots, \sigma_{i|var(A)}, \sigma_{j|var(A)}, \cdots \rangle$.

The projection of a maximal execution of a transformed algorithm $\mathscr{T}(A)$ over a process P_x, $\tilde{\Xi}_{P_x}$, is defined in the similar fashion.

Definition 2 (Correct Global Snapshot) *Let $\vartheta_{x|var(A)}$ be the projection of the local state ϑ_x of process P_x on the use algorithm A obtained by removing variables not belonging to the use algorithm A from ϑ_x. The local copy of the global snapshot at process P_y, obtained by inspecting the token, is termed as* correct global snapshot *if it contains the current values of $\vartheta_{x|var(A)}$ for all $P_x \in \Pi - \{P_y\}$.*

Definition 3 (1-Step Consistent Global Snapshot)
Let $\tilde{\Xi}_A = \langle \cdots, \sigma_{i|var(A)}, \sigma_{j|var(A)}, \cdots \rangle$ be the projection of a maximal execution of a transformed algorithm $\mathscr{T}(A)$ over the use algorithm A. Let σ_i be the global system state where process P_y executed an action of use algorithm A most recently. Let σ_k be the global system state in which some other process P_x executed an action of the use algorithm A such that k is the largest index satisfying the condition $i > k$ and ϑ_{kx} be the local state of a process P_x in the global state σ_k. The local copy of a global snapshot at a process P_y is said to be 1-step consistent *if for every process $P_x \in \Pi - \{P_y\}$, process P_y's copy of the local state of P_x is $\vartheta_{kx|var(A)}$.*

Note that 1-step consistency of a global snapshot as defined above is weaker than correctness. It captures scenarios where a process' copy of the global snapshot might be "outdated" by a single step due to an execution step of some other processes in the system.

Theorem 1 (based on [10]). *Every execution of the transformed algorithm $\mathscr{T}(A)$, irrespective of the starting state, self-stabilizes to a state where there is exactly one token circulating in the system within $O(n^2)$ rounds under any weakly fair scheduler.*

As a result of the self-stabilization of the spanning tree and mutual exclusion layers (Theorem 1), exactly one of the processes in the system assumes the role of a root process and coordinates the token circulation (see [10] for the proof). We refer to this distinguished process as P_{root} in the following.

Lemma 1. *Process P_{root} has the correct local global snapshot within $O(n)$ rounds after the transformed algorithm $\mathscr{T}(A)$ has self-stabilized to a state with exactly one circulating token.*

Proof. The mutual exclusion algorithm circulates the token on the spanning tree and the token traverses the tree in a depth-first manner. It defines an Euler tour (a virtual ring) over the spanning tree. This virtual ring has $2n-2$ virtual nodes. Process P_{root} gets the token and thereby a chance to access its critical section at least once every $4n-4$ rounds irrespective of the fact whether the mutual exclusion algorithm has stabilized or not (Theorem 1). Let σ_i be the global state in which process P_{root} gets the token for the first time after the mutual exclusion algorithm has self-stabilized. The global snapshot that P_{root} gets along with the token in global state σ_i might be incorrect and thus P_{root} may execute a guarded command that is enabled with the help of an incorrect global snapshot. However, the assignment part of each of the guarded commands writes the current values of the local variables belonging to use algorithm A to the global snapshot part of the token. Let σ_j be the global state after P_{root} executes its guarded command. Thus, when P_{root} passes on the token to its first child (defined by the ordering of outgoing edges in P_{root}) the global snapshot has the current local state $\sigma_{j_{\text{root}}|\text{var}(A)}$ of P_{root}.

Let σ_k be a global state after the state σ_j such that a non-root process P_z gets the token from its parent process. It can be argued that –irrespective of the correctness of the global snapshot which is passed on to process P_z along with the token– the token has the correct local state $\sigma_{lz|\text{var}(A)}$ of P_z, where σ_l is the resultant state after the execution of guarded commands of A, when it is passed on to the descendants of P_z. A process accesses its critical section (and thus executes use algorithm A) only when it gets the token from its parent process although it gets the token more than once while routing it through its sub-tree and back to its parent. This implies that once any process passes on the token to its descendants, the projection of its local state on the use algorithm does not change. Let $P_{\text{root}} \rightarrow P_x \rightarrow P_y \rightarrow P_z$ be the path traversed by the token in the spanning tree before it reaches process P_z. The argument above can be used to infer that P_z gets the current values of the local variables of processes P_{root}, P_x and P_z (belonging to use algorithm A) when it receives the token from its parent. This argument can be further extended to infer that every process gets the current values of variables belonging to use algorithm A of processes that possessed the token before and updates its local state prior to passing the token to its successors.

Every process gets a chance to access its critical section once in $4n-4$ rounds after the mutual exclusion algorithm has self-stabilized. Let σ_m be the global state in which P_{root} gets the token for the first time after the state σ_i. All the other processes update their current local states to the global snapshot between global states σ_i and σ_m and do not change them thereafter. Thus, when P_{root} gets the token in σ_m for the second time, it has the current value of every $\vartheta_{mx|\text{var}(A)}$ for all $x \in \{1, \cdots, n\} \setminus \{root\}$. □

Lemma 2. *Every process in the system has a* 1*-consistent global snapshot within* $O(n)$ *rounds following the state in which the process* P_{root} *gets the token exclusively for the second time.*

Proof. As stated in the proof of Lemma 1, every process gets a chance to execute its critical section once in $4n-4$ rounds after the mutual exclusion algorithm has self-stabilized. Also, process P_{root} gets the correct global snapshot when it gets the token exclusively for the second time (Lemma 1). Let σ_i be the global state in which P_{root} has a correct global snapshot. P_{root}, irrespective of actions of use algorithm A, updates its current local state $\vartheta_{j_{\text{root}}|\text{var}(A)}$ to the token before passing it on. Let P_α be the first child of P_{root} in the spanning tree. P_α gets the token right after P_{root}. As a result of token possession, P_α might change its local state with σ_α being the resultant global state. This makes the P_{root} copy of the local state of P_α outdated. Note that there cannot be an execution sequence (after the mutual exclusion algorithm has self-stabilized) such that between two global states σ_κ and σ_j where P_α accessed its critical section, no global state exists where P_{root} accessed its critical section. Thus, in global state σ_α, P_{root}'s copy of the local state of P_α has the value which corresponds to the global state σ_ι resulted from the execution of the critical section of P_α and ι is the largest index such that $\iota < i$.

Let $P_{\text{root}} \rightarrow P_\alpha \cdots \rightarrow P_\varsigma \rightarrow P_\beta$ be the sequence in which the token traverses the spanning tree after state σ_i. Let σ_β be the global state resulting from the execution of the critical section of process P_β. As a result of accessing its critical section (thereby possibly executing a guarded command of use algorithm A), P_β might change its local state. This action will make the copies of the local state of P_β outdated in all the processes that possessed the token before P_β. However, as we argued above, there cannot be an execution sequence with two states σ_ϖ and σ_β where P_β accessed its critical section and none of processes in the set $\{P_{\text{root}}, \cdots, P_\varsigma\}$ access their respective critical section. Thus, each process in $\{P_{\text{root}}, \cdots, P_\varsigma\}$ will have $\sigma_{\varpi\beta|\text{var}(A)}$ as local state of P_β where ϖ is the largest index such that $\varpi < \iota$ for each $\iota \in \{i, \alpha, \cdots, \varsigma\}$. This argument can be extended inductively for all the non-root process. Hence, it can be inferred that once process P_{root} gets the token again after state σ_i, all the processes in the system have a 1-step consistent global snapshot. □

Lemma 3. *A process executes an action of the use algorithm A in a global state with exactly one token if it has a correct global snapshot.*

Proof. Let σ_i be the global state where process P_{root} receives the correct global state. P_{root} also has the token in this state σ_i (from Lemma 1) which allows P_{root} to potentially execute an enabled guarded command of the use algorithm A (by construction). Thus, P_{root} executes an action of use algorithm A only if it has a correct global snapshot. Let P_β be a process that gets the token in some state after σ_i. P_β gets the correct snapshot when it gets the token (from Lemmata 1 and 2). Possession of the token also enables P_β to execute an enabled guarded command of use algorithm A if it has an enabled guarded command when it receives the token (by construction). This completes the proof. □

Lemma 4. *If an action of use algorithm A is executed by any process in a global state with exactly one token, then this execution step of the transformed algorithm $\mathscr{T}(A)$ leads to a decrease in the value of the ranking function Δ_A.*

Proof. A process P_i executes a guarded command of the use algorithm A only if it has the token (by construction). Also, an action of the use algorithm A is executed only if P_i has a correct global snapshot (from Lemma 3). The guards in every process are strengthened such that an assignment statement is executed only if its execution leads to a decrease in Δ_A. This, in conjunction with Lemma 3 completes the proof. □

Lemma 5. *After the root process gets a correct global snapshot for the first time, the projection of any execution of the transformed algorithm $\mathscr{T}(A)$ over the use *algorithm A under any weakly fair scheduler is an execution of use algorithm A under its originally used scheduler $\mathbb{D}_A$.*

Proof. Use algorithm A has some liveness property under scheduler $\mathbb{D}_A$ which is exhibited by the existence of the ranking function Δ_A. This implies that 1) in every state of the system executing use algorithm A there exists at least one process with an enabled guarded command $\mathcal{G}_{ij}$ and 2) in every state at least one of the processes with enabled guarded commands has a guarded command $\mathcal{G}_{ij}$ enabled such that decrease(Δ_A) holds until $\mathcal{P}_A$ is satisfied. Thus, in every state of the system executing the transformed algorithm $\mathscr{T}(A)$ there exists at least one process with a modified guarded command $\grave{\mathcal{G}}_{ij}$ being enabled until $\mathcal{P}_A$ holds. Let ε be a fragment of the projection $\tilde{\Xi}_A$ of a maximal execution $\hat{\Xi}_{\mathscr{T}(A)}$ of $\mathscr{T}(A)$ under any weakly fair scheduler such that its first state is the state where process P_{root} gets the correct global snapshot for the first time and $\mathcal{P}_A$ does not hold in any state of ε. Let $\sigma_{i|\text{var}(A)} \to \sigma_{j|\text{var}(A)}$ be an execution step of A in ε. There exists no execution in which a process without the token causes a change in the values of variables of A (by construction). Only one process can execute a guarded command of use algorithm A after the mutual exclusion algorithm has self-stabilized (Theorem 1). Thus, $\sigma_{i|\text{var}(A)} \to \sigma_{j|\text{var}(A)}$ can only realized through the execution of a guarded command of a single process of use algorithm A.

Let P_x be the process which executes the guarded command to do so. P_x executes an enabled modified guarded command in $\sigma_{i|\text{var}(A)}$ based on latest global information (Lemma 3), and this step leads to a decrease in Δ_A (Lemma 4). As shown above, the use algorithm A has an enabled guarded command that leads to a decrease in Δ_A in any state under it original scheduler $\mathbb{D}_A$. Note that the assignment statements of use algorithm A remain unchanged in the transformation. Thus, there exists an execution step of A under $\mathbb{D}_A$ which corresponds to $\sigma_{i|\text{var}(A)} \to \sigma_{j|\text{var}(A)}$ under $\mathbb{D}_A$. An execution step of A in $\sigma_{i|\text{var}(A)}$ is only possible in process P_x. Also note that, P_x cannot be denied a token indefinitely because the mutual exclusion layer ensures that each process gets the token infinitely often in any execution under any weakly fair scheduler. Hence, an execution step of A cannot be delayed indefinitely. This argument can be extended to build an execution of A under $\mathbb{D}_A$ which corresponds to $\tilde{\Xi}_A$ until $\mathcal{P}_A$ holds.

Let $\tilde{\Xi}_A$ be the projection of a maximal execution of $\mathscr{T}(A)$ such that $\tilde{\Xi}_A$ is not maximal. Let $\varepsilon_{\mathcal{P}_A}$ be a suffix of $\tilde{\Xi}_A$ such that all states satisfy $\mathcal{P}_A$. Thus, $\varepsilon_{\mathcal{P}_A}$ consists of states where a guarded command in a process is enabled but is never executed. This is, however, not possible because the mutual exclusion algorithm ensures that each process gets the token infinitely often in any maximal execution. A process executes an enabled guarded command of use algorithm A based on the latest global state (Lemma 3) when it gets token. The result of such an execution is the same as that of use algorithm A under $\mathbb{D}_A$ as the assignment parts of the guarded commands remain unchanged in the transformation. □

Lemma 6. *If a use algorithm A converges to a predicate $\mathcal{P}_A$ under the scheduler $\mathbb{D}_A$, then the transformed algorithm $\mathscr{T}(A)$ converges to the predicate $\mathcal{P}_A$ under any weakly fair scheduler.*

Proof. Let $\tilde{\Xi}_A$ be the projection of a maximal execution $\hat{\Xi}_{\mathscr{T}(A)}$ of the transformed algorithm $\mathscr{T}(A)$ over use algorithm A. Let ε be the suffix of $\tilde{\Xi}_A$ such that σ_i is the first state of ε and P_{root} has a correct global snapshot in σ_i for the first time (Lemma 1). ε is a maximal execution of A under $\mathbb{D}_A$ (Lemma 5). Let ε have no suffix that converges to a state satisfying $\mathcal{P}_A$. This implies that there exists a maximal execution of use algorithm A under its scheduler $\mathbb{D}_A$ which does not converge to a state satisfying the predicate $\mathcal{P}_A$. This, however, contradicts the precondition of the lemma statement. Also, $\varDelta_A$ is a monotonous function defined over a well-founded domain. Therefore $\mathscr{T}(A)$ reaches a state satisfying $\mathcal{P}_A$ in a finite number of execution steps. This completes the proof. □

Theorem 2. *The transformed algorithm $\mathscr{T}(A)$ is self-stabilizing with respect to predicate $\mathcal{P}_A$ under any weakly fair scheduler if $\mathcal{P}_A$ is closed under any weakly fair scheduler.*

Proof. The convergence of the transformed algorithm $\mathscr{T}(A)$ follows from Lemma 6. Its closure follows from Lemma 5. □

Optimization of the Transformation Method. The method presented so far relies on global mutual exclusion in order to preserve the self-stabilization property of a use algorithm. As a result of this transformation, concurrency inherent in the system is reduced to a bare minimum because in any state exactly one process can execute a guarded command of the use algorithm. This is optimal for the scenario where the online calculation of the ranking function needs local state information of all the processes in the system. However, there may be scenarios where a process can decide whether an execution of its guarded command is conducive to convergence of the system or not by inspecting the local states of the processes in its k-neighborhood where $2 \leq k < n$ only. In such scenarios, self-stabilizing *k-local* mutual exclusion algorithms (see *e.g.* [15,16]) can be used for local coordination instead of the global mutual exclusion algorithm to increase concurrency while preserving the convergence property of the use algorithm. The structure of the ranking function can be used to determine the locality of the k-local mutual exclusion algorithm. We omit the details due to the lack of space.

4 Related Work

The difficulty of designing a self-stabilizing algorithm from scratch has led to the development of various methods for transforming algorithms that are not self-stabilizing in the first place into respective self-stabilizing algorithms. A transformer that employs a *supervisor* process to reset a global system state has been proposed in [17]. The supervisor process periodically requests a global snapshot and resets the system to a pre-defined initial state in case the snapshot violates some (safety) predicate. This methods assumes the existence of a distinguished process in the system. Awerbuch and Varghese [18] presented a transformer that converts a synchronous distributed algorithm into an asynchronous self-stabilizing algorithm via a *resynchronizer*. A resynchronizer simulates a synchronous algorithm under an asynchronous environment in a self-stabilizing manner. The underlying principle of the resynchronizer-based transformer is to check the output of each process after T rounds, where T is the time complexity of the algorithm, and restart the algorithm if any inconsistency is detected during the checking phase. It has been shown in [19,13] that for some problems, it is sufficient to check the inconsistency *locally*. A *local stabilizer* that transforms *online* synchronous distributed algorithms into respective self-stabilizing algorithms is presented in [20]. Each process implementing a local stabilizer maintains a data structure termed as *pyramid*; a pyramid contains d values, where d is the diameter of the system, such that i^{th} entry represents the local states of the processes within i hops in $t - i$ rounds. An inconsistent system state is detected if pyramids of two neighboring processes do not match or if a process is not able to reconstruct its current local state using relevant entries of the pyramids of its neighbors. In case an inconsistency is detected, the system is repaired via pyramids of non-faulty processes. Beauquier *et al.* presented a set of transient fault detectors for various families of tasks [21]; a transient fault detector ensures that eventually, a transient fault is detected by at least one process in the system. Such a transient fault detector can be composed with a self-stabilizing reset algorithm [22] to transform a distributed algorithm into a self-stabilizing algorithm.

A lock-based transformer is presented in [23,24] to transfer the self stabilization property of an algorithm from a sequential scheduler to a distributed scheduler. The transformer ensures that a process can execute its guarded command only if it gets the lock. The conflict among multiple processes competing for the lock is resolved on the basis of timestamps sent along with the request for the lock. This transformer, however, may not preserve the self-stabilization property if convergence requires a fair scheduler. Self-stabilizing solutions to the classical problem of Dining Philosphers [25] have been proposed to transfer the self-stabilization property of an algorithm proven under a weakly fair scheduler to a distributed scheduler [26,27,28]. A transformation to preserve self-stabilization under unfair schedulers via composition is presented in [29]. Beauquier *et al.* [30] showed that for a specific class of self-stabilizing algorithms all k-fair schedulers are equivalent. A self-stabilizing algorithm to implement strong fairness under a weakly fair scheduler is presented in [31]. This algorithm emulates the behavior under a strongly fair scheduler by ensuring that whenever a process executes its guarded command, it does so exclusively in its 2-neighborhood. A *maximal* algorithm to emulate strong fairness is presented [32] where maximality implies that the algorithm is able to produce *all possible* strongly fair scheduling strategies. However, this algorithm is not self-stabilizing.

5 Conclusion and Future Work

We showed how to transform a self-stabilizing algorithm whose convergence property has been proven under a given (potentially restrictive) scheduler to a self-stabilizing algorithm under any weakly fair distributed scheduler. The transformation was achieved by "embedding" a known ranking function, previously used to prove convergence under the given scheduler, in the guarded commands of the algorithm. The transformation also preserves self-stabilization in the read/write atomicity model because it uses lower-layer algorithms designed for the read/write atomicity model [10, p. 71–73]. We also briefly described how the structure of a ranking function can be used to fine-tune the transformer.

The transformer presented in this paper can be used to develop a semi-automatic method to synthesize large self-stabilizing algorithms from smaller components. Another interesting extension is to see the effect of the transformation on cost and dependability metrics of the original self-stabilizing algorithm. In addition, we plan to analyze the effect of the component algorithms in the additional layers on the convergence time of the transformed algorithm. It is also be worthwhile to investigate the effect of choice of lower layers algorithms on the dependability of the transformed algorithm.

References

1. Perlman, R.: Interconnections: bridges, routers, switches, and internetworking protocols. Addison-Wesley Longman, Amsterdam (1999)
2. Tsuchiya, T., Nagano, S., Paidi, R.B., Kikuno, T.: Symbolic model checking for self-stabilizing algorithms. IEEE Trans. Parallel Distrib. Syst. 12, 81–95 (2001)
3. Burns, J.E., Gouda, M.G., Miller, R.E.: On Relaxing Interleaving Assumptions. In: Proc. MCC Workshop on Self-Stabilization. MCC Tech. Rep. STP, pp. 379–389 (1989)
4. Gouda, M.G., Haddix, F.F.: The alternator. Distributed Computing 20, 21–28 (2007)
5. Gouda, M.G.: The Theory of Weak Stabilization. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 114–123. Springer, Heidelberg (2001)
6. Devismes, S., Tixeuil, S., Yamashita, M.: Weak vs. Self vs. Probabilistic Stabilization. In: ICDCS, pp. 681–688. IEEE Computer Society, Los Alamitos (2008)
7. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority Scheduling of Distributed Systems Based on Model Checking. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
8. Balaban, I., Pnueli, A., Zuck, L.D.: Modular Ranking Abstraction. Int. J. Found. Comput. Sci. 18, 5–44 (2007)
9. Völzer, H., Varacca, D., Kindler, E.: Defining Fairness. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 458–472. Springer, Heidelberg (2005)
10. Dolev, S.: Self-Stabilization. The MIT Press, Cambridge (2000)
11. Kessels, J.L.W.: An Exercise in Proving Self-Stabilization with a Variant Function. Inf. Process. Lett. 29, 39–42 (1988)
12. Theel, O.: Exploitation of Ljapunov Theory for Verifying Self-Stabilizing Algorithms. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 209–222. Springer, Heidelberg (2000)
13. Afek, Y., Kutten, S., Yung, M.: The Local Detection Paradigm and Its Application to Self-Stabilization. Theor. Comput. Sci. 186, 199–229 (1997)
14. Dolev, S., Israeli, A., Moran, S.: Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. Distributed Computing 7, 3–16 (1993)
15. Boulinier, C., Petit, F.: Self-stabilizing wavelets and rho-hops coordination. In: Intl. Sym. Parallel and Distributed Processing, pp. 1–8. IEEE, Los Alamitos (2008)
16. Danturi, P., Nesterenko, M., Tixeuil, S.: Self-Stabilizing Philosophers with Generic Conflicts. ACM Trans. Autonomous and Adaptive Systems 4 (2009)
17. Katz, S., Perry, K.J.: Self-Stabilizing Extensions for Message-Passing Systems. Distributed Computing 7, 17–26 (1993)
18. Awerbuch, B., Varghese, G.: Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols. In: FOCS, pp. 258–267 (1991)
19. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-Stabilization By Local Checking and Correction. In: FOCS, pp. 268–277 (1991)

20. Afek, Y., Dolev, S.: Local Stabilizer. J. Parallel Distrib. Comput. 62, 745–765 (2002)
21. Beauquier, J., Delaët, S., Dolev, S., Tixeuil, S.: Transient Fault Detectors. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 62–74. Springer, Heidelberg (1998)
22. Arora, A., Gouda, M.G.: Distributed Reset. IEEE ToC 43, 1026–1038 (1994)
23. Mizuno, M., Kakugawa, H.: A Timestamp Based Transformation of Self-Stabilizing Programs for Distributed Computing Environments. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 304–321. Springer, Heidelberg (1996)
24. Kakugawa, H., Mizuno, M., Nesterenko, M.: Development of Self-Stabilizing Distributed Algorithms using Transformation: Case Studies. In: WSS, pp. 16–30. Carleton University Press (1997)
25. Chandy, K.M., Misra, J.: The Drinking Philosopher's Problem. ACM ToPLaS 6, 632–646 (1984)
26. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-Stabilizing Local Mutual Exclusion and Daemon Refinement. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 223–237. Springer, Heidelberg (2000)
27. Nesterenko, M., Arora, A.: Stabilization-Preserving Atomicity Refinement. J. Parallel Distrib. Comput. 62, 766–791 (2002)
28. Boulinier, C., Petit, F., Villain, V.: When Graph Theory helps Self-Stabilization. In: Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing, pp. 150–159 (2004)
29. Beauquier, J., Gradinariu, M., Johnen, C.: Cross-Over Composition - Enforcement of Fairness under Unfair Adversary. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 19–34. Springer, Heidelberg (2001)
30. Beauquier, J., Johnen, C., Messika, S.: All k-Bounded Policies Are Equivalent for Self-stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 82–94. Springer, Heidelberg (2006)
31. Karaata, M.H.: Self-Stabilizing Strong Fairness under Weak Fairness. IEEE Trans. Parallel Distrib. Syst. 12, 337–345 (2001)
32. Lang, M., Sivilotti, P.A.G.: A Distributed Maximal Scheduler for Strong Fairness. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 358–372. Springer, Heidelberg (2007)

On Byzantine Containment Properties of the *min* + 1 Protocol*

Swan Dubois[1], Toshimitsu Masuzawa[2], and Sébastien Tixeuil[1]

[1] LIP6 - UMR 7606 Université Pierre et Marie Curie - Paris 6 & INRIA, France
[2] Osaka University, Japan

Abstract. Self-stabilization is a versatile approach to fault-tolerance since it permits a distributed system to recover from any transient fault that arbitrarily corrupts the contents of all memories in the system. Byzantine tolerance is an attractive feature of distributed systems that permits to cope with arbitrary malicious behaviors.

We consider the well known problem of constructing a breadth-first spanning tree in this context. Combining these two properties prove difficult: we demonstrate that it is impossible to contain the impact of Byzantine processes in a strictly or strongly stabilizing manner. We then adopt the weaker scheme of *topology-aware strict stabilization* and we present a similar weakening of strong stabilization. We prove that the classical $min + 1$ protocol has optimal Byzantine containment properties with respect to these criteria.

1 Introduction

The advent of ubiquitous large-scale distributed systems advocates that tolerance to various kinds of faults and hazards must be included from the very early design of such systems. *Self-stabilization* [1,2,3] is a versatile technique that permits forward recovery from any kind of *transient* faults, while *Byzantine Fault-tolerance* [4] is traditionally used to mask the effect of a limited number of *malicious* faults. Making distributed systems tolerant to both transient and malicious faults is appealing yet proved difficult [5,6,7] as impossibility results are expected in many cases.

Two main paths have been followed to study the impact of Byzantine faults in the context of self-stabilization:

– *Byzantine fault masking.* In completely connected synchronous systems, one of the most studied problems in the context of self-stabilization with Byzantine faults is that of *clock synchronization.* In [8,5], probabilistic self-stabilizing protocols were proposed for up to one third of Byzantine processes, while in [9,10] deterministic solutions tolerate up to one fourth and one third of Byzantine processes, respectively.

* This work has been supported in part by ANR projects SHAMAN, ALADDIN, SPADES, by MEXT Global COE Program and by JSPS Grant-in-Aid for Scientific Research ((B) 22300009).

– *Byzantine containment.* For *local* tasks (*i.e.* tasks whose correctness can be checked locally, such as vertex coloring, link coloring, or dining philosophers), the notion of *strict stabilization* was proposed [7,11,12]. Strict stabilization guarantees that there exists a *containment radius* outside which the effect of permanent faults is masked, provided that the problem specification makes it possible to break the causality chain that is caused by the faults. As many problems are not local, it turns out that it is impossible to provide strict stabilization for those.

Our Contribution. In this paper, we investigate the possibility of Byzantine containment in a self-stabilizing setting for tasks that are global (*i.e.* for which there exists a causality chain of size r, where r depends on n the size of the network), and focus on a global problem, namely breadth-first spanning (BFS) tree construction. A good survey on self-stabilizing solutions to this problem can be found in [13]. In particular, one of the simplest solution is known under the name of $min + 1$ protocol (see [14,15]). This name is due to the construction of the protocol itself. Each process has two variables: one pointer to its parent in the tree and one level in this tree. The protocol is reduced to the following rule: each process chooses as its parent the neighbor which has the smallest level (min part) and updates its level in consequence (+1 part). [14] proves that this protocol is self-stabilizing. In this paper, we propose a complete study of Byzantine containment properties of this protocol.

First, we study space Byzantine containment properties of this protocol. As strict stabilization is impossible with such global tasks (see [7]), we use the weaker scheme of *topology-aware strict stabilization* (see [16]). In this scheme, we weaken the containment constraint by relaxing the notion of containment radius to containment area, that is Byzantine processes may disturb infinitely often a set of processes which depends on the topology of the system and on the location of Byzantine processes. We show that the $min+1$ protocol has optimal containment area with respect to topology-aware strict stabilization.

Secondly, we study time Byzantine containment properties of this protocol using the concept of *strong stabilization* (see [17,18]). We first show that it is impossible to find a strongly stabilizing solution to the BFS tree construction problem. It is why we weaken the concept of strong stabilization using the notion of containment area to obtain *topology-aware strong stabilization.* We show then that the $min + 1$ protocol has also optimal containment area with respect to topology-aware strong stabilization.

2 Distributed System

A *distributed system* $S = (P, L)$ consists of a set $P = \{v_1, v_2, \dots, v_n\}$ of processes and a set L of bidirectional communication links (simply called links). A link is an unordered pair of distinct processes. A distributed system S can be seen as a graph whose vertex set is P and whose link set is L, so we use graph terminology to describe a distributed system S.

Processes u and v are called *neighbors* if $(u, v) \in L$. The set of neighbors of a process v is denoted by N_v, and its cardinality (the *degree* of v) is denoted by $\Delta_v (= |N_v|)$. The degree Δ of a distributed system $S = (P, L)$ is defined as $\Delta = \max\{\Delta_v \mid v \in P\}$. We do not assume existence of a unique identifier for each process. Instead we assume each process can distinguish its neighbors from each other by locally arranging them in some arbitrary order: the k-th neighbor of a process v is denoted by $N_v(k)$ $(1 \leq k \leq \Delta_v)$.

In this paper, we consider distributed systems of arbitrary topology. We assume that a single process is distinguished as a *root*, and all the other processes are identical.

We adopt the *shared state model* as a communication model in this paper, where each process can directly read the states of its neighbors.

The variables that are maintained by processes denote process states. A process may take actions during the execution of the system. An action is simply a function that is executed in an atomic manner by the process. The actions executed by each process is described by a finite set of guarded actions of the form $\langle\text{guard}\rangle \longrightarrow \langle\text{statement}\rangle$. Each guard of process u is a boolean expression involving the variables of u and its neighbors.

A global state of a distributed system is called a *configuration* and is specified by a product of states of all processes. We define C to be the set of all possible configurations of a distributed system S. For a process set $R \subseteq P$ and two configurations ρ and ρ', we denote $\rho \overset{R}{\mapsto} \rho'$ when ρ changes to ρ' by executing an action of each process in R simultaneously. Notice that ρ and ρ' can be different only in the states of processes in R. For completeness of execution semantics, we should clarify the configuration resulting from simultaneous actions of neighboring processes. The action of a process depends only on its state at ρ and the states of its neighbors at ρ, and the result of the action reflects on the state of the process at ρ'.

We say that a process is *enabled* in a configuration ρ if the guard of at least one of its actions is evaluated at true in ρ.

A *schedule* of a distributed system is an infinite sequence of process sets. Let $Q = R^1, R^2, \ldots$ be a schedule, where $R^i \subseteq P$ holds for each i $(i \geq 1)$. An infinite sequence of configurations $e = \rho_0, \rho_1, \ldots$ is called an *execution* from an initial configuration ρ_0 by a schedule Q, if e satisfies $\rho_{i-1} \overset{R^i}{\mapsto} \rho_i$ for each i $(i \geq 1)$. Process actions are executed atomically, and we also assume that a *distributed daemon* schedules the actions of processes, *i.e.* any subset of processes can simultaneously execute their actions. We say that the daemon is *central* if it schedules action of only one process at any step.

The set of all possible executions starting from $\rho_0 \in C$ is denoted by E_{ρ_0}. The set of all possible executions is denoted by E, that is, $E = \bigcup_{\rho \in C} E_\rho$. We consider *asynchronous* distributed systems where we can make no assumption on schedules except that any schedule is *fair*: a process which is infinitely often enabled in an execution can not be never activated in this execution.

In this paper, we consider (permanent) *Byzantine faults*: a Byzantine process (*i.e.* a Byzantine-faulty process) can make arbitrary behavior independently from its actions. If v is a Byzantine process, v can repeatedly change its variables arbitrarily.

3 Self-stabilizing Protocol Resilient to Byzantine Faults

Problems considered in this paper are so-called *static problems*, *i.e.* they require the system to find static solutions. For example, the spanning-tree construction problem is a static problem, while the mutual exclusion problem is not. Some static problems can be defined by a *specification predicate* (shortly, specification), *spec*(v), for each process v: a configuration is a desired one (with a solution) if every process v satisfies *spec*(v). A specification *spec*(v) is a boolean expression on variables of P_v ($\subseteq P$) where P_v is the set of processes whose variables appear in *spec*(v). The variables appearing in the specification are called *output variables* (shortly, *O-variables*). In what follows, we consider a static problem defined by specification *spec*(v).

A *self-stabilizing protocol* ([1]) is a protocol that eventually reaches a *legitimate configuration*, where *spec*(v) holds at every process v, regardless of the initial configuration. Once it reaches a legitimate configuration, every process never changes its O-variables and always satisfies *spec*(v). From this definition, a self-stabilizing protocol is expected to tolerate any number and any type of transient faults since it can eventually recover from any configuration affected by the transient faults. However, the recovery from any configuration is guaranteed only when every process correctly executes its actions from the configuration, *i.e.*, we do not consider existence of permanently faulty processes.

3.1 Strict Stabilization

When (permanent) Byzantine processes exist, Byzantine processes may not satisfy *spec*(v). In addition, correct processes near the Byzantine processes can be influenced and may be unable to satisfy *spec*(v). Nesterenko and Arora [7] define a *strictly stabilizing protocol* as a self-stabilizing protocol resilient to unbounded number of Byzantine processes.

Given an integer c, a *c-correct process* is a process defined as follows.

Definition 1 (c-correct process). *A process is c-correct if it is correct (*i.e. *not Byzantine) and located at distance more than c from any Byzantine process.*

Definition 2 ((c, f)-containment). *A configuration ρ is* (c, f)-contained *for specification spec if, given at most f Byzantine processes, in any execution starting from ρ, every c-correct process v always satisfies spec(v) and never changes its O-variables.*

The parameter c of Definition 2 refers to the *containment radius* defined in [7]. The parameter f refers explicitly to the number of Byzantine processes, while [7] dealt with unbounded number of Byzantine faults (that is $f \in \{0 \dots n\}$).

Definition 3 ((c, f)-strict stabilization). *A protocol is* (c, f)-strictly stabilizing *for specification spec if, given at most f Byzantine processes, any execution $e = \rho_0, \rho_1, \ldots$ contains a configuration ρ_i that is (c, f)-contained for spec.*

An important limitation of the model of [7] is the notion of *r-restrictive* specifications. Intuitively, a specification is r-restrictive if it prevents two processes that are at least r hops away to be simultaneously in some given states. An important consequence related to Byzantine tolerance is that the containment radius of protocols solving those specifications is at least r. For some problems, such as the spanning tree construction we consider in this paper, r can not be bounded by a constant. We can show that there exists no $(o(n), 1)$-strictly stabilizing protocol for the spanning tree construction.

3.2 Strong Stabilization

To circumvent the impossibility result, [17] defines a weaker notion than the strict stabilization. Here, the requirement to the containment radius is relaxed, *i.e.* there may exist processes outside the containment radius that invalidate the specification predicate, due to Byzantine actions. However, the impact of Byzantine triggered action is limited in times: the set of Byzantine processes may only impact processes outside the containment radius a bounded number of times, even if Byzantine processes execute an infinite number of actions.

In the following of this section, we recall the formal definition of strong stabilization adopted in [18]. From the states of c-correct processes, *c-legitimate configurations* and *c-stable configurations* are defined as follows.

Definition 4 (c-legitimate configuration). *A configuration ρ is c-legitimate for* spec *if every c-correct process v satisfies spec(v).*

Definition 5 (c-stable configuration). *A configuration ρ is c-stable if every c-correct process never changes the values of its O-variables as long as Byzantine processes make no action.*

Roughly speaking, the aim of self-stabilization is to guarantee that a distributed system eventually reaches a c-legitimate and c-stable configuration. However, a self-stabilizing system can be disturbed by Byzantine processes after reaching a c-legitimate and c-stable configuration. The *c-disruption* represents the period where c-correct processes are disturbed by Byzantine processes and is defined as follows

Definition 6 (c-disruption). *A portion of execution $e = \rho_0, \rho_1, \ldots, \rho_t$ $(t > 1)$ is a c-disruption if and only if the following holds: (1) e is finite, (2) e contains at least one action of a c-correct process for changing the value of an O-variable, (3) ρ_0 is c-legitimate for* spec *and c-stable, and (4) ρ_t is the first configuration after ρ_0 such that ρ_t is c-legitimate for* spec *and c-stable.*

Now we can define a self-stabilizing protocol such that Byzantine processes may only impact processes outside the containment radius a bounded number of times, even if Byzantine processes execute an infinite number of actions.

Definition 7 ((t, k, c, f)-time contained configuration). *A configuration ρ_0 is (t, k, c, f)-time contained for* spec *if given at most f Byzantine processes, the following properties are satisfied: (1) ρ_0 is c-legitimate for* spec *and c-stable, (2) every execution starting from ρ_0 contains a c-legitimate configuration for* spec *after which the values of all the O-variables of c-correct processes remain unchanged (even when Byzantine processes make actions repeatedly and forever), (3) every execution starting from ρ_0 contains at most t c-disruptions, and (4) every execution starting from ρ_0 contains at most k actions of changing the values of O-variables for each c-correct process.*

Definition 8 ((t, c, f)-strongly stabilizing protocol). *A protocol A is (t, c, f)-strongly stabilizing if and only if starting from any arbitrary configuration, every execution involving at most f Byzantine processes contains a (t, k, c, f)-time contained configuration that is reached after at most l rounds. Parameters l and k are respectively the (t, c, f)-stabilization time and the (t, c, f)-process-disruption times of A.*

Note that a (t, k, c, f)-time contained configuration is a (c, f)-contained configuration when $t = k = 0$, and thus, a (t, k, c, f)-time contained configuration is a generalization (relaxation) of a (c, f)-contained configuration. Thus, a strongly stabilizing protocol is weaker than a strictly stabilizing one (as processes outside the containment radius may take incorrect actions due to Byzantine influence). However, a strongly stabilizing protocol is stronger than a classical self-stabilizing one (that may never meet their specification in the presence of Byzantine processes).

The parameters t, k and c are introduced to quantify the strength of fault containment, we do not require each process to know the values of the parameters.

4 Topology-Aware Byzantine Resilience

4.1 Topology-Aware Strict Stabilization

In Section 3.1, we saw that there exist a number of impossibility results on strict stabilization due to the notion of r-restrictive specifications. To circumvent this impossibility result, we describe here a weaker notion than the strict stabilization: the *topology-aware strict stabilization* (denoted by TA strict stabilization for short) introduced by [16]. Here, the requirement to the containment radius is relaxed, *i.e.* the set of processes which may be disturbed by Byzantine ones is not reduced to the union of c-neighborhood of Byzantine processes but can be defined depending on the graph and Byzantine processes location.

In the following, we recall the formal definitions of [16]. From now, B denotes the set of Byzantine processes and S_B (which is function of B) denotes a subset of V (intuitively, this set gathers all processes which may be disturbed by Byzantine processes).

Definition 9 (S_B-correct process). *A process is S_B-*correct *if it is a correct process (*i.e. *not Byzantine) which not belongs to S_B.*

Definition 10 (S_B-legitimate configuration). *A configuration ρ is* S_B-legitimate *for spec if every S_B-correct process v is legitimate for spec (*i.e. *if spec(v) holds).*

Definition 11 ((S_B, f)-topology-aware containment). *A configuration ρ_0 is* (S_B, f)-topology-aware contained *for specification spec if, given at most f Byzantine processes, in any execution $e = \rho_0, \rho_1, \ldots$, every configuration is S_B-legitimate and every S_B-correct process never changes its O-variables.*

The parameter S_B of Definition 11 refers to the *containment area*. Any process which belongs to this set may be infinitely disturbed by Byzantine processes. The parameter f refers explicitly to the number of Byzantine processes.

Definition 12 ((S_B, f)-topology-aware strict stabilization). *A protocol is* (S_B, f)-topology-aware strictly stabilizing *for specification spec if, given at most f Byzantine processes, any execution $e = \rho_0, \rho_1, \ldots$ contains a configuration ρ_i that is (S_B, f)-topology-aware contained for spec.*

Note that, if B denotes the set of Byzantine processes and $S_B = \{v \in V | min\{d(v, b), b \in B\} \leq c\}$, then a (S_B, f)-topology-aware strictly stabilizing protocol is a (c, f)-strictly stabilizing protocol. Then, a TA strictly stabilizing protocol is generally weaker than a strictly stabilizing one, but stronger than a classical self-stabilizing protocol (that may never meet their specification in the presence of Byzantine processes).

The parameter S_B is introduced to quantify the strength of fault containment, we do not require each process to know the actual definition of the set. Actually, the protocol proposed in this paper assumes no knowledge on the parameter.

4.2 Topology-Aware Strong Stabilization

In the same way as previous, we can weaken the notion of strong stabilization using the notion of containment area. Then, we obtain the following definition:

Definition 13 (S_B-stable configuration). *A configuration ρ is S_B-stable if every S_B-correct process never changes the values of its O-variables as long as Byzantine processes make no action.*

Definition 14 (S_B-TA-disruption). *A portion of execution $e = \rho_0, \rho_1, \ldots, \rho_t$ ($t > 1$) is a S_B-TA-disruption if and only if the following hold: (1) e is finite, (2) e contains at least one action of a S_B-correct process for changing the value of an O-variable, (3) ρ_0 is S_B-legitimate for spec and S_B-stable, and (4) ρ_t is the first configuration after ρ_0 such that ρ_t is S_B-legitimate for spec and S_B-stable.*

Definition 15 ((t, k, S_B, f)-TA time contained configuration). *A configuration ρ_0 is (t, k, S_B, f)-TA time contained for* spec *if given at most f Byzantine processes, the following properties are satisfied: (1) ρ_0 is S_B-legitimate for* spec *and S_B-stable, (2) every execution starting from ρ_0 contains a S_B-legitimate configuration for* spec *after which the values of all the O-variables of S_B-correct* *processes remain unchanged (even when Byzantine processes make actions repeatedly and forever), (3) every execution starting from ρ_0 contains at most t S_B-TA-disruptions, and (4) every execution starting from ρ_0 contains at most k actions of changing the values of O-variables for each S_B-correct process.*

Definition 16 ((t, S_B, f)-TA strongly stabilizing protocol). *A protocol A is (t, S_B, f)-TA strongly stabilizing if and only if starting from any arbitrary configuration, every execution involving at most f Byzantine processes contains a (t, k, S_B, f)-TA-time contained configuration that is reached after at most l actions of each S_B-correct process. Parameters l and k are respectively the (t, S_B, f)-stabilization time and the (t, S_B, f)-process-disruption time of A.*

5 BFS Spanning Tree Construction

In this section, we are interested in the problem of BFS spanning tree construction. That is, the system has a distinguished process called the root (and denoted by r) and we want to obtain a BFS spanning tree rooted to this root. We made the following hypothesis: the root r is never Byzantine.

To solve this problem, each process v has two O-variables: the first is $prnt_v \in N_v \cup \{\perp\}$ which is a pointer to the neighbor that is designated to be the parent of v in the BFS tree and the second is $level_v \in \{0, \dots, D\}$ which stores the depth of v in this tree. Obviously, Byzantine process may disturb (at least) their neighbors. For example, a Byzantine process may act as the root. It is why the specification of the BFS tree construction we adopted states in fact that there exists a BFS spanning forest such that any root of this forest is either the real root of the system or a Byzantine process. More formally, we use the following specification of the problem.

Definition 17 (BFS path). *A path $(v_0, \dots, v_k)$ $(k \geq 1)$ of S is a* BFS path *if and only if:*

1. $prnt_{v_0} = \perp$, $level_{v_0} = 0$, *and* $v_0 \in B \cup \{r\}$,
2. $\forall i \in \{1, \dots, k\}, prnt_{v_i} = v_{i-1}$ *and* $level_{v_i} = level_{v_{i-1}} + 1$, *and*
3. $\forall i \in \{1, \dots, k\}, level_{v_{i-1}} = \min\limits_{u \in N_{v_i}} \{level_u\}$.

We define the specification predicate $spec(v)$ of the BFS spanning tree construction as follows.

$$spec(v) : \begin{cases} prnt_v = \perp \text{ and } level_v = 0 \text{ if } v \text{ is the root } r \\ \text{there exists a BFS path } (v_0, \dots, v_k) \text{ such that } v_k = v \text{ otherwise} \end{cases}$$

In the case where any process is correct, note that $spec$ implies the existence of a BFS spanning tree rooted to the real root. The well-known $min + 1$ protocol solves this problem in a self-stabilizing way (see [14]). In the following of this section, we assume that some process may be Byzantine and we study the Byzantine containment properties of this protocol. We show that this self-stabilizing protocol has moreover optimal Byzantine containment properties.

In more details, we prove first that there exists neither strictly nor strongly stabilizing solution to the BFS spanning tree construction (see Theorems 1 and 2). Then, we demonstrate in Theorems 3 and 4 that the $min + 1$ protocol is both (S_B, f)-TA strictly and (t, S_B^*, f)-TA strongly stabilizing where $f \leq n-1$, $t = n\Delta$, and

$$S_B = \left\{ v \in V \;\middle|\; \underset{b \in B}{min} \left(d(v,b)\right) \leq d(r,v) \right\}$$
$$S_B^* = \left\{ v \in V \;\middle|\; \underset{b \in B}{min} \left(d(v,b)\right) < d(r,v) \right\}$$

Finally, we show that these containment areas are in fact optimal (see Theorems 5 and 6).

5.1 Impossibility Results

Theorem 1. *Even under the central daemon, there exists no* $(c, 1)$*-strictly stabilizing protocol for BFS spanning tree construction where* c *is any (finite) integer.*

Proof. This result is a direct application of Theorem 4 of [7] (note that the specification of BFS tree construction is D-restrictive in the worst case where D is the diameter of the system).

Theorem 2. *Even under the central daemon, there exists no* $(t, c, 1)$*-strongly stabilizing protocol for BFS spanning tree construction where* t *and* c *are any (finite) integers.*

Proof. Let t and c be (finite) integers. Assume that there exists a $(t, c, 1)$-strongly stabilizing protocol $\mathcal{P}$ for BFS spanning tree construction under the central daemon. Let $S = (V, E)$ be the following system $V = \{p_0 = r, p_1, \ldots, p_{2c+2}, p_{2c+3} = b\}$ and $E = \{\{p_i, p_{i+1}\}, i \in \{0, \ldots, 2c+2\}\}$. Process p_0 is the real root and process b is a Byzantine one.

Assume that the initial configuration ρ_0 of S satisfies: $level_r = level_b = 0$, $prnt_r = prnt_b = \bot$ and other variables of b (if any) are identical to those of r. Assume now that b takes exactly the same actions as r (if any) immediately after r (note that $d(r, b) > c$ and hence $level_r = 0$ and $prnt_r = \bot$ still hold by closure and then $level_b = 0$ and $prnt_b = \bot$ still hold too). Then, by construction of the execution and by convergence of $\mathcal{P}$ to *spec*, we can deduce that the system reaches in a finite time a configuration ρ_1 in which: $\forall i \in \{1, \ldots, c+1\}, level_{p_i} = i$ and $prnt_{p_i} = p_{i-1}$ and $\forall i \in \{c+2, \ldots, 2c+2\}, level_{p_i} = 2c+3-i$ and $prnt_{p_i} = p_{i+1}$ (because this configuration is the only one in which all correct process v such that $d(v, b) > c$ satisfies $spec(v)$ when $level_r = level_b = 0$ and $prnt_r = prnt_b = \bot$). Note that ρ_1 is 0-legitimate and 0-stable and *a fortiori* c-legitimate and c-stable.

Assume now that the Byzantine process acts as a correct process and executes correctly $\mathcal{P}$. Then, by convergence of $\mathcal{P}$ in fault-free systems (remember that a $(t, c, 1)$-strongly stabilizing protocol is a special case of self-stabilizing protocol), we can deduce that the system reaches in a finite time a configuration ρ_2 in which:

$\forall i \in \{1, \ldots, 2c+3\}, level_{p_i} = i$ and $prnt_{p_i} = p_{i-1}$ (because this configuration is the only one in which all process v satisfies $spec(v)$). Note that the portion of execution between ρ_1 and ρ_2 contains at least one c-perturbation (p_{c+2} is a c-correct process and modifies at least once its O-variables) and that ρ_2 is 0-legitimate and 0-stable and *a fortiori* c-legitimate and c-stable.

Assume now that the Byzantine process b takes the following state: $level_b = 0$ and $prnt_b = \perp$. This step brings the system into configuration ρ_3. From this configuration, we can repeat the execution we constructed from ρ_0. By the same token, we obtain an execution of $\mathcal{P}$ which contains c-legitimate and c-stable configurations (see ρ_1) and an infinite number of c-perturbations which contradicts the $(t, c, 1)$-strong stabilization of $\mathcal{P}$.

5.2 Byzantine Containment Properties of the $min + 1$ Protocol

In the $min + 1$ protocol, as in many self-stabilizing tree construction protocols, each process v checks locally the consistence of its $level_v$ variable with respect to the one of its neighbors. When it detects an inconsistency, it changes its $prnt_v$ variable in order to choose a "better" neighbor. The notion of "better" neighbor is based on the global desired property on the tree (here, the BFS requirement implies to choose one neighbor with the minimum level).

When the system may contain Byzantine processes, they may disturb their neighbors by providing alternatively "better" and "worse" states.

The $min+1$ protocol chooses an arbitrary one of the "better" neighbors (that is, a neighbor with the minimal level). Actually this strategy allows us to achieve the (S_B, f)-TA strict stabilization but is not sufficient to achieve the (t, S_B^*, f)-TA strong stabilization. To achieve the (t, S_B^*, f)-TA strong stabilization, we must bring a slight modification to the protocol: we choose a "better" neighbor with a round robin order (along the set of its neighbor).

Algorithm 5.2 presents our BFS spanning tree construction protocol $\mathcal{SSBFS}$ which is both (S_B, f)-TA strictly and (t, S_B^*, f)-TA strongly stabilizing (where $f \leq n-1$ and $t = n\Delta$) providing that the root is never Byzantine.

In the following of this section, we provide sketches of proof of topology-aware strict and strong stabilization of $\mathcal{SSBFS}$[1]. First at all, remember that the real root r can not be a Byzantine process by hypothesis. Note that the subsystems whose set of processes are respectively $V \setminus S_B$ and $V \setminus S_B^*$ are connected by construction.

$(S_B, n-1)$-TA strict stabilization

Given a configuration $\rho \in C$ and an integer $d \in \{0, \ldots, D\}$, let us define the following predicate:

$$I_d(\rho) \equiv \forall v \in V, level_v \geq min\left\{d, \min_{u \in B \cup \{r\}} \{d(v,u)\}\right\}$$

[1] Due to the lack of place, complete proofs are omitted but are available in the companion research report (see [19]).

Algorithm 1. $\mathcal{SSBFS}$: A TA strictly and TA strongly stabilizing protocol for BFS tree construction

```
Data:
    $N_v$: totally ordered set of neighbors of $v$
Variables:
    $prnt_v \in N_v \cup \{\bot\}$: pointer on the parent of $v$ in the tree.
    $level_v \in \mathbb{N}$: integer
Macro:
    For any subset $A \subseteq N_v$, $choose(A)$ returns the first element of $A$ which is bigger
    than $prnt_v$ (in a round-robin fashion).
Rules:
    $(\boldsymbol{R_r})$ :: $(v = r) \wedge ((prnt_v \neq \bot) \vee (level_v \neq 0)) \longrightarrow prnt_v := \bot; level_v := 0$
    $(\boldsymbol{R_v})$ :: $(v \neq r) \wedge \Big((prnt_v = \bot) \vee (level_v \neq level_{prnt_v} + 1) \vee$
                          $(level_{prnt_v} \neq \min_{q \in N_v}\{level_q\})\Big)$
                          $\longrightarrow prnt_v := choose\left(\left\{p \in N_v \middle| level_p = \min_{q \in N_v}\{level_q\}\right\}\right);$
                            $level_v := level_{prnt_v} + 1$
```

Let d be an integer such that $d \in \{0, \ldots, D\}$. Let $\rho \in C$ be a configuration such that $I_d(\rho) = true$ and $\rho' \in C$ be a configuration such that $\rho \overset{R}{\mapsto} \rho'$ is a step of $\mathcal{SSBFS}$. We can prove that in this case $I_d(\rho') = true$ that induces the following lemma.

Lemma 1. *For any integer $d \in \{0, \ldots, D\}$, the predicate I_d is closed.*

Let $\mathcal{LC}$ be the following set of configurations:

$$\mathcal{LC} = \{\rho \in C \,|(\rho \text{ is } S_B\text{-legitimate for } spec) \wedge (I_D(\rho) = true)\}$$

Let ρ be a configuration of $\mathcal{LC}$. By construction, ρ is S_B-legitimate for *spec*. If we assume that there exists a process $v \in V \setminus S_B$ enabled by a rule of $\mathcal{SSBFS}$ in ρ, then we can prove that the activation of this rule in a step $\rho \overset{R}{\mapsto} \rho'$ leads to $I_D(\rho') = false$, that contradicts the closure of I_D. Then, we can state that:

Lemma 2. *Any configuration of $\mathcal{LC}$ is $(S_B, n-1)$-TA contained for spec.*

In order to prove the convergence of $\mathcal{SSBFS}$, we prove the following property by induction on $d \in \{0, \ldots, D\}$:

$(\mathcal{P}_d)$: Starting from any configuration, any run of $\mathcal{SSBFS}$ reaches a configuration ρ such that $I_d(\rho) = true$ and in which any process $v \notin S_B$ such that $d(v, r) \leq d$ satisfies $spec(v)$.

The initialization part is easy. For the induction part, we assume that $(\mathcal{P}_{d-1})$ is true and we define the following set of processes $E_d = \{v \in V | min\{d(v, u), u \in B \cup \{r\}\} \geq d\}$. Then, we can prove that any process $v \in E_d$ such that $level_v = d - 1$ is activated in a finite time. In consequence, we can deduce that the system reaches in a finite time a configuration such that I_d holds. Then, we study processes of $V \setminus S_B$ such that $d(r,v) = d$. We prove that any process of this set which satisfies *spec* is never activated and that any process of this set which does not satisfy *spec* is activated in a finite time and then satisfies *spec*.

We can now deduce that $(\mathcal{P}_D)$ implies the following result.

Lemma 3. *Starting from any configuration, any execution of $\mathcal{SSBFS}$ reaches a configuration of $\mathcal{LC}$ in a finite time.*

Lemmas 2 and 3 prove respectively the closure and the convergence of $\mathcal{SSBFS}$ and imply the following theorem.

Theorem 3. *$\mathcal{SSBFS}$ is a $(S_B, n-1)$-TA strictly stabilizing protocol for spec.*

$(n\Delta, S_B^*, n-1)$-TA strong stabilization

Let $E_B = S_B \setminus S_B^*$ (*i.e.* E_B is the set of process v such that $d(r,v) = \underset{b \in B}{min}\{d(v,b)\}$).

The construction of sets $\mathcal{LC}$ and E_B, the closure of I_D and the construction of the macro *choose* imply the following result.

Lemma 4. *If ρ is a configuration of $\mathcal{LC}$, then any process $v \in E_B$ is activated at most Δ_v) times in any execution starting from ρ.*

Let ρ be a configuration of $\mathcal{LC}$ and v be a process such that $v \in E_B$. Assume that there exists an execution starting from ρ such that (i) $spec(v)$ is infinitely often false in e and (ii) v is never activated in e. For any configuration ρ, let us denote by $P_v(\rho) = (v, v_1 = prnt_v, v_2 = prnt_{v_1}, \ldots, v_k = prnt_{v_{k-1}}, p_v = prnt_{v_k})$ the maximal sequence of processes following pointers *prnt* (maximal means here that either $prnt_{p_v} = \bot$ or p_v is the first process such that there $p_v = v_i$ for some $i \in \{1, \ldots, k\}$).

In the case where $prnt_v \in V \setminus S_B$ in ρ, we can prove that $spec(v)$ remains true in any execution starting from ρ. This contradicts the assumption (i) on e. In the contrary case, we demonstrate that that there exists at least one process which is infinitely often activated in e. We can show this leads to a contradiction with assumption (ii) on e.

These contradictions allow us to state the following lemma.

Lemma 5. *If ρ is a configuration of $\mathcal{LC}$ and v is a process such that $v \in E_B$, then for any execution e starting from ρ either v is activated in e or there exists a configuration ρ' of e such that $spec(v)$ is always satisfied after ρ'.*

Let us define: $\mathcal{LC}^* = \{\rho \in C \,|(\rho \text{ is } S_B^*\text{-legitimate for } spec) \wedge (I_D(\rho) = true)\}$

Note that, as $S_B^* \subseteq S_B$, we can deduce that $\mathcal{LC}^* \subseteq \mathcal{LC}$. Hence, properties of Lemmas 4 and 5 also apply to configurations of $\mathcal{LC}^*$. In consequence, Lemmas 4 and 5 lead to the following result.

Lemma 6. *Any configuration of $\mathcal{LC}^*$ is $(n\Delta, \Delta, S_B^*, n-1)$-TA time contained for spec.*

Let ρ be an arbitrary configuration. We know by Lemma 3 that any execution starting from ρ reaches in a finite time a configuration ρ' of $\mathcal{LC}$. Let v be a process of E_B. By Lemmas 4 and 5, we know that v takes at most Δ_v actions in any execution starting from ρ'. Moreover, we know that v satisfies $spec(v)$ after its last action (otherwise, we obtain a contradiction between the two lemmas). This implies that any execution starting from ρ' reaches a configuration ρ'' such that any process v of E_B satisfies $spec(v)$. It is easy to see that $\rho'' \in \mathcal{LC}^*$. We can now state that:

Lemma 7. *Starting from any configuration, any execution of $\mathcal{SSBFS}$ reaches a configuration of $\mathcal{LC}^*$ in a finite time under a distributed fair scheduler.*

Lemmas 6 and 7 prove respectively the closure and the convergence of $\mathcal{SSBFS}$ and imply the following theorem.

Theorem 4. *$\mathcal{SSBFS}$ is a $(n\Delta, S_B^*, n-1)$-TA strongly stabilizing protocol for spec.*

5.3 Optimality of Containment Areas of the *min* + 1 Protocol

Theorem 5. *Even under the central daemon, there exists no $(A_B, 1)$-TA strictly stabilizing protocol for BFS spanning tree construction where $A_B \subsetneq S_B^*$.*

Proof. This is a direct application of the Theorem 2 of [16].

Theorem 6. *Even under the central daemon, there exists no $(t, A_B, 1)$-TA strongly stabilizing protocol for BFS spanning tree construction where $A_B \subsetneq S_B$ and t is any (finite) integer.*

Proof. Let $\mathcal{P}$ be a $(t, A_B, 1)$-TA strongly stabilizing protocol for BFS spanning tree construction protocol where $A_B \subsetneq S_B^*$ and t is a finite integer. We must distinguish the following cases:

Consider the following system: $V = \{r, u, u', v, v', b\}$ and $E = \{\{r, u\}, \{r, u'\}, \{u, v\}, \{u', v'\}, \{v, b\}, \{v', b\}\}$ (b is a Byzantine process). We can see that $S_B^* = \{v, v'\}$. Since $A_B \subsetneq S_B^*$, we have: $v \notin A_B$ or $v' \notin A_B$. Consider now the following configuration ρ_0: $prnt_r = prnt_b = \bot$, $level_r = level_b = 0$, $prnt$ and $level$ variables of other processes are arbitrary (other variables may have arbitrary values but other variables of b are identical to those of r).

Assume now that b takes exactly the same actions as r (if any) immediately after r. Then, by symmetry of the execution and by convergence of $\mathcal{P}$ to $spec$, we can deduce that the system reaches in a finite time a configuration ρ_1 in which: $prnt_r = prnt_b = \bot$, $prnt_u = prnt_{u'} = r$, $prnt_v = prnt_{v'} = b$, $level_r = level_b = 0$ and $level_u = level_{u'} = level_v = level_{v'} = 1$ (because this configuration is the only one in which every correct process v satisfies $spec(v)$ when $prnt_r = prnt_b = \bot$ and $level_r = level_b = 0$). Note that ρ_1 is A_B-legitimate for $spec$ and A_B-stable (whatever A_B is).

Assume now that b behaves as a correct process with respect to $\mathcal{P}$. Then, by convergence of $\mathcal{P}$ in a fault-free system starting from ρ_1 which is not legitimate (remember that a strictly-stabilizing protocol is a special case of self-stabilizing protocol), we can deduce that the system reaches in a finite time a configuration ρ_2 in which: $prnt_r = \perp$, $prnt_u = prnt_{u'} = r$, $prnt_v = u$, $prnt_{v'} = u'$, $prnt_b = v$ (or $prnt_b = v'$), $level_r = 0$, $level_u = level_{u'} = 1$ $level_v = level_{v'} = 2$ and $level_b = 3$. Note that processes v and v' modify their O-variables in the portion of execution between ρ_1 and ρ_2 and that ρ_2 is A_B-legitimate for $spec$ and A_B-stable (whatever A_B is). Consequently, this portion of execution contains at least one A_B-TA-disruption (whatever A_B is).

Assume now that the Byzantine process b takes the following state: $prnt_b = \perp$ and $level_b = 0$. This step brings the system into configuration ρ_3. From this configuration, we can repeat the execution we constructed from ρ_0. By the same token, we obtain an execution of $\mathcal{P}$ which contains c-legitimate and c-stable configurations (see ρ_1) and an infinite number of A_B-TA-disruption (whatever A_B is) which contradicts the $(t, A_B, 1)$-TA strong stabilization of $\mathcal{P}$.

6 Conclusion

In this article, we are interested in the BFS spanning tree construction in presence of both systemic transient faults and permanent Byzantine failures. As this task is global, it is impossible to solve it in a strictly stabilizing way. We proved then that there exists no solution to this problem even if we consider the weaker notion of strong stabilization.

Then, we provide a study of Byzantine containment properties of the well-known $min + 1$ protocol. This protocol is one of the simplest self-stabilizing protocols for this problem. However, it achieves optimal area containment with respect to the notion of topology-aware strict and strong stabilization.

Using the result of [20] about r-operators, we can easily extend results of this paper to some others problems as depth-first search or reliability spanning trees. This work raises the following open questions. Has any other global static task as leader election or maximal matching a topology-aware strictly or/and strongly stabilizing solution ? We can also wonder about non static tasks as mutual exclusion (recall that local mutual exclusion has a strictly stabilizing solution provided by [7]).

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. ACM Commun. 17(11), 643–644 (1974)
2. Dolev, S.: Self-stabilization. MIT Press, Cambridge (March 2000)
3. Tixeuil, S.: Self-stabilizing Algorithms. Chapman & Hall/CRC Applied Algorithms and Data Structures. In: Algorithms and Theory of Computation Handbook, 2nd edn., 26.1–26.45. pp. CRC Press/ Taylor & Francis Group (2009)
4. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
5. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. J. ACM 51(5), 780–799 (2004)

6. Daliot, A., Dolev, D.: Self-stabilization of byzantine protocols. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, pp. 48–67. Springer, Heidelberg (2005)
7. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: 21st Symposium on Reliable Distributed Systems, p. 22. IEEE Computer Society, Los Alamitos (2002)
8. Ben-Or, M., Dolev, D., Hoch, E.N.: Fast self-stabilizing byzantine tolerant digital clock synchronization. In: Bazzi, R.A., Patt-Shamir, B. (eds.) PODC, pp. 385–394. ACM, New York (2008)
9. Dolev, D., Hoch, E.N.: On self-stabilizing synchronous actions despite byzantine attacks. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 193–207. Springer, Heidelberg (2007)
10. Hoch, E.N., Dolev, D., Daliot, A.: Self-stabilizing byzantine digital clock synchronization. In: [21], pp. 350–362
11. Sakurai, Y., Ooshita, F., Masuzawa, T.: A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 283–298. Springer, Heidelberg (2005)
12. Masuzawa, T., Tixeuil, S.: Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. International Journal of Principles and Applications of Information Science and Technology (PAIST) 1(1), 1–13 (2007)
13. Gartner, F.C.: A survey of self-stabilizing spanning-tree construction algorithms. Technical report ic/2003/38, EPFL (2003)
14. Huang, S.T., Chen, N.S.: A self-stabilizing algorithm for constructing breadth-first trees. Inf. Process. Lett. 41(2), 109–117 (1992)
15. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing 7(1), 3–16 (1993)
16. Dubois, S., Masuzawa, T., Tixeuil, S.: The Impact of Topology on Byzantine Containment in Stabilization. In: DISC (to appear 2010), Technical report available at http://hal.inria.fr/inria--00481836/en/
17. Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. In: [21], pp. 440–453
18. Dubois, S., Masuzawa, T., Tixeuil, S.: Self-stabilization with byzantine tolerance for global tasks. Research report inria-00484645, INRIA (May 2010), http://hal.inria.fr/inria-00484645/en/
19. Dubois, S., Masuzawa, T., Tixeuil, S.: On byzantine containment properties of the min+1 protocol. Research report inria-00487091, INRIA (May 2010), http://hal.inria.fr/inria-00487091/en/
20. Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators. Distributed Computing 14(3), 147–162 (2001)
21. Datta, A.K., Gradinariu, M. (eds.): SSS 2006. LNCS, vol. 4280. Springer, Heidelberg (2006)

Efficient Self-stabilizing Graph Searching in Tree Networks

Jean Blair[1], Fredrik Manne[2], and Rodica Mihai[2,⋆]

[1] Department of EE and CS, United States Military Academy,
West Point, NY 10996, USA
Jean.Blair@usma.edu

[2] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
{fredrikm,rodica}@ii.uib.no

Abstract. The graph search problem asks for a strategy that enables a minimum sized team of searchers to capture a "fugitive" while it evades and potentially multiplies through a network. It is motivated by the need to eliminate fast spreading viruses and other malicious software agents in computer networks.

The current work improves on previous results with a self-stabilizing algorithm that clears an n node tree network using only $1+\log n$ searchers and $O(n \log n)$ moves after initialization. Since $\Theta(\log n)$ searchers are required to clear some tree networks even in the sequential case, this is the best that any self-stabilizing algorithm can do. The algorithm is based on a novel multi-layer traversal of the network.

1 Introduction

Networks of computing devices enable fast communication, pervasive sharing of information, and effective distributed computations that might otherwise be infeasible. Unfortunately, this comes at the cost of fast spreading viruses and malicious software agents. The fact that modern networks are constantly changing exacerbates the problem. Thus, it is important to regularly search a network in order to eliminate malicious software.

This setting has been formalized as various *graph search* problems where one asks for a strategy that will clear a graph of any unwanted "intruders" typically using as few operations as possible. One can think of a searcher as a separate software agent that must be run on the individual network devices in order to clear it. Thus, minimizing the number of searchers is also important as each searcher uses resources that the system could otherwise have used for productive work. One might further want to limit the number of concurrent searchers when there may be a cost associated with the maximum number used at any given time, for instance due to software licences.

There is a significant body of work focused on sequential algorithms for computing the minimum number of searchers required to search a graph and the

⋆ Now, International Research Institute of Stavanger, N-5008 Bergen, Norway.

corresponding searching strategy. See [4] for an annotated bibliography. Peng et. al. and Skodinis gave linear-time sequential node and edge searching algorithms for trees [9,10] and in [6,7] it was proven that in general $\Theta(\log n)$ searchers are required for tree networks. Distributed graph searching algorithms have also received considerable attention. See [8] for a list of the most recent works. For tree networks [3] gives a distributed algorithm to compute the minimum number of searchers necessary to clear the edges.

The first self-stabilizing algorithm for solving the node search problem in a tree was introduced in [8]. Given a tree T with n nodes and height h, their algorithm stabilizes after $O(n^3)$ time steps under the distributed adversarial daemon and the number of searchers used to clear the tree T is h.

In this paper we give an efficient self-stabilizing algorithm that improves on the results in [8]. Our algorithm is based on integrating two existing self-stabilizing algorithms with a new search algorithm in order to continuously search a tree network with $n \geq 2$ nodes using only $1 + \lfloor \log n \rfloor$ searchers and $O(n \log n)$ moves after initialization. As our algorithm is non-silent and self-stabilizing it will adapt to any transient faults or changes in the configuration of the network. Moreover, if an intruder is (re)introduced in a cleared network, the algorithm will immediately clear the network again.

We use the leader election algorithm from [2] for rooting the tree and then apply the efficient multiwave algorithm introduced in [1] to initialize the tree. This is then followed by our new search algorithm to clear the tree. The search algorithm works by recursively splitting the graph into smaller components and then clearing each of these. In this way the algorithm behaves as if it was composed of a number of layered algorithms each with its own set of variables. However, the clearing is achieved with just one efficient algorithm and with the number of variables linear in the size of the graph.

The paper is organized as follows. In Section 2 we give preliminaries, followed by a presentation and motivation of our algorithm in Section 3. In Section 4 we show the correctness of our algorithm before concluding in Section 5.

2 Preliminaries

The current focus is on a variant of the graph search problem known as *node searching*. A node search strategy for a graph $G = (V, E)$ consists of a sequence of steps where, at each step, searchers may be both added to and removed from the nodes of G. A node is cleared once a searcher is placed on it and an edge is cleared when searchers occupy both of its endpoints at the same time.[1] A node that has a searcher on it is guarded and cannot be recontaminated as long as the searcher is present on that node. However, cleared edges and cleared nodes without searchers on them are assumed to be recontaminated instantly iff there exists a path of unguarded nodes from them to an uncleared node or edge of the

[1] As will be discussed in the concluding remarks, our results can easily be adapted to solve other graph search variants such as *edge search* or *mixed search*.

graph. The graph search problem then asks for a search strategy that ensures that the entire graph is cleared while using as few searchers as possible.

In our distributed computational model each node of G has a unique identifier and also stores a number of local variables which it can manipulate. A node can read the local variables of its neighbors, i.e. the shared memory model. As is typical, we present our self-stabilizing algorithms as a set of rules where the predicate of each rule only depends on variables local to a node and those of its neighbors. Further, we assume that rules can only be executed during fixed time steps and that a distributed unfair daemon governs which node(s) get to execute a move at any given time step. This means that any non-empty subset of enabled rules may execute during a time step. Note that this is the most general type of daemon. We measure complexity both in terms of the number of rules that have been executed (moves-complexity) and also in terms of the number of rounds, where a round is the smallest sub-sequence of an execution in which every node that was eligible at the beginning of the round either makes a move or has had its predicate disabled since the beginning of the round.

The goal of a self-stabilizing algorithm is for the global system to reach a stable configuration (i.e. where no moves can be made) that conforms to a given specification, independent of its initial configuration and without external intervention. A *non-silent* self-stabilizing algorithm will also reach a configuration that conforms to the specification, but will continue to execute indefinitely once it has done so. Moreover, in the absence of transient faults, the algorithm will continue to conform to the specification.

3 The Algorithm

The overall graph searching algorithm integrates three separate self-stabilizing processes. Initially, we use the leader election algorithm from [2] to elect an $n/2$-separator as the root r (i.e. no component of $G - \{r\}$ contains more than $n/2$ nodes). Moreover, this algorithm ensures that each node knows in which direction r is. Following this, r uses a variant of the multi-wave Propagate-Information-with-Feedback (PIF) process from [1] to get the network ready for searching. This is accomplished by r continuously iterating through a circular list of its children, and for each iteration concurrently initiating two PIF processes. When r sets $child_r$ to point to a node in $N(r)$ the subtree rooted at $child_r$ transitions to the ACTIVE state while all other subtrees are either in the SLEEP state or are in the process of transitioning to SLEEP. Only nodes in the ACTIVE state can participate in the ensuing search process. When $child_r$ signals that the search of its subtree is complete and the next node in r's neighbor list signals that it is SLEEP, r will advance $child_r$ and repeat the process.

The overall process is outlined in Algorithm 1. Loosely speaking, after the system first reaches line 6 the network will have reached a "normal configuration" and thereafter the search process behaves as expected. Note that the algorithm is non-silent.

The entire process uses $5+2\delta(u)$ variables on each node u, where $\delta(u) = |N(u)|$ and $N(u)$ denotes the neighbors of u. For the leader election process we maintain

Algorithm 1. The overall graph searching process

```
1: Elect an $n/2$-separator as root $r$ and set all $p_u$ to point towards $r$; /* L1-L5 */
2: $child_r \leftarrow$ an arbitrary neighbor of $r$;                      /* R1 */
3: Signal all $v \in N(r) - \{child_r\}$ to go to SLEEP;   /* consequence of R1 */
4: loop
5:   when (Next($child_r$) is SLEEP) & ($child_r$ signals "search completed") do
6:     $prev \leftarrow child_r$; $child_r \leftarrow$ Next($child_r$);          /* R2 */
7:     do in parallel
8:       Transition subtree rooted at $child_r$ to ACTIVE;        /* T1-T4 */
9:       Transition subtree rooted at $prev$ to SLEEP;            /* T5 */
10:    Search the subtree rooted at $child_r$;                   /* S1-S4 */
```

a parent pointer p_u and for each neighbor $v \in N(u)$ a $size_u(v)$ value that will hold the size of the subgraph containing u in the original graph with the edge (u, v) removed. The results in [2] guarantee that once the election process stabilizes, all p_u and $size_u(v)$ values are correct. Since those values are not changed in any other part of the algorithm, in the absence of spurious faults they remain correct throughout. The remaining $4+\delta(u)$ variables are used in the other two processes and will be described in the follow-on subsections.

We use the leader election algorithm, which we refer to as the L-algorithm implemented as rules L1-L5, exactly as specified in [2] and therefore will not further describe it here. The next subsection explains the transition process and our implementation of it. Subsection 3.2 describes the new search process.

3.1 Transition - Lines 8 and 9

The transition processes in lines 8 and 9 together implement a full 2-wave PIF process similar to that designed by Bein, Datta, and Karaata in [1]. For each wave except the last, their algorithm broadcasts information down from the root and then propagates feedback up from the leaves. As presented in [1] the last wave is cut short, only broadcasting down. We, however, include this last propagation of feedback up, since we need the feedback to guarantee that the last broadcast reached the leaves before starting the search process.

The complete state-transition process from the perspective of one node u transitions its variable $state_u$ through a series of five states (SLEEP $\rightarrow$ AWAKE $\rightarrow$ CSIZE-UP $\rightarrow$ CSIZE-DOWN $\rightarrow$ ACTIVE) as depicted in Figure 1. After initialization, an arbitrary node u rests in the SLEEP state and expects its parent p_u to be in the SLEEP state as well. Only when u sees that p_u is AWAKE will u transition away from SLEEP, moving to the AWAKE state and thereby playing its part in a "wake-up" broadcast. Following this, u waits in the AWAKE state until all of its children transition themselves from SLEEP through AWAKE and finally to CSIZE-UP; only then will u itself transition to CSIZE-UP, thereby providing "I'm awake" feedback to p_u. The leaves are the first to switch from the broadcast down state AWAKE to the feedback up state CSIZE-UP, since they have no children. The switch from this first propagate feedback ("I'm awake") stage to the second broadcast ("get

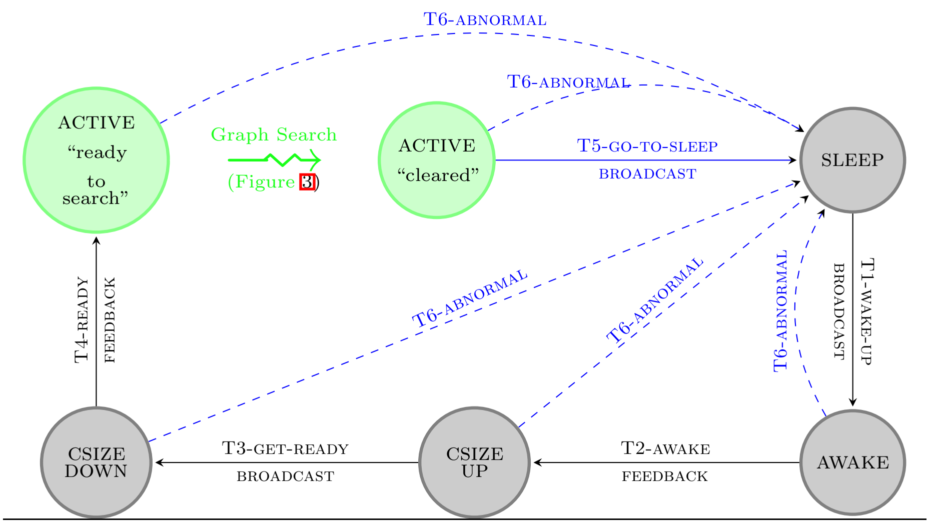


Fig. 1. The T moves process is a PIF process

ready to search") stage occurs at the root, where once its neighbor $child_r$ is in the CSIZE-UP state it effectively moves itself through CSIZE-UP to CSIZE-DOWN, thereby initiating the second wave transitioning all nodes to CSIZE-DOWN once their parent is CSIZE-DOWN. Again, the leaves turn the broadcast down wave into "I'm ready to search" feedback by transitioning through CSIZE-DOWN and to ACTIVE. Only when all of u's children are ACTIVE does u transition itself to ACTIVE. Once ACTIVE, a node is eligible to participate in the search process described in the next subsection.

If at any point during the search process u sees that p_u is no longer ACTIVE (i.e., has transitioned to SLEEP), then u itself will transition to SLEEP. During normal processing this will be initiated as a broadcast down wave from the root only when the entire subtree containing u has been cleared. If, at any point after leader election is complete, the states that p_u, u, and u's children are in do not make sense with respect to this transition process, then u will respond to the abnormal configuration by jumping to SLEEP.

There are three main tasks that we need the transition process to accomplish in order to correctly implement the search using only $\lfloor \log n \rfloor + 1$ searchers. First, the transition process computes in each node the size of the current subtree assuming the edge between the root and the subtree does not exist. These values are collectively stored in the $csize()$ vectors in exactly the same way as the $size()$ vectors holds the size of the subtrees of the entire graph. The $csize()$ values are computed in two waves. During the first feedback ("I'm awake") wave the size of the subtree below each node is calculated using the propagated up $csize()$ values from its children. Then, during the second broadcast ("get ready to search") wave the size of the subtree above each node (i.e., the portion of the subtree reached through the parent) is calculated using the propagated down $csize()$ values from p_u.

```
function ParentState $(u)$:
    if Root $(p_u)$
    then if $child_{p_u} = u$
        then if $state_u =$ CSIZE-UP
            then return CSIZE-DOWN
            elseif $state_u =$ SLEEP
            then return AWAKE
            else return $state_u$
        else return SLEEP
    else return $state_{p_u}$

function CalculateComponents $(u)$:
    foreach $v \in N(u)$
        $csize_u(v) \leftarrow 1 + \sum_{x \in N(u)-\{v\}} csize_x(u)$
    return $csize_u$

function ReadyToSearch $(u)$:
    /* check if leaf */
    if $|N(u)| = 1$
    then if $csize_{p_u}(u) = 0$
        then return true
        else return false
    $n \leftarrow csize_u(p_u) + csize_{p_u}(u)$
    if $(\forall v \in N(u),\ csize_v(u) < n/2)$
    then return true
    if $(\exists v \in N(u), csize_v(u) > n/2)$
    then return false
    if $(\exists v \in N(u), csize_v(u) = n/2)$
    then if $(ID_u > ID_v)$
        then return true
        else return false
```

Fig. 2. Auxilliary functions

The second main task that the transition process accomplishes is to initialize, during the last feedback ("I'm ready to search") wave, the other variables needed to begin the search process. Combining these first two tasks with the transition process from [1] we end up with a six rule implementation that accomplishes the following. (Red font highlights the difference between our implementation and the rules given in [1]).

- T1-WAKE-UP (rule *iB* in [1]): broadcast down a "wake-up" call (i.e., transition to AWAKE).
- T2-AWAKE (iF, lF): propagate up both "I'm awake" feedback (i.e., transition to CSIZE-UP) and the size of the subtree rooted at u.
- T3-GET-READY (iB): broadcast down both a "get ready to search" call (i.e., transition to CSIZE-DOWN) and the size of the subtree above.
- T4-READY (iF, lF): propagate up "I'm ready to search" feedback (i.e., transition to ACTIVE) and initialize variables needed for the search.
- T5-GO-TO-SLEEP (iB, lB): broadcast down a "go to sleep" call (i.e., transition to SLEEP).
- T6-ABNORMAL (iCa, lCa): jump to SLEEP if any *state* value in the neighborhood does not make sense.

We refer to these six rules as the T-algorithm. Note that for the T-algorithm u uses $1+\delta(u)$ additional variables: the state variable $state_u$ and for each neighbor $v \in N(u)$, $csize_u(v)$.

The third task that the T-algorithm accomplishes is to admit the search process in only one subtree of the root at a time. We accomplish this by using a **ParentState**(u) function to report the state of a node u's parent p_u.

If p_u is not the root, the function simply returns p_u's state. However, when p_u is the root the state it returns is dependent on whether or not $u = child_r$ (see

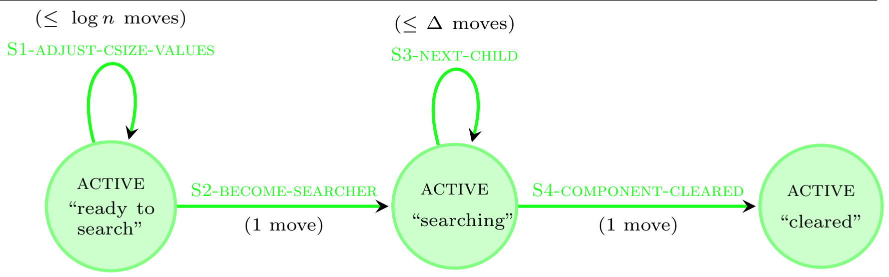


Fig. 3. The S moves process implements graph search

Figure 2 where red font again highlights differences from the implementation in [1]). To see how this works, consider a call to **ParentState**(u) when the parent p_u is the root. Then the return value is SLEEP if u is not the root's current $child_r$; otherwise the return value is the appropriate state in the transition process relative to u's current state. For example, if $u = child_r$ has not yet begun transitioning away from SLEEP then the function returns AWAKE in order to initiate a "wake-up" broadcast in u's subtree.

The results in [1] prove that rules T1 - T6 are executed in a well structured manner, giving the following result.

Lemma 1. *When the root initiates a multi-wave broadcast-feedback process in a subtree C_1 after initialization the following properties hold:*

(a) Every node in C_1 will execute, in order, rules T1, T2, T3, and then T4.
(b) For any $v \in C_1$, when v executes T2, all descendants of v will have already executed T2.
(c) All nodes in C_1 will execute T2 before any node in C_1 executes T3.
(d) For any $v \in C_1$, when v executes T3, all ancestors of v will have already executed T3.
(e) For any $v \in C_1$, when v executes T4, all descendants of v will have already executed T4.

3.2 Search - Lines 2, 3, 5, 6, and 10

The rules used to implement the search process are divided into two sets called the R-algorithm (implementing lines 2, 3, 5, and 6) and the S-algorithm (implementing line 10). The R-algorithm executes only on the root and the S-algorithm executes only on non-root nodes. As mentioned above, the S-algorithm takes place in one subtree of the root at a time. We call that subtree a component of the graph and recursively search components by placing a searcher at the $n'/2$-separator of the component, where n' denotes the number of vertices in the component. We denote such a separator as a *center* node of the component and use this to partition the current component into smaller components, searching each of these in turn. Once a component is cleared, the last searcher in that component is released before returning control to a previous level in the recursion. The root always hosts a searcher. It is this recursive behavior of splitting the graph at its center that admits a graph searching process that uses only $\lfloor \log n \rfloor + 1$ searchers.

The R- and S-algorithms use the remaining 3 variables. Boolean variables $searcher_u$ and $cleared_u$ are used respectively to indicate the presence of a searcher on u and to signal that after u's most recent T4 move u has been searched and either currently maintains a searcher as a guard or is cleared and guarded somewhere else. Finally, a pointer $child_u$ is used to point to the current neighboring subcomponent that is being searched. We also assume that each node u has a list of its neighbors beginning with **FirstChild**(u) and ending with p_u. The function **Next**$(child_u)$ advances $child_u$ through this list.

The S-algorithm includes five rules that systematically progress through the ACTIVE state as shown in Figure 3. Details are given as Algorithm 2. Rule S1 adjusts the vector $csize_u()$ using the auxiliary function **CalculateComponents** shown in Figure 2. With this the $csize_u()$ values can reflect the reduced size of the current component. When a node u has up-to-date $csize_u()$ values and it

Algorithm 2. Search process on non-root node u (S-algorithm)

```
/* S1 - S4 only possible for ACTIVE non-root with entire neighborhood
   ACTIVE                                                              */
S1-ADJUST-CSIZE-VALUES:
1: if $(\neg\, cleared_u)$ & $(\exists v \in N(u), csize_u(v) \neq 1 + \sum_{x \in N(u)-\{v\}} csize_x(u))$ then
2:     $csize_u \leftarrow$ CalculateComponents$(u)$;
S2-BECOME-SEARCHER:
1: if (ReadyToSearch$(u)$) & $(\neg\, cleared_u)$ & $(child_u = \text{NULL})$ then
2:     $searcher_u \leftarrow$ true; $cleared_u \leftarrow$ true;        /* clears $u$ */
3:     $child_u \leftarrow$ FirstChild$(u)$; $csize_u(child_u) \leftarrow 0$;
S3-NEXT-CHILD:
1: if $(child_u \neq p_u)$ & $(child_{child_u} = u)$ & $(\neg\, searcher_{child_u})$ then
2:     $child_u \leftarrow$ Next$(child_u)$; $csize_u(child_u) \leftarrow 0$;
S4-COMPONENT-CLEARED:
1: if $(child_u = p_u)$ & $(child_{p_u} = u)$ & $(searcher_u)$ & $(searcher_{p_u})$ then
2:     $searcher_u \leftarrow$ false; /* edge $(u, p_u)$ is cleared; release $u$'s searcher */
S5-ABNORMAL:
1: if $(child_u \notin N(u) \cup \{\text{NULL}\})$ || $((child_u \in N(u))$ & $(csize_u(child_u) \neq 0))$
     || $((child_u = \text{NULL})$ & $(searcher_u$ || $cleared_u))$
     || $((child_u \in N(u))$ & $(p_{child_u} = u)$ & $(child_{child_u} = u)$ & $(\neg\, searcher_u))$ then
2:     $child_u \leftarrow p_u$; $csize_u(p_u) \leftarrow 0$;
3:     $searcher_u \leftarrow$ false; $cleared_u \leftarrow$ true;
```

is the center of the current component, the Boolean function **ReadyToSearch** given in Figure 2 evaluates to true and S2 is enabled on u. An S2 move places a searcher on the node u that is executing the move and starts the search process in one of its neighboring subcomponents (the one pointed to by $child_u$). The S3 move is enabled only after the current $child_u$ subcomponent is cleared, at which time u advances its child pointer to the next child. After u has progressed through each of its non-parent neighbors, S3 is no longer enabled. The S4 move is used to release the searcher on u only after all children components are cleared and the parent p_u also has a searcher on it to guard u's cleared component. Note that just before the S4 move releases the searcher on u, we also know that the edge (u, p_u) has been cleared.

After initialization in the root r's neighborhood (either explicitly through execution of R1 or implicitly with initial values in $N(r)$), the R-algorithm is a continuous series of R2 moves executed by r, each followed by a period of waiting until the current $child_r$ is cleared. An ACTIVE $child_r$ and the subtree rooted at $child_r$ is considered to be cleared when $child_{child_r} = r$ and $child_r$ has released its searcher (i.e., $\neg searcher_{child_r}$). Details are given as Algorithm 3.

Algorithm 3. Search process on root node u (R-algorithm)

```
/* R1 - R2 only possible for root with consistent size values in
   neighborhood                                                   */
R1-ROOT-INITIALIZE:
1: if $(\neg\, searcher_u)$ & $(child_u \notin N(u))$ & $(\exists v \in N(u), csize_u(v) \neq 0)$ then
2:     $searcher_u \leftarrow$ true;                 /* clears $u$ */;
3:     $\forall v \in N(u)$, $csize_u(v) \leftarrow 0$;
4:     $child_u \leftarrow$ FirstChild$(u)$;

R2-ROOT-NEXT-CHILD:
1: if $(child_{child_u} = u)$ & $(\neg\, searcher_{child_u})$ & $(state_{child_u} =$ ACTIVE$)$ &
   $(state_{\mathbf{Next}(child_u)} =$ SLEEP$)$ then
2:     $child_u \leftarrow$ Next$(child_u)$;
```

4 Correctness

We show the correctness of our algorithm in five parts. The first part is an immediate consequence of the results in [2], where they prove that at most $O(n^2)$ time steps (or h rounds, where h is the diameter of G) contain L moves. Thus if we can show that at any time prior to stabilization of the L-algorithm the R-, T- and S-algorithms will stabilize given that there are no L moves, then it will follow that the L-algorithm will stabilize with the $n/2$-separator as the global root, designated by r, and with every parent pointer p_v set appropriately. The fact that the R-, T- and S-algorithms stabilize will follow from the remainder of the correctness proof.

Next we argue in Subsection 4.1 below that the R-algorithm behaves as expected, assuming the T- and S-algorithms stabilize in the absence of any L- or

R moves. This argument shows that after stabilization of the L-algorithm r will begin executing R2 moves and that just prior to any R2 move, $state_y = \text{SLEEP}$ for $y = \mathbf{Next}(child_r)$. The results in [1] then guarantee that every node in the subtree rooted at y will be in the SLEEP state before it begins the T-algorithm that will be initiated by r with the next R2 move.

We then show in Subsection 4.2 that each R2 move will initiate the T-algorithm in the subtree rooted at the assigned-in-R2 pointer $child_r$ and that each node in that subtree will have the correct initial values for the S-algorithm before it is eligible to execute any S move. Subsection 4.3 then addresses the key correctness result, proving that starting from a normal configuration the S-algorithm correctly searches the subtree rooted at $child_r$ using no more than $\lfloor \log n \rfloor$ searchers at any point in time. For the final portion of the correctness proof Subsection 4.4 argues that the T- and S-algorithms will stabilize correctly in the absence of any L or R moves regardless of the initial configuration and that before the L-algorithm stabilizes, the R-algorithm will stabilize in the absence of any L moves.

Note that the premise required for the first part, that before stabilization of the L-algorithm the R-, T- and S-algorithms stabilize in the absence of any L moves, follows from these results. That is, the results in Subsections 4.2 and 4.3 ensure, respectively, that starting from a normal configuration the T-, and S-algorithms stabilize. The results in Subsection 4.4 ensure that before the L-algorithm stabilizes and/or starting from an abnormal configuration all three algorithms stabilize.

Due to space limitations we do not give the details of most proofs. Details are, however, available upon request.

4.1 The R-algorithm

We focus in this subsection on R moves made after the L-algorithm has stabilized with the $n/2$-separator as r.

Only a node believing it is r is enabled to execute any R move. Moreover, since no T- or S move is privileged on r, R moves are the only possible moves r can make. Furthermore, because the T- and S-algorithms will stabilize (Subsections 4.2 and 4.3), we know that r will be given an opportunity to execute enabled R moves. In light of this, we prove here the following lemma.

Lemma 2. *After the L-algorithm has stabilized:*

(a) r will execute R2 with appropriate variable values and
(b) just before any R2 move, $state_y = \text{SLEEP}$ for $y = \mathbf{Next}(child_r)$, ensuring that every node v in the subtree rooted at y is either in the SLEEP state or will transition to SLEEP before any subsequent S move.

4.2 The T-algorithm

In this subsection we consider what happens just after r executes an R2 move. Let C be the component of $G-\{r\}$ containing $child_r$ following the R2 move. We will show that each $v \in C$ will execute the T-algorithm and that, after v's last T move, its variables will be correctly initialized for the start of the S-algorithm. To that end, we say that a $csize_v(x)$ value is *correct-with-respect-to* C if $v \in C$ and $csize_v(x)$ is equal to the size of the component in $C \backslash \{(v, x)\}$ containing v.

From Lemma 2 we know that each node in C will either be in the SLEEP state just after r makes the R2 move, or will subsequently transition to SLEEP before its parent (and hence it) can begin the T-algorithm. Then it follows from Lemma 1 that each node in C will perform T1 through T4 in sequence. This is the foundation needed to prove that the configuration at each $v \in C$ following its T4 move is what is needed before it begins the S-algorithm.

Lemma 3. *Starting from a normal configuration, after a vertex $v \in C$ makes a T4 move and before it makes any S move, the following properties hold:*

(a) $cleared_v =$ false.
(b) $searcher_v =$ false.
(c) $child_v =$ NULL.
(d) For all $x \in N(v)$, $csize_v(x)$ and $csize_x(v)$ are correct-with-respect-to C.

4.3 The S-algorithm

In this subsection we will show that once r has executed an R2 move setting $child_r = v$ the subtree C rooted at v will be searched and cleared. Moreover, the search will stabilize with v's variable values enabling an R2 move on r, thus continuing the search in G. We will also show that the process of searching C never uses more than $\log n$ searchers simultaneously and that the total number of moves is at most $O(|C| \log |C|)$.

It follows from the discussion in the previous section that every node in C will execute the entire T-algorithm before it can start executing the S-algorithm. Thus even though both the T- and S-algorithms might execute concurrently any node that has not yet finished the T-algorithm is eligible to continue executing the T-algorithm. Also, no values that are set by the S-algorithm are used in the predicates of the T-algorithm. Note in particular that $csize()$ values used by the T-algorithm are only influenced by other $csize()$ values from nodes where the T-algorithm is still running. We will therefore assume that the T-algorithm has run to completion on C when the S-algorithm starts. Thus we assume that when the S-algorithm starts we know from Lemma 3 that all $csize()$ values are correct-with-respect-to C, $searcher_v =$ false and $child_v =$ false for each $v \in C$. The value of $searcher_v$ can only be set to true if v executes an S2 move. To keep track of the searchers that are used at any given time, we will label a node that executes an S2 move as an L_i searcher where $i - 1$ is the number of searchers in C just prior to the move. For completeness we define $L_0 = r$.

It is conceivable that more than one node in C could execute an S2 move during the same time step and thus we could have more than one L_i searcher at the same time. But, as we will show, only one node in C can execute an S2 move during any time step. We will also show that the searchers are placed and removed in a first-in-last-out fashion. Thus every L_j, $j < i$, will still be a searcher when L_i ceases to be so.

With these assumptions we define the *components* of $G - \{L_0, \ldots, L_{i-1}\}$ that are incident on any L_{i-1} as the C_i components of G. It follows then that C is a C_1 component. Again for completeness we define $C_0 = G$. Additionally, if the value of $child_{L_{i-1}}$ is pointing to a node $v \in C_i \cap N(L_{i-1})$ with $csize_{L_{i-1}}(v) = 0$, then we denote the C_i component containing v as the *active* C_i component. As we will show all active components are in C and are nested within each other.

The *center* of a component C_i, denoted by $center(C_i)$, is the $|C_i|/2$-separator. Ties are broken using the highest ID. Let C_i be a component and let $v \in C_i$ and $x \in N(v)$. Then the *actual size* of the subtree containing v in $C_i - \{x\}$, is denoted by $actual_v^i(x)$.

Again, looking at $C = C_1$ it is clear that until r makes a subsequent R2 move, C_1 remains active and that following the T-algorithm $csize_v(x) = actual_v^1(x)$ for every $v \in C_1$. Before proceeding we need the following result about how S1 moves will update the $csize()$ values in an active component.

Lemma 4. *Let $x_0 = L_{i-1}$ be incident on an active component C_i and let $x_1, x_2 \cdots, x_j$ be any path in C_i where the following properties hold:*

- *$x_1 \in N(x_0)$ and $csize_{x_0}(x_1) = 0$.*
- *$csize_{x_k}(x_{k+1}) > B$ for some constant $B \geq |C_i|$ and each k, $0 < k < j$.*
- *$csize_y(x_k)$ is correct-with-respect-to $C_i - \{x_k\}$ for each x_k, $0 < k \leq j$, and $y \in N(x_k)$ where y is not on the path $x_0, \cdots, x_j$.*

Assume further that the only type of moves that are executed on any node are S1 moves by the nodes $x_1, x_2, \cdots, x_j$. Then for each x_k, $0 < k \leq j$:

(a) The S1 moves will stabilize with $csize_{x_k}(x_{k+1}) = actual_{x_k}^i(x_{k+1})$.
(b) At each time step prior to the move where $csize_{x_k}(x_{k+1})$ obtains its final value $csize_{x_k}(x_{k+1}) > B$.

Lemma 4 identifies the very structured way in which any $csize_v(x)$ value will evolve with a series of S1 moves in the absence of any other S moves; essentially they will decrease, jumping from one actual value to some subsequent actual value, ending when v becomes the center of an active component. As it turns out, this characteristic of the changing $csize_v(x)$ values persists even in the presence of other S moves within the active components.

We will use the following defined characteristic of an active component when we later show how searchers are activated.

Definition 1. *An active component C_i where $|C_i| > 0$ is* ready-for-searching *if the following is true immediately following the S2 or S3 move by an L_{i-1} node that defined C_i:*

(a) Every L_j, $0 \leq j < i$, such that there exists an edge (L_j, v) for some $v \in C_i$ has $child_{L_j} = v$ and $csize_{L_j}(child_{L_j}) = 0$.
(b) For every $v \in C_i$, $cleared_v =$ false.
(c) For each $v \in C_i$ let $x \in N(v)$ be the neighbor of v on the path from v to L_{i-1}. Then $csize_v(x) = actual_v^{i-1}(x)$.

(d) *For each $v \in C_i$ let $x \in N(v)$ be a neighbor of v not on the path from v to L_{i-1}. Then $csize_v(x) \geq |C_i|$.*

Note that following the T-algorithm on $C = C_1$ all $csize_v()$ for $v \in C_1$ are correct-with-respect-to C_1. Thus the only move that can be executed in C_1 is S2 by $center(C_1)$ which then becomes the first (and currently only) L_1 node. If $|C_1| > 1$ then $child_{L_1}$ is set to some $v \in C_1$ thus defining one or more C_2 components. Since at that time $csize_{L_1}(v) = 0$ it follows that the C_2 component of $C_1 - \{L_1\}$ containing v is also active. Moreover, it is not hard to see that this C_2 component is ready-for-searching. Since the $csize()$ values surrounding any other C_2 components have not changed these will remain stable until L_1 makes a subsequent S3 move, changing a $csize()$ value next to the other C_2 component.

As the next lemma shows, the S-algorithm will recursively continue to create new nested components that are ready-for-searching. Moreover, it will do so in a sequential fashion.

Lemma 5. *When the S-algorithm runs on a ready-for-searching component C_i, $i > 1$:*

(i) The first node in C_i to execute an S2 move will be $center(C_i)$.
(ii) If $|C_i| > 1$ then the C_{i+1} component that $child_{center(C_i)}$ points to just after its S2 move will be ready-for-searching at that time.
(iii) The nodes in $C_i - C_{i+1} - \{center(C_i)\}$ can only execute S1 moves until $center(C_i)$ makes an S3 move.

Since L_1 is the only initial searcher in C_1 it follows from Lemma 5 that the S-algorithm will continue to create new nested components $C_2, \ldots, C_i$ one at a time until either a node L_i is a leaf or a node L_i sets $child_{L_i} = L_j$ where $j < i$. In both cases, L_i then sees $|C_{i+1}| = 0$. In the following we show how the recursion returns and that when it does so each component has been cleared and will not be recontaminated. First we need the following definition of a subtree that has been searched.

Definition 2. *Let $(v, w) \in E$ be where $w = p_v$. The subtree H of $G - \{w\}$ with v as its root is* cleared-and-guarded *if:*

- *For every $x \in V(H) - \{v\}$ $searcher_v =$ true and $searcher_x =$ false,*
- *Every $x \in V(H)$ has $child_x = p_x$, and*
- *Every $x \in V(H)$ has been cleared and no recontamination has occurred.*

Note that it is only the node v closest to r that can possibly make a move in a subtree that is cleared-and-guarded and this can only happen if p_v has $child_{p_v} = v$, at which point v is eligible to execute an S4 move.

We can now show that the S-algorithm, when started correctly on a component C_i, will return with the entire C_i being cleared-and-guarded.

Lemma 6. *Let C_i be a component of G such that C_i is ready-for-searching and every subgraph of $G - C_i$ adjacent to C_i except possibly the one closest to r*

is cleared-and-guarded. Further, let $(v, w) \in E$ be such that $v \in C_i$ and $w \notin C_i$ while $p_v = w$. Then following the S2 or S3 move that defined C_i, the S-algorithm will reach a point where the component of $G - \{w\}$ containing v, will be cleared-and-guarded. In doing so the algorithm will not use more than $\lfloor \log |C_i| \rfloor + 1$ new searchers simultaneously.

Because each S2 and S3 move designates exactly one of its adjacent subcomponents to next recursively start the S-algorithm, we can prove the following.

Lemma 7. *The number of moves executed by the S-algorithm on a component C_i which satisfies the conditions for Lemma 6 is $O(|C_i| \log |C_i|)$.*

Note that the proof of Lemma 7 establishes that, for any $v \in C_i$, the number of times that v executes each S move is as specified in Figure 3.

4.4 Initialization

In this subsection we show that the algorithm will reach a normal configuration. We do this by first showing that in the absence of any L, R and T moves the S-algorithm will stabilize, regardless of the initial configuration. This result, together with the results in [1], ensures that in the absence of any L or R moves, the T- and S-algorithms together will stabilize in a normal configuration. Our other initialization result, Lemma 9, uses this to then show that in the absence of any L moves, the combined R-, T- and S-algorithms stabilize in a normal configuration. Since the results in [2] guarantee that as long as this is the case the L-algorithm will stabilize we will then have proven that the integrated self-stabilizing algorithm that includes the five L rules, the two R rules, the six T rules and the five S rules will reach a normal configuration.

Lemma 8. *Let $H = (V_H, E_H)$ be a maximal connected subgraph of G such that every $v \in V_H$ has $state_v =$ ACTIVE and let $n = |V_H|$. Then in the absence of any L, T or R moves, the S-algorithm will stabilize in H using $O(n^2)$ moves.*

Lemma 9. *Let $v =$ **FirstChild**(r) after an R1 move by r and let C_1 be the component of $G - \{r\}$ containing v. Then in the absence of any L moves, the T- and S-algorithms will stabilize in C_1 with $child_v = r$ and $searcher_v =$ false.*

4.5 The Integrated Algorithm

We are now ready to prove the final results.

Theorem 1. *Starting from an arbitrary configuration the L-, T-, R- and S-algorithms combined reach a point when the $n/2$-separator of G is r, all p_v point in the direction of r, and the entire network is in a normal configuration when the r executes the R2 move.*

Proof. Follows from Lemmas 2, 8, 9 and the results in [2] and [1].

Theorem 2. *After reaching a normal configuration, following any R2 move on r the combined L-, T-, R- and S-algorithm will clear all of G in $O(n \log n)$ moves using no more than $1 + \lfloor \log n \rfloor$ searchers.*

5 Concluding Remarks

We have given an efficient non-silent self-stabilizing algorithm for graph searching in trees. The algorithm integrates three separate self-stabilizing processes and ensures that each behaves as expected even in the presence of the other processes.

Although as presented the algorithm addresses the node search problem, it can be easily modified to perform edge searching or mixed searching. In the edge search variant searchers are slid through the edges. Let us consider a node u and its parent v. If v has a searcher on it and u is the next node that will receive a searcher then instead of placing the searcher directly on u we can place the searcher on v and then slide it to u. By doing this we can transform the node search strategy to an edge search strategy.

Due to the sequential nature of our algorithm, the number of rounds for it to execute could be on the same order as the number of moves.

Our algorithm can be used to solve other types of problems that require recursive decomposition of the graph by identifying the centers at each level of the decomposition. For example, it is straight-forward to adapt our algorithm to find a 2-center of a tree using only $O(n)$ moves after initialization, improving over the algorithm given in [5].

References

1. Bein, D., Datta, A.K., Karaata, M.H.: An optimal snap-stabilizing multi-wave algorithm. The Computer Journal 50, 332–340 (2007)
2. Blair, J.R.S., Manne, F.: Efficient self-stabilizing algorithms for tree networks. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 912–921 (2003)
3. Coudert, D., Huc, F., Mazauric, D.: A distributed algorithm for computing and updating the process number of a forest. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 500–501. Springer, Heidelberg (2008)
4. Fomin, F.V., Thilikos, D.M.: An annotated bibliography on guaranteed graph searching. Theor. Comput. Sci. 399, 236–245 (2008)
5. Huang, T.C., Lin, J.C., Chen, H.J.: A self-stabilizing algorithm which finds a 2-center of a tree. Computers and Mathematics with Applications 40, 607–624 (2000)
6. Kinnersley, N.G.: The vertex separation number of a graph equals its path-width. Inf. Process. Lett. 42, 345–350 (1992)
7. Korach, E., Solel, N.: Tree-width, path-width, and cutwidth. Discrete Appl. Math. 43, 97–101 (1993)
8. Mihai, R., Mjelde, M.: A self-stabilizing algorithm for graph searching in trees. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 563–577. Springer, Heidelberg (2009)
9. Peng, S., Ho, C., Hsu, T., Ko, M., Tang, C.: Edge and node searching problems on trees. Theor. Comput. Sci. 240, 429–446 (2000)
10. Skodinis, K.: Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. J. Algorithms 47, 40–59 (2003)

Adaptive Containment of Time-Bounded Byzantine Faults

Yukiko Yamauchi[1], Toshimitsu Masuzawa[2], and Doina Bein[3]

[1] Nara Institute of Science and Technology, Japan
[2] Osaka University, Japan
[3] The Pennsylvania State University, USA
y-yamauchi@is.naist.jp, masuzawa@ist.osaka-u.ac.jp, siona@psu.edu

Abstract. In this paper, we introduce a novel Byzantine fault model called *time-bounded Byzantine fault* that imposes an upper bound on the number of malicious actions of a Byzantine faulty process. We also propose a new method for adaptive fault-containment against time-bounded Byzantine faults that guarantees that the number of perturbed processes depends on the number of malicious actions at Byzantine processes. The proposed information diffusion method imposes k consecutive state changes on a process so that the process diffuses information to processes at distance k. We present an example of a leader election protocol to show the adaptive containment of the proposed method.

Keyword: Distributed system, fault-tolerance, self-stabilization, Byzantine fault, fault-containment, leader election.

1 Introduction

A distributed system consists of a collection of processes that communicate with each other so that the entire system satisfies a given specification. In large-scale networks such as P2P networks, mobile ad hoc networks, and wireless sensor networks, it is expected that the system guarantees scalability, reliability, availability, etc. Fault-tolerance is one of the main challenges in the design of distributed systems because a distributed system is more prone to faults such as memory crashes and malicious users as the number of processes increases.

Self-stabilization provides an excellent autonomous adaptability against any finite number of transient faults. A transient fault changes the memory contents at processes arbitrarily. A self-stabilizing protocol promises that the system eventually satisfies its specification and after that, it never violates its specification. Since Dijkstra first introduced the notion of self-stabilization [4], many self-stabilizing protocols were proposed [5,12].

Though self-stabilization was originally designed for transient faults, the worst fault model for self-stabilization is the Byzantine fault that allows arbitrary (malicious) actions at faulty processes [9,10,11,14]. Self-stabilization assumes that during the convergence, all processes behave according to the protocol while

Byzantine processes continue their malicious actions during and after convergence. Hence, the goal of Byzantine fault resilient self-stabilization is to contain the effect of malicious actions of Byzantine processes to a bounded number of non-Byzantine processes. Existing Byzantine fault resilient self-stabilization protocols are designed for unbounded number of malicious actions at Byzantine processes [9,10,11].

We focus on a weaker type of Byzantine fault model, called *time bounded Byzantine fault* (TB-Byzantine fault, for short) in which the number of malicious actions of a faulty process is finite. It is obvious that a self-stabilizing protocol eventually satisfies its specification after a finite number of malicious actions. We focus on the fault-containment against TB-Byzantine processes and propose an adaptive containment method in which the scale of perturbation caused by Byzantine faulty processes depends on the number of malicious actions of Byzantine processes.

Related work. Though self-stabilization is a brilliant design paradigm for reliability against any finite number of transient faults, it allows the effect of a single transient fault to spread over the entire network. In the context of self-stabilization, a system configuration satisfying the system specification is called a *legitimate configuration*. Many researchers focused on localization of a fault in a legitimate configuration: time-adaptive self-stabilization [7,3], fault-containing self-stabilization [6], local stabilizer [1], and fault-local distributed mending [8]. However, these papers focus on a single transient fault in a legitimate configuration that changes the memory contents at process(es) just once.

The challenge against Byzantine processes is how to obtain and keep consensus among correct processes in the presence of Byzantine faulty processes because each Byzantine faulty process takes malicious actions during and after convergence. Nesterenko and Arora proposed the notion of *strict stabilization* that contains the effect of a Byzantine faulty process to a constant distance from the process and the remaining processes realize self-stabilization property [11]. The basic method is to discard fictitious information locally at each process. They proposed a strict stabilizing protocol for vertex coloring problem and dining philosophers problem. Masuzawa and Tixeuil proposed a strict stabilizing protocol for link coloring protocol [9]. Their strategy is to put a priority on the communication to delay messages from Byzantine processes and to process the messages from correct processes first. Masuzawa and Tixeuil also proposed the notion of *strong stabilization* by relaxing requirements for strict stabilization [10]: permanent influence of Byzantine processes are contained within their neighbors and temporary influence is allowed to spread over the network. However, all these papers focus on unbounded Byzantine faults in which a faulty process permanently takes malicious actions. Additionally, these existing strict stabilizing protocols except [10] are designed for local distributed problems that need consensus among direct neighbors.

There exist many variants of Byzantine processes for non-self-stabilizing protocols, such as mortal Byzantine faults [13] and Byzantine faults with recovery [2]. They are motivated by the fact that the assumption of classical Byzantine fault model is too strong, because once a process takes an arbitrary behavior, the process is considered to be faulty forever.

Our contribution. In this paper, we first introduce a novel fault model called *time bounded Byzantine fault* (TB-Byzantine fault, for short) in which the number of malicious actions is finite. Then, we propose a novel fault-containment mechanism against TB-Byzantine faults, called *pumping*. The proposed protocol provides *adaptive fault-containment* in the sense that when each of f TB-Byzantine processes changes its state at most k times during recovery, the effect is contained to at most $f \cdot k$ processes.

The TB-Byzantine fault model is a subclass of the Byzantine fault model. However, this bounded fault model is worth considering from a practical aspect. For a Byzantine process, the number of malicious actions is a critical cost because if a process repeats malicious actions, then it can be detected by an outside observer or users of the system. Additionally, in practice, in a wireless communication network, a message transmission following a malicious action drains the battery at the TB-Byzantine process and in a cellular network, the carrier charges fee for bandwidth usage caused by any message transmission.

We focus on the leader election problem on an oriented ring which is a global distributed problem. More specifically, we focus on information diffusion in the presence of TB-Byzantine processes. Each process broadcasts its ID to all other processes and selects as the leader the process with the minimum ID among the received IDs. However, a TB-Byzantine process may diffuse fictitious minimum ID to prevent correct processes from electing a leader. To contain the effect of TB-Byzantine processes, it is necessary to contain the information diffused by TB-Byzantine faults.

In the proposed method, a novel mechanism called pumping is introduced, which requires a process to keep on changing its state to diffuse information to distant processes. When a process wants to diffuse information to a process at distance k, the process has to change its state k times. Hence, a TB-Byzantine process has to make k malicious actions to diffuse a fictitious minimum ID to a process at distance k. By imposing state changes as a cost for diffusing information, we achieve the containment of the effect of a TB-Byzantine process. Though we focus on the leader election problem on an oriented ring, the pumping mechanism is easily applied to any distributed problem on any topology.

Organization of this paper. In Section 2, we define the system model and the TB-Byzantine fault. In Section 3, we show an adaptive fault-containing leader election protocol on an oriented ring. The correctness proofs and performance evaluation are shown in Section 4. We conclude this paper with Section 5.

2 Preliminary

2.1 System Model

A system is represented by a graph $G = (V, E)$ where the vertex set V is the set of processes and the edge set $E \subseteq V \times V$ is the set of bidirectional communication links. Two processes P_i and P_j are *neighboring* if $(P_i, P_j) \in E$. The distance between two processes is the length of the shortest path between the two.

Each process P_i has a unique ID denoted by ID_i and maintains a set of local variables; a subset of the local variables at each process is called the *output variables.* The state of a process is an assignment of values to all local variables. We adopt the *state reading model* for communication among processes. Process P_i can read the values of the local variables at its immediate neighbors, while P_i can change the values of local variables at P_i.

Each process P_i changes its state according to a protocol that consists of a finite number of guarded actions of the form $\langle label \rangle : \langle guard \rangle \to \langle action \rangle$. A *guard* is a Boolean expression involving the local variables of the process and of its neighboring processes. An *action* is a statement that changes the values of the local variables at P_i. A guard is *enabled* if it is evaluated to *true.* A process with an enabled guard is called *enabled.*

A *configuration* of a system is an assignment of all local variables of all processes. A *schedule* of a distributed system is an infinite sequence of sets of processes. Let $S = R^1, R^2, \cdots$ be a schedule where $R^i \subseteq V$ holds for each i $(i \geq 1)$. For a process set R and two configurations σ and σ', we denote $\sigma \xrightarrow{R} \sigma'$ when σ changes to σ' by executing an action of each process in R simultaneously. If a selected process has no enabled guard then it does not change its state. On the other hand, if a selected process has multiple enabled guards, then the process executes the action corresponding to only one of the enabled guards. We assume that the selection of enabled guards (and its action) is weakly fair, *i.e.,* a guard enabled infinitely often is selected infinitely often. The evaluation of guards and the execution of the corresponding action are *atomic*, *i.e.*, these computations are done without any interruption. An infinite sequence of configurations $E = \sigma_0, \sigma_1, \cdots$ is called an *execution* from an initial configuration σ_0 by schedule S if $\sigma_i \xrightarrow{R^{i+1}} \sigma_{i+1}$ holds for each $i \geq 0$. We say a process in R^{i+1} is *activated* in configuration σ_i.

We adopt the *distributed daemon* as a scheduler that allows any subset of processes to execute actions simultaneously. The distributed daemon is *weakly fair*, that is, if a process is enabled infinitely often, then the process is activated infinitely often.

A distributed daemon allows *asynchronous* executions. In an asynchronous execution, the time is measured by *rounds.* Let $E = \sigma_0, \sigma_1, \cdots$ be an asynchronous execution by schedule $S = R^1, R^2, \cdots$. The first round $\sigma_0, \sigma_1, \cdots, \sigma_j$ is the minimum prefix of E such that $\bigcup_{i=1}^{j} R^i = V$. The second round and the latter rounds are defined recursively by applying the definition of the first round to the remaining suffix $E' = \sigma_j, \sigma_{j+1}, \cdots$ and $S' = R^{j+1}, R^{j+2}, \cdots$.

A *Byzantine faulty process* behaves arbitrarily and independently from the protocol. A state change at a process is a *malicious action* if and only if the state change does not conform to the protocol, otherwise a *normal action.* When a Byzantine process P_i is activated, P_i consumes one normal action or one malicious action. If a Byzantine process does not change its state by ignoring the behavior of the protocol when it is activated, P_i consumes one malicious action.

A *time-bounded Byzantine fault* (*TB-Byzantine fault* for short) is a subclass of Byzantine fault model such that the number of malicious actions at each Byzantine process is finite. An execution $E = \sigma_0, \sigma_1, \cdots$ is *(f, k_f)-faulty* if and only if the number of Byzantine processes is f and the number of malicious actions at each Byzantine process is at most k_f. For an (f, k_f)-faulty execution, a Byzantine process is called k_f-TB-Byzantine process. An (f, k_f)-faulty execution contains at most $f \cdot k_f$ malicious actions. An execution is *correct* if it contains no malicious action.

2.2 Self-stabilization and Fault-Containment

In this paper, we focus on autonomous adaptability of a protocol in two aspects: *self-stabilization* and *fault-containment*. A self-stabilizing protocol guarantees that in any correct execution, the system eventually satisfies its specification. Self-stabilization promises autonomous adaptability against any finite number of any type of faults by considering the configuration obtained by the last fault as the initial configuration. Hence, the stabilization in the presence of a bounded number of malicious actions (*i.e.*, TB-Byzantine processes) is clear. However, self-stabilization guarantees nothing in the presence of faults during convergence, and even for faults in a legitimate configuration, it does not provide containment of the effect of the fault.

A *problem* (*task*) $\mathcal{T}$ is defined by a *validity predicate* on output variables at all processes. Intuitively, a configuration of a protocol is legitimate if it satisfies the validity predicate of $\mathcal{T}$. However, the protocol may have local variables other than the output variables and the legitimacy of its configuration may depend on the values of all the local variables (including the output variables). Hence, we define a *legitimate configuration* σ of a protocol $\mathcal{P}_\mathcal{T}$ for problem $\mathcal{T}$ as the one such that any configuration (including σ itself) appearing in any correct execution starting from σ satisfies the validity predicate of $\mathcal{T}$. The set of legitimate configurations are denoted by $\mathcal{C}_L(\mathcal{P}_\mathcal{T})$. (We omit $\mathcal{P}$ and $\mathcal{T}$ when they are clear.)

Definition 1. *Self-stabilization*
Protocol $\mathcal{P}$ is self-stabilizing for a problem $\mathcal{T}$ if the system eventually reaches a legitimate configuration of $\mathcal{P}$ for $\mathcal{T}$ in any correct execution starting from any configuration.

The *convergence time* is the maximum (worst) number of rounds that is necessary for the system to reach a legitimate configuration in any correct execution starting from any configuration.

As stated above, any self-stabilizing protocol eventually reaches a legitimate configuration in any (f, k_f)-faulty execution. However, malicious actions during convergence may delay the convergence to a legitimate configuration. To measure the disturbance of the convergence by malicious actions, we introduce the disruption as follows. For an (f, k_f)-faulty execution $E = \sigma_0, \sigma_1, \cdots$, the *disruption* is the minimal prefix of E, denoted by $E' = \sigma_0, \sigma_1, \cdots \sigma_j$ such that σ_j is a legitimate configuration. The disruption of E represents the convergence in spite of or after malicious actions. A disruption is called (f', k'_f)-disruption if it contains malicious actions of f' processes and at most k'_f malicious actions for each of the processes. Hence, in an (f', k'_f)-disruption, there are at most $f' \cdot k'_f$ malicious actions. An (f, k_f)-faulty execution can contain an (f', k'_f)-disruption for $f' \leq f$ and $k'_f \leq k'_f$. The (f', k'_f)-*disruption time* is the maximum (worst) number of rounds of any (f', k'_f)-disruption. We note that when $f' \cdot k'_f = 0$, the execution is correct and the disruption time is equal to the convergence time.

We consider fault-containment of TB-Byzantine faults in a legitimate configuration. In other words, we consider an (f, k_f)-faulty execution that starts from a legitimate configuration immediately followed by at least one malicious actions (*i.e.*, the first transition contains at least one malicious action). It is expected that the effect of TB-Byzantine processes does not spread over the entire system and is contained to a restricted number (or distance) of processes around the faulty process.

Consider an (f, k_f)-faulty execution $E = \sigma_0, \sigma_1, \cdots$ starting from a legitimate configuration σ_0 immediately followed by at least one malicious actions. The *perturbation* of E is the minimal prefix of E, denoted by $E' = \sigma_0, \sigma_1, \cdots \sigma_j$ $(j \geq 1)$ such that σ_j is a legitimate configuration. We say that a correct process is *perturbed* in E' if and only if the process changes its output in E'. When the perturbation contains malicious actions of f' processes and at most k'_f malicious actions for each of the processes, we call it (f', k'_f)-*perturbation.* The (f', k'_f)-*perturbation number* is the maximum (worst) number of perturbed processes in any (f', k'_f)-perturbation. The (f', k'_f)-*perturbation time* is the maximum (worst) number of rounds of any (f', k'_f)-perturbation. Note that we define the perturbation number based only on the output variables.

Definition 2. *TB-Byzantine resilient fault-containment*
A self-stabilizing protocol $\mathcal{P}$ is TB-Byzantine resilient fault-containing if (f', k'_f)-perturbation number depends on $\min\{f' \cdot k'_f, n\}$ *and/or $(f' \cdot k'_f)$-perturbation time depends on* $\min\{f' \cdot k'_f, n\}$.

3 Proposed Method

3.1 Overview

In this paper, we consider the *leader election problem* on an oriented (but bidirectional) ring based on ID diffusion. The *leader election problem* is to make all processes in the system recognize a single process that is called *leader*. A straightforward strategy for the leader election problem is the minimum (or maximum) ID finding: Each process broadcasts its ID to all other processes and among received IDs, it selects a process with the minimum (or maximum) ID as the leader. Our proposed leader election protocol is based on this simple strategy.

Because we focus on the proposed containment method, for simplicity, we use an oriented ring $G = (V, E)$ of n processes. Each process P_i in V has two neighbors $P_{i-1 \bmod n}$ and $P_{i+1 \bmod n}$ $(i \in [0..n-1])$[1]. P_{i-1} (P_{i+1}) is called *predecessor* (*successor*, respectively) of P_i. The output variable at each process is the leader's ID variable. Additionally, each process maintains local variables to store the IDs of other processes in the system. In a legitimate configuration, each process stores only the IDs of all processes in V and its leader's ID variable takes the smallest ID in the ID list of the process. Note that in a legitimate configuration, no process stores an ID of a non-existent process.

A malicious action at each TB-Byzantine process can perturb the entire network in two ways: First, it can diffuse a fictitious ID of any non-existent process. If the fictitious ID is smaller than any IDs of existing processes, the ID is chosen as the leader's ID at each process and forwarded to the entire network. Secondly, a TB-Byzantine process can stop forwarding the smallest ID that should be chosen as the leader's ID. To avoid choosing an incorrect ID as the leader's ID, self-stabilization requires processes to repeatedly check the consistency of the chosen ID.

To achieve fault-containment, we introduce a mechanism called *pumping*, which makes each process keep on changing its state to push an ID further and further. This forces a TB-Byzantine process to consume a malicious action in pushing a faulty ID one hop further and in stopping forwarding a received ID.

The proposed leader election protocol $\mathcal{PLE}$ is based on an information diffusion part $\mathcal{PUMP}$ based on pumping mechanism which is the main challenge against TB-Byzantine faults in this paper. $\mathcal{PLE}$ provides minimum ID finding among received IDs which also needs careful design for fault-containment.

Our strategy for fault-containment is to impose each process P_i to change its state k times in order to diffuse ID_i to all process at distance at most k. This pumping method is implemented with a sequence of waves. Each wave generated at P_i is a message that consists of P_i's ID and a TTL value that takes a positive integer. We call P_i as *source* of the waves with ID_i and P_{i-1} as *tail*. Each process $P_{i+1}, P_{i+2}, \cdots, P_{i-1}$ forwards the message by decrementing TTL until it reaches zero. The source process repeatedly generates waves of its own ID with increasing the TTL by one at each generation. (The initial wave has TTL of one.) Hence, P_i has to generate a sequence of k waves with incrementing the TTL values to diffuse ID_i to process P_{s+k}. The diffusion is finished when a wave reaches P_{i-1}.

To achieve self-stabilization, it is necessary that each process *diffuses* its ID and *removes* the locally-stored IDs of non-existent processes. However, both functionality should be designed carefully because TB-Byzantine processes also use them. Consider a protocol that allows each process to discard a locally stored ID if and only if its predecessor does not store the same ID. If one process P_i removes an ID, then its successor P_{i+1} removes that ID. After that, P_{i+2} removes that ID. This behavior becomes global and starts a removal wave that spreads fast over the entire system. In the proposed protocol, propagation of removal of locally stored IDs is also implemented with slow waves.

[1] For the rest of the paper, we omit *mod*.

3.2 Pumping Protocol $\mathcal{PUMP}$

The ID diffusion part called $\mathcal{PUMP}$ is shown as Protocol 3.1. Given two processes, *source* P_s and *tail* P_t ($s < t$) for ID k, $\mathcal{PUMP}$ diffuses ID k at P_s to all processes P_i ($s < i \leq t$). Each processes P_i ($s < i < t$) is called *forwarder*.

$\mathcal{PUMP}$ has four parameters: T_i for local table at P_i, and three predicates $IsSource_i(k)$, $IsForwarder_i(k)$, and $IsTail_i(k)$ to define the source, the forwarders, and the tail for ID k. The Boolean predicate $IsSource_i(k)$ ($IsTail_i(k)$) holds only at the source (tail, respectively) process and $IsForwarder_i(k)$ holds at each forwarder P_i ($s < i < t$).

Each process maintains a table T_i of received waves. Each entry of T_i is in the form of (k, ℓ) where k is an ID and ℓ is the counter (*i.e.*, TTL) of the most recently received wave[2]. For any entry $(k, \ell) \in T_i$, the second element is denoted by $C_i(k)$: for (k, ℓ), $C_i(k)$ returns ℓ. The counter value $C_i(k)$ returns either a positive integer, $\perp$, ϕ or *undef*. The value $\perp$ is a restart signal that is returned to the source and ϕ is an acknowledgment signal that a tail returns to the source. If T_i does not have an entry with ID of k, $C_i(k)$ returns *undef*. The output variables of $\mathcal{PUMP}$ at process P_i are the set of IDs in T_i.

There are four operations for each entry of these tables: *hold*, *remove*, update, and comparison. Operation $hold((k, \ell), T_i)$ creates an entry (k, ℓ) if T_i does not have an entry with ID of k, otherwise, sets the value of $C_i(k)$ to ℓ. Operation $remove((k, C_i(k)), T_i)$ removes the entry with ID k from T_i. The update operation changes the value of $C_i(k)$ of the tuple $(k, C_i(k))$ in T_i, and is simply described as an assignment to $C_i(k)$ in Protocol 3.1. The comparison operation on the value of $C_i(k)$ is also possible. When $C_i(k)$ stores $\perp$, ϕ, or *undef*, the comparison operation returns false.

A source process P_s starts the diffusion by executing S_1 and S_2. Then, P_{s+1} changes $C_{s+1}(k)$ to 0 by executing S_5. After that, by executing S_3, P_s continues to generate waves with incrementing TTL values. Then P_{s+i} forwards the message with $TTL = \tau$ by decrementing the TTL of the received message (S_6). When a wave reaches the tail process P_t, P_t generates an acknowledgement signal by the execution of S_{11} and the acknowledgment wave is returned to P_s by the execution of S_9 and S_4. When a forwarder process finds inconsistency among the C values of neighboring processes, it generates a reset signal by executing S_7 and the reset signal is returned to P_s by the execution of S_8. When P_s receives the reset wave, it starts with a wave with $TTL = 0$ (S_1).

Figure 1 shows an example of ID diffusion from a source P_i where $k = ID_i$ to a tail P_{i-1}. The wave $(k, 1)$ generated at P_i is forwarded to P_{i+1} and the wave $(k, 3)$ generated at P_i is forwarded to P_{i+2}. In this way, the wave $(k, n-1)$ generated at P_i is forwarded to P_{i-1}.

The containment is achieved by the forwarders. In an (f', k'_f)-perturbation, when a TB-Byzantine process starts diffusing a new ID, it should generate a

[2] For simplicity, we assume each entry in T_i, has a unique ID value. Hence, for any k if (k, ℓ) in T_i, no entry (k, ℓ') with $\ell \neq \ell'$ exists in T_i. This is a data structure consistency problem and a solution for it can be easily implemented and applied to T_i. We do not address this problem in the paper.

Protocol 3.1 Protocol $\mathcal{PUMP}(T_i, \mathit{IsSource}_i(k), \mathit{IsForwarder}_i(k), \mathit{IsTail}_i(k))$ at process P_i for ID k

```
Parameters at $P_i$
    Local variable
        $C_i(k)$: the counter value of entry $(k, C_i(k))$ in $T_i$
    Output at $P_i$
        The set of IDs in $T_i$.
    Predicates at $P_i$
        $\mathit{IsSource}_i(k)$: Boolean predicate that takes true if $P_i$ is the source for ID $k$,
                otherwise false.
        $\mathit{IsForwarder}_i(k) \equiv \neg\mathit{IsSource}_i(k) \wedge \neg\mathit{IsTail}_i(k) \wedge (k, C_{i-1}(k)) \in T_{i-1}$
        $\mathit{IsTail}_i(k)$: Boolean predicate that takes true if $P_i$ is the tail for $k$,
                otherwise false.

        $\mathit{CntCons}_i(k) = C_i(k) > 0 \wedge \{C_{i-1}(k) - C_i(k) = 1 \vee C_{i-1}(k) - C_i(k) = 2\}$
        $\mathit{AckIncons}_i(k) = \{C_i(k) = \perp \vee 0 \leq C_i(k)\} \wedge C_{i-1}(k) = \phi$

Actions at process $P_i$
    // Source
        $S_1$  $\mathit{IsSource}_i(k) \wedge$
                $\{C_i(k) = \perp \vee C_i(k) = \mathit{undef} \vee (C_{i+1}(k) = \perp \wedge 1 \leq C_i(k)) \vee$
                $(C_{i+1}(k) = \mathit{undef} \wedge 1 < C_i(k)) \vee \neg\mathit{CntCons}_i(k)\}$
                $\longrightarrow \mathit{hold}((k, 0), T_i)$
        $S_2$  $\mathit{IsSource}_i(k) \wedge C_i(k) = 0 \longrightarrow C_i(k) = 1$
        $S_3$  $\mathit{IsSource}_i(k) \wedge C_{i+1}(k) - C_i(k) = 1 \longrightarrow C_i(k) + +$
        $S_4$  $\mathit{IsSource}_i(k) \wedge C_{i+1}(k) = \phi \wedge 0 < C_i(k) \longrightarrow C_i(k) = \phi$
    // Forwarder
        $S_5$  $\mathit{IsForwarder}_i(k) \wedge C_{i-1}(k) = 1 \wedge$
                $(0 < C_i(k) \vee C_i(k) = \mathit{undef} \vee C_i(k) = \phi \vee C_i(k) = \perp)$
                $\longrightarrow \mathit{hold}((k, 0), T_i)$
        $S_6$  $\mathit{IsForwarder}_i(k) \wedge C_{i-1}(k) - C_i(k) = 2 \wedge$
                $\{(0 < C_i(k) \wedge C_{i+1}(k) - C_i(k) = 1) \vee C_i(k) = 0\}$
                $\longrightarrow C_i(k) + +$
        $S_7$  $\mathit{IsForwarder}_i(k) \wedge \{\neg\mathit{CntCons}_i(k) \vee \mathit{AckIncons}_i(k) \vee$
                    $(C_{i-1}(k) - C_i(k) = 2 \wedge C_i(k) - C_{i+1}(k) = 2) \vee$
                    $(C_{i+1}(k) = \phi \wedge (\neg\mathit{CntCons}_i(k) \vee C_i(k) = 0) \vee$
                    $(C_{i+1}(k) = \mathit{undef} \wedge (1 < C_i(k) \vee C_i(k) = \phi))\} \vee$
               $\mathit{IsTail}_i(k) \wedge \{\neg\mathit{CntCons}_i(k) \vee \mathit{AckIncons}_i(k)\}$
                $\longrightarrow C_i(k) = \perp$
        $S_8$  $\mathit{IsForwarder}_i(k) \wedge (0 \leq C_i(k) \vee C_i(k) = \phi) \wedge C_{i+1}(k) = \perp$
                $\longrightarrow C_i(k) = \perp$
        $S_9$  $\mathit{IsForwarder}_i(k) \wedge \mathit{CntCons}_i(k) \wedge C_{i+1}(k) = \phi \longrightarrow C_i(k) = \phi$
    // Tail
        $S_{10}$ $\mathit{IsTail}_i(k) \wedge C_i(k) = 1 \wedge C_i(k) = \mathit{undef} \longrightarrow \mathit{hold}((k, 0), T_i)$
        $S_{11}$ $\mathit{IsTail}_i(k) \wedge 0 = C_i(k) \wedge \mathit{CntCons}_i(k) \longrightarrow C_i(k) := \phi$
```

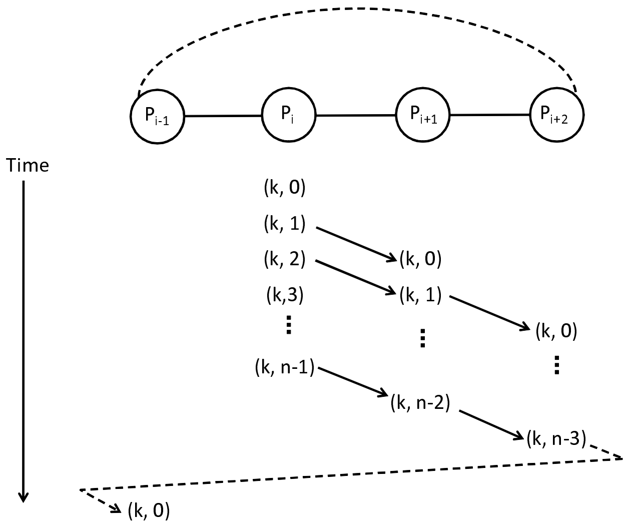


Fig. 1. Diffusion of waves from P_i

sequence of waves. At each forwarder P_i, the entry for the new ID is created only when the predecessor P_{i-1} receives the head of the wave (*i.e.*, the TTL value of one) by the execution of S_5. Then, P_i forwards a new wave only when it has received the previous wave. Hence, to diffuse a new ID to a process at distance k, a TB-Byzantine process has to generate a sequence of waves with TTL $0, 1, 2, \cdots, k$. In this way, the diffusion is slowed down by the forwarders.

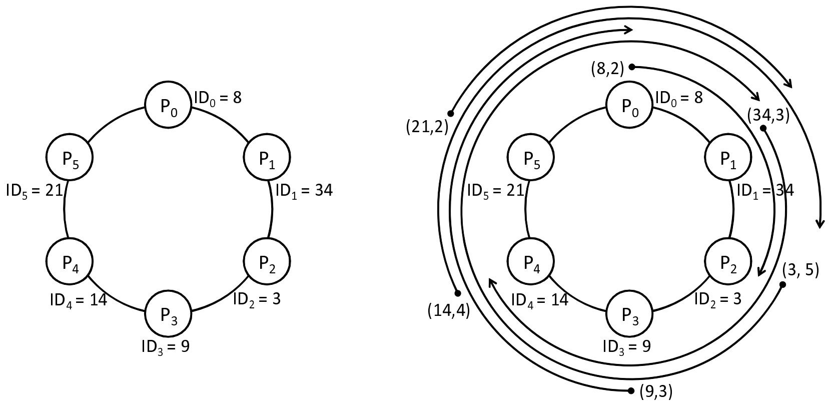
We say the counter value $C_i(k)$ at P_i is *consistent* if and only if $C_{i-1}(k) - C_i(k) = 1$ or 2 holds, otherwise *inconsistent*.

A configuration is legitimate for $\mathcal{PUMP}$ for ID k, if the following predicate $\ell_{\mathcal{PUMP}}$ holds at any process P_i in V:

$$\ell_{\mathcal{PUMP}} \equiv$$
$$\forall i : s \leq i \leq t \text{ s.t. } \mathit{IsSource}_s(k) = \mathit{true} \text{ and } \mathit{IsTail}_t(k) = \mathit{true} : C_i(k) = \phi$$

3.3 Leader Election Protocol $\mathcal{PLE}$

Now, we present the leader election protocol $\mathcal{PLE}$ as Protocol 3.2. In $\mathcal{PLE}$, $\mathcal{PUMP}$ is used to propagate ID of each process P_i (where the source is P_i and the tail is P_{i-1}) and to propagate removal of fictitious IDs of non-existent processes. Each process P_i maintains a local variable LID_i and two tables of received waves: a diffusion table DfT_i and a removal table RmT_i. LID_i is the output variable at P_i that stores leader's ID. The diffusion table DfT_i is used to diffuse IDs of processes and the removal table RmT_i is used to remove IDs. A wave diffused by using diffusion table is called a *diffusion wave* and a wave diffused by using removal table is called a *removal wave*.



(a) Ring of five processes (b) Diffusion waves

Fig. 2. Diffusion waves on a ring in $\mathcal{PLE}$

Figure 2 (b) shows an example of diffusion waves in a ring of five processes presented in Figure 2 (a). Each process diffuses its ID downstream and when all diffusions are finished, the LID value at each process is updated.

In $\mathcal{PLE}$, $\mathcal{PUMP}$ is executed in S_1 for diffusing ID_i and also in S_2 for removal waves. The source, forwarder, and the tail for diffusion waves are defined by the three predicates $IsDS_i(k)$, $IsDF_i(k)$, and $IsDT_i(k)$. $IsDS_i(k)$ is evaluated to *true* if and only if $ID_i = k$ holds. Then, $IsDT_i(k)$ is evaluated to *true* at P_{i-1}. The source, forwarder, and the tail for removing waves are defined by the three predicates $IsRS_i(k)$, $IsRF_i(k)$, and $IsRT_i(k)$. Process P_i is the source for removal of ID k when it finds its successor P_{i+1} stores (k, ℓ) while $C_i(k)$ is *undef* at P_i. Process P_i becomes the tail for removal of ID k when it finds it stores (k, ℓ) while $C_{i+1}(k)$ is *undef* at its successor P_{i+1}.

After the diffusion of removal waves for ID k is finished, ID k is removed by the execution of S_3 at the tail, each forwarder, and the source. When the suspicious ID is stored locally, it is removed by the execution of S_4. Finally, the removal wave is discarded by S_5. The leader's ID variable is changed only when the diffusion of IDs and removal waves are finished after the execution of S_6.

The self-stabilization of $\mathcal{PLE}$ is guaranteed due to the self-stabilization property of $\mathcal{PUMP}$. Because each process keeps on diffusing its ID, eventually all processes receive IDs of all existent processes. IDs of non-existent processes are also removed during a correct execution.

The fault-containment of $\mathcal{PLE}$ is derived from the fault-containment property of $\mathcal{PUMP}$. Each diffusion wave or a removal wave spreads slowly in $\mathcal{PLE}$ and after a malicious action, the predecessor of a TB-Byzantine process finds inconsistency and generates a reset signal for each removed ID or becomes a source of the removal wave for new ID of non-existent processes. The reset signal for

Protocol 3.2 Protocol $\mathcal{PLE}$ at process P_i

```
Local variables at $P_i$
    $ID_i$: ID of process $P_i$
    $LID_i$: leader's ID
    $DC_i(k)$: the counter value of entry $(k, DC_i(k))$ in $DfT_i$
    $RC_i(k)$: the counter value of entry $(k, RC_i(k))$ in $RmT_i$
Output variable at $P_i$
    $LID_i$
Predicates at $P_i$
    $IsDS_i(k) \equiv k = ID_i$
    $IsDF_i(k) \equiv k \neq ID_i \wedge ID_{i+1} \neq k \wedge (k, DC_{i-1}(k)) \in DfT_{i-1}$
    $IsDT(k) \equiv ID_{i+1} = k$
    $IsRS_i(k) \equiv DC_i(k) = undef \wedge (k, DC_{i+1}(k)) \in DfT_{i+1} \wedge k \neq ID_{i+1}$
    $IsRF_i(k) \equiv (k, DC_{i-1}(k)) \in DfT_{i-1} \wedge (k, DC_i(k)) \in DfT_i \wedge$
                  $(k, DC_{i+1}(k)) \in DfT_{i+1}$
    $IsRT(k) \equiv (k, DC_i(k)) \in DfT_i \wedge DC_{i+1}(k) = undef$

Actions at process $P_i$
    // Diffusion wave
    $S_1$  $true \longrightarrow$
        $\forall k$ execute $\mathcal{PUMP}(DfT_i, IsDS_i(k), IsDF_i(k), IsDT_i(k))$
    // Removal wave
    $S_2$  $\neg\{\exists k \neq ID_i : DC_{i-1}(k) = undef \wedge (k, DC_i(k)) \in DfT_i \wedge DC_{i+1}(k) = undef\}$
        $\longrightarrow \forall k$ execute $\mathcal{PUMP}(RmT_i, IsRS_i(k), IsRF_i(k), IsRT_i(k))$
    // Discarding IDs of non-existent processes
    $S_3$  $\exists k \neq ID_i : RC_{i-1}(k) = \phi \wedge RC_i(k) = \phi \wedge RC_{i+1}(k) = undef \longrightarrow$
        $remove((k, DC_i(k)), DfT_i)$; $remove((k, RC_i(k)), RmT_i)$
    // Discarding locally
    $S_4$  $\exists k \neq ID_i : DC_{i-1}(k) = undef \wedge (k, DC_i(k)) \in DfT_i \wedge DC_{i+1}(k) = undef$
        $\longrightarrow remove((k, DC_i(k)), DfT_i)$; $remove((k, RC_i(k)), RmT_i)$
    // Discarding removal wave
    $S_5$  $\exists k : (k, RC_i(k)) \in RmT_i \wedge \{k = ID_i \vee (RC_{i-1}(k) = undef \wedge \neg IsRS_i(k)\}$
        $\longrightarrow remove((k, RC_i(k)), RmT_i)$
    // Selecting leader's ID
    $S_6$  $\forall (k, DC_i(k)) \in DfT_i : DC_i(k) = \phi \wedge RC_i(k) = undef \longrightarrow$
        $LID_i = \min\{k | (k, DC_i(k)) \in DfT_i\}$
```

an ID of an existing process causes pumping of n waves, and the recovery may entail global recomputation.

A configuration is legitimate for $\mathcal{PLE}$, if the following predicate $\ell_{\mathcal{PLE}}$ holds at any process P_i in V:

$$\begin{aligned}\ell_{\mathcal{PLE}} \equiv & \{\forall P_j \in V : DC_i(ID_j) = \phi \wedge (ID_j, RC_i(ID_j)) \notin RmT_i\} \\ & \wedge\{\forall (k, DC_i(k)) \in DfT_i : \exists P_j \in V : ID_j = k\} \\ & \wedge\{LID_i = \min\{ID_\ell | P_\ell \in V)\}\}\end{aligned}$$

4 Correctness Proof

Before we start correctness proofs, we first show a trivial upper bound of (f', k'_f)-disruption time and (f', k'_f)-perturbation time. In the worst case, every malicious action occurs just before the system reaches a legitimate configuration.

Remark 1. The (f', k'_f)-disruption time of a self-stabilizing protocol $\mathcal{P}$ with convergence time T is at most $(f' \cdot k'_f + 1)T$ and the (f', k'_f)-perturbation time of a TB-Byzantine fault-resilient fault-containing self-stabilizing protocol $\mathcal{P}$ with convergence time T is at most $(f' \cdot k'_f + 1)T$.

We show the correctness proofs and performance evaluation of $\mathcal{PUMP}$ and $\mathcal{PLE}$. In the following, we first show self-stabilization and convergence time of $\mathcal{PUMP}$. From Protocol 3.1, we obtain the following lemma. (Due to page restriction, we omit the detailed proof.)

Lemma 1. *In any legitimate configuration of $\mathcal{PUMP}$, there exists at least one enabled process.*

Let P_s be a source and P_t be a tail for ID k. The ℓ-th forwarder be the process $P_{s+\ell}$ between P_s and P_t $(s < s + \ell < t)$. Let $P_{s+\ell}$ be a forwarder that satisfies (i) $C_i(k)$ at P_i and $C_{i+1}(k)$ at P_{i+1} are consistent for all $s \le i < s + \ell$, and (ii) $C_{s+\ell}(k)$ at $P_{s+\ell}$ and $C_{s+\ell+1}(k)$ at $P_{s+\ell+1}$ are inconsistent. We call $P_{s+\ell}$ as *head* of the wave for k.

Lemma 2. *In a correct execution, in each $(\ell^2 + 5\ell + 2)/2$ rounds, the head of a wave at ℓ-th forwarder progresses to $(\ell + 1)$-th forwarder.*

Proof. Let the head of the wave for ID k be $P_{s+\ell}$. In the worst case, $P_{s+\ell}$ generates a reset signal and after that, the source P_s restarts pumping with generating $(\ell + 1)$ waves. Then, the number of rounds for the head at $P_{s+\ell}$ to progress to $P_{s+\ell+1}$ is

$$\ell + (1 + 2 + \cdots + \ell + (\ell + 1)) = \ell + \frac{(\ell + 1)(\ell + 2)}{2} = \frac{\ell^2 + 5\ell + 2}{2}. \qquad \square$$

From Lemma 1 and Lemma 2, eventually the diffusion is completed and the system reaches a legitimate configuration.

Theorem 1. *$\mathcal{PUMP}$ is self-stabilizing and for diffusion from P_s to P_t where $t - s = \ell$, the convergence time is $O(\ell^2)$ rounds.*

Next, we show the fault-containment of $\mathcal{PUMP}$. Because the output variables of $\mathcal{PUMP}$ at P_i are the IDs stored in T_i, if P_i stores a new ID, then P_i is perturbed.

In $\mathcal{PUMP}$, there are two types of waves. The propagation of ϕ and $\perp$ is implemented with *fast* waves that are forwarded immediately to the neighbors. On the other hand, the diffusion of IDs is implemented with *slow* waves that the source should keep on generating waves with incrementing the TTL value and the forwarders allow these waves to propagate in the FIFO order. These slow waves guarantee the upper bound on the number of perturbed processes.

Consider an (f, k_f)-faulty execution $E = \sigma_0, \sigma_1, \cdots$ starting from a legitimate configuration with at least one malicious action. Because of the self-stabilization property of $\mathcal{PUMP}$, there exists a finite length of disruption in E. Let E' be the (f', k'_f)-disruption of E. Because each TB-Byzantine process takes at most k'_f malicious action during E', it generates at most k'_f waves for each fictitious ID. Because in a legitimate configuration, there exists no fictitious IDs at any process, each TB-Byzantine process has to generate ℓ waves to diffuse a fictitious ID to a process at distance ℓ. In the worst case, correct processes forwards these waves if *IsForwarder* is evaluated to *true*. Hence, in a (f', k'_f)-disruption, there are at most f' TB-Byzantine processes each of which perturbs at most k'_f correct processes. Then, we obtain the following theorem.

Theorem 2. *$\mathcal{PUMP}$ is TB-Byzantine resilient fault-containing and the (f', k'_f)-perturbation number is* $\min\{f' \cdot k'_f, n\}$.

Because $\mathcal{PUMP}$ causes the restart of pumping from the source, the disruption time (perturbation time) depends on the convergence time, f', and k'_f for any (f', k'_f)-disruption ((f', k'_f)- perturbation, respectively).

Theorem 3. *The (f', k'_f)-disruption time and the (f', k'_f)-perturbation time of $\mathcal{PUMP}$ are $O(f'k'_f\ell^2)$ for diffusion from P_s to P_t where $t - s = \ell$.*

Secondly, we show the correctness proofs of $\mathcal{PLE}$ based on the stabilization and the fault-containment property of $\mathcal{PUMP}$. (Due to page restriction, we omit the detailed proof.) We showed that $\mathcal{PUMP}$ prevents fictitious IDs from spreading over the entire network by using slow waves. However, $\mathcal{PUMP}$ does not remove these fictitious IDs after it is propagated to a small number of perturbed processes. By using $\mathcal{PUMP}$ for diffusion waves and removal waves, $\mathcal{PLE}$ achieves self-stabilization and fault-containment for the leader election problem.

Starting from an arbitrary initial configuration, the propagation of removal waves eventually finishes and each process keeps on pushing its ID until it is received by all other processes.

Theorem 4. *$\mathcal{PLE}$ is self-stabilizing.*

From the perturbation number shown in Theorem 2, all diffusion waves and removal waves generated at a TB-Byzantine process reach processes at distance at most k'_f in any (f', k'_f)-disruption. Then, we have the following theorem.

Theorem 5. *$\mathcal{PLE}$ is TB-Byzantine resilient fault-containing and the (f', k'_f)-perturbation number is* $\min\{f' \cdot k'_f, n\}$.

5 Conclusion

In this paper, we proposed a novel fault model called TB-Byzantine fault and introduce the notion of adaptive containment of TB-Byzantine processes. The proposed information diffusion method, called pumping, guarantees fault-containment of TB-Byzantine faults in the sense of perturbation number. We note that the pumping method can be easily extended to any arbitrary topology and any arbitrary problem because the pumping method is implemented on an arbitrary topology with broadcast waves. Though the perturbation number is bounded by the number of malicious actions during a disruption, the disruption time and the perturbation time of the proposed protocol depends on the convergence time and the number of malicious actions. This paper also proposes to analyze these time complexity measures in the presence of malicious actions. Our future work is to develop a fast stabilization and containment technique against TB-Byzantine faults.

Acknowledgment

This work is supported in part by MEXT Global COE Program, JSPS Grant-in-Aid for Scientific Research ((B)22300009), and JSPS Grant-in-Adi for Young Scientists (Start-up) (21800031).

References

1. Afek, Y., Dolev, S.: Local stabilizer. Journal of Parallel and Distributed Computing 62, 745–765 (2002)
2. Biely, M., Huttle, M.: Consensus when all processes may be byzantine for some time. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 120–132. Springer, Heidelberg (2009)
3. Burman, J., Herman, T., Kutten, S., Patt-Shamir, B.: Asynchronous and fully self-stabilizing time-adaptice majority consensus. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 146–160. Springer, Heidelberg (2006)
4. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of ACM 17(11), 643–644 (1974)
5. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
6. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing algorithms. In: Proceedings of the 15th PODC, pp. 45–54 (May 1996)
7. Kutten, S., Patt-Shamir, B.: Time-adaptive self stabilization. In: Proceedings of the 16th PODC, pp. 149–158 (1997)
8. Kutten, S., Peleg, D.: Fault-local distributed mending. In: Proceedings of the 14th PODC, pp. 20–27 (August 1995)
9. Masuzawa, T., Tixeuil, S.: A self-stabilizing link-coloring protocol resilient to unbounded byzantine faults in arbitrary networks. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 118–129. Springer, Heidelberg (2006)
10. Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 440–453. Springer, Heidelberg (2006)
11. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: Proceedings of the 21st SRDS, pp. 22–29 (October 2002)
12. Schneider, M.: Self-stabilization. ACM Computing Survey 25(1), 45–67 (1993)
13. Widder, J., Gridling, G., Weiss, B., Blanquart, J.: Synchronous consensus with mortal byzantines. In: Proceedings of the 37th DSN, pp. 102–112 (June 2007)
14. Zhao, Y., Bastani, F.: A self-adjusting algorithm for byzantine agreement. Distributed Computing 5, 219–226 (1992)

Brief Announcement: Fast Convergence in Route-Preservation

Jorge A. Cobb

Department of Computer Science
The University of Texas at Dallas
cobb@utdallas.edu

Introduction. Optimal-routing in a computer network consists of building a spanning-tree such that two conditions hold: a) the root of the tree is a distinguished node, and b) weights are assigned to the network links, and each path along the tree to the root is optimal with respect to these weights [4]. This differs from spanning-tree protocols used in leader election, in which any node can be the root, and usually the tree need not be optimal with respect to link weights.

We have developed a stabilizing routing protocol with the following properties: a) it is suitable for all maximizable metrics [4], not just strictly-bounded metrics, b) stabilization time is proportional to L, the longest network path, c) nodes have no knowledge of an upper bound on L (and hence of the network size) and d) it is loop-free and suitable for route-preservation. Route-preservation [5], is the ability to ensure that data messages reach the root irrespective of changes in the routing tree.

In [2], we presented a stabilizing routing protocol for maximizable metrics without assuming an upper bound on L, and whose convergence time is bounded by L. However, it is not suitable for route-preservation because nodes become temporarily disconnected from the root. We refer to this protocol as the *base* protocol.

Adaptive Routing Protocol. We propose running two protocols in parallel, the base protocol and an *adaptive* protocol. The purpose of the adaptive protocol is to adapt its routing tree to match the routing tree of the base protocol, while maintaining the integrity of its own routing tree (i.e., always be loop-free and have no disconnected nodes).

Let input $u.\widehat{pr}$ contain the parent chosen for node u by the base protocol. The adaptive protocol is unaware of when the base protocol has converged; it simply takes the current value of $u.\widehat{pr}$ and assigns it to its parent variable $u.pr$. To ensure loop-freedom, we perform a diffusing computation before this assignment takes place, similar to that of other loop-free routing protocols [6].

Each node can be in one of two states: high or low. During steady-state operation, i.e., when both the adaptive and base routing protocols have the same tree ($u.pr = u.\widehat{pr}$ for all u), all nodes are in the high state. When $u.pr \neq u.\widehat{pr}$, node u changes its state to low, and causes all of its descendants to change their state to low. Node u can then choose a new parent only if the new parent's state is high. The specification of a non-root node u is given below (more details can be found in [1]).

```
node $u$
inp
  $u.N$            :   set of node id's          {neighbors of $u$}
  $u.\widehat{pr}$ :   element of $u.N$          {base parent of $u$}
var
  $u.pr$           :   element of $u.N$          {parent of $u$}
  $u.st$           :   element of {high, low}    {state of $u$}
  $u.req$          :   boolean                   {request lowering of descendants}
  $u.end$          :   subset of $u.N$           {neighbors where diffusion ended}
par
  $g$              :   element of $u.N$          {any neighbor of $u$}
begin
  {start request if new parent}
  $u.high \wedge \neg u.req \wedge u.pr \neq u.\widehat{pr}$   $\rightarrow$
        $u.req :=$ true;   $u.end := \emptyset$

▯ {propagate request}
  $(u.high \wedge \neg u.req) \wedge ((u.pr).high \wedge (u.pr).req)$   $\rightarrow$
        $u.req :=$ true;   $u.end := \emptyset$

▯ {add neighbor to end set}
  $g.pr \neq u \vee g.low$   $\rightarrow$
        $u.end := u.end \bigcup \{g\}$

▯ {terminate request and lower state}
  $u.high \wedge u.req \wedge u.end = u.N$   $\rightarrow$
        $u.st := low$

▯ {change to new parent}
  $u.low \wedge (u.\widehat{pr}).high \wedge \neg(u.\widehat{pr}).req$   $\rightarrow$
        $u.pr := u.\widehat{pr}$;
        $u.st := high$;   $u.req :=$ false
end
```

Route Preserving Policy. Similar to [5], we define a routing policy that is *route-preserving*. I.e., we define a set of guidelines to route data messages toward the root, such that, even though edge weights are currently changing, and hence, the routing tree is changing, data messages are guaranteed to reach the root. We assume each message has a one-bit flag that is reserved for the routing policy. The flag is set whenever the message arrives at a node whose state is low, or if the message is queued at a node whose state transitioned from high to low. Once the flag is set in a message, it may not be cleared. Once a message is flagged and has reached a node in the high state, we require the message to continue only along nodes with a high state. This ensures no parent changes along its path, and thus, it eventually arrives at the root. More details on this policy may be found in [1].

Stabilization. To stabilize the adaptive protocol, the main obstacle is the detection and elimination of routing loops, and having nodes that were formerly in a loop rejoin the routing tree. There are two traditional ways to detect routing loops. The first is simply not to do anything specific to break loops, provided the routing metric is strictly-bounded [4], i.e., the metric of nodes along a loop becomes progressively worse, until the metric of the routing tree is better than that of the loop. Another approach is to maintain a hop-count to the root, and the loop is broken if the hop count of a node reaches the upper bound on L [3].

Our approach also uses a hop-count, but without assuming an upper bound on L. Instead, loops are assumed to exist if the hop-count of a child is not equal to the hop-count of its parent plus one (note that this condition must always exist in all loops). To prevent false-positives, during normal operation, the hop-count of a node must always be consistent with its parent. However, due to the dynamic structure of the routing tree, this is not possible. Thus, the hop-count is allowed to be inconsistent at the crossover point between low nodes and high nodes.

A loop may be broken by a node u simply by setting its parent variable $u.pr$ to itself ($u.pr := u$). To prevent forming new loops, u must not choose a descendant as a new parent. This can easily be achieved by u choosing a new parent only if u has no children (and hence, no descendants). A child v of u can observe that u has no parent (i.e., $u.pr = u$), and v can also set its parent to itself ($v.pr := v$). Thus, u eventually has no children, and is free to choose any node as its parent.

However, as shown in [1], this can lead to a race-condition. We prevent the race-condition via an additional diffusing computation. The objective of this new diffusing computation is to ensure that the subtree of a parent-less node has been completely disassembled before the parent-less node is allowed to choose a new parent.

References

1. Cobb, J.A.: Fast convergence in route preservation, Department of Computer Science Technical Report, The University of Texas at Dallas (June 2010), http://www.utdallas.edu/~jcobb/PublishedPapers/TR/RP.pdf
2. Cobb, J.A., Huang, C.T.: Stabilization of maximal-metric routing without knowledge of network size. In: Second International Workshop on Reliability, Availability, and Security (WRAS 2009), Hiroshima, Japan (2009)
3. Gouda, M.G., Schneider, M.: Maximum flow routing. In: Proceedings of the Second Workshop on Self-Stabilizing Systems, Technical Report, Department of Computer Science, University of Nevada, Las Vegas (1995)
4. Gouda, M.G., Schneider, M.: Maximizable routing metrics. IEEE/ACM Trans. Netw. 11(4), 663–675 (2003)
5. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 184–198. Springer, Heidelberg (2003)
6. Merlin, P.M., Segall, A.: A failsafe distributed routing protocol. IEEE Transactions on Communications COM-27(9) (1979)

Authenticated Broadcast with a Partially Compromised Public-Key Infrastructure

S. Dov Gordon, Jonathan Katz[⋆], and Ranjit Kumaresan, and Arkady Yerukhimovich

Dept. of Computer Science
University of Maryland
{gordon,jkatz,ranjit,arkady}@cs.umd.edu

Abstract. Given a public-key infrastructure (PKI) and digital signatures, it is possible to construct broadcast protocols tolerating any number of corrupted parties. Almost all existing protocols, however, do not distinguish between *corrupted* parties (who do not follow the protocol), and *honest* parties whose secret (signing) keys have been compromised (but who continue to behave honestly). We explore conditions under which it is possible to construct broadcast protocols that still provide the usual guarantees (i.e., validity/agreement) to the latter.

Consider a network of n parties, where an adversary has compromised the secret keys of up to t_c honest parties and, in addition, fully controls the behavior of up to t_a other parties. We show that for any fixed $t_c > 0$, and any fixed t_a, there exists an efficient protocol for broadcast if and only if $2t_a + \min(t_a, t_c) < n$. (When $t_c = 0$, standard results imply feasibility.) We also show that if t_c, t_a are not fixed, but are only guaranteed to satisfy the bound above, then broadcast is impossible to achieve except for a few specific values of n; for these "exceptional" values of n, we demonstrate a broadcast protocol. Taken together, our results give a complete characterization of this problem.

1 Introduction

Although Public Key Infrastructures (PKI) are heavily used in the design of cryptographic protocols, in practice they are often subject to key leakage, cryptanalysis and side channel attacks. Such attacks can make the resulting construction insecure as it's security depends on the security of the PKI. In this work, we consider the security that can be guaranteed even with a compromised PKI. In particular, we study the problem of *broadcast* in a setting where some of the honest players' signatures can be forged.

[⋆] Work done in part while visiting IBM. Supported by NSF, the U.S. DoD/ARO MURI program, and the US Army Research Laboratory and the UK Ministry of Defence under agreement number W911NF-06-3-0001.

Broadcast protocols allow a designated player (the *dealer*) to distribute an input value to a set of parties such that (1) if the dealer is honest, all honest parties output the dealer's value (**validity**), and (2) even if the dealer is dishonest, the outputs of all honest parties agree (**agreement**). Broadcast protocols are fundamental for distributed computing and secure computation: they are crucial for simulating a broadcast channel over a point-to-point network, and thus form a critical sub-component of various higher-level protocols.

Classical results of Pease, Shostak, and Lamport [12,8] show that broadcast (and, equivalently, Byzantine agreement) is achievable in a synchronous network of n parties if and only if the number of corrupted parties t satisfies $t < n/3$. To go beyond this bound, some form of set-up is required. The most commonly studied set-up assumption is the existence of a public-key infrastructure (PKI) such that each party P_i has a public signing key pk_i that is known to all other parties (in addition to the cryptographic assumption that secure digital signatures exist). In this model, broadcast is possible for any $t < n$ [12,8,1].

With few exceptions [3,5] (see below), prior work in the PKI model treats each party as either totally honest, or as completely corrupted and under the control of a single adversary; the assumption is that the adversary cannot forge signatures of any honest parties. However, in many situations it makes sense to consider a middle ground: parties who honestly follow the protocol but whose signatures might be forged (e.g., because their signing keys have been compromised). Most existing work treats any such party P_i as corrupt, and provides no guarantees for P_i in this case: the output of P_i may disagree with the output of other honest parties, and validity is not guaranteed when P_i is the dealer. Clearly, it would be preferable to ensure agreement and validity for honest parties who have simply had the misfortune of having their signatures forged.

Here, we consider broadcast protocols providing exactly these guarantees. Specifically, say t_a parties in the network are actively corrupted; as usual, such parties may behave arbitrarily and we assume their actions are coordinated by a single adversary $\mathcal{A}$. We also allow for t_c parties who follow the protocol honestly, but whose signatures can be forged by $\mathcal{A}$; this is modeled by simply giving $\mathcal{A}$ their secret keys. We refer to such honest-behaving parties as *compromised*, and require agreement and validity to hold even for compromised parties.

Say t_a, t_c *satisfy the threshold condition* with respect to some total number of parties n if $2t_a + \min(t_a, t_c) < n$. We show:

1. For any n and any t_a, t_c satisfying the threshold condition with respect to n, there is an efficient (i.e., polynomial in n) protocol achieving the notion of broadcast outlined above.
2. When the threshold condition is *not* satisfied, broadcast protocols meeting our notion of security are impossible. (With the exception of the "classical" case where $t_c = 0$; here standard results like [1] imply feasibility.)
3. Except for a few "exceptional" values of n, there is no *fixed* n-party protocol that tolerates all t_a, t_c satisfying the threshold condition with respect to n. (The positive result mentioned above relies on two different protocols,

depending on whether $t_a \leq t_c$.) For the exceptional values of n, we show protocols that *do* tolerate any t_a, t_c satisfying the threshold condition.

Taken together, our results provide a complete characterization of the problem.

Motivating the Problem. Compromised parties are most naturally viewed as honest parties whose secret (signing) keys have been obtained somehow by the adversary. E.g., perhaps an adversary was able to hack into an honest user's system and obtain their secret key, but subsequently the honest party's computer was re-booted and now behaves honestly. Exactly this scenario is addressed by *proactive* cryptosystems [11] and leakage-resilient cryptosystems [2], though in somewhat different contexts.

We remark, however, that our model is meaningful even if such full-scale compromise of honest users' secret keys is deemed unlikely. Specifically, our work provides important guarantees whenever there is a possibility that an honest user's signature might be *forged* (whether or not the adversary learns the user's actual secret key). Signature forgery can potentially occur due to cryptanalysis, poor implementation of cryptographic protocols [9,10], or side-channel attacks [6,7]. In all these cases, it is likely that an adversary might be able to forge signatures of a small number of honest parties without being able to forge signatures of everyone.

Prior Work. Gupta et al. [5] also consider broadcast protocols providing agreement and validity for honest-behaving parties whose secret keys have been compromised. Our results improve upon theirs in several respects. First, we construct *efficient* protocols whenever $2t_a + \min(t_a, t_c) < n$, whereas the protocols presented in the work of Gupta et al. have message complexity exponential in n. Although Gupta et al. [5] also claim impossibility when $2t_a + \min(t_a, t_c) \geq n$, our impossibility result is simpler and stronger in that it holds relative to a weaker adversary.[1] Finally, Gupta et al. treat t_a, t_c as known and do not consider the question of designing a fixed protocol achieving broadcast for any t_a, t_c satisfying the threshold condition (as we do in the third result mentioned above).

Fitzi et al. [3] consider broadcast in a model where the adversary can either corrupt a few players and forge signatures of *all* parties, or corrupt more players but forge no signatures. In our notation, their work handles the two extremes $t_a < n/3$, $t_c = n$ and $t_a < n/2$, $t_c = 0$. Our work addresses the intermediate cases, where an adversary might be able to forge signature of some honest parties but not others.

Organization. Section 2 introduces our model and provides a formal definition of broadcast in our setting. In Section 3 we show that for every n, t_a, t_c satisfying the threshold condition, there exists an efficient broadcast protocol. We show our impossibility results in Section 4: namely, broadcast is impossible whenever t_a, t_c do not satisfy the threshold condition (except when t_c is fixed to 0), and (other

[1] In [5], the adversary is assumed to have access to the random coins used by the compromised parties when running the protocol, whereas we do not make this assumption.

than for the exceptional values of n) there does not exist a single, fixed protocol achieving broadcast for all t_a, t_c satisfying the threshold condition. In Section 5 we give positive results for the exceptional values of n. Although dealing with these "outliers" may seem like a minor point, in fact all the exceptional values of n are small and so are more likely to arise in practice. Furthermore, dealing with these exceptional values is, in some sense, the most technically challenging part of our work.

2 Model and Definitions

We consider the standard setting in which n players communicate in synchronous rounds via authenticated channels in a fully connected, point-to-point network. (See below for further discussion regarding the assumption of authenticated channels.) We assume a public-key infrastructure (PKI), established as follows: each party P_i runs a key-generation algorithm Gen (specified by the protocol) to obtain public key pk_i along with the corresponding secret key sk_i. Then all parties begin running the protocol holding the same vector of public keys $(pk_1, \ldots, pk_n)$, and with each P_i holding sk_i.

A party that is *actively corrupted* (or "Byzantine") may behave arbitrarily. All other parties are called *honest*, though we further divide the set of honest parties into those who have been *compromised* and those who have not been compromised, as discussed below. We view the set of actively corrupted players as being under the control of a single adversary $\mathcal{A}$ coordinating their actions. We always assume such parties are *rushing*, and may wait to see the messages sent by honest parties in a given round before deciding on their own messages to send in that round. Actively corrupted parties may choose their public keys arbitrarily and even dependent on the public keys of honest parties. We continue to assume, however, that all honest parties hold the same vector of public keys.

Some honest players may be *compromised*; if P_i is compromised then the adversary $\mathcal{A}$ is given that P_i's secret key sk_i. We stress that compromised players follow the protocol as instructed: the only difference is that $\mathcal{A}$ is now able to forge signatures on their behalf. On the other hand, we assume $\mathcal{A}$ is unable to forge signatures of any honest players who have *not* been compromised.

We assume authenticated point-to-point channels between *all* honest parties, even those who have been compromised. In other words, although the adversary can forge the signature of an honest party P_i who has been compromised, it cannot falsely inject a point-to-point message on P_i's behalf. It is worth noting that this is a common assumption in many previous works relating to information-theoretic broadcast, byzantine agreement, and secure computation: for each of these problems, shared cryptographic keys cannot be used to ensure authenticated and/or secret channels against an all-powerful adversary. In practice, authenticated channels would be guaranteed using pairwise symmetric keys (that are less easily compromised or cryptanalyzed than signing keys), or could also be ensured via physical means in small-scale networks. Note that without the assumption of authenticated channels, no meaningful results are possible.

Definition 1. *A protocol for parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, where a distinguished dealer $D \in \mathcal{P}$ holds an initial input M, achieves* **broadcast** *if the following hold:*

Agreement. *All honest parties output the same value.*

Validity. *If the dealer is honest, then all honest parties output M.*

We stress that "honest" in the above includes those honest parties who have been compromised.

Although the above refers to an arbitrary input M for the dealer, we assume for simplicity that the dealer's input is a single bit. Broadcast for arbitrary length messages can be obtained from binary broadcast using standard techniques.

An adversary $\mathcal{A}$ is called a (t_a, t_c)*-adversary* if $\mathcal{A}$ actively corrupts up to t_a parties and additionally compromises up to t_c of the honest parties. In a network of n players, we call $\mathcal{A}$ a *threshold adversary* if $\mathcal{A}$ chooses t_a, t_c subject to the restriction $2t_a + \min(t_a, t_c) < n$; actively corrupts up to t_a parties; and compromises up to t_c honest parties.

3 Broadcast for (t_a, t_c)-Adversaries

In this section, we prove the following result:

Theorem 1. *Fix n, t_a, t_c with $2t_a + \min(t_a, t_c) < n$. Then there exists a protocol achieving broadcast in the presence of a (t_a, t_c)-adversary.*

The case of $t_a \leq t_c$ is easy: $t_a \leq t_c$ implies $3t_a < n$ and the parties can thus run a standard (unauthenticated) broadcast protocol [12,8] where the PKI is not used at all. (In this case, it makes no difference whether honest players are compromised or not.) The challenge is to design a protocol for $t_c < t_a$, and we deal with this case for the remainder of this section.

Let DS refer to the Dolev-Strong protocol [1] that achieves broadcast with a PKI, in the usual sense (i.e., when no honest parties' keys can be compromised), for any $t < n$ corrupted parties. (The Dolev-Strong protocol is reviewed in Appendix A.) We say that P_i calls an execution of the DS protocol *dirty* if P_i receives valid signatures by the dealer on two different messages, or never receives any valid signed messages from the dealer; i.e., if P_i detects that the dealer is either corrupted or compromised. P_i declares the execution *clean* otherwise. The following is easy to prove (the proof is omitted due to lack of space):

Lemma 1. *Consider an execution of protocol* DS *in the presence of t_a adversarial parties and t_c compromised honest parties, where $t_a + t_c < n$. Then:*

1. *All honest parties agree on whether an execution of* DS *is clean or dirty.*
2. *Agreement holds. (I.e., the outputs of all honest players are identical.)*
3. *If the dealer is honest and has not been compromised, then validity holds (i.e., all honest parties agree on the dealer's input) and the execution is clean. If the dealer is honest and the execution is clean, then validity also holds.*

Protocol 1

Inputs: Let D be the dealer, with input bit b.

Computation:

1. D sends b to all other players. Let b_i be the value received by P_i from D in this step (if the dealer sends nothing to P_i, then b_i is taken to be some default value).
2. In parallel, each party P_i acts as the dealer in an execution of $\mathsf{DS}(b_i)$ (the original dealer D runs $\mathsf{DS}(b)$). We let $|\mathsf{CLEAN}_0|$ (resp., $|\mathsf{CLEAN}_1|$) denote the number of executions of DS that are both *clean* and result in output 0 (resp., 1).

Output: If $|\mathsf{CLEAN}_0| \geq |\mathsf{CLEAN}_1|$ then all parties output 0; otherwise, all parties output 1.

Fig. 1. A broadcast protocol for $t_c < t_a$ and $2t_a + t_c < n$

Thus, DS fails to satisfy Definition 1 only when the dealer is *honest* but *compromised*. Our protocol (cf. Figure 1) guarantees validity even in this case (while leaving the other cases unaffected).

Theorem 2. *Let $\mathcal{A}$ be a (t_a, t_c)-adversary with $t_c < t_a$ and $2t_a + t_c < n$. Then Protocol 1 achieves broadcast in the presence of $\mathcal{A}$.*

Proof. We prove agreement and validity. Note that $n > t_a + t_c$, so Lemma 1 applies.

Agreement: By Lemma 1, the output of each honest player is the same in every execution of DS in step 2, and all honest parties agree on whether any given execution of DS is clean or dirty. So all honest players agree on $|\mathsf{CLEAN}_0|$ and $|\mathsf{CLEAN}_1|$, and agreement follows.

Validity: Assume the dealer is honest (whether compromised or not). Letting t_h denote the number of honest, non-compromised players, we have $t_h + t_a + t_c = n > 2t_a + t_c$ and so $t_h > t_a$. Thus, there are t_h honest and non-compromised dealers in step 2 of Protocol 1, and (since D is honest) each of these runs $\mathsf{DS}(b)$ where b is the initial input of D. By Lemma 1, all honest players output b in (at least) these t_h executions, and each of these t_h executions is clean. Furthermore, there can be at most t_a clean executions resulting in output $1 - b$, as only adversarial players will possibly run $\mathsf{DS}(1 - b)$ in step 2. The majority value output by the honest players is therefore always equal to the original dealer's input b.

4 Impossibility Results

In this section we show two different impossibility results. First, we show that there is no protocol achieving broadcast in the presence of a (t_a, t_c)-adversary when $n \leq 2t_a + \min(t_a, t_c)$ and $t_c > 0$, thus proving that Theorem 1 is tight. We

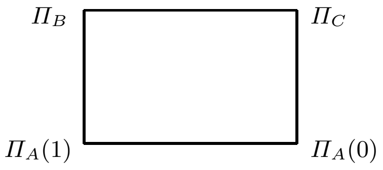


Fig. 2. A mental experiment involving a four-node network

then consider the case when t_a, t_c are not fixed, but instead all that is guaranteed is that $2t_a + \min(t_a, t_c) < n$. (In the previous section, unauthenticated broadcast was used to handle the case $t_a \leq t_c$ and Protocol 1 assumed $t_c < t_a$. Here we seek a *single* protocol that handles both cases.) We show that in this setting, broadcast is impossible for almost all n.

4.1 The Three-Player Case

We first present a key lemma that will be useful for the proofs of both results described above. For this, define a general adversary $\mathcal{A}$ as follows:

Definition 2. *Let $\mathcal{S}$ be a set of pairs $\{(S_a^1, S_c^1), (S_a^2, S_c^2), \ldots\}$ where $S_a^i, S_c^i \subset \{P_1, \ldots, P_n\}$. An $\mathcal{S}$-*adversary *can choose any i, and actively corrupt any subset of the players in S_a^i and additionally compromise the secret keys of any subset of the players in S_c^i.*

We restrict our attention to the case of three parties (named A, B, and C) and $\mathcal{S}$ defined as follows:

$$\mathcal{S} = \left\{ \begin{array}{c} (\{A\}, \emptyset) \\ (\{B\}, \{A\}) \\ (\{C\}, \{A\}) \end{array} \right\}. \tag{1}$$

Lemma 2. *In the presence of an $\mathcal{S}$-adversary, for $\mathcal{S}$ defined as above, there does not exist a protocol achieving broadcast for dealer A.*

Proof. Suppose, towards a contradiction, that there exists a protocol Π for computing broadcast in the presence of an $\mathcal{S}$-adversary when A is the dealer. Let Π_A, Π_B, Π_C denote the code specified by Π for players A, B, and C, respectively.

Consider an experiment in which four machines are arranged in a rectangle (see Figure 2). The top left and top right nodes will run Π_B and Π_C, respectively. The bottom left node will run Π_A using input 1, and the bottom right node will run Π_A using input 0. Public and secret keys for A, B, and C are generated honestly, and both executions of Π_A use the same keys.

Claim. In the experiment of Figure 2, Π_B outputs 1.

Proof. Consider an execution in the real network of three players, in the case where A holds input 1 and the adversary corrupts C and compromises the secret key of A. The adversary then simulates the right edge of the rectangle from Figure 2 while interacting with the (real) honest players A and B (running the code for $\Pi_A(1)$ and Π_B, respectively). That is, every time the corrupted player C receives a message from B the adversary forwards this message to its internal copy of Π_C, and every time C receives a message from A the adversary forwards this message to its internal copy of $\Pi_A(0)$. Similarly, any message sent by Π_C to Π_B is forwarded to the real player B, and any message sent by $\Pi_A(0)$ to $\Pi_A(1)$ is forwarded to the real player A. (Messages between Π_C and $\Pi_A(0)$ are forwarded internally.) This defines a legal $\mathcal{S}$-adversary. If Π is a secure protocol, validity must hold and so B in the real network (and hence Π_B in the mental experiment) must output 1.

Claim. In the experiment of Figure 2, Π_C outputs 0.

The proof is the same as above.

Claim. In the experiment of Figure 2, Π_B and Π_C output the same value.

Proof. Consider an execution in the real network of three players when the adversary corrupts A (and does not further compromise anyone). The adversary then simulates the bottom edge of the rectangle when interacting with the real players B and C, in the obvious way. Since this defines a legal $\mathcal{S}$-adversary, security of Π implies that agreement must hold between B and C in the real network and so the outputs of Π_B and Π_C must agree in the mental experiment.

The three claims are contradictory, and so we conclude that no secure protocol Π exists.

We remark that impossibility holds even if we relax our definition of broadcast and allow agreement/validity to fail with negligible probability.

4.2 Impossibility of Broadcast for $2t_a + \min(t_a, t_c) \geq n$

Theorem 3. *Fix n, t_a, t_c with $t_c > 0$ and $2t_a + \min(t_a, t_c) \geq n$. There is no protocol achieving broadcast in the presence of a (t_a, t_c)-adversary.*

Proof. We prove the theorem by demonstrating that a broadcast protocol Π secure in the presence of a (t_a, t_c)-adversary with $2t_a + \min(t_a, t_c) \geq n$, yields a protocol Π' for 3-player broadcast in the presence of an $\mathcal{S}$-adversary for $\mathcal{S}$ as defined in the previous section. Using Lemma 2, this shows that such a protocol Π cannot exist. In fact, we show this even assuming the dealer is fixed in advance.

Assume that such a protocol Π exists. We construct a protocol Π' for 3-player broadcast by having each player simulate a subset of the players in the n-player protocol Π. The simulation proceeds in the obvious way, by having each of the 3 players run the code of the parties they simulate in Π. They forward any messages sent by the simulated parties to the player simulating the destination party, who uses these as incoming messages for his simulated players. To provide a PKI for the simulated protocol we view the keys of each of the 3 players as consisting of multiple keys. Player A's public key is $PK_A = (pk_1, \ldots, pk_a)$ and his secret key is $SK_A = (sk_1, \ldots, sk_a)$ for some number of simulated players a. The players in the 3-player protocol determine their outputs from the outputs of the players they simulate. If all players simulated by A output the same value b in the simulated protocol, then A outputs b. Otherwise, A outputs a special value $\perp$. Note that an adversarial player can only simulate adversarial players and an honest but compromised player can only simulate compromised players since the adversary learns all the secret keys of player A's simulated players when A's key is compromised.

We let A simulate a set of at most $\min(t_a, t_c)$ players, including the dealer, and let B and C each simulate at most t_a players. Since $2t_a + \min(t_a, t_c) \geq n$, it is possible to do this in such a way that each of the n original players is simulated by one of A, B, or C. We now consider each of the three allowed types of corruption for the adversary $\mathcal{A}$ as per Definition 2, and demonstrate that the corresponding corruption in the n-player protocol is also "legal": that is, we demonstrate that the allowed actions for $\mathcal{A}$ translate into adversarial actions for which the non-faulty players in Π terminate correctly, achieving broadcast in the simulated n-player protocol. This implies a secure 3-player broadcast protocol in the presence of $\mathcal{A}$.

Recall that, by assumption, Π is secure against a (t_a, t_c)-adversary; as long as no more than t_a players are corrupt, and no more than t_c are compromised, Π satisfies the requirements of authenticated broadcast. If $\mathcal{A}'$ chooses the pair $(\{A\}, \emptyset)$, all players simulated by A in Π' are corrupt and the players simulated by B and C are honest and non-compromised. Since, $\min(t_a, t_c) \leq t_a$, this is an allowed corruption for a (t_a, t_c)-adversary, and Π executes correctly implying that Π' terminates with the correct output. Next, if $\mathcal{A}'$ chooses $(\{B\}, \{A\})$ this will result in a $(t_a, \min(t_a, t_c))$ corruption. Since $\min(t_a, t_c) \leq t_c$, this corruption type is also permitted in Π, and Π' executes correctly. Finally, the corruption type $(\{C\}, \{A\})$ is handled identically to that of $(\{B\}, \{A\})$. Since we proved in Lemma 2 that no such protocol Π' exists, this proves the theorem.

4.3 Impossibility of Broadcast with a Threshold Adversary

We now turn to the case of the threshold adversary. Recall that in this setting the exact values of t_a and t_c used by the adversary are not known; we only know that they satisfy $2t_a + \min(t_a, t_c) < n$ (and we do allow $t_c = 0$). In what follows, we show that secure broadcast is impossible if $n \notin \{2, 3, 4, 5, 6, 8, 9, 12\}$. For the "exceptional" values of n, we demonstrate feasibility in Section 5.

Theorem 4. *If $n \leq 2\left\lfloor \frac{n-1}{3} \right\rfloor + \left\lfloor \frac{n-1}{2} \right\rfloor$, then there does not exist a secure broadcast protocol for n players in the presence of a threshold adversary. (Note that $n \leq 2\left\lfloor \frac{n-1}{3} \right\rfloor + \left\lfloor \frac{n-1}{2} \right\rfloor$ for all $n > 1$ except $n \in \{2, 3, 4, 5, 6, 8, 9, 12\}$.)*

Proof. Assume there exists a protocol Π for n satisfying the stated inequality. We show that this implies a protocol Π' for broadcast with 3 players in the presence of the adversary $\mathcal{A}$ from Definition 2. By Lemma 2, we conclude that Π cannot exist. In fact, we show this even assuming the dealer is fixed in advance.

We construct Π' using a player simulation argument as in the previous section. Let A simulate a set of at most $\lfloor \frac{n-1}{2} \rfloor$ players, and including the dealer. B and C each simulate at most $\lfloor \frac{n-1}{3} \rfloor$ players and at least one player. By the stated inequality, it is possible to do this in such a way that A, B, and C simulate all n players. We now show that the three allowed types of corruption for $\mathcal{A}$ (in the 3-party network) are also allowed corruption patterns for the n-player threshold adversary $\mathcal{A}'$.

If $\mathcal{A}$ corrupts A, this corresponds to corruption of $\lfloor \frac{n-1}{2} \rfloor$ players in Π (and no compromised players). Since $2\lfloor \frac{n-1}{2} \rfloor < n$, this is a legal corruption pattern for a threshold adversary and Π should remain secure. If $\mathcal{A}$ corrupts B and compromises A, this corresponds to $t_a = \lfloor \frac{n-1}{3} \rfloor$ players and $t_c = \lfloor \frac{n-1}{2} \rfloor$ players in Π. Since $2 \lfloor \frac{n-1}{3} \rfloor + \min\{\lfloor \frac{n-1}{3} \rfloor, \lfloor \frac{n-1}{2} \rfloor\} = 3 \lfloor \frac{n-1}{3} \rfloor < n$, this is again a legal corruption pattern for a threshold adversary and Π should remain secure. The case when C is corrupted and A is compromised is exactly analogous.

5 Handling the Exceptional Values of n

We refer to $\{2, 3, 4, 5, 6, 8, 9, 12\}$ as the set of *exceptional values* for n. (These are the only positive, integer values of n for which Theorem 4 does not apply.) We show for any exceptional value of n a broadcast protocol that is secure against any threshold adversary. Designing protocols in this setting is more difficult than in the setting of Section 3, since the honest parties are no longer assumed to "know" whether $t_a \leq t_c$.

Our protocol, which we refer to as authLSP, is an authenticated version of the exponential protocol of Lamport et al. [8]; see Figure 3. Although the message complexity of this protocol is exponential in the number of players, the maximum number of players considered here is 12. In this full version of this work [4], we provide a more efficient protocol under the assumption that there is at least one honest and uncompromised player.

We say a message M is *valid* if it has the form $(v, s_{P_1}, \ldots, s_{P_i})$, where all P_j's are distinct, the string s_{P_j} is a valid signature on $(v, s_{P_1}, \ldots, s_{P_{j-1}})$ relative to the verification key of P_j, and one of the s_{P_j} is the signature of the dealer. (We note that authLSP is defined recursively, and the criteria for deciding if a message is valid is defined with respect to the dealer of the *local* execution.) We also assume implicitly that each message has a tag identifying which execution it belongs to. These tags (together with uncompromised signatures) will prevent malicious players from substituting the messages of one execution for those of another execution. We refer to v as the *content* of such a message. When we say that an execution of authLSP satisfies agreement or validity (cf. Definition 1), we mean that the output is a valid message whose *content* satisfies these properties. We note that in the protocol authLSP, it is possible for honest players to have invalid input. In this case, we change the definition of validity slightly to require that all honest players (including the dealer) output messages with content 0. Finally, we let $t_h = n - t_c - t_a$ denote the number of honest and uncompromised parties. One useful observation about threshold adversaries that we will repeatedly use is that when $t_a > \lfloor \frac{n-1}{3} \rfloor$, it follows that $t_h > t_a$.

Protocol $\mathsf{authLSP}(m)$

Inputs: The protocol is parameterized by an integer m. Let D be the dealer with input M of the form $M = (v, s_{P_1}, \ldots, s_{P_i})$ with $0 \leq i \leq n$ (M is not necessarily valid).

Case 1: $m = 0$

1. If the content of M is not in $\{0, 1\}$, D sets $M = 0$.[a] D sends $M_d = (M, \mathsf{Sign}_{sk_D}(M))$ to all other players and outputs M_d.
2. Letting M_i denote the message received by P_i, P_i ouputs M_i.

Case 2: $m > 0$

1. If the content of M is not in $\{0, 1\}$, D sets $M = 0$. D sends $M_d = (M, \mathsf{Sign}_{sk_D}(M))$ to all other players and outputs M_d.
2. Let $\mathcal{P}' = \mathcal{P} \setminus \{D\}$. For $P_i \in \mathcal{P}'$, let M_i denote the message received by P_i from D. P_i plays the dealer in $\mathsf{authLSP}(m-1)$ for the rest of the players in $\mathcal{P}'$, using message M_i as its input.
3. Each P_i locally does the following: for each $P_j \in \mathcal{P}'$, let M_j be the output of P_i when P_j played the dealer in $\mathsf{authLSP}(m-1)$ in step 2. For each M_j, P_i sets value b_j as follows:

$$b_j = \begin{cases} \text{the content of } M_j & \text{if } M_j \text{ is valid} \\ \qquad\perp & \text{otherwise} \end{cases}$$

   (We stress that the above also includes the output of P_i when he was dealer in step 2.) P_i computes $b^* = \mathsf{majority}(b_j)$. If there is no value in strict majority, P_i outputs 0.
4. P_i outputs the first valid message M_j (lexicographically) with content b^*.

[a] As mentioned in the text, we assume the dealer also includes the appropriate tag identifying which execution M belongs to. We do not mention this again going forward.

Fig. 3. Protocol $\mathsf{authLSP}$

The next two lemmas follow readily from [8]; we do not prove them here.

Lemma 3. *If $n > 3m$ and $m \geq t_a$, then* $\mathsf{authLSP}(m)$ *achieves validity and agreement.*

Lemma 4. *If the dealer D is honest and $n > 2t_a + m$, then* $\mathsf{authLSP}(m)$ *achieves validity and agreement.*

We now prove several additional lemmas about $\mathsf{authLSP}$.

Lemma 5. *If the dealer is honest and uncompromised, then* $\mathsf{authLSP}(m)$ *achieves validity and agreement for any m.*

Proof. Let D be the dealer with input that has content b_d. (Recall that if $b_d \notin \{0, 1\}$, then D switches his input for valid input with content $b_d = 0$.) It follows from the protocol description that D outputs a valid message with content b_d. Furthermore, when an honest player is dealer in the recursive call in step 2, it has input and output with content b_d. Therefore, when honest P_i computes $\mathsf{majority}(\mathsf{b_j})$ in step 3 of $\mathsf{authLSP}$, it sets the value $b_i = b_d$. On the other hand, since D is honest and uncompromised, the adversary cannot produce a valid message with content $1 - b_d$ (recall that for a message to be valid, it must contain the signature of the dealer). It follows then that $b_j \neq 1 - b_d$ for all values used to compute majority in step 3. Validity and agreement follow.

Lemma 6. *If the dealer is honest and compromised, and $t_h > t_a$, then protocol* $\mathsf{authLSP}(m)$ *achieves validity and agreement for any m.*

Proof. It is easy to see that the lemma holds for $m = 0$. Let us assume the lemma holds for $\mathsf{authLSP}(m-1)$, and consider $\mathsf{authLSP}(m)$. If an honest and uncompromised player is the dealer in step 2 of $\mathsf{authLSP}(m)$ (i.e. in the recursive call to $\mathsf{authLSP}(m-1)$), then by Lemma 5 this run achieves validity and agreement. If an honest but compromised player is the dealer in step 2, then it still holds in the recursive execution that $t_h > t_a$, since the dealer is not counted in t_h, and all other players participate in the execution of $\mathsf{authLSP}(m-1)$; by the induction hypothesis this execution achieves validity and agreement on output b_d as well. It follows that in step 3 of $\mathsf{authLSP}(m)$, for each honest player P_i, at least $n - t_a - 1$ of the b_j values equal b_d and at most t_a of the b_j values equal $(1 - b_d)$. Since $n - t_a - 1 \geq t_h > t_a$, b_d is the majority value for each honest player, and the lemma follows.

Theorem 5. *For any value $n \in \{2, 3, 4, 5, 6, 8, 9, 12\}$ there exists a protocol for n players that achieves broadcast in the presence of a threshold adversary.*

Proof. The case $n = 2$ is trivial. When $n = 3$, it follows from our constraints that $t_a \leq 1$ and $t_c = 0$, so we can run any authenticated byzantine agreement protocol. When $n = 4$, it follows from our constraints that $t_a \leq 1$, and therefore that $n > 3t_a$, so we can ignore the PKI and run a protocol that is secure without authentication. The remainder of the proof deals with $n \in \{5, 6, 8, 9, 12\}$.

Lemma 7. *For $n \in \{5, 6, 8\}$,* $\mathsf{authLSP}(\lfloor \frac{n-1}{3} \rfloor + 1)$ *achieves broadcast in the presence of a threshold adversary.*

Proof. We prove the lemma by considering all possible types of dealers. We let b_d denote the input bit of the dealer D.

D is honest and not compromised: This case follows from Lemma 5.

D is honest and compromised: Consider the following two scenarios:

$\mathbf{t_a} \leq \lfloor \frac{\mathbf{n-1}}{\mathbf{3}} \rfloor$: For $n \in \{5, 6, 8\}$, we have $n > 2\lfloor \frac{n-1}{3} \rfloor + \lfloor \frac{n-1}{3} \rfloor + 1 \geq 2t_a + m$, where the first inequality holds because of our assumption on n, and the second from our assumption on t_a. Applying Lemma 4 we get validity and agreement as claimed.

$\mathbf{t_a} > \lfloor\frac{\mathbf{n-1}}{\mathbf{3}}\rfloor$: Since we assume a threshold adversary, in this case we have $t_h > t_a$ (cf. section 2). Applying Lemma 6, agreement and validity follow.

D is malicious: Since we assume a threshold adversary, we have that $n > 2t_a$. We note that the malicious dealer is excluded from each of the executions of $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor)$ in step 2, and therefore, of the $n-1$ players that participate in those executions, only $t_a - 1$ are malicious. The reader can verify that for $n \in \{5, 6, 8\}$, $n - 1 > 3\lfloor\frac{n-1}{3}\rfloor$, and that $\lfloor\frac{n-1}{3}\rfloor \geq t_a - 1$ (recalling that $t_a < \frac{n}{2}$). Applying Lemma 3, we have agreement in each execution of $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor)$ in step 2. Agreement in $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 1)$ follows when the players compute their output in steps 3 and 4.

Lemma 8. *For $n \in \{9, 12\}$, Protocol* $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 2)$ *achieves broadcast in the presence of a threshold adversary.*

Proof. We prove the lemma by considering all possible types of dealers.

D is honest and uncompromised: This case follows from Lemma 5.

D is honest and compromised: Consider the following two scenarios:

$\mathbf{t_a} \leq \lfloor\frac{\mathbf{n-1}}{\mathbf{3}}\rfloor$: For $n \in \{9, 12\}$, we have $n > 2\lfloor\frac{n-1}{3}\rfloor + \lfloor\frac{n-1}{3}\rfloor + 2 \geq 2t_a + m$, where the first inequality holds by our assumption on n, and the second holds by our assumption on t_a. Applying Lemma 4 we get validity and agreement as claimed.

$\mathbf{t_a} > \lfloor\frac{\mathbf{n-1}}{\mathbf{3}}\rfloor$: Because we assume a threshold adversary, we have $t_h > t_a$ (cf. section 2). Applying Lemma 6, agreement and validity follow.

D is malicious: We consider the recursive execution of $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 1)$ in step 2, and prove agreement for each of the $n-1$ dealers. When the dealer in step 2 is honest and uncompromised, by Lemma 5 we have agreement in his execution. If the dealer is honest and compromised we consider two further possibilities. If $t_a \leq \lfloor\frac{n-1}{3}\rfloor$, then among the $n-1$ players participating in this recursive execution, of which at most $t_a - 1$ are malicious, we have

$$\begin{aligned} n - 1 > 3\left\lfloor\frac{n-1}{3}\right\rfloor - 1 &= 2\left(\left\lfloor\frac{n-1}{3}\right\rfloor - 1\right) + \left(\left\lfloor\frac{n-1}{3}\right\rfloor + 1\right) \\ &\geq 2(t_a - 1) + \left(\left\lfloor\frac{n-1}{3}\right\rfloor + 1\right). \end{aligned}$$

By Lemma 4, agreement follows. If the dealer is honest and compromised and $t_a > \lfloor\frac{n-1}{3}\rfloor$, then $t_h > t_a$ and by Lemma 6 agreement follows. If the dealer in step 2 is malicious, consider what happens in the *next* recursive step when the players execute $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor)$. Now two malicious dealers have been excluded: both the dealer in $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 2)$ and the dealer in $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 1)$. Noting that the maximum number of malicious players is 4 when $n = 9$ and 5 when $n = 12$ (because we have a threshold adversary), it follows that among the remaining $n-2$ players, $n-2 > 3(\lfloor\frac{n-1}{3}\rfloor)$ and $\lfloor\frac{n-1}{3}\rfloor \geq t_a - 2$. Applying Lemma 3, we have agreement for all dealer types in $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor)$, and agreement follows for all malicious dealers in the executions of $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 1)$. Since we have proven agreement for all dealer types in $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 1)$, we have agreement in the execution of $\mathsf{authLSP}(\lfloor\frac{n-1}{3}\rfloor + 2)$ as well.

This concludes the proof of Theorem 5.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the US Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

References

1. Dolev, D., Strong, H.: Authenticated algorithms for Byzantine agreement. SIAM Journal on Computing 12(4), 656–666 (1983)
2. Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: 49th Annual Symposium on Foundations of Computer Science (FOCS), pp. 293–302. IEEE, Los Alamitos (2008), http://eprint.iacr.org/2008/240
3. Fitzi, M., Holenstein, T., Wullschleger, J.: Multi-party computation with hybrid security. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 419–438. Springer, Heidelberg (2004)
4. Gordon, S., Katz, J., Kumaresan, R., Yerukhimovich, A.: Authenticated broadcast with a partially compromised public-key infrastructure (2009), http://eprint.iacr.org/2009/410
5. Gupta, A., Gopal, P., Bansal, P., Srinathan, K.: Authenticated Byzantine generals in dual failure model. In: Distributed Computing and Networking (ICDCN). LNCS, vol. 5935, pp. 79–91. Springer, Heidelberg (2010)
6. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
7. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
8. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine generals problem. ACM Trans. Programming Language Systems 4(3), 382–401 (1982)
9. MS00-008: Incorrect registry setting may allow cryptography key compromise. Microsoft Help and Support, http://support.microsoft.com/kb/259496
10. Nguyen, P.Q.: Can we trust cryptographic software? Cryptographic flaws in GNU privacy guard v1.2.3. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 555–570. Springer, Heidelberg (2004)
11. Ostrovsky, R., Yung, M.: How to withstand mobile virus attacks. In: 10th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 51–59. ACM Press, New York (1991)
12. Pease, M., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)

A The Dolev-Strong Protocol

For completeness, we present a modified version of the Dolev-Strong [1] protocol for authenticated broadcast. (See Figure 4.) A message M is called (v, i)-*valid* if it was received in round i and has the form $(v, s_{P_1}, \ldots, s_{P_i})$, where $P_1 = D$, all P_j's are distinct, and for every $j = 1, \ldots, i$ the string s_{P_j} is a valid signature on $(v, s_{P_1}, \ldots, s_{P_{j-1}})$ relative to the verification key of P_j. We refer to v as the *content* of a valid message. If the dealer is honest and uncompromised, there will only be $(v, \star)$-valid messages for a single value v, in which case the players consider the execution *clean*. Otherwise, the execution is called *dirty*. We note that the protocol differs from the original Dolev-Strong [1] protocol only in round complexity: we always require $n + 1$ rounds to ensure that all players agree whether the run was dirty. We also assume at least one honest and uncompromised player.

DS

Inputs: Let D be the dealer with input $b_d \in \{0,1\}^*$ and secret key sk_D.

1. (Round $r = 0$) D sends $(b_d, \mathsf{Sign}_{sk_D}(b_d))$ to every player.
2. In round $r = 1$ to n:
    1. Every player P_i checks every incoming message and discards any that are not $(\cdot, r)$-valid or that already contain P_i's signature. P_i orders the remaining messages lexicographically.
        - If the content, v, of all remaining messages is identical, P_i appends its signature to the first message (thus forming a $(v, r + 1)$-valid message) and sends the result to all players.
        - If there exist 2 messages with different content, P_i appends its signature to the first 2 such messages and sends the result to all players.
    2. Termination:
        1. If P_i ever received valid messages with different content, then it outputs a default value.
        2. If P_i only received valid messages for one value v, then it outputs v.
        3. If P_i never received a valid message for either $v \in \{0,1\}$ then it outputs a default value.

Fig. 4. The Dolev-Strong protocol for broadcast

On Applicability of Random Graphs for Modeling Random Key Predistribution for Wireless Sensor Networks

Tuan Manh Vu, Reihaneh Safavi-Naini, and Carey Williamson

University of Calgary, Calgary, AB, Canada

Abstract. We study the applicability of random graph theory in modeling secure connectivity of wireless sensor networks. Specifically, our work focuses on the highly influential random key predistribution scheme by Eschenauer and Gligor to examine the appropriateness of the modeling in finding system parameters for desired connectivity. We use extensive simulation and theoretical results to identify ranges of the parameters where i) random graph theory is not applicable, ii) random graph theory may lead to estimates with excessive errors, and iii) random graph theory gives very accurate results. We also investigate the similarities and dissimilarities in the structure of random graphs and key graphs (i.e., graphs describing key sharing information between sensor nodes). Our results provide insights into research relying on random graph modeling to examine behaviors of key graphs.

1 Introduction

Wireless sensor networks (WSNs) are *ad-hoc* networks that consist of hundreds to thousands of small sensor nodes communicating wirelessly to collect and deliver data to base stations. Generally, sensor networks rely on symmetric key algorithms to avoid the high computation cost of public key crypto-systems such as Diffie-Hellman key exchange [3]. Furthermore, traditional methods of key establishment that use a trusted authority (e.g., Kerberos protocol [16]) are not suitable due to the frequently used unattended deployments of WSNs.

Eschenauer and Gligor (EG) [6] pioneered an innovative randomized approach to key establishment that provides an efficient and self-organising way of constructing pairwise keys for sensor nodes with guaranteed security. In the EG scheme, every sensor receives a random subset of m keys called a *key ring*, from a *key pool* of N keys. Once deployed, sensor nodes broadcast the identifiers of keys in their key rings to discover wireless neighbors with whom they have at least one key in common, and then establish *secure links*.

The *key graph* for a WSN of n sensor nodes is a graph with n vertices in which each node is represented by a vertex, and two vertices are joined by an edge if the two corresponding nodes share at least one key. The important question is how to choose key ring size m and key pool size N so that the key graph is connected with desired probability c. A connected key graph implies that any two nodes can set up a pairwise key. Eschenauer and Gligor modeled the key graph as an *Erdős-Rényi random graph* $G(n,p)$, a graph of n nodes where each possible edge exists independently with a fixed probability p. This modeling allows to determine m given N and n such that secure connectivity is guaranteed with probability c. The EG scheme has generated a flurry of research on key predistribution for WSNs that employ Erdős-Rényi's theory of random graphs to study connectivity [2,4,10,13,14,15]. This modeling makes two crucial assumptions:

1. *Asymptotic results provide accurate prediction for all network sizes.* Erdős-Rényi[5] studied asymptotic behavior of random graphs, showing that as $n \to \infty$ there are certain properties that will *almost surely* appear in the graph, once the edge probability exceeds a threshold that depends on the property. Connectivity is one such property; a random graph with edge probability $p = \frac{\mathtt{ln}(n)-\mathtt{ln}(-\mathtt{ln}(c))}{n}$ is connected with probability c. It is however unclear if choosing p as above for all possible values of n will achieve connectivity with probability c.
2. *Edge probability in key graphs is fixed and edge probabilities are independent.* In key graphs, although the probability that an edge between two arbitrarily given nodes exists is fixed, these probabilities are not independent of the existence of other edges in the graph [17]. For example consider a key graph for a network with three nodes X, Y, and Z, and key ring size $m = 2$. Using the results from [6] for sufficiently large key pool size, say $N > 10$, the probability that two nodes share at least one key is less than 50% (see Equation 5). However, if the two pairs (X, Y) and (Y, Z) each share a key, then the probability that X and Z have at least one key in common exceeds 50%, regardless of N. This is because both X and Z have at least one key in a fixed set of two keys (i.e., key ring of Y). This is referred to as the *transitivity* property. There are also certain values of (N, m) for which the key graph is clearly not an Erdős-Rényi random graph. For instance, if each node has only one key (i.e., $m = 1$), then the set of all nodes having the same key forms a complete graph.

The goals of this paper are i) to study the applicability of random graph theory in estimating key ring size (for desired connectivity probability) in key graphs, and ii) to compare structural properties of the two families of graphs. The latter study is motivated by application of random key predistribution in secure group connectivity in sensor and *ad-hoc* networks.

1.1 Our Work

We consider the following two questions:

Q1 Can random graphs be always used to model secure connectivity achieved by random key predistribution systems, and when applicable how accurate are the estimated key ring sizes?

Q2 How similar are the structural properties of the two graph families?

We provide two types of results: (i) analytical results and (ii) simulation results. The latter results are obtained by constructing many random instances of key graphs and determining the relative frequency of instances that have the property of interest. For this, we developed a simulator [19] that can efficiently generate key graphs from the prescribed random key assignment process. Using results from statistics we estimate the measurement error and confidence intervals for the obtained values.

Our results for Q1: The simulated key ring sizes for a network of size n differ from the analytical results. The differences can be categorized into four intervals delineated by three threshold values $n_1^{[N,c]}, n_2^{[N,c]}, n_3^{[N,c]}$.

1. Interval I: $n \leq n_1^{[N,c]}$. Random graph theory cannot be used as it results in $p > 1$ which is an invalid edge probability. The value $n_1^{(N,c)}$ is independent of key pool size N and is determined only by connectivity c.
2. Interval II: $n_1^{[N,c]} < n < n_2^{[N,c]}$. Random graph theory can be used in this interval, however key ring sizes predicted by random graph modeling are slight over-estimation. We define over-estimation as the relative error $\geq 5\%$ *and* the absolute error > 2 (see Section 3.2 for more details). When the network size n and connectivity c are fixed, the value of $n_2^{[N,c]}$ increases as N grows. Nevertheless, the empirical simulation results suggest that for wide ranges of parameters (see Table 2), $n_2^{[N,c]}$ never exceeds 100.
3. Interval III: $n_2^{[N,c]} \leq n < n_3^{[N,c]}$. In this interval, random graph modeling provides very good estimates for m (i.e., compared to key ring sizes obtained by simulations, either the relative error $< 5\%$ or the absolute error ≤ 2), and the predicted key ring size ≥ 2. The value $n_3^{[N,c]}$ depends on connectivity c and key pool size N.
4. Interval IV: $n \geq n_3^{[N,c]}$. In this interval, random graph modeling always gives $m = 1$. This however does not provide connectivity except for the trivial case where all nodes have the same key. Simulation results suggest that m should be 2 instead.

The results suggest that random graph theory, when used within appropriate parameter ranges, provides very good estimates of key predistribution parameters for achieving desired secure connectivity of WSNs.

Our results for Q2: We consider the following structural properties:

1. Global clustering coefficient.
2. The size of the maximal clique.
3. The number of cliques with respect to clique sizes.

These properties are important in studying key predistribution as well as routing in WSN. Global clustering is a measure of transitivity that also enables us to explain behavior of the other two properties. Cliques are used in many *ad-hoc* algorithms for constructing group-wise keys, detecting intrusion and choosing group leaders (i.e., clusterheads in WSNs), as well as algorithms for finding capacity, quality of service, and routing [8,11,12,21]. Our study shows that:

1. Global clustering coefficient of key graphs deviates significantly from that of random graphs for smaller key ring sizes, but starts to converge as the key ring size increases.
2. Key graphs contain many more cliques than do random graphs of the same size.
3. The maximal clique size observed in a key graph is much larger than in random graphs of the same size.

The rest of this paper is organized as follows. In Section 2, we give background preliminaries and definitions. Section 3 explains our methodology and results for estimating key ring size. Structural properties of the two graph families are compared in Section 4 and finally, Section 5 provides concluding remarks and directions for future work.

2 Preliminaries

2.1 Random Graph

An Erdős-Rényi random graph, denoted by $G_r(n, p)$, is a graph of size n generated through the following random process. First, we start with n vertices and no edges. Next, each of $\frac{n\cdot(n-1)}{2}$ possible edges between vertex pairs is added to the graph with probability p, determined by a biased coin flip. A graph obtained through this random and independent edge generation process is an instance from a family of random graphs.

Connectivity
The probability of $G_r(n, p)$ being connected is the probability that the random and independent edge generation process with parameters n and p results in a connected graph. Erdős and Rény showed that,

$$\lim_{n\to\infty} \mathtt{Pr}\left[G_r\left(n,p\right) \text{ is connected}\right] = c, \text{ where } p = \frac{\mathtt{ln}(n) - \mathtt{ln}(-\mathtt{ln}(c))}{n} \quad (1)$$

Global clustering coefficient
Holland and Leinhardt [9] introduced the notion of global clustering coefficient C. This is an important measurement in studying social and real-world networks to examine the property, "a friend of my friend is likely to be my friend as well". In other words, C implies transitivity relation between pairs of vertices: if there exists an edge between vertices (X, Y) and an edge between vertices (Y, Z), then there is a high probability that there is an edge between vertices (X, Z). Global clustering coefficient is defined as a metric for a particular graph. In the family of random graphs, since each edge occurs independently, we have

$$\mathtt{E}[C] = p. \quad (2)$$

Number of cliques
A clique in an undirected graph G is a subset S of vertices such that every two vertices in S are connected by an edge. Let the random variable $\Gamma_k(G)$ denote the number of cliques of size k in G. Given a subset S of k vertices in $G_r(n,p)$, the number of pairs of vertices is $\frac{k\cdot(k-1)}{2}$, and thus the probability of S being a clique is $p^{k\cdot(k-1)/2}$ [1] and

$$\mathtt{E}[\Gamma_k(G_r(n,p))] = \binom{n}{k} \cdot p^{k\cdot(k-1)/2}. \tag{3}$$

Maximal clique size
Let the random variable $\Upsilon(G)$ denote the maximal size of a clique in an undirected graph G. Specifically, $\Upsilon(G) = \mathtt{max}\{k : \Gamma_k(G) > 0\}$. Grimmett and McDiarmid [7] studied asymptotic behavior of the maximal clique size in random graphs, showing that

$$\lim_{n\to\infty} \frac{\Upsilon(G_r(n,p))}{\mathtt{ln}(n)} = \frac{2}{\mathtt{ln}(1/p)}. \tag{4}$$

2.2 Key Graph

A key graph $G_k(n,N,m)$ describing key sharing information between nodes is constructed through the following random key assignment process. We start with n nodes, each with a randomly chosen key ring of size m from a key pool of size N. For every two nodes X and Y that share at least one key, we add the edge (X,Y). All instances of graphs corresponding to all possible key assignments with parameters (n,N,m) define a family of key graphs.

2.3 Modeling Key Graphs Using Erdős-Rényi Random Graph Theory

Two arbitrary nodes are joined by an edge if their assigned key rings intersect. As shown by Eschenauer and Gligor [6], this occurs with probability

$$p_{\mathtt{key\ sharing}} = 1 - \frac{((N-m)!)^2}{N!\cdot(N-2\cdot m)!}. \tag{5}$$

Assuming $G_k(n,N,m)$ is $G_r(n,p_{\mathtt{key\ sharing}})$, Equation 1 suggests that to achieve connectivity c, $p_{\mathtt{key\ sharing}}$ should be at least $\frac{\mathtt{ln}(n)-\mathtt{ln}(-\mathtt{ln}(c))}{n}$. This allows key ring size m to be estimated as a function of n, N, and c.

3 Applicability of Random Graph Theory in Estimating Key Ring Size

In the following, we give our theoretical results and explain how simulation data are obtained, and then discuss our observations.

3.1 Framework

Using Random Graph Theory: As explained in Section 2.3, for network size n, key pool size N, and desired connectivity c, key ring size m is estimated as the smallest integer that satisfies the following

$$1 - \frac{((N-m)!)^2}{N! \cdot (N - 2 \cdot m)!} \geq \frac{\mathtt{ln}(n) - \mathtt{ln}(-\mathtt{ln}(c))}{n}. \tag{6}$$

Using Simulation: To find true connectivity probability c for key graphs with parameters (n, N, m), we need to generate all key graphs with these parameters and find the ratio of the connected graphs to the total number. The number of key assignments, however, grows exponentially with n, N, and m, which makes this calculation infeasible. We therefore use random sampling of the set of key graphs to estimate connectivity. By selecting a sufficiently large sample size, the simulation results give, with high confidence, an accurate estimate of c.

We use two algorithms. Algorithm 1 takes inputs n, N, m, and the size S of the sample set, generates S random instances of $G_k(n, N, m)$, examines their connectivity, and calculates the ratio of connected key graphs to S. This is the estimation of connectivity $\hat{c}$ of $G_k(n, N, m)$. Algorithm 2 performs binary search on the given interval (i.e., the algorithm inputs) to determine the smallest m such that $\hat{c} \geq c$. This method works correctly since when network size n and key pool size N are fixed, connectivity probability c monotonically increases as key ring size m grows. The pseudocode for both algorithms are given in the appendix.

Error and Confidence Interval: In the experiments, we use a sample size of $S = 10,000$ to achieve a reasonable statistical behavior. In particular, if $\hat{c}$ is the estimated connectivity of $G_k(n, N, m)$ obtained by Algorithm 1, the standard deviation of $\hat{c}$ is $\sigma = \sqrt{\frac{\hat{c}\cdot(1-\hat{c})}{S}} = \frac{\sqrt{\hat{c}\cdot(1-\hat{c})}}{100}$. With the 99% confidence level, the true connectivity c falls within $z^* \cdot \sigma$ from $\hat{c}$, where z^* is the critical value corresponding to the desired confidence level. For the 99% confidence level, we have $z^* = 2.58$. Table 1 summarizes the confidence intervals for different values of $\hat{c}$, showing that 10,000 random samples give a very good estimate of graph connectivity. For instance, if $G_k(n, N, m)$ is connected with probability $\hat{c} = 0.8$ in the experiment, then the true connectivity c lies in the interval (0.8 ± 0.0103) 99% of the time.

Table 1. Accuracy of graph connectivity simulation results with 99% confidence level

Connectivity $\hat{c}$	Std error σ	Margin of error ($z^* \cdot \sigma = 2.58 \cdot \sigma$)	Confidence interval
0.500	0.0050	0.0129	0.500 ± 0.0129
0.700	0.0046	0.0119	0.700 ± 0.0119
0.900	0.0030	0.0077	0.900 ± 0.0077
0.999	0.0003	0.0008	0.999 ± 0.0008

Data Sets: We obtain key ring size m theoretically and using simulation over wide ranges of n, N, and c as indicated in Table 2. Our observations are discussed in the next section.

Table 2. Data sets

Parameter	Range
Network size n	$\{3..100,\ i \cdot 10^j \mid i = 2..10, j = 2..3\}$
Key pool size N	$\{10,\ \frac{1}{4} \cdot 10^i,\ \frac{1}{2} \cdot 10^i,\ \frac{3}{4} \cdot 10^i,\ 10^i \mid i = 2..5\}$
Desired connectivity c	$\{0.5, 0.6, 0.7, 0.8, 0.9, 0.99, 0.999\}$

3.2 The Results

We first compare m obtained theoretically and by simulation when both N and c are fixed, and discuss how the value of n affects the applicability and accuracy of random graph modeling. We then extend these results to all n, N, and $c \geq 50\%$.

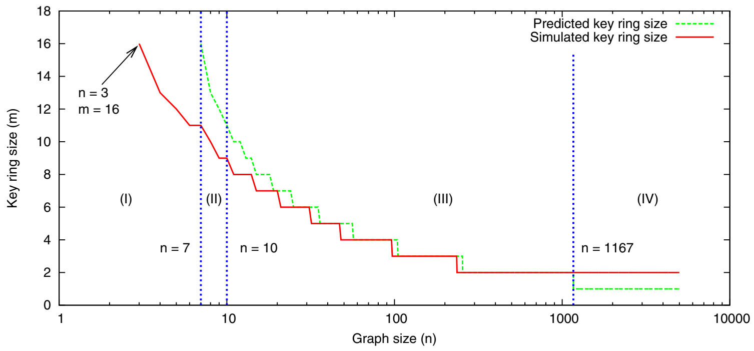


Fig. 1. Key ring size with respect to network size when key pool size $N = 100$ and desired connectivity $c = 99\%$

Figure 1 plots key ring sizes with respect to network size n when $N = 100$ and $c = 99\%$. In Figure 1, we can identify four distinct intervals.

Interval I, $n \leq n_1^{[N,c]}$ - Small graphs: Figure 1 does not have the plot for theoretical key ring size for $n \leq 6$. This is because, for $n \leq 6$, $N = 100$, and $c = 99\%$, random graph theory estimates edge probability $p > 1$ (see Equation 1), which cannot be achieved.

Finding $n_1^{[N,c]}$: Random graph theory suggests that $p \geq \frac{\mathtt{ln}(n)-\mathtt{ln}(-\mathtt{ln}(c))}{n}$ where connectivity $c \in (0, 1)$. Additionally, $n \in [3, \infty)$ as only networks with at least three nodes are of interest. We want to determine for which values of c and n in their respective domains, $\frac{\mathtt{ln}(n)-\mathtt{ln}(-\mathtt{ln}(c))}{n}$ becomes an invalid probability. Due to the limited space, we do not go into the details but summarize the results instead. Basically, we examine the variation of $p(n) = \frac{\mathtt{ln}(n)-\mathtt{ln}(-\mathtt{ln}(c))}{n}$, and find when $p(n)$ is equal to 0 and 1, and reaches the maximum. From that, we make the following observations.

1. When c is close to 0, there is a bound $n_0^{[N,c]}$ of n such that $\forall n \in [3, n_0^{[N,c]}]$: $p(n) < 0$. Explicitly, let $n_0^{[N,c]}$ be the solution to $p(n) = 0$, $n_0^{[N,c]} = -\mathtt{ln}(c)$. It follows that, $n_0^{[N,c]} \geq 3$ (i.e., $-\mathtt{ln}(c) \geq 3$) if and only if $c < c^\dagger$ where $c^\dagger = e^{-3} \simeq 0.05$. In short, $\forall c < c^\dagger, \forall n \in [3, -\mathtt{ln}(c)] : p(n) < 0$. In such cases, random graph theory suggests that choosing edge probability 0 (i.e., key ring size $m = 0$) will achieve connectivity c. However, if $m = 0$, the key graph is surely disconnected, and c is 0. Nevertheless, since c is too small (i.e., $c < 0.05$) to be considered in an actual WSN deployment, we are not interested in this case.
2. When c is close to 1, there is a bound $n_1^{[N,c]}$ of n such that $\forall n \in [3, n_1^{[N,c]}]$: $p(n) > 1$. That means the edge probability estimated by random graph theory is larger than 1, which cannot be achieved. This range is noted as (I) in Figure 1 for the particular parameters $N = 100$ and $c = 0.99$. In such cases, random graph theory is not applicable. We define $n_1^{[N,c]}$ as

$$n_1^{[N,c]} = \mathtt{max}\{n : p(n) > 1 \ \mathtt{and}\ n \geq 3\}.$$

The value of $n_1^{[N,c]}$ depends on c only, and grows as c increases. Let c^* be the solution to $\frac{\mathtt{ln}(3)-\mathtt{ln}(-\mathtt{ln}(x))}{3} = 1$, we have $c^* = e^{-e^{\mathtt{ln}(3)-3}} \simeq .86$, and $\forall c < c^*, \forall n \geq 3 : p(n) < 1$ (i.e., $n_1^{[N,c]}$ does not exist). Some examples of $n_1^{[N,c]}$ are presented in Table 3. One cannot use random graph theory to estimate m given n and c if $c \geq c^*$ and $n \in [3, n_1^{[N,c]}]$.

Table 3. Lower bound of network size n with respect to connectivity c

Desired connectivity c	$n_1^{[N,c]}$	Desired connectivity c	$n_1^{[N,c]}$
0.8	n/a	0.9	3
0.99	6	0.999	9

Interval II - Over-estimation: We compare the theoretical and simulated key ring sizes. We define *over-estimation* when the relative error $\geq 5\%$ and the absolute error > 2. Both conditions are required since considering only one kind of error may be misleading. For example, if theoretical and simulated values of m are 5 and 4, respectively, then the random graph modeling estimate is off by only one key and we consider it as a good estimation despite the relative error being 25%. On the other hand, if the two values are 111 and 107, respectively, then the relative error is less than 4% while the absolute error is 4 keys.

For $n \in [7, 10)$, $N = 100$, and $c = 99\%$, random graph theory results in over-estimation. For wide ranges of parameters where N is up to 10^5, n is up to 10^4, and c is from 50% to 99.9%, the comparison results suggest that there is a small interval of n in which using random graph theory gives over-estimation. The left side of this interval is $n_1^{[N,c]} + 1$ if $c \geq c^*$, or 3 otherwise.

Finding $n_2^{[N,c]}$*:* For fixed c, the right hand side of the interval increases as N grows but over the ranges of parameters summarized in Table 3, it never exceeds 100. Thus, when $n < 100$, our simulation results show that an excessive over-estimation *may* occur. [1] One can always use the simulator [19] to obtain key ring size m by simulations for better estimate.

Interval III - Good estimation: Figure 1 shows that when $N = 100$ and $c = 99\%$, random graph theory gives accurate estimates for $n \in [10, 1167)$. Specifically, the theoretical key ring sizes perfectly match the simulated ones in most cases. In other cases, random graph theory over-estimates by only one key. We did not observe any under-estimation. When $n \geq 1167$, the theoretical results show $m = 1$, while the simulation experiments yield $m = 2$. This phenomenon as well as at which values of n it occurs (e.g., $n = 1167$ for $N = 100$ and $c = 99\%$) is further discussed later in this section.

Let us assume that $n_3^{[N,c]}$, which is a function of c and N, is a lower bound for n such that random graph modeling estimates $m = 1, \forall n \geq n_3^{[N,c]}$. *We call the interval* $[100, n_3^{[N,c]})$ *safe for using random graph theory for estimating* m*.* Table 5 presents selected comparison results of key ring sizes for $c = 99.9\%$ and different values of n and N. Aside from the correct estimates and a few over-estimates with low relative errors, the theoretical key ring size is 1 when $N = 100$ for some cases. In these cases, n is outside the safe interval (i.e., $n \geq n_3^{[100,0.999]}$). Generally, when the key pool size is large and network size is small, over-estimation may occur, but the relative error is less than 5%. For small key pool sizes, if there is an over-estimation, the absolute error is only one or two keys.

Tables 6a-d in Appendix show the difference between theoretical and simulated key ring sizes with respect to n, N, and c. A light-gray cell represents an under-estimation while dark-gray cell indicates an over-estimation. Again, we can see that there is a pattern of under-estimating key ring sizes when n is large and N is small in all four tables. In those cases, m is estimated by random graph theory as 1, and n is outside the safe interval. There are rare situations in which random graph theory under-estimates m when $n \in [100, n_3^{[N,c]})$. One such case is when $n = 1000$, $N = 50000$, and $c = 70\%$ as shown in Table 6b. We believe this is due to statistical error.

As noted earlier, over-estimation occurs when N is very large and n is small; the relative error in such cases, however, is less than 5%. In general, when n lies in the safe interval, the data in Tables 6a-d supports the claim that, *the estimate for key ring size based on random graph theory is very precise for* $n \in [100, n_3^{[N,c]})$.

[1] Over-estimation leads to extra keys in the key rings. In the case of an eavesdropping adversary, this only results in less efficient (larger key ring size) systems. In the case of a node capturing adversary, larger key rings result in higher probability of edge compromise.

Interval IV - Large graphs: For a fixed N, the key sharing probability for $m = 1$ is $\frac{1}{N}$. As the network size n increases, the edge probability p that is required for connectivity c given by $p = \frac{\mathtt{ln}(n)-\mathtt{ln}(-\mathtt{ln}(c))}{n}$ decreases. For sufficiently large n, we will have $\frac{\mathtt{ln}(n)-\mathtt{ln}(-\mathtt{ln}(c))}{n} \leq \frac{1}{N}$. Therefore, according to random graph theory, connectivity c can be achieved with $m = 1$. In this case, nodes that have the same key can be grouped together and so the graph can be decomposed into disjoint cliques. The key graph is connected only if all n nodes have the same key and this happens with probability $\frac{1}{N^{n-1}}$, which could be much lower than desired connectivity c. We define $n_3^{[N,c]}$ as follows

$$n_3^{[N,c]} = \mathtt{min}\{n : \frac{\mathtt{ln}(n) - \mathtt{ln}(-\mathtt{ln}(c))}{n} \leq \frac{1}{N}\}.$$

Given desired connectivity c and key pool N, random graph theory always gives incorrect estimate for key ring size (i.e., m is estimated as 1) when $n \geq n_3^{[N,c]}$. Figure 2 plots $n_3^{[N,c]}$ with respect to key pool size N and connectivity c.

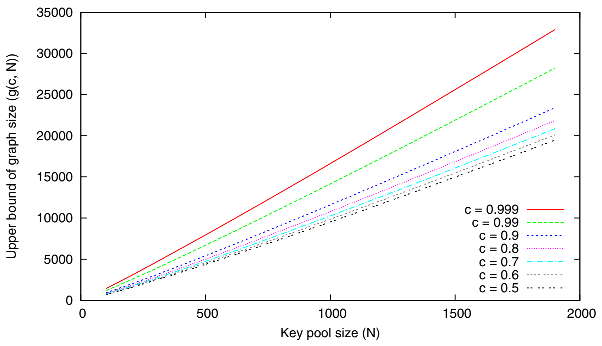


Fig. 2. Upper bounds of network size n with respect to connectivity c and key pool size N

Summary. Table 4 describes four intervals of network size n for given connectivity c and key pool N. These intervals provide insights into the applicability of random graph theory in estimating key ring size to achieve connectivity c when $c \geq 50\%$. For $c < 50\%$, one would expect similar interval structure.

4 Structural Properties

We examine and compare structural properties of random graphs and key graphs for given n and p, that is the same network size and the same edge probability. We use the following approach:

1. For each set of parameters (n, N, m), we calculate the edge probability p according to Equation 5.

Table 4. Four intervals of network size n with respect to key pool size N and desired connectivity c

		$0.5 \leq c < c^*$	$c \geq c^*$
n	Interval I	n/a	$[3,\ n_1^{[N,c]}]$
	Interval II	$[3,\ 100)$	$(n_1^{[N,c]},\ 100)$
	Interval III	$[100,\ n_3^{[N,c]})$	
	Interval IV	$[n_3^{[N,c]},\ +\infty)$	

2. We then generate α random instances of $G_k(n, N, m)$ and measure the average value of property X.
3. Finally, we compare the simulation result with the theoretical values obtained for $G_r(n, p)$. The parameters n, N, and m are chosen to achieve 'reasonable' edge probability p.

The number of random instances of key graphs in each set of simulation experiments is $\alpha = 1,000$. In Figures 3-5, the simulation results are plotted with the error bars indicating the 99% confidence intervals for the true mean.

4.1 Global Clustering Coefficient

In the first set of simulation experiments, we measure the global clustering coefficient C in the two graph families. Recall that in a random graph, each edge occurs independently, and thus C is always equal to the edge probability. However, this value in key graphs gives the probability that two nodes share keys if they both share keys with some common node. In the experiments, we choose $n = 100$, $N = 1000$, and observe C as m varies. Figure 3 provides our comparison results. It can be seen that for very small values of m, C in key graphs is much higher than that in random graphs. This is because if nodes X and Y share key k_1, nodes Y and Z share key k_2, and m is small, then it is likely that k_1 is k_2, which means X and Z have at least one key in common. Nevertheless, when m is large enough, C in key graphs converges to C in random graphs. In the case of $n = 100$ and $N = 1000$ in our experiments, when the key ring size is 25 or more, there is hardly any difference between the global clustering coefficients C in the two graph families.

4.2 Size of the Maximal Clique

In the second set of simulation experiments, we study the maximal clique in key graphs. The simulations assume that N is 1000 and m is 11 (i.e., edge probability $p \simeq 0.1$). The comparison results summarized in Figure 4 show that the average size of the maximal clique in key graphs increases linearly with the network size n. On the other hand, the expected size of the maximal clique in random graphs grows very slowly as n increases. In the following, we give a lower bound of the size of the maximal clique in a key graph, to explain the linear behavior.

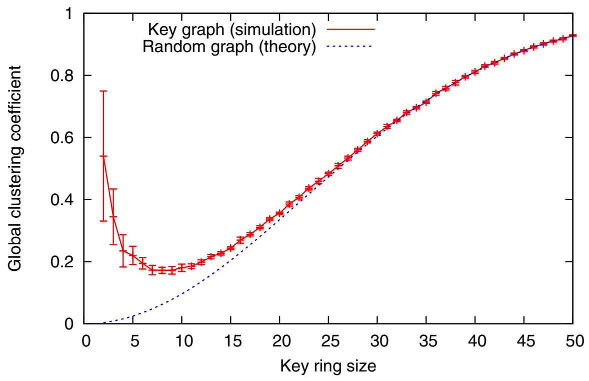


Fig. 3. Global clustering coefficient with respect to key ring size ($n = 100$, $N = 1000$)

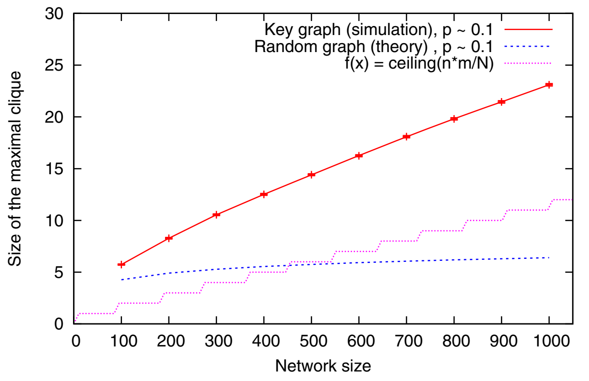


Fig. 4. Size of the maximal clique with respect to network size ($N = 1000$, $m = 11$)

Proposition 1. *The maximal clique size a in key graph is at least* $\lceil \frac{n \cdot m}{N} \rceil$.

Proof: Each of n nodes has m keys, and the total number of keys including duplicated ones is $n \cdot m$ keys. Since the number of distinct keys is at most N, by pigeon hole principle, there exists some key k duplicated at least $\lceil \frac{n \cdot m}{N} \rceil$ times. In other words, at least $\lceil \frac{n \cdot m}{N} \rceil$ nodes have the same key k, hence they form a clique. Since keys are distributed randomly, the frequencies of occurrence for each key may vary in a given random key assignment. That is, the number of nodes having the same key can well exceed $\lceil \frac{n \cdot m}{N} \rceil$. Furthermore, it is not required that all the nodes in a clique must have the same key. Thus, $\lceil \frac{n \cdot m}{N} \rceil$ is a loose lower bound for the size of the maximal clique in a key graph. When N and m are fixed, this lower bound increases linearly as n increases. That means the actual size of the maximal clique grows at least linearly with the network size. □

4.3 Number of Cliques with Respect to Clique Sizes

Figure 5 plots the number of cliques in key graphs and random graphs. In these experiments, we choose $n = 100$ and $N = 1000$. We use the two values of $m = 11$

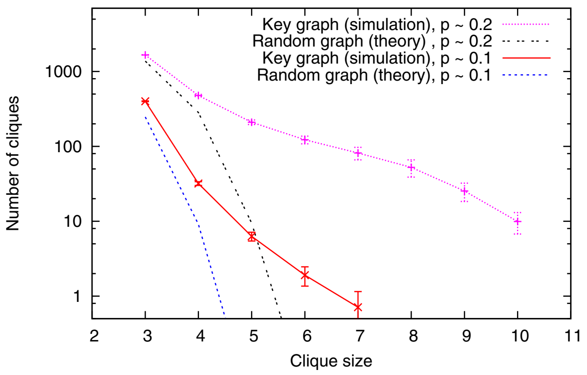


Fig. 5. Number of cliques in a key graph and a random graph for different edge probabilities when $n = 100$ and $N = 1000$

and $m = 15$ so that edge probability is approximately 0.1 and 0.2, respectively. The comparison results suggest that given the same number of nodes and the same edge probability, cliques tend to be larger and more plentiful in key graphs than in random graphs. Specifically, $G_k(100, 1000, 15)$ contains more than 200 cliques of size 5 on average, while there are fewer than 10 cliques of that size in a random graph with the same number of nodes and the same edge probability. Moreover, cliques of size up to 10 can be observed in $G_k(100, 1000, 15)$, as opposed to the random graphs where the expected number of cliques of size 6 (or larger) is negligible.

Explanation of the larger number of cliques in key graphs: Let S_k be the set of all nodes that have the key k. In key graphs, any set $S_k \neq \emptyset$, forms a clique regardless of edge probability. In random graphs, however the expected number of cliques is a function of n and p (see Equation 3). Additionally, because of i) the maximal clique in key graphs is much larger than that in random graphs (see Figure 4), and ii) any non-empty subset of nodes in a clique is also a clique itself, one expects more cliques in key graphs.

We can also use the global clustering coefficient to explain the formation of cliques in key graphs. As noted earlier, when m is small, if two nodes have a common neighbor in the key graph, there is a higher chance that they are connected by an edge. In general, the more common neighbors that two nodes X and Y have, the higher the probability that the edge XY exists. Thus, given a set of nodes such that many pairs of nodes are connected by an edge, the transitivity property implies there would be even more pairs that are directly connected, and the probability of this set of nodes being a clique increases. In contrast, a set of nodes S in random graphs forms a clique only when all the independent edge formation events between every pair of nodes in S occur at the same time.

5 Conclusions and Future Work

We study the applicability of random graph theory in modeling secure connectivity of wireless sensor networks. We identify ranges of parameters for which random graph modeling is not applicable and suggest how one can estimate key predistribution parameters for such cases. Besides, we determine ranges of parameters for which random graph theory may give estimates with excessive error, as well as other ranges of parameters where random graph theory provides very accurate results. We also study various structural properties in two graph families, observing and discussing the similarities and differences in the structure of random graphs and key graphs. In future work, we may extend the study of applicability of random graph modeling when the wireless connectivity is taken into account. Finally, there are other structural properties that we may investigate as well.

References

1. Bollobás, B., Erdős, P.: Cliques in Random Graphs. Mathematical Proceedings of the Cambridge Philosophical Society 80(3), 419–427 (1976)
2. Chan, H., Perrig, A., Song, D.: Random Key Predistribution Schemes for Sensor Networks. In: Proceedings of IEEE Security and Privacy Symposium, pp. 197–213 (May 2003)
3. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
4. Du, W., Deng, J., Han, Y., Varshney, P., Katz, J., Khalili, A.: A Pairwise Key predistribution Scheme for Wireless Sensor Networks. In: Proceedings of 10th ACM Conference on Computer and Communications Security, pp. 42–51 (October 2003)
5. Erdős, P., Rényi, A.: On Random Graphs. Publicationes Mathematicae 6, 290–297 (1959)
6. Eschenauer, L., Gligor, V.: A Key Management Scheme for Distributed Sensor Networks. In: Proceedings of the 9th ACM Conference on Computer and Communication Security, pp. 41–47 (November 2002)
7. Grimmett, G., McDiarmid, C.: On Coloring Random Graphs. Mathematical Proceedings of the Cambridge Philosophical Society 77, 313–324 (1975)
8. Gupta, R., Musacchio, J., Walrand, J.: Sufficient rate constraints for QoS flows in ad-hoc networks. Ad-hoc Network 5(4), 429–443 (2007)
9. Holland, P., Leinhardt, S.: Transitivity in Structural Models of Small Groups. Comparative Group Studies 2, 107–124 (1971)
10. Huang, D., Mehta, M., Medhi, D., Harn, L.: Location-aware key management scheme for wireless sensor networks. In: Proceedings of the 2nd ACM Workshop on Security of Ad-hoc and Sensor Networks, pp. 29–42 (October 2004)
11. Huang, X., Bensaou, B.: On Max-min Fairness and Scheduling in Wireless Ad Hoc Networks: Analytical Framework and Implementation. In: Proceedings of the 2nd ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 221–231 (October 2001)
12. Huang, Y., Lee, W.: A Cooperative Intrusion Detection System for Ad Hoc Networks. In: Proceedings of the 1st ACM Workshop on Security of Ad Hoc and Sensor Networks, pp. 135–147 (2003)

13. Hwang, J., Kim, Y.: Revisiting random key predistribution schemes for wireless sensor networks. In: Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks, pp. 43–52 (October 2004)
14. Liu, D., Ning, P.: Location-based pairwise key establishments for static sensor networks. In: Proceedings of the 1st ACM Workshop on Security of Ad-hoc and Sensor Networks, pp. 72–82 (October 2003)
15. Liu, D., Ning, P., Liu, R.: Establishing Pairwise Keys in Distributed Sensor Networks. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 52–61 (October 2003)
16. Massachusetts Institute of Technology, Kerberos: The Network Authentication Protocol, http://web.mit.edu/kerberos/
17. Pietro, R., Mancini, L., Mei, A., Panconesi, A., Radhakrishnan, J.: Redoubtable Sensor Networks. ACM Transactions on Information and System Security 11(3), 1–22 (2008)
18. Spencer, J.: The Strange Logic of Random Graphs. In: Algorithms and Combinatorics, vol. 22. Springer, Heidelberg (2000), ISBN 3-540-41654-4
19. Vu, T., Williamson, C., Safavi-Naini, R.: Simulation Modeling of Secure Wireless Sensor Networks. In: Proceedings of ValueTools 2009, Pisa, Italy (October 2009)
20. Watts, D., Strogatz, S.: Collective Dynamics of 'Small-world' Networks. Nature 393, 440–442 (1998)
21. Xue, Y., Li, B., Nahrstedt, K.: Optimal Resource Allocation in Wireless Ad Hoc Networks: A Price-Based Approach. IEEE Transactions on Mobile Computing 5(4), 347–364 (2006)

Appendix

The attached appendix contains algorithmic details and additional tabular results for the paper.

```
Input:
  – $n$: network size
  – $N$: key pool size
  – $m$: key ring size
  – $S$: sample size

Output: connectivity $c$

$counter \leftarrow 0$
for $i \leftarrow 1$ to $S$ do
    construct a key graph $G_k(n, N, m)$
    if $G_k(n, N, m)$ is connected then
        $counter{+}{+}$
    end
end
return $\frac{counter}{S}$
```

Algorithm 1: Determining connectivity

```
Input:
  – $n$: network size
  – $N$: key pool size
  – $lowerBound$: lower bound on key ring size
  – $upperBound$: upper bound on key ring size
  – $S$: sample size in each simulation experiment
  – $c$: desired connectivity

Output: Key ring size $m$

$lBound \leftarrow lowerBound$
$uBound \leftarrow upperBound$
while $uBound - lBound > 1$ do
    $mid \leftarrow (uBound + lBound)/2$
    if $connectivity(n, N, mid, S) \geq c$ then
        $uBound \leftarrow mid$
    else
        $lBound \leftarrow mid$
    end
end
return $uBound$
```

Algorithm 2: Binary search for key ring size

Table 5. Theoretical and simulated key ring sizes to achieve connectivity 99.9%

		Key pool size N							
		100	500	1,000	5,000	10,000	50,000	100,000	
Network size n	100	4	8	11	25	35	77	107	Simulation
		4	8	12	25	35	79	111	Theory
	500	2	4	5	12	17	37	51	Simulation
		2	4	6	12	17	37	52	Theory
	1,000	2	3	4	9	12	26	37	Simulation
		2	3	4	9	12	27	38	Theory
	5,000	2	2	2	4	6	13	18	Simulation
		1	2	2	4	6	13	18	Theory
	10,000	2	2	2	3	5	9	13	Simulation
		1	1	2	3	5	9	13	Theory

Table 6. Difference between theoretical and simulated key ring sizes

		Key pool size N						
		10^2	$\frac{10^3}{2}$	10^3	$\frac{10^4}{2}$	10^4	$\frac{10^5}{2}$	10^5
Network size n	10^2	0	0	0	0	0	0	1
	$\frac{10^3}{2}$	0	0	0	0	0	0	0
	10^3	-1	0	0	0	0	0	0
	$\frac{10^4}{2}$	-1	-1	0	0	0	0	0
	10^4	-1	-1	-1	0	0	0	0

(a) Desired connectivity $c = 50\%$

		Key pool size N						
		10^2	$\frac{10^3}{2}$	10^3	$\frac{10^4}{2}$	10^4	$\frac{10^5}{2}$	10^5
Network size n	10^2	0	0	0	1	1	0	1
	$\frac{10^3}{2}$	0	0	0	0	0	1	1
	10^3	-1	-1	0	0	0	-1	0
	$\frac{10^4}{2}$	-1	-1	0	0	0	0	0
	10^4	-1	-1	0	0	0	0	0

(b) Desired connectivity $c = 70\%$

		Key pool size N						
		10^2	$\frac{10^3}{2}$	10^3	$\frac{10^4}{2}$	10^4	$\frac{10^5}{2}$	10^5
Network size n	10^2	0	0	0	0	0	1	1
	$\frac{10^3}{2}$	0	0	0	0	1	0	0
	10^3	-1	0	0	0	0	0	0
	$\frac{10^4}{2}$	-1	0	0	0	0	0	0
	10^4	-1	-1	0	0	0	0	0

(c) Desired connectivity $c = 90\%$

		Key pool size N						
		10^2	$\frac{10^3}{2}$	10^3	$\frac{10^4}{2}$	10^4	$\frac{10^5}{2}$	10^5
Network size n	10^2	0	0	0	0	1	2	2
	$\frac{10^3}{2}$	0	0	0	0	0	1	0
	10^3	0	0	0	0	0	0	1
	$\frac{10^4}{2}$	-1	0	0	0	0	0	0
	10^4	-1	-1	0	0	0	0	0

(d) Desired connectivity $c = 99\%$

"Slow Is Fast" for Wireless Sensor Networks in the Presence of Message Losses*

Mahesh Arumugam[1], Murat Demirbas[2], and Sandeep S. Kulkarni[3]

[1] Cisco Systems Inc., San Jose, CA, 95134
maarumug@cisco.com
[2] SUNY Buffalo, Buffalo, NY, 14260
demirbas@cse.buffalo.edu
[3] Michigan State University, East Lansing, MI, 48824
sandeep@cse.msu.edu

Abstract. Transformations from shared memory model to wireless sensor networks (WSNs) quickly become inefficient in the presence of prevalent message losses in WSNs, and this prohibits their wider adoption. To address this problem, we propose a variation of the shared memory model, the SF shared memory model, where the actions of each node are partitioned into slow actions and fast actions. The traditional shared memory model consists only of fast actions and a lost message can disable the nodes from execution. Slow actions, on the other hand, enable the nodes to use slightly stale state from other nodes, so a message loss does not prevent the nodes from execution. We quantify over the advantages of using slow actions under environments with varying message loss probabilities, and find that a slow action has asymptotically better chance of getting executed than a fast action when the message loss probability increases. We also present guidelines for helping the protocol designer identify which actions can be marked as slow so as to enable the transformed program to be more loosely-coupled, and tolerate communication problems (latency, loss) better.

1 Introduction

Several computation models have been proposed for distributed computing, including shared memory model, read/write model, and message passing model. These models differ with respect to the level of abstraction they provide. Low level models such as the message passing model permits one to write programs that are closer to the actual system implementation and, hence, the programs can potentially be implemented more efficiently. However, since such programs need to analyze low level communication issues such as channel contention, message delays, etc, they are difficult to design and prove. Using a high level abstraction enables the designers to ignore low-level details of process communication and facilitates the design and verification of the protocols. For example, shared memory model, which allows a node to simultaneously read all its neighbors and

* This work was partially sponsored by NSF CNS 0914913, NSF 0916504, and Air Force Contract FA9550-10-1-0178.

update its own state, has been used extensively in the distributed systems literature. The drawback of using a high level abstraction model is that the system implementation requires more effort. While transformations from shared memory model to read/write model or message passing model have been considered in the literature [10,5,9], the efficiency of the transformed program suffers.

Wireless sensor networks (WSNs) warrant a new computation model due to their wireless broadcast communication mode, not captured in any of the above-mentioned models. Write-all-with-collision (WAC) model has been proposed in [6] to capture the important features of wireless broadcast communication for WSNs. In this model, in one step, a node can write its own state and communicate it to its neighbors. Due to the nature of shared medium, if one node is being updated by two (or more) of its neighbors simultaneously then the update fails (message collisions leads to message loss). While WAC model enables us to analyze the energy efficiency and message cost of protocols in WSNs more easily, it is not easy to design and prove protocols in WAC model compared to a higher level model such as the shared memory model.

Transformations from shared memory model to WAC model exist [6,8], however, these transformations have practical problems prohibiting their wider adoption. Although the shared memory model and the WAC model are similar in spirit (in that the former allows a node to read all its neighbors whereas the latter allows the node to write to all its neighbors), direct transformation becomes inefficient when message losses are considered. In [6], a CSMA based transformation, Cached Sensornet Transform (CST), from shared memory model to WAC model has been presented. In CST, a single message loss may violate the correctness of the resultant concrete program in the WAC model. The proof in [6] shows that if the abstract program was designed to be self-stabilizing and no other message loss occurs for a sufficiently long period, the concrete program will stabilize and start making progress. Thus, given the message loss rates at WSNs, this transformation incurs heavy correctness and performance loss at the WAC level. In [8], a transformation from read/write model to WAC model has been presented, and as we show in Section 3, it also applies for transformation from shared memory model to WAC model. This transformation employs a TDMA schedule to reduce message losses in the WAC model, however, due to interference, fading, or sleeping nodes, message losses are still likely in the real deployment. Message losses do not violate safety in this transformation, but they reduce the performance because the loss of a broadcast from a node prevents the evaluation of the actions at other nodes that depended on that information.

Contributions of the paper. To address the performance problems of transformations to WAC model, we propose a variation of the shared memory model, the SF shared memory model. In the SF shared memory model, a node is allowed to read the state of its neighbors and write its own state. However, actions of each node are partitioned into 'slow' actions and 'fast' actions. If a node j determines that a fast action is enabled then j must execute the action immediately before j's neighbors change their state. Otherwise, j must verify whether that action is still enabled the next time it evaluates its guards. On the other hand, if j determines that a slow action is enabled then j can execute the action at any point later as long as j does not execute any other action in between. Note that neighbors of j could change their state in the meanwhile.

We show that the use of SF shared memory model improves the performance of the transformed program under environments with message loss. The traditional shared memory model consists only of fast actions, and a lost message can disable the nodes from execution. Slow actions, on the other hand, enable the nodes to use slightly stale state from other nodes, so a message loss does not prevent the node from execution. We show that a slow action has asymptotically better chance of getting executed than a fast action when the message loss probability increases. By the same token, SF model also allows us to deal with another aspect of WSNs where nodes sleep periodically to save energy.

We present guidelines for the protocol designer to identify slow and fast actions. The designer can mark an action as slow only if 1) guard is stable, 2) guard depends only on local variables (this covers a rich set of programs), or 3) guard is a "locally stable" predicate. These conditions are increasingly more general; stable predicate implies locally stable predicates, but not vice versa. Local stable predicate with respect to j can change after j executes (then other neighbors can execute as the local stable contract is over at that time).

We also introduce slow-motion execution of fast actions. Under continuous message losses, fast actions may never get to execute. This results in bad performance and also violates strong fairness. In order to ensure strong fairness and to achieve graceful degradation of performance under environments with high message loss, we use slow-motion execution. Slow motion execution deliberately slows down the actions that the fast-action depends on, so that the fast action can execute as a pseudo-slow action. The fast action does not need to use the latest state, but it can use a recent consistent state.

Last but not least, our work draws lessons for protocol designers working at the shared memory model level. In order to preserve performance during the transformation, the designers should try to write actions as slow actions. This enables the concrete system to be more loosely-coupled, and tolerate communication problems (latency, loss) better.

Organization of the paper. First, in Section 2, we introduce the structure of programs and the computational models considered in this paper. In Section 3, we present the transformation from shared memory model to WAC model. Then, in Section 4, we introduce the notion of slow and fast actions. Subsequently, in Section 5, we provide an illustrative example. And, in Section 6, we analyze the effect of slow and fast actions. In Section 7, we present an approach for slow-motion execution of fast actions. In Section 8, we discuss some of the questions raised by this work, and finally, in Section 9, we make concluding remarks.

2 Preliminaries

A program is specified in terms of its processes. Each process consists of a set of variables and a set of guarded commands that update a subset of those variables [4]. Each guarded command (respectively, action) is of the form

$$guard \quad \longrightarrow \quad statement,$$

where *guard* is a predicate over program variables and *statement* updates the program variables. An action $g \longrightarrow st$ is enabled when g evaluates to true and to execute that action st is executed. A computation consists of a sequence $s_0, s_1, \ldots,$ where s_{l+1} is obtained from s_l $(0 \leq l)$ by executing one or more actions in the program.

Observe that a process can read variables of other processes while evaluating guards of its actions. The copies of these variables can be used in updating the process variables. Hence, we allow declaration of constants in the guard of an action. Intuitively, these constants *save* the value of the variable of the other process so that it can be used in the execution of the statement. As an illustration, consider a program where there are two processes j and k with variables $x.j$ and $x.k$ respectively. Hence, an action where j copies the value of $x.k$ when $x.j$ is less than $x.k$ is specified as follows:

Let $y = x.k$
$x.j < y \longrightarrow x.j = y$

Note that in a distributed program, for several reasons, it is necessary that a process can only read the variables of a small subset of processes called the neighborhood. More precisely, the neighborhood of process j consists of all the processes whose variables can be read by j.

A computation model limits the variables that an action can read and write. We now describe shared memory model and WAC model.

Shared memory model. In shared memory model, in one atomic step, a process can read its state as well as the state of all its neighbors and write its own state. However, it cannot write the state of other processes.

Write all with collision (WAC) model. In WAC model, each process (or *node*) consists of write actions (to be precise, write-all actions). In one atomic action, a process can update its own state and the state of all its neighbors. However, if two or more processes simultaneously try to update the state of another process, say l, then the state of l remains unchanged. Thus, this model captures the broadcast nature of shared medium.

3 Basic Shared Memory Model to WAC Model

In this section, we present an algorithm (adapted from [8]) for transforming programs written in shared memory model into programs in WAC model. First, note that in WAC model, there is no equivalent of read action. Hence, an action by which node j reads the state of k in shared memory model needs to be modeled in WAC model by requiring k to write its state at j. When k executes this write action, no other neighbor of j can execute simultaneously. Otherwise, due to collision, j remains unchanged. To deal with collisions, TDMA (e.g., [7,3,2]) is used to schedule execution of actions. Figure 1 outlines the transformation from

```
Input: Program $p$ in shared memory model
begin
Step 1: Slot computation
    compute TDMA schedule using a slot assignment algorithm (e.g., [7,3,2])
Step 2: Maintain state of neighbors
    for each variable $v.k$ at $k$, node $j$ ($k \in N.j$, where $N.j$ denotes neighbors
    of $j$) maintains a copy $copy_j.v.k$ that captures the value of $v.k$
Step 3: Transformation
    if slot $s$ is assigned to node $j$
        for each action $g_i \longrightarrow st_i$ in $j$
            evaluate $g_i$
                if $g_i$ mentions variable $v.k, k \neq j$
                    use the $copy_j.v.k$ to evaluate $g_i$
                end-if
        end-for
        if some guard $g_i$ is enabled
            execute $st_i$
        else
            skip;
        end-if
        for all neighbors $k$ in $N.j$
            for each variable $v.j$ in $j$, $copy_k.v.j = v.j$
        end-for
    end-if
end
```

Fig. 1. TDMA based transformation algorithm

shared memory model to WAC model. This algorithm assumes that message losses (other than collisions) do not occur.

In the algorithm, each node maintains a copy of all (public) variables of its neighbors. And, each node evaluates its guards and executes an enabled action in the slots assigned to that node. Suppose slot s is assigned to node j. In slot s, node j first evaluates its guards. If a guard, say g, includes variable $v.k$ of neighbor k, j uses $copy_j.v.k$ to evaluate g. And, if there are some guards that are enabled then j executes one of the enabled actions. Subsequently, j writes its state at all its neighbors. Since j updates its neighbors only in its TDMA slots, collisions do not occur during the write operation.

In this algorithm, under the assumption of no message loss, whenever a node writes its state at its neighbors, it has an immediate effect. Thus, whenever a node is about to execute its action, it has *fresh* information about the state of all its neighbors. Hence, if node j executes an action based on the copy of the state of its neighbors then it is utilizing the most recent state. Moreover, the algorithm in Figure 1 utilizes TDMA and, hence, when node j is executing its action, none of its neighbors are executing. It follows that even if multiple nodes execute their shared memory actions at the same time, their effect can be serialized. Thus, the execution of one or more nodes in a given time instance is equivalent to a serial execution of one or more shared memory actions.

Theorem 1. *Let p be the given program in shared memory model. And, let p' be the corresponding program in WAC model transformed using the algorithm in Figure 1. For every computation of p' in WAC model there is an equivalent computation of p in shared memory model.* □

4 Slow and Fast Actions

According to Section 3, when a process executes its shared memory actions, it utilizes the copy of the neighbors' state. However, when message losses occur, it is possible that the information j has is stale. In this section, we discuss how a node can determine whether it is safe to execute its action.

For the following discussion, let $g \longrightarrow st$ be a shared memory action A at node j. To execute A, j needs to read the state of some of its neighbors to evaluate g and then execute st if g evaluates to true. Let N denote the set of neighbors whose values need to be read to evaluate g. In the context of WSNs, j obtains its neighbors' values by allowing the neighbors to write the state of j. In addition to the algorithm in Figure 1, we require the update to be associated with a timestamp which can be implemented easily and efficiently [1]. Next, we focus on how j can determine whether g evaluates to true.

4.1 When Do We Evaluate the Guard?

The first approach to evaluate g is to ensure that the knowledge j has about the state of nodes in N is up-to-date. Let Cur denote the current time and let t_k denote the time when k notified j of the state of k. The information j has about nodes in N is latest iff for every node k in N, k was not assigned any TDMA timeslot between (t_k, Cur).

Definition 1 (Latest). *We say that j has the latest information with respect to action* A *iff latest(j, A) is true, where*

$$\mathit{latest}(j, \mathrm{A}) = (\forall k : k \in \mathrm{N} : \begin{array}{l} k \textit{ updated the state of } j \textit{ at time } t_k \textit{ and} \\ k \textit{ does not have a TDMA slot in the interval } (t_k, \mathrm{Cur}), \\ \textit{where } \mathrm{Cur} \textit{ denotes the current time.)} \end{array}$$

Clearly, if $\mathit{latest}(j, \mathrm{A})$ is true and g evaluates to true then g is true in the current global state, and, j can execute action A. Of course, if action A depends upon several neighbors then in the presence of message loss or sleeping nodes, it is difficult for j to ensure that g holds true in the current state. For this reason, we change the algorithm in Figure 1 as follows: Instead of maintaining just

[1] In the context of TDMA and the algorithm in Figure 1, the timestamp information can be relative. Based on the results in Section 6, it would suffice if only 2-4 bits are maintained for this information.

one copy for its neighbors, j maintains several copies with different time values (i.e., snapshots). Additionally, whenever a node updates its neighbors, instead of just including the current time, it includes an interval (t_1, t_2) during which this value remains unchanged. Based on these, we define the notion that j has a consistent information about its neighbors although the information may not be most recent.

Definition 2 (Consistent). *We say that j has consistent information as far as action* A *is concerned iff* $\mathit{consistent}(j, t, \mathrm{A})$ *is true, where*

$$\mathit{consistent}(j,t,\mathrm{A}) = (\forall k : k \in \mathrm{N} : k \textit{ updated the state of } j \textit{ at time } t_k \textit{ and } k \textit{ does not have a TDMA slot in the interval } (t_k, t))$$

Observe that if $\mathit{consistent}(j, t, \mathrm{A})$ is true and g evaluates to true based on most up-to-date information at time t then this implies that it is safe to execute A at time t. After j executes it can discard the old snapshots, and start collecting new snapshots. As we show in Section 6, at most 3 or 4 snapshots is enough for finding a consistent cut, so the memory overhead is low.

Even though satisfying $\mathit{latest}(j, \mathrm{A})$ may be difficult due to message losses and/or sleeping nodes, satisfying $\mathit{consistent}(j, t, \mathrm{A})$ is easier (cf. Section 6). If j misses an update from its neighbor, say k, in one timeslot then j may be able to obtain it in the next timeslot. Moreover, if state of k had not changed in the interim, j will be able to detect if a guard involving variables of k evaluates to true. Furthermore, if action A involves several neighbors of j then it is straightforward to observe that the probability that $\mathit{consistent}(j, t, \mathrm{A})$ is true for some t is significantly higher than the probability that $\mathit{latest}(j, \mathrm{A})$ is true.

The notion of consistency can be effectively used in conjunction with sleeping nodes. If node k is expected to sleep during an interval (t_1, t_2), it can include this information when it updates the state of j. This will guarantee j that state of k will remain unchanged during the interval (t_1, t_2) thereby making it more feasible to ensure that it can find a consistent state with respect to its neighbors.

4.2 When Do We Execute the Action?

The problem with the notion of consistency is that even though the guard of an action evaluated to true at some point in the past, it may no longer be true. Towards this end, we introduce the notion of a slow action and the notion of a fast action. (We call the resulting model as SF shared memory model.)

Definition 3 (slow action). *Let* A *be an action of j of the form $g \longrightarrow st$. We say that* A *is a slow action iff the following constraint is true:*

$$\begin{aligned}&(g \textit{ evaluates true at time } t) \wedge\\ &(j \textit{ does not execute any action between interval } [t, t'])\\ &\qquad\qquad\Rightarrow\ (g \textit{ evaluates true at time } t')\end{aligned}$$

Rule 1: Rule for execution of a slow action. Let A be a slow action of node j. Node j can execute A provided there exists t such that $\mathit{consistent}(j, t, \mathrm{A})$ is true and j has not executed any action in the interval [t, Cur) where Cur denotes the current time.

Definition 4 (fast action). *Let* A *be an action of* j *of the form* $g \longrightarrow st$*. We say that* A *is a fast action iff it is not a slow action.*

Rule 2: Rule for execution of a fast action. Let A be a fast action of node j. Node j can execute A provided $latest(j, \mathrm{A})$ is true.

If the algorithm in Figure 1 is modified based on the above two rules, i.e., slow actions can be executed when their guard evaluates to true at some time in the past and fast actions are executed only if their guard evaluates to true in the current state, then we can prove the following theorems:

Theorem 2. *Let* j *and* k *be two neighboring nodes with actions* A_1 *and* A_2 *respectively. If both* A_1 *and* A_2 *are slow actions then their execution by Rule 1 is serializable.* □

Theorem 3. *Let* j *and* k *be two neighboring nodes with actions* A_1 *and* A_2 *respectively. If both* A_1 *and* A_2 *are fast actions then their execution by Rule 2 is serializable.* □

Theorem 4. *Let* j *and* k *be two neighboring nodes with actions* A_1 *and* A_2 *respectively. Let* A_1 *be a slow action and let* A_2 *be a fast action. Then, their execution according to Rules 1 and 2 is serializable.* □

5 An Illustrative Example

In this section, we use the tree program from [1] to illustrate the notion of slow and fast actions. In this tree program (cf. Figure 2), each node j maintains three variables: *P.j*, that denotes the parent of node j, *root.j* that denotes the node that j believes to be the root, and *color.j* that is either green (i.e., the tree is not broken) or red (i.e., the tree is broken). Each node j also maintains an auxiliary variable *up.j* that denotes whether j is *up* or whether j has failed.

The protocol consists of five actions. The first three are program actions whereas the last two are environment actions that cause a node to fail and recover respectively. The first action allows a node to detect that the tree that it is part of may be broken. In particular, if j finds that its parent has failed then it sets its color to red. This action also fires if there is a parent and the parent is colored red. Observe that with the execution of this action, if a node is red then it will eventually cause its descendents to be red. The second action allows a red node to separate from the current tree and form a tree by itself provided it has no children. The third action allows one node to join the tree of another node. In particular, if j observes that its neighbor k has a higher root value and both j and k are colored green then j can change its tree by changing *P.j* to k and *root.j* to *root.k*. The fourth action is a fault action that causes a node to fail (i.e., $up.j = false$). Due to the execution of this action, the first action will be enabled at the children. And, finally, the last action allows a node to recover. When a node recovers, it sets its color to red.

```
$AC1:$ $color.j = green \wedge$
       $(\neg up.(P.j) \vee color.(P.j) = red)$   $\longrightarrow color.j = red$
$AC2:$ $color.j = red \wedge$
       $(\forall k : k \in Nbr.j : P.k \neq j)$   $\longrightarrow color.j, P.j, root.j = green, j, j$
$AC3:$ Let $x = root.k$
       $root.j < x \wedge color.j = green \wedge$
       $color.k = green$                          $\longrightarrow P.j, root.j = k, x$
$AC4:$ $up.j$                                     $\longrightarrow up.j = false$
$AC5:$ $\neg up.j$                                $\longrightarrow up.j, color.j = true, red$
```

Fig. 2. Coloring tree program

We can make the following observations about this program.

Theorem 5. *$AC1$ and $AC2$ are slow actions.*

Proof. If a node detects that its parent has failed or its parent is red then this condition is stable until that node (child) separates from the tree by executing action $AC2$. Hence, $AC1$ is a slow action. Likewise, if a node is red and has no children then it cannot acquire new children based on the guard of $AC3$. □

Theorem 6. *$AC3$ is a fast action.*

Proof. After j evaluates its own guard for $AC3$, it is possible that the guard becomes false subsequently if k changes its color by executing $AC1$ or if k changes its root by executing $AC3$. Hence, $AC3$ is a fast action. □

6 Effect of Slow versus Fast Actions during Execution

In this section, we evaluate the execution conditions of slow and fast actions in the presence of message loss. To execute a fast action, each node j needs to evaluate whether it has obtained the latest information about the state of its neighbors. If $latest(j, \mathrm{A})$ evaluates to true for some action then j can evaluate the guard of that action and execute the corresponding statement. If $latest(j, \mathrm{A})$ is false for all actions then j must execute a 'skip' operation and see if it can obtain the latest information in the next TDMA round. For the execution of a slow action, j proceeds in a similar fashion. However, if j obtains consistent information about its neighbors that is not necessarily from the latest TDMA round, j can execute its action.

Next, we evaluate the probability that j can obtain the necessary consistent and latest state information. Let p be the probability of a message loss and let N denote the number of neighbors whose status needs to be known to evaluate the guard of the action. If j cannot successfully obtain consistent and/or latest state information in one TDMA round then it tries to do that in the next round. Hence, we let m denote the number of TDMA rounds that j tries to obtain the consistent and/or latest information. Assuming that states of the neighbors do not change during these m rounds, we calculate the probability that j can obtain the required consistent and/or latest state information[2].

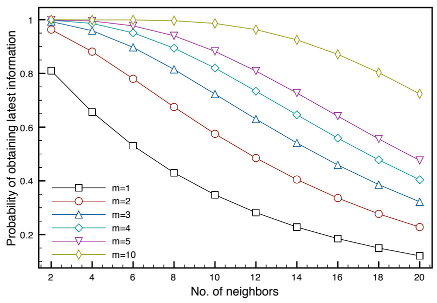
Probability of obtaining latest information. To obtain the latest information in one TDMA round, j needs to successfully receive message from each of its neighbors. Probability of successfully receiving message from one neighbor is $(1 - p)$. Hence, the probability of obtaining latest information in one round is $(1 - p)^N$. And, the probability of not obtaining the latest information in one round is $1 - (1 - p)^N$. Therefore, probability of not obtaining the latest information in any one of m rounds is $(1 - (1 - p)^N)^m$. Thus, the probability that j can obtain the latest information in at least one m rounds is $(1 - (1 - (1 - p)^N)^m)$.

Probability of obtaining consistent information. To obtain consistent information in the earliest of m rounds, j needs to obtain information from each of its neighbors in some round. (Observe that since the nodes include the intervals where their value is unchanged, receiving a message from each node at some round is enough for identifying the first round as the consistent cut.) The probability that j does not receive message from one of its neighbors in either of m rounds is p^m. Hence, probability of successfully receiving message from one neighbor is $(1 - p^m)$. Therefore, probability of successfully receiving message from every neighbor is $((1 - p^m))^N$. Furthermore, there is an additional conditional probability where j fails to get consistent information in the first (earliest) round but obtains it in the next round. We account for this in our calculations and graphs, but omit the full formula here for the sake of brevity.

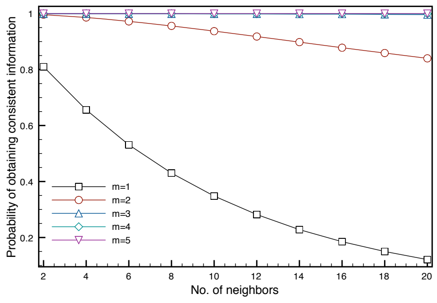
Figures 3-5 show the probabilities for $p = 10\%$, $p = 20\%$, and $p = 30\%$ respectively. First, we note that the probability of obtaining latest information decreases as N increases for different values of m. A given node has to receive updates from all its neighbors in order to obtain the latest information. Hence, as N increases, latest probability decreases. Moreover, in a high message loss environment (e.g., $p = 20\%$ and $p = 30\%$), latest probabilities decrease significantly as N increases. For small neighborhoods, the probability of getting latest information improves as m increases. This suggests that if the neighbors remain silent for some rounds then the probability of obtaining latest information improves. On the other hand, although the probability of obtaining consistent information decreases as N increases, for $m \geq 3$, it remains close to 1. By choosing $m = 3$, the probability of finding a consistent cut is virtually certain at $p = 10\%$.

Thus, the probability of obtaining the consistent information is significantly higher than that of latest information. This suggests that it is better to utilize protocols that have slow actions verses protocols that have fast actions. In particular, it is better if actions that depend on the value of several neighbors are slow actions. On the other hand, if protocols must have fast actions, then it is better if they rely on a small number (preferably 1) of neighbors.

[2] We can relax this assumption by requiring the nodes to include their old values in previous rounds with their broadcast. These values are then used for finding a consistent cut in the past. Our results show that it suffices for the node to include values from the last 3 rounds for most cases. Observe that this method does not help "latest" because learning an older snapshot does not allow executing a fast action.

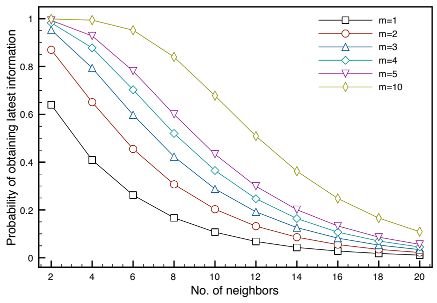


(a) Latest probabilities

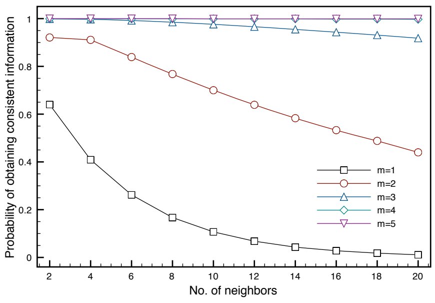


(b) Consistent probabilities

Fig. 3. Latest and consistent probabilities for $p = 10\%$



(a) Latest probabilities



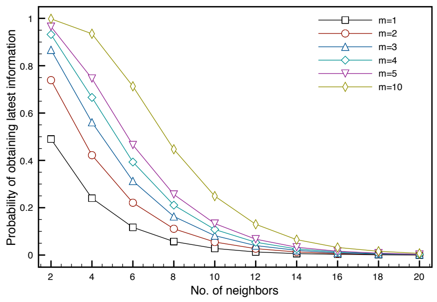
(b) Consistent probabilities

Fig. 4. Latest and consistent probabilities for $p = 20\%$

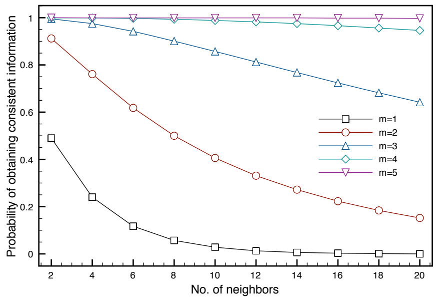
7 Pseudo-slow Action

The results in Section 6 show that if actions of a program are slow then their execution is expected to be more successful. Thus, the natural question is what happens if all program actions were fast? Can we allow such a program to utilize an old consistent state to evaluate its guard. We show that for a subset of the original actions, this is feasible if we analyze the original shared memory program to identify *dependent* actions.

We illustrate our approach in the context of the tree example in Section 5. For the sake of discussion, let us assume that all actions are fast actions; this is reasonable since it adds more restrictions on how each action can be executed. Furthermore, let us consider the case that we want j to be able to execute $AC3$ by utilizing a consistent state although not necessarily the latest state. Recall that action $AC3$ causes j to join the tree of k. If j is using a consistent state that is not necessarily the latest state, it is possible that k has changed its state in the interim. Observe that if k had increased the value of $root.k$ by executing $AC3$ then it is still safe for j to execute action $AC3$. However, if k executes $AC1$ and changes its color to red, subsequently observes that it has no children and



(a) Latest probabilities



(b) Consistent probabilities

Fig. 5. Latest and consistent probabilities for $p = 30\%$

executes $AC2$ then it may not be safe for j to join the tree of k. Thus, if we want to allow j to execute $AC3$ using a consistent state that is not necessarily latest then k must be prevented from executing either $AC1$ or $AC2$. Again, for the sake of discussion, let us assume that we want to restrict k from executing $AC2$. Hence, in this case, we will say that pseudo-slow execution of $AC3.j$ is dependent upon slowing down $AC2.k$.

In this approach, we allow j to utilize consistent snapshots for up to x previous TDMA rounds (i.e., j can execute $AC3.j$ if it obtains a consistent state that is no more than x rounds before the current time and evaluates that the guard of $AC3$ is true). However, in this case, if k ever wants to execute action $AC2$ then it must stay silent for at least $x+1$ TDMA rounds before executing action $AC2$. Note that this will essentially disable execution of action $AC3.j$ (i.e., at the end of $x+1$ silent rounds k knows that j cannot simultaneously execute $AC3.j$ and interfere with the execution of $AC2.k$)[3].

We generalize this approach in terms of the following 4-step algorithm.

Step 1: Identify pseudo-slow actions. First, the designer needs to identify the set of actions, $\mathbb{A}$, that are fast actions but it is desired that they can execute as slow actions, where a node can utilize consistent (but not necessarily the latest) information about the state of neighbors. The choice of $\mathbb{A}$ is application dependent, i.e., it is based on the designers' belief/observation that quick execution of these actions is likely to help execution of the program. We denote the actions in $\mathbb{A}$ as a set of pseudo-slow actions since they are not slow actions but behave similar to the slow actions.

Step 2: Identify dependent actions. Let $\mathcal{A}_j$ be one of the pseudo-slow actions in $\mathbb{A}$ that is to be executed by node j. Let $\mathcal{A}_j$ be of the form $g \longrightarrow st$.

[3] We can relax this $x+1$ silent rounds requirement. For this, we modify the algorithm in Figure 1 slightly where a node, say j, not only notifies its neighbors about its own state but also includes a timestamp information about messages received from its neighbors. With this change, k can either execute its action $AC2$ if it stops transmitting for $x+1$ rounds or if it checks that j is aware of its color being red and, hence, will not execute action $AC3.j$.

Since $\mathcal{A}_j$ is a fast action, this implies that if the guard of $\mathcal{A}_j$ is true in some state then it can become false by execution of actions of one or more neighbors of j. Hence, the goal of this step is to identify the set of actions, say $\mathfrak{A}$, such that if (1) g evaluates to true in some state, (2) no action from $\mathfrak{A}$ is executed, and (3) no action of j is executed then it is still acceptable to execute the statement st in the given shared memory program. The value obtained for $\mathfrak{A}$ is called the dependent actions of $\mathcal{A}_j$.

In this step, for each action in $\mathbb{A}$, we identify the corresponding set of dependent actions. The dependent actions for $\mathbb{A}$ is obtained by taking the union of these dependent actions. Step 2 is successful if $\mathbb{A}$ and its dependent actions are disjoint. If there is an overlap between these two sets then the set of pseudo-slow actions needs to be revised until this condition is met.

Step 3: Choosing the delay value. The next step is to identify how much old information can be used in evaluating the guard of an action. Essentially, this corresponds to the choice of x in the above example. We denote this as the delay value of the corresponding action. The delay value x chosen for efficient implementation of pseudo-slow actions is also user dependent. The value will generally depend upon the number of neighbors involved in the execution of the pseudo-slow action. Based on the analysis from Section 6, we expect that a value of 3-4 is expected to be sufficient for this purpose.

Step 4: Revising the transformation algorithm. The last step of the algorithm is to utilize $\mathbb{A}$ identified in Step 1, the corresponding dependent actions identified in Step 2 and the delay value identified in Step 3 to revise the transformation algorithm. In particular, we allow a pseudo slow action at j to execute if (1) j obtains consistent state information about its neighbors, (2) j does not have more recent information about its neighbors than the one it uses, and (3) no more than x TDMA rounds have passed since obtaining the consistent state.

Additionally, a dependent action at j can execute if j does not transmit its own state for at least $x + 1$ rounds. (It is also possible for j to execute a dependent action earlier based on the knowledge j got about the state of its neighbors. However, for reasons of space, we omit the details.)

8 Discussion

What is specific to write-all in our transformation algorithm? Why is this transformation not applicable for message passing?
Write-all-with-collision (i.e., wireless broadcast) model helps a lot for our transformation, but is not strictly necessary. Our transformation is also applicable for message-passing, if on execution of an action at k at its TDMA slot, its state is made available to all of its neighbors before the next slot starts. It may not be easy and inexpensive to guarantee this condition for message passing, whereas for write-all with TDMA this condition is easily and inexpensively satisfied.
Can we relax the TDMA communication assumption?
The definitions of "latest" and "consistent" depend on the assumption that "k does not have another (missed) TDMA slot until the cut". This is used for

ensuring that k does not execute any action in that interval, so k's new state is not out of sync with the cached state in the cut. Without using TDMA, the same condition can be achieved by using an alternative mechanism to communicate that k will not update its state for a certain duration. For example k can include a promise in its message that it will not update its state for some interval (e.g., until its next scheduled update, or until its sleep period is over).

Given that our transformation can tolerate message losses in the concrete model, dropping the TDMA mechanism would not hurt the performance of the transformed program significantly. The round concept could be used without the TDMA slots, and the nodes would utilize CSMA to broadcast their messages.

What are the rules of thumb for marking actions as slow?
As mentioned in the Introduction, an action be marked slow only if 1) guard is a stable predicate, 2) guard depends only on local variables, or 3) guard is a "locally stable" predicate. While the first two conditions are easy to detect, the locally stable condition requires reasoning about the program execution. We expect the protocol designer to understand his program.

A big problem is marking a fast action as slow, as this would violate correctness! It is better to err on the side of safety and mark the action as fast if there is some doubt about it being a slow action. Marking a slow action as fast does not violate correctness, but would just reduce the performance.

Do we need to use slow-motion execution for every program?
If the designer can mark all program actions as slow, there is obviously no need for slow-motion execution as there is no fast action remaining. Even when there are some fast actions remaining, if most of the actions are slow actions and message loss rates are not very high, these fast actions may not reduce the performance of the program significantly. However, if message loss rates increase further, it could be more beneficial to switch to slow-motion execution than to suffer from message losses voiding the latest cut and blocking the fast actions.

9 Conclusion

We have presented an extension to the shared memory model, by introducing the concept of slow action. A slow action is one such that once it is enabled at a node j, it can be executed at any later point at j provided that j does not execute another action in between. Slow actions mean that the process can tolerate slightly stale state from other processes, which enables the concrete system to be more loosely-coupled, and tolerate communication problems better. We quantified the improvements possible by using a slow action, and gave practical rules that help a programmer mark his program actions as slow and fast. For reducing the performance penalty of fast actions under heavy message loss environments, we also introduced the notion of slow-motion execution for fast actions.

Our work enables a good performance for transformed programs in realistic WSN environments with message loss. In future work, we will investigate adaptive switching to slow-motion execution to curb the performance penalty that message losses incur on fast actions. To this end, we will determine the break-even point for switching to the slow-motion execution mode, and middleware for switching to and back from the slow-motion mode seamlessly.

References

1. Arora, A.: Efficient reconfiguration of trees: A case study in the methodical design of nonmasking fault-tolerance. Science of Computer Programming (1996)
2. Arumugam, M.: A distributed and deterministic TDMA algorithm for write-all-with-collision model. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 4–18. Springer, Heidelberg (2008)
3. Arumugam, M., Kulkarni, S.S.: Self-stabilizing deterministic time division multiple access for sensor networks. AIAA Journal of Aerospace Computing, Information, and Communication (JACIC) 3, 403–419 (2006)
4. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11) (1974)
5. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing 7, 3–16 (1993)
6. Herman, T.: Models of self-stabilization and sensor networks. In: Das, S.R., Das, S.K. (eds.) IWDC 2003. LNCS, vol. 2918, pp. 205–214. Springer, Heidelberg (2003)
7. Kulkarni, S.S., Arumugam, M.: SS-TDMA: A self-stabilizing mac for sensor networks. In: Sensor Network Operations. Wiley/IEEE Press (2006)
8. Kulkarni, S.S., Arumugam, M.: Transformations for write-all-with-collision model. Computer Communications 29(2), 183–199 (2006)
9. Mizuno, M., Nesterenko, M.: A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. Information Processing Letters 66(6), 285–290 (1998)
10. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. Journal of Parallel and Distributed Computing 62(5), 766–791 (2002)

Modeling and Analyzing Periodic Distributed Computations

Anurag Agarwal⋆, Vijay K. Garg⋆⋆, and Vinit Ogale⋆ ⋆ ⋆

The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Abstract. The earlier work on predicate detection has assumed that the given computation is finite. Detecting violation of a liveness predicate requires that the predicate be evaluated on an infinite computation. In this work, we develop the theory and associated algorithms for predicate detection in infinite runs. In practice, an infinite run can be determined in finite time only if it consists of a recurrent behavior with some finite prefix. Therefore, our study is restricted to such runs. We introduce the concept of *d-diagram*, which is a finite representation of infinite directed graphs. Given a d-diagram that represents an infinite distributed computation, we solve the problem of determining if a global predicate ever became true in the computation. The crucial aspect of this problem is the stopping rule that tells us when to conclude that the predicate can never become true in future. We also provide an algorithm to provide vector timestamps to events in the computation for determining the dependency relationship between any two events in the infinite run.

1 Introduction

Correctness properties of distributed programs can be classified either as safety properties or liveness properties. Informally, a safety property states that the program never enters a bad (or an unsafe) state, and a liveness property states that the program eventually enters into a good state. For example, in the classical dining philosopher problem a safety property is that "two neighboring philosophers never eat concurrently" and a liveness property is that "every hungry philosopher eventually eats." Assume that a programmer is interested in monitoring for violation of a correctness property in her distributed program. It is clear how a runtime monitoring system would check for violation of a safety property. If it detects that there exists a consistent global state[1] in which two neighboring philosophers are eating then the safety property is violated. The literature in the area of global predicate detection deals with the complexity and algorithms for such tasks [2,3]. However, the problem of detecting violation of the liveness

⋆ Currently at Google.

⋆⋆ Part of the work was performed at the University of Texas at Austin supported in part by the NSF Grants CNS-0509024, CNS-0718990, Texas Education Board Grant 781, SRC Grant 2006-TJ-1426, and Cullen Trust for Higher Education Endowed Professorship.

⋆ ⋆ ⋆ Currently at Microsoft.

property is harder. At first it appears that detecting violation of a liveness property may even be impossible. After all, a liveness property requires something to be true eventually and therefore no finite observation can detect the violation. We show in this paper a technique that can be used to infer violation of a liveness property in spite of finite observations. Such a technique would be a basic requirement for detecting a temporal logic formula[4] on a computation for runtime verification.

There are three important components in our technique. First, we use the notion of a *recurrent* global state. Informally, a global state is recurrent in a computation γ if it occurs more than once in it. Existence of a recurrent global state implies that there exists an infinite computation δ in which the set of events between two occurrences of can be repeated ad infinitum. Note that γ may not even be a prefix of δ. The actual behavior of the program may not follow the execution of δ due to nondeterminism. However, we know that δ is a legal behavior of the program and therefore violation of the liveness property in δ shows a bug in the program.

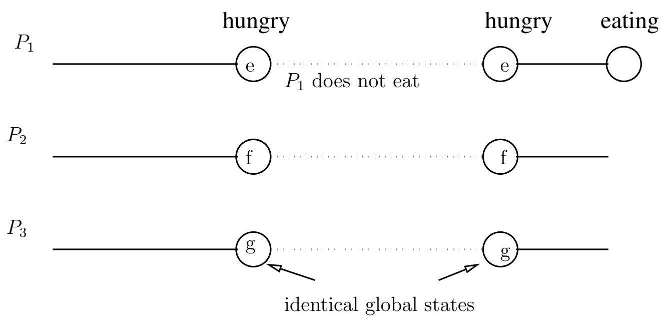


Fig. 1. A finite distributed computation C of dining philosophers

For example, in figure 1, a global state repeats where the same philosopher P_1 is hungry and has not eaten in between these occurrences. P_1 does get to eat after the second occurrence of the *recurrent* global state; and, therefore a check that "every hungry philosopher gets to eat" does not reveal the bug. It is simple to construct an infinite computation δ from the observed finite computation γ in which P_1 never eats. We simply repeat the execution between the first and the second instance of the recurrent global state. This example shows that the approach of capturing a periodic part of a computation can result in detection of bugs that may have gone undetected if the periodic behavior is not considered.

The second component of our technique is to develop a finite representation of the infinite behavior γ. Mathematically, we need a finite representation of the infinite but periodic poset γ. In this paper, we propose the notion of d-diagram to capture infinite periodic posets. Just as finite directed acyclic graphs (dag's) have been used to represent and analyze finite computations, d-diagrams may be used for representing periodic infinite distributed computations for monitoring or logging purposes. The logging may be

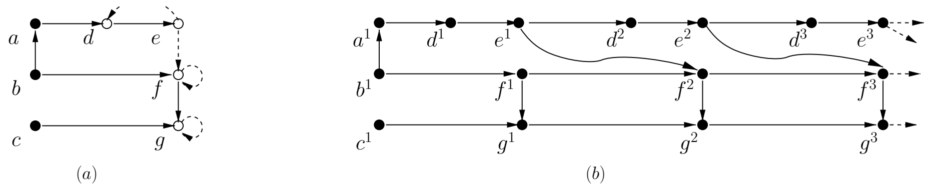


Fig. 2. (a) A d-diagram and (b) its corresponding infinite poset

useful for replay or offline analysis of the computation. Figure 2 shows a d-diagram and the corresponding infinite computation. The formal semantics of a d-diagram is given in Section 3. Intuitively, the infinite poset corresponds to the infinite unrolling of the recurrent part of the d-diagram.

The third component of our technique is to develop efficient algorithms for analyzing the infinite poset given as a d-diagram. Two kinds of computation analysis have been used in past for finite computations. The first analysis is based on vector clocks which allows one to answer if two events are dependent or concurrent, for example, works by Fidge[5] and Mattern[6]. We extend the algorithm for timestamping events of a finite poset to that for the infinite poset. Of course, since the set of events is infinite, we do not give explicit timestamp for all events, but only an implicit method that allows efficient calculation of dependency information between any two events when desired. The second analysis we use is to detect a global predicate B on the infinite poset given as a d-diagram. In other words, we are interested in determining if there exists a consistent global state which satisfies B. Since the computation is infinite, we cannot employ the traditional algorithms [2,3] for predicate detection. Because the behavior is periodic it is natural that a finite prefix of the infinite poset may be sufficient to analyze. The crucial problem is to determine the number of times the recurrent part of the d-diagram must be unrolled so that we can guarantee that B is true in the finite prefix *iff* it is true in the infinite computation. We show in this paper that it is sufficient to unroll the d-diagram N times where N is the number of processes in the system.

We note here that there has been earlier work in detection of temporal logic formulas on distributed computation, such as [7,8,9,10]. However, the earlier work was restricted to verifying the temporal logic formula on the finite computation with the interpretation of liveness predicates modified to work for finite posets. For example, the interpretation of "a hungry philosopher never gets to eat" was modified to "a hungry philosopher does not eat by the end of the computation." This interpretation, although useful in some cases, is not accurate and may give false positives when employed by the programmer to detect bugs. This paper is the first one to explicitly analyze the periodic behavior to ensure that the interpretation of formulas is on the infinite computation.

In summary, this paper makes the following contributions:

– We introduce the notion of recurrent global states in a distributed computation and propose a method to detect them.
– We introduce and study a finite representation of infinite directed computations called d-diagrams.

- We provide a method of timestamping nodes of a d-diagram so that the happened-before relation can be efficiently determined between any two events in the given infinite computation.
- We define the notion of core of a d-diagram that allows us to use any predicate detection algorithm for finite computations on infinite computations as well.

2 Model of Distributed Computation

We first describe our model of a distributed computation. We assume a message passing asynchronous system without any shared memory or a global clock. A distributed program consists of N sequential processes denoted by $P = \{P_1, P_2, \ldots, P_N\}$ communicating via asynchronous messages. A *local computation* of a process is a sequence of events. An event is either an internal event, a send event or a receive event. The predecessor and successor events of e on the process on which e occurs are denoted by $pred(e)$ and $succ(e)$.

Generally a distributed computation is modeled as a partial order of a set of events, called the *happened-before relation* [11]. In this paper, we instead use directed graphs to model distributed computations as done in [9]. When the graph is acyclic, it represents a distributed computation. When the distributed computation is infinite, the directed graph that models the computation is also infinite. An infinite distributed computation is *periodic* if it consists of a subcomputation that is repeated forever.

Given a directed graph $G = \langle E, \rightarrow \rangle$, we define a *consistent cut* as a set of vertices such that if the subset contains a vertex then it contains all its incoming neighbors. For example, the set $C = \{a^1, b^1, c^1, d^1, e^1, f^1, g^1\}$ is a consistent cut for the graph shown in figure 3(b). The set $\{a^1, b^1, c^1, d^1, e^1, g^1\}$ is not consistent because it includes g^1, but does not include its incoming neighbor f^1. The set of finite consistent cuts for graph G is denoted by $\mathcal{C}(G)$.

In this work we focus only on finite consistent cuts (or *finite order ideals* [12]) as they are the ones of interest for distributed computing.

A *frontier* of a consistent cut is the set of those events of the cut whose successors, if they exist, are not contained in the cut. Formally,

$$frontier(C) = \{x \in C | succ(x) \text{ exists } \Rightarrow succ(x) \notin C\}$$

For the cut C in figure 3(b), $frontier(C) = \{e^1, f^1, g^1\}$. A consistent cut is uniquely characterized by its frontier and in this paper we always identify a consistent cut by its frontier.

Two events are said to be *consistent* iff they are contained in the frontier of some consistent cut, otherwise they are inconsistent. It can be verified that events e and f are consistent iff there is no path in the computation from $succ(e)$ to f and from $succ(f)$ to e.

3 Infinite Directed Graphs

From distributed computing perspective, our intention is to provide a model for an infinite computation of a distributed system which eventually becomes periodic. To this end, we introduce the notion of *d-diagram* (directed graph diagram).

Definition 1 (d-diagram). *A d-diagram Q is a tuple (V, F, R, B) where V is a set of vertices or nodes, F (forward edges) is a subset of $V \times V$, R (recurrent vertices) is a subset of V, and B (shift edges) is a subset of $R \times R$. A d-diagram must satisfy the following constraint: If u is a recurrent vertex and $(u, v) \in F$ or $(u, v) \in B$, then v is also recurrent.*

Figure 2(a) is an example of a d-diagram. The recurrent vertices and non-recurrent vertices in the d-diagram are represented by hollow circles and filled circles respectively. The forward edges are represented by solid arrows and the shift-edges by dashed arrows. The recurrent vertices model the computation that is periodic.

Each d-diagram generates an infinite directed graph defined as follows:

Definition 2 (directed graph for a d-diagram). *The directed graph $G = \langle E, \rightarrow \rangle$ for a d-diagram Q is defined as follows:*

- *$E = \{u^1 | u \in V\} \cup \{u^i | i \geq 2 \wedge u \in R\}$*
- *The relation $\rightarrow$ is the set of edges in E given by:*
  (1) if $(u, v) \in F$ and $u \in R$, then $\forall i : u^i \rightarrow v^i$, and (2) if $(u, v) \in F$ and $u \notin R$, then $u^1 \rightarrow v^1$, and (3) if $(v, u) \in B$, then $\forall i : v^i \rightarrow u^{i+1}$.

The set E contains infinite instances of all recurrent vertices and single instances of non-recurrent vertices. For a vertex u^i, we define its index as i.

It can be easily shown that if the relation F is acyclic, then the resulting directed graph for the d-diagram is a poset. Figure 2 shows a d-diagram along with a part of the infinite directed graph generated by it. Two vertices in a directed graph are said to be *concurrent* if there is no path from one to other.

Note that acyclic d-diagrams cannot represent all infinite posets. For example, any poset P defined by an acyclic d-diagram is well-founded. Moreover, there exists a constant k such that every element in P has the size of the set of its upper covers and lower covers[12] bounded by k. Although acyclic d-diagrams cannot represent all infinite posets, they are sufficient for the purpose of modeling distributed computations. Let the *width* of a directed graph be defined as the maximum size of a set of pairwise concurrent vertices. A distributed computation generated by a finite number of processes has finite width and hence we are interested in only those d-diagrams which generate finite width directed graphs. The following property of the posets generated by acyclic d-diagrams is easy to show.

Lemma 1. *A poset P defined by an acyclic d-diagram has finite width iff for every recurrent vertex there exists a cycle in the graph $(R, F \cup B)$ which includes a shift-edge.*

Proof. Let $k > 0$ be the number of shift-edges in the shortest cycle involving $u \in R$. By transitivity we know that there is a path from vertex u^i to u^{i+k} in R. Therefore, at most $k - 1$ instances of a vertex $u \in R$ are concurrent. Since R is finite, the largest set of concurrent vertices is also finite.

Conversely, if there exists any recurrent vertex v that is not in a cycle involving a shift-edge, then v^i is concurrent with v^j for all i, j. Then, the set

$$\{v^i | i \geq 1\}$$

contains an infinite number of concurrent vertices. Thus, G has infinite width.

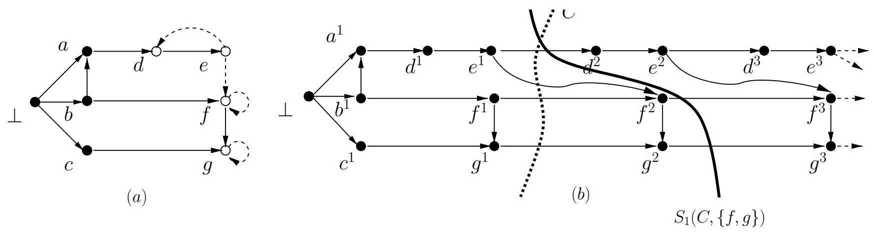


Fig. 3. (a) A d-diagram (b) The computation for the d-diagram showing a cut and the shift of a cut

Figure 2 shows a d-diagram and the corresponding computation. Note that for any two events x, y on a process, either there is a path from x to y or from y to x, i.e., two events on the same process are always ordered.

The notion of *shift* of a cut is useful for analysis of periodic infinite computations. Intuitively, the shift of a frontier C produces a new cut by moving the cut C forward or backward by a certain number of iterations along a set X of recurrent events in C. Formally,

Definition 3 (d-shift of a cut). *Given a frontier C, a set of recurrent events $X \subseteq R$ and an integer d, a* d-shift cut *of C with respect to X, is represented by the frontier $S_d(C, X)$*

$$\{e^i | e^i \in C \wedge e \notin X\} \cup \{e^m | e^i \in C \wedge e \in X \wedge m = \mathit{max}(1, i + d)\}$$

We denote $S_d(C, R)$ simply by $S_d(C)$.

Hence $S_d(C, X)$ contains all events e^i that are not in X, and the shifted events for all elements of X. Note that in the above definition d can be negative. Also, for a consistent cut C, $S_d(C, X)$ is not guaranteed to be consistent for every X.

As an example, consider the infinite directed graph for the d-diagram in figure 3. Let C be a cut given by the frontier $\{e^1, f^1, g^1\}$ and $X = \{f, g\}$. Then $S_1(C, X)$ is the cut given by $\{e^1, f^2, g^2\}$. Figure 3 shows the cut C and $S_1(C, X)$. Similarly for C given by $\{a^1, f^1, g^1\}$, $S_1(C) = \{a^1, f^2, g^2\}$. Note that in this case, a^1 remains in the frontier of $S_1(C)$ and the cut $S_1(C)$ is not consistent.

4 Vector Clock Timestamps

In this section, we present algorithms to assign vector timestamps to nodes in the infinite computation given as a d-diagram. The objective is to determine dependency between any two events, say e^i and f^j, based on the vector clocks assigned to these events rather than exploring the d-diagram. Since there are infinite instances of recurrent events, it is clear that we can only provide an implicit vector timestamp for the events. The explicit vector clock can be computed for any specific instance of i and j.

Timestamping events of the computation has many applications in debugging distributed programs [5]. Given the timestamps of recurrent events e and f, our algorithm enables answering queries of the form:

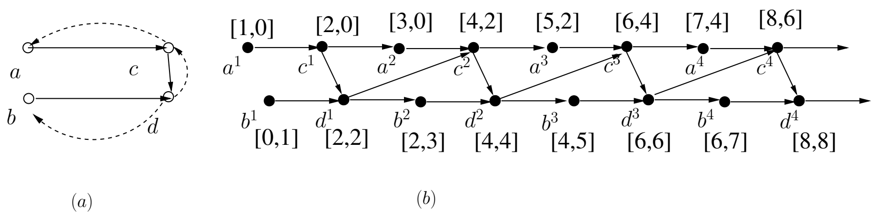


Fig. 4. (a) A d-diagram(b) The corresponding computation with vector timestamps

1. Are there any instances of e and f which are concurrent, i.e., are there indices i and j such that e^i is concurrent with f^j? For example, when e and f correspond to entering in the critical section, this query represents violation of the critical section.
2. What is the maximum value of i such that e^i happened before a given event such as f^{256}?
3. Is it true that for all i, e^i happened before f^i?

We show in this section, that there exists an efficient algorithm to timestamp events in the d-diagram. As expected, the vectors corresponding to any recurrent event e^i eventally become periodic. The difficult part is to determine the threshold after which the vector clock becomes periodic and to determine the period.

We first introduce the concept of *shift-diameter* of a d-diagram. The shift-diameter provides us with the threshold after which the dependency of any event becomes periodic.

Definition 4 (shortest path in a d-diagram). *For a d-diagram Q, the shortest path between any two vertices is a path with the minimum number of shift-edges.*

Definition 5 (shift-diameter of a d-diagram). *For a d-diagram Q, the shift-diameter $\eta(Q)$ is the maximum of the number of shift-edges in the shortest path between any two vertices in the d-diagram.*

When Q is clear from the context, we simply use η to denote $\eta(Q)$. For the d-diagram in Figure 3, $\eta = 1$. In figure 4, we can see that $\eta = 2$. We first give a bound on η.

Lemma 2. *For a d-diagram Q corresponding to a computation with N processes, $\eta(Q) \leq 2N$.*

Proof. Consider the shortest path between two vertices $e, f \in V$. Clearly this path does not have a cycle; otherwise, a shorter path which excludes the cycle exists. Moreover, all the elements from a process occur consecutively in this path. As a result, the shift-edges that are between events on the same process are traversed at most once in the path. Moving from one process to another can have at most one shift-edge. Hence, $\eta(Q) \leq 2N$.

For an event $x \in E$, we denote by $J(x)$, the least consistent cut which includes x. The least consistent cut for $J(e^i)$ will give us the vector clock for event e^i. We first show that the cuts $J(e^i)$ *stabilize* after some iterations i.e. the cut $J(e^j)$ can be obtained from $J(e^i)$ by a simple shift for $j > i$. This allows us to *predict* the structure of $J(e^i)$ after certain iterations.

The next lemma shows that the cut $J(f^j)$ does not contain recurrent events with iterations very far from j.

Lemma 3. *If* $e^i \in frontier(J(f^j))$, $e \in R$, *then* $0 \leq j - i \leq \eta$.

Proof. If $e^i \in frontier(J(f^j))$, then there exists a path from e^i to f^j and $\forall k > i$ there is no path from e^k to f^j. Therefore the path from e^i to f^j corresponds to the shortest path between e and f in the d-diagram. Therefore, by the definition of η, $j - i \leq \eta$.

The following theorem proves the result regarding the stabilization of the cut $J(e^i)$. Intuitively, after a first few iterations the relationship between elements of the computation depends only on the difference between their iterations.

Theorem 1. *For a recurrent vertex* $e \in R$, $J(e^{\beta+1}) = S_1(J(e^\beta))$ *for all* $\beta \geq \eta + 1$.

Proof. We first show that $S_1(J(e^\beta)) \subseteq J(e^{\beta+1})$. Consider $f^j \in S_1(J(e^\beta))$. If $f \in V \setminus R$ (i.e., f is not a recurrent vertex), then $f^j \in J(e^\beta)$, because the shift operator affects only the recurrent vertices. This impies that there is a path from f^j to e^β, which in turn implies the path from f^j to $e^{\beta+1}$. Hence, $f^j \in J(e^{\beta+1})$. If f is recurrent, then $f^j \in S_1(J(e^\beta))$ implies $f^{j-1} \in J(e^\beta)$. This impies that there is a path from f^{j-1} to e^β, which in turn implies the path from f^j to $e^{\beta+1}$, from the property of d-diagrams. Therefore, $S_1(J(e^\beta)) \subseteq J(e^{\beta+1})$.

Now we show that $J(e^{\beta+1}) \subseteq S_1(J(e^\beta))$. Consider $f^j \in J(e^{\beta+1})$. If $j > 1$, then given a path from f^j to $e^{\beta+1}$, there is a path from f^{j-1} to e^β. Hence $f^j \in S_1(J(e^\beta))$. Now, consider the case when j equals 1. $f^1 \in J(e^{\beta+1})$ implies that there is a path from f^1 to $e^{\beta+1}$. We claim that for $\beta > \eta$, there is also a path from f^1 to e^β. Otherwise, the shortest path from f to e has more than η shift-edges, a contradiction.

When d-diagram generates a poset, Theorem 1 can be used to assign timestamps to vertices in the d-diagram in a way similar to vector clocks. The difference here is that a timestamp for a recurrent vertex is a concise way of representing the timestamps of infinite instances of that vertex.

Each recurrent event, e, has a special *p-timestamp* ($PV(e)$) associated with it, which lets us compute the time stamp for any arbitrary iteration of that event. Therefore, this result gives us an algorithm for assigning p-timestamp to a recurrent event. The p-timestamp for a recurrent event e, $PV(e)$ would be a list of the form

$$(V(e^1), \ldots, V(e^\beta); I(e))$$

where $I(e) = V(e^{\beta+1}) - V(e^\beta)$ and $V(e^j)$ is the timestamp assigned by the normal vector clock algorithm to event e^j. Now for any event $e^j, j > \beta$, $V(e^j) = V(e^\beta) + (j - \beta) * I(e)$.

In figure 4, $\eta = 2$, $\beta = 3$. $V(a^3) = [5, 2]$ and $V(a^4) = [7, 4]$. $I(a) = [2, 2]$. Hence $PV(a) = ([1, 0], [3, 0], [5, 2]; [2, 2])$. Now, calculating $V(a^j)$ for an arbitrary j is trivial. For example, if $j = 6$, then $V(a^6) = [5, 2] + (6 - 3) * [2, 2] = [11, 8]$.

This algorithm requires $O(\eta n)$ space for every recurrent vertex. Once the timestamps have been assigned to the vertices, any two instances of recurrent vertices can be compared in $O(n)$ time.

The notion of vector clock also allows us to keep only the *relevant* events[13] of the d-diagram. Any dependency related question on the relevant events can be answered by simply examining the vector timestamps instead of the entire d-diagram.

5 Detecting Global Predicates

We now consider the problem of detecting predicates in d-diagrams. A predicate is a property defined on the states of the processes and possibly channels. An example of a predicate is "more than one philosopher is waiting."

Given a consistent cut, a predicate is evaluated with respect to the values of the variables resulting after executing all the events in the cut. If a predicate p evaluates to true for a consistent cut C, we say that C satisfies p. We further assume that the truthness of a predicate on a consistent cut is governed only by the *labels* of the events in the frontier of the cut. This assumption implies that the predicates do not involve shared state such as the channel state. We define $\mathcal{L} : G \rightarrow L$ to be an onto mapping from the set of vertices in d-diagram to a set of labels L with the constraint that $\forall e \in V$: $\mathcal{L}(e^i) = \mathcal{L}(e^j)$. This is in agreement with modeling the recurrent events as repetition of the same event.

It is easy to see that it does not suffice to detect the predicate on the d-diagram without unrolling it. As a simple example, consider figure 4, where though $\{a^1, d^1\}$ is not a consistent cut, but $\{a^2, d^1\}$ is consistent.

In this section, we define a finite extension of our d-diagram which enables us to detect any property that could be true in the infinite poset corresponding to the d-diagram. We show that it is sufficient to perform predicate detection on that finite part.

We mainly focus on the recurrent part of the d-diagram as that is the piece which distinguishes this problem from the case of finite directed graph. We identify certain properties of the recurrent part which allows us to apply the techniques developed for finite directed graphs to d-diagrams.

Predicate detection algorithms explore the lattice of global states in BFS order as in Cooper-Marzullo [2] algorithm, or a particular order of events as in Garg-Waldecker [14] algorithm. For finite directed graphs, once the exploration reaches the final global state it signals that the predicate could never become true. In the case of infinite directed graphs, there is no final global state. So, the key problem is to determine the stopping rule that guarantees that if the predicate ever becomes true then it would be discovered before the stopping point. For this purpose, we show that for every cut in the computation, a subgraph of the computation called the *core* contains a cut with the same label. The main result of this section is that the core of the periodic infinite computation is simply the set of events in the computation with iteration less than or equal to N, the number of processes.

Definition 6 (core of a computation). *For a d-diagram Q corresponding to a computation with N processes, we define $U(Q)$, the* core *of Q, as the directed graph given by*

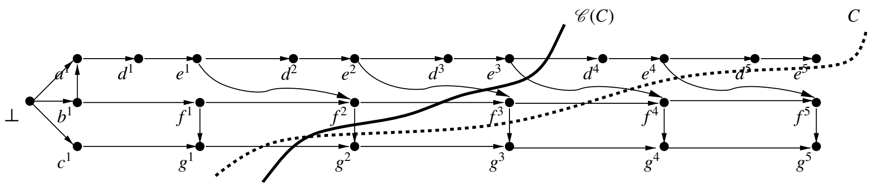


Fig. 5. Compression operation being applied on a cut

the set of events $E' = \{e^j | e \in R \wedge 2 \leq j \leq N\} \cup \{e^1 | e \in V\}$ *and the edges are the restriction of* $\rightarrow$ *to set* E'.

The rest of the section is devoted to proving the completeness of the core of a computation. The intuition behind the completeness of the core is as follows: For any frontier C, we can perform a series of shift operations such that the resulting frontier is consistent and lies in the core. We refer to this operation as a *compression* operation and the resulting cut is denoted by by the frontier $\mathscr{C}(C)$. Figure 5 shows the cut $C = \{e^5, f^3, g^1\}$ and the compressed cut $\mathscr{C}(C) = \{e^3, f^2, g^1\}$.

For proving the completeness of the core, we define the notion of a *compression* operation. Intuitively, compressing a consistent cut applies the shift operation multiple times such that the final cut obtained lies in the core of the computation and has the same labeling.

Definition 7 (Compression). *Given a frontier* C *and index* i, *define* $\mathscr{C}(C, i)$ *as shifting of all events with index greater than* i *by sufficient iterations such that in the shifted frontier the event with next higher index than* i *is* $i + 1$. *The cut obtained after all possible compressions is denoted as* $\mathscr{C}(C)$.

In Figure 5, consider the cut $C = \{e^5, f^3, g^1\}$. When we apply $\mathscr{C}(C, 1)$, we shift events e^5 and f^3 back by 1. This results in the cut $\{e^4, f^2, g^1\}$. The next higher index in the cut now is 2 in f^2. We now apply another compression at index 2, by shifting event e^4, and the compressed cut $\mathscr{C}(C) = \{e^3, f^2, g^1\}$. As another example, consider a cut $C = \{e^7, f^4, g^4\}$. We first apply $\mathscr{C}(C, 0)$ to get the cut $\{e^4, f^1, g^1\}$. Applying the compression at index 1, we finally get $\{e^2, f^1, g^1\}$.

Note that the cut resulting from the compression of a cut C has the same labeling as the cut C. The following lemma shows that it is safe to apply compression operation on a consistent cut i.e. compressing the gaps in a consistent cut results in another consistent cut. This is the crucial argument in proving completeness of the core.

Lemma 4. *If* C *is the frontier of a consistent cut, then* $\mathscr{C}(C, l)$ *corresponds to a consistent cut for any index* l.

Proof. Let $C' = \mathscr{C}(C, l)$ for convenience. Consider any two events $e^i, f^j \in C$. If $i \leq l, j \leq l$ or $i > l, j > l$, then the events corresponding to e^i and f^j in C' are also consistent. When $i > l$ and $j > l$, events corresponding to e^i and f^j in C' get shifted by the same number of iterations.

Now assume $i \leq l$ and $j > l$. Then e^i remains unchanged in C' and f^j is mapped to f^a such that $a \leq j$. Since $i < a$, there is no path from $succ(f^a)$ to e^i. If there is a path from $succ(e^i)$ to f^a, then there is also a path from $succ(e^i)$ to f^j as there is a path from f^a to f^j. This contradicts the fact that e^i and f^j are consistent. Hence, every pair of vertices in the cut C' is consistent.

Now we can use the compression operation to compress any consistent cut to a consistent cut in the core. Since the resulting cut has the same labeling as the original cut, it must satisfy any non-temporal predicate that the original cut satisfies. The following theorem establishes this result.

Theorem 2. *If there is a cut $C \in \mathcal{C}(\langle E, \rightarrow \rangle)$, then there exists a cut $C' \in \mathcal{C}(U(Q))$ such that $\mathcal{L}(C) = \mathcal{L}(C')$.*

Proof. Let $C' = \mathscr{C}(C)$. By repeated application of the lemma 4, we get that C' is a consistent cut and $\mathcal{L}(C) = \mathcal{L}(C')$. Moreover, by repeated compression, no event in C' has index greater than N. Therefore, $C' \in U(Q)$.

The completeness of the core implies that the algorithms for predicate detection on finite directed graphs can be used for d-diagrams as well after unrolling the recurrent events N times. This result holds for any global predicate that is non-temporal (i.e., defined on a single global state). Suppose that the global predicate B never becomes true in the core of the computation, then we can assert that there exists an infinite computation in which B never becomes true (i.e., the program does not satisfy that eventually B becomes true). Similarly, if a global predicate B is true in the recurrent part of the computation, it verifies truthness of the temporal predicate that B becomes true infinitely often.

6 Recurrent Global State Detection Algorithm

We now briefly discuss a method to obtain a d-diagram from a finite distributed computation. The *local state* of a process is the value of all the variables of the process including the program counter. The *channel* state between two processes is the sequence of messages that have been sent on the channel but not received. A *global state* of a computation is defined to be the cross product of local states of all processes and all the channel states at any cut. Any consistent cut of the computation determines a unique consistent global state. A global state is *recurrent* in a computation, if there exist consistent cuts Y and Z such that the global states for Y and Z are identical and Y is a proper subset of Z. Informally, a global state is recurrent if there are at least two distinct instances of that global state in the computation.

We now give an algorithm to detect recurrent global states of a computation. We assume that the system logs the message order and nondeterministic events so that the distributed computation can be exactly replayed. We also assume that the system supports a vector clock mechanism.

The first step of our recurrent global state detection (RGSD) algorithm consists of computing the global state of a distributed system. Assuming FIFO, we could use the classical Chandy and Lamport's algorithm[1] for this purpose. Otherwise, we can use any of the algorithms, such as [15,16,17]. Let the computed global snapshot be G. Let Z be the vector clock for the global state G.

The second step consists of replaying the distributed computation while monitoring the computation to determine the least consistent cut that matches G. We are guaranteed to hit such a global state because there exists at least one such global state (at vector time Z) in the computation. Suppose that the vector clock of the detected global state is Y. We now have two vector clocks Y and Z corresponding to the global state G. If Y equals Z, we continue with our computation. Otherwise, we have succeeded in finding a recurrent global state G.

Note that replaying a distributed computation requires that all nondeterministic events (including the message order) be recorded during the initial execution [18]. Monitoring the computation to determine the least consistent cut that matches G can be done using algorithms for conjunctive predicate detection [3,19].

When the second step fails to find a recurrent global state, the first step of the algorithm is invoked again after certain time interval. We make the following observation about the recurrent global state detection algorithm.

Theorem 3. *If the distributed computation is periodic then the algorithm will detect a recurrent global state. Conversely, if the algorithm returns a recurrent global state G, then there exists an infinite computation in which G appears infinitely often.*

Proof. The RGSD algorithm is invoked periodically and therefore it will be invoked at least once in repetitive part of the computation. This invocation will compute a global state G. Since the computation is now in repetitive mode, the global state G must have occurred earlier and the RGSD algorithm with declare G as a recurrent global state.

We prove the converse by constructing the infinite computation explicitly. Let Y and Z be the vector clocks corresponding to the global state recurrent global state G. Our infinite computation will first execute all events till Y. After that it will execute the computation that corresponds to events executed between Y and Z. Since Y and Z have identical global state, the computation after Y is also a legal computation after Z. By repeatedly executing this computation, we get an infinite legal computation in which G appears infinitely often.

It is important to note that our algorithm does not guarantee that if there exists any recurrent global state, it will be detected by the algorithm. It only guarantees that if the computation is periodic, then it will be detected.

We note here that RGSD algorithm is also useful in debugging applications in which the distributed program is supposed to be terminating and presence of a recurrent global state itself indicates a bug.

7 Related Work

A lot of work has been done in identifying the classes of predicates which can be efficiently detected [7,9]. However, most of the previous work in this area is mainly restricted to finite traces.

Some examples of the predicates for which the predicate detection can be solved efficiently are: *conjunctive* [7,20], *disjunctive* [7], *observer-independent* [21,7], *linear* [7], *non-temporal regular* [22,9] predicates and *temporal* [8,23,24].

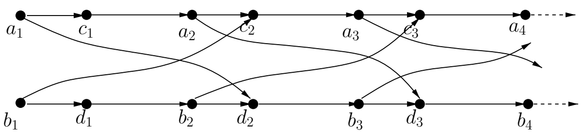


Fig. 6. A poset which cannot be captured using MSC graphs or HMSC

Some representations used in verification explicitly model concurrency in the system using a partial order semantics. Two such prominent models are message sequence charts (MSCs) [25] and petri nets [26]. MSCs and related formalisms such as time sequence diagrams, message flow diagrams, and object interaction diagrams are often used to specify design requirements for concurrent systems. An MSC represents one (finite) execution scenario of a protocol; multiple MSCs can be composed to depict more complex scenarios in representations such as MSC graphs and high-level MSCs (HMSC). These representations capture multiple posets but they cannot be used to model all the posets (and directed graphs) that can be represented by d-diagrams. In particular, a message sent in a MSC node must be received in the same node in MSC graph or HMSC. Therefore, some infinite posets which can be represented through d-diagrams cannot be represented through MSCs. Therefore, an infinite poset such as the one shown in figure 6 is not possible to represent through MSCs.

Petri nets [26] are also used to model concurrent systems. Partial order semantics in petri nets are captured through net unfoldings [27]. Unfortunately, unfoldings are usually infinite sets and cannot be stored directly. Instead, a finite initial part of the unfolding, called the finite complete prefix [28] is generally used to represent the unfolding. McMillan showed that reachability can be checked using the finite prefix itself. Later Esparza [29] extended this work to use unfoldings to efficiently detect predicates from a logic involving the **EF** and **AG** operators. Petri nets are more suitable to model the behavior of a complete system whereas d-diagrams are more suitable for modeling distributing computations in which the set of events executed by a process forms a total order. They are a simple extension of process-time diagrams[11] which have been used extensively in distributed computing literature.

8 Conclusion

In this paper, we introduce a method for detecting violation of liveness properties in spite of observing a finite behavior of the system. Our method is based on (1) determining recurrent global states, (2) representing the infinite computation by a d-diagram, (3) computing vector timestamps for determining dependency and (4) computing the core of the computation for predicate detection. We note here that intermediate steps are of independent interest. Determining recurrent global states can be used to detect if a terminating system has an infinite trace. Representing an infinite poset with d-diagram is useful in storing and replaying an infinite computation.

Our method requires that the recurrent events be unrolled N times. For certain computations, it may not be necessary to unroll recurrent event N times. It would be interesting to develop a method which unrolls each recurrent event just the minimum number of times required for that prefix of the computation to be core.

In this paper, we have restricted ourselves to very simple unnested temporal logic formulas. Detecting a general temporal logic formula efficiently in the model of d-diagram is a future work.

References

1. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems 3(1), 63–75 (1985)
2. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, ACM/ONR, pp. 163–173 (1991)
3. 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)
4. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual IEEE-ACM Symposium on Foundations of Computer Science, pp. 46–57 (1977)
5. Fidge, C.J.: Partial orders for parallel debugging. In: Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, vol. 24(1), pp. 183–194 (1989), published in ACM SIGPLAN Notices
6. Mattern, F.: Virtual Time and Global States of Distributed Systems. In: Proc. of the Int'l Workshop on Parallel and Distributed Algorithms (1989)
7. Garg, V.K.: Elements of Distributed Computing. John Wiley & Sons, Chichester (2002)
8. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: 7th International Conference on Principles of Distributed Systems, La Martinique, France (2003)
9. Mittal, N., Garg, V.K.: Computation Slicing: Techniques and Theory. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, p. 78. Springer, Heidelberg (2001)
10. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 420–434. Springer, Heidelberg (2007)
11. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558–565 (1978)
12. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
13. Agarwal, A., Garg, V.K.: Efficient dependency tracking for relevant events in shared-memory systems. In: Aguilera, M.K., Aspnes, J. (eds.) PODC, pp. 19–28. ACM, New York (2005)
14. Garg, V.K., Waldecker, B.: Detection of unstable predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA. ACM/ONR (1991)
15. Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. Journal of Parallel and Distributed Computing, 423–434 (1993)
16. Garg, R., Garg, V.K., Sabharwal, Y.: Scalable algorithms for global snapshots in distributed systems. In: Proceedings of the ACM Conference on Supercomputing. ACM, New York (2006)
17. Kshemkalyani, A.D.: A symmetric o(n log n) message distributed snapshot algorithm for large-scale systems. In: Cluster, pp. 1–4. IEEE, Los Alamitos (2009)
18. Le Blanc, M.-C.: Debugging parallel programs with instant replay. IEEETC: IEEE Transactions on Computers 36 (1987)
19. Garg, V.K., Chase, C.M., Kilgore, R.B., Mitchell, J.R.: Efficient detection of channel predicates in distributed systems. J. Parallel Distrib. Comput. 45(2), 134–147 (1997)

20. Hurfin, M., Mizuno, M., Raynal, M., Singhal, M.: Efficient detection of conjunctions of local predicates. IEEE Transactions on Software Engineering 24(8), 664–677 (1998)
21. Charron-Bost, B., Delporte-Gallet, C., Fauconnier, H.: Local and temporal predicates in distributed systems. ACM Transactions on Programming Languages and Systems 17(1), 157–179 (1995)
22. Garg, V.K., Mittal, N.: On Slicing a Distributed Computation. In: Proc. of the 15th Int'l Conference on Distributed Computing Systems, ICDCS (2001)
23. Sen, A., Garg, V.K.: Detecting temporal logic predicates in the happened before model. In: International Parallel and Distributed Processing Symposium (IPDPS), Florida (2002)
24. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 420–434. Springer, Heidelberg (2007)
25. Z.120. ITU-TS recommendation Z.120: Message Sequence Chart (MSC) (1996)
26. Petri, C.A.: Kommunikation mit Auto-maten. PhD thesis, Bonn: Institut fuer Instru- mentelle Mathematik (1962)
27. Nielsen, M., Winskel, G.P., Petri, G.: nets, event structures and domains. Theoretical Computer Science 13(1), 85–108 (1980)
28. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
29. Esparza, J.: Model checking using net unfoldings. Science of Computer Programming 23(2), 151–195 (1994)

Complexity Issues in Automated Model Revision without Explicit Legitimate State[*]

Fuad Abujarad and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
{abujarad,sandeep}@cse.msu.edu
http://www.cse.msu.edu/~{abujarad,sandeep}

Abstract. Existing algorithms for the automated model revision incur an impediment that the designers have to identify the legitimate states of original model. Experience suggests that of the inputs required for model revision, identifying such legitimate states is the most difficult. In this paper, we consider the problem of automated model revision without explicit legitimate states. We note that without the explicit legitimate states, in some instances, the complexity of model revision increases substantially (from P to NP-hard). In spite of this, we find that this formulation is relatively complete, i.e., if it was possible to perform model revision *with* explicit legitimate states then it is also possible to do so *without* the explicit identification of the legitimate states. Finally, we show if the problem of model revision can be solved with explicit legitimate states then the increased cost of solving it without explicit legitimate states is very small.

In summary, the results in this paper identify instances of model revision where the explicit knowledge of legitimate state is beneficial and where it is not very crucial.

Keywords: Model Revision, Program Synthesis.

1 Introduction

There are several instances where one needs to revise an existing model. In particular, model revision is required to account for bug fixes, newly discovered faults or changes in the environment. Quite often, model revisions are done manually and, hence, create several concerns. For one, it requires a significant effort and resources. Also, the new (i.e., revised) model may violate other properties that the original model provided. Consequently, this approach entails that the

[*] This work was partially sponsored by the Air Force ContractFA9550-10-1-0178 NSF CNS 0914913.

new model must be rechecked and re-tested to verify if it still preserves the old properties in addition to the new properties.

Another approach for revising existing models/programs is through *automated model revision*. The goal of the automated model revision is to automatically revise an existing model to generate a new model, which is correct-by-construction [13]. Such model will also preserve the existing model properties and satisfy new properties. In its basic form, the problem of automated model revision (also known as incremental synthesis) focuses on modifying an existing model, say p, into a new model, say p'. It is required that p' satisfies the new property of interest. Additionally, p' continues to satisfy existing properties of p using the same transitions that p used.

Current approaches for automated model revision for revising an existing model to add fault-tolerance include [7,8,13,15]. These approaches describe the model as an abstract program. They require the designer to specify (1) the existing abstract program that is correct in the absence of faults, (2) the program specification, (3) the faults that have to be tolerated, and (4) the program legitimate states, from where the existing program satisfies its specification (c.f. Figure 1). We call this problem as *the problem of model revision with explicit legitimate states.*

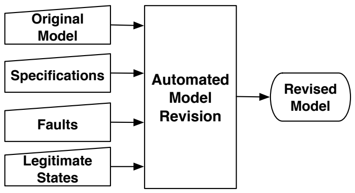


Fig. 1. Model Revision with Explicit Legitimate States

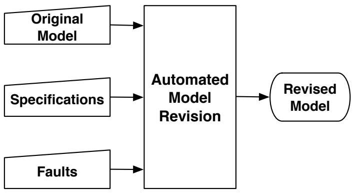


Fig. 2. Model Revision *without* Explicit Legitimate States

Of these four inputs, the first three are easy to identify and/or unavoidable. For example, one is expected to utilize model revision only if they have an existing model that fails to satisfy a required property. Thus, if model revision is applied in the context of newly identified faults, original model and faults are already available. Likewise, specification identifies what the model was supposed to do. Clearly, requiring it is unavoidable.

Identifying the legitimate states from where the fault-intolerant program satisfies its specification is however a difficult task. Our experience in this context shows that while identifying the other three arguments is often straightforward, identifying precise legitimate states requires significant effort. With this motivation, in this paper, we focus on the problem of model revision where the input only consists of the fault-intolerant program, faults and the specification, i.e., it does not include the legitimate states. We call this problem as *the problem of model revision without explicit legitimate states* (cf. Figure 2).

There are several important questions that have to be addressed for such a new formulation.

Q. 1 Is the new formulation relatively complete? (i.e., if it is possible to perform model revision using the problem formulation in Figure 1, is it guaranteed that it could be solved using the formulation in Figure 2.)
An affirmative answer to this question will indicate that reduction of designers' burden does not affect the solvability of the corresponding problem.

Q. 2 Is the complexity of both formulations in the same class? (By same class, we mean polynomial time reducibility, where complexity is computed in the size of state space.)
An affirmative answer to this question will indicate that the reduction in the designers' burden does not significantly affect the complexity.

Q. 3 Is the increased time cost, if any, small comparable to the overall cost of program revision?
While Question 2 focuses on qualitative complexity, assuming that the answer is affirmative, Question 3 will address the quantitative change in complexity.

The contributions of the paper are as follows:

– We show that the answer to Q. 1 is affirmative (cf. Theorem 1).
– Regarding Q.2, we consider two versions of the problem (*partial* revision and *total revision*):
  • We note that the answer to Q.2 is negative for partial revision.
  • We show that the answer to Q.2 is affirmative for total revision.
  • We show that there is a class of problems where partial revision can be achieved in polynomial time. This class includes all instances where it is possible to successfully revise a program based on the formulation in Figure 1.
– Regarding Q. 3, we show that for instances where the answer to the question in Figure 1 is affirmative, the extra computation cost of solving the problem using an approach in Figure 2 is small.

Organization of the paper. The rest of the paper is organized as follows: We define distributed program and specification in Section 2. In Section 3, we give the automated model revision problem statement. In Sections, 4, 5 and 6, we answer the above three questions respectively. We discuss related work in Section 7 and conclude in Section 8.

2 Programs, Specifications and Faults

In this section, we formally define programs, specifications, faults and fault-tolerance. We also identify the problem of automated program revision with explicit legitimate states and without legitimate states. Part of those definitions is based on the ones given by Arora and Gouda [3].

2.1 Programs and Specifications

Since we focus on the design of distributed programs, we specify the state space of the program in terms of its variables. Each variable is associated with its domain. A state of the program is obtained by assigning each of its variables a value from the respective domain. The state space of the program is the set of all states. Thus, a program p is a tuple $\langle S_p, \delta_p \rangle$ where S_p is a finite set of states which identifies the program state space, and δ_p is a subset of S_pxS_p which identifies the program transitions. A state predicate, say S, of $p(= \langle S_p, \delta_p \rangle)$ is any subset of S_p. Since a state predicate can be characterized by the set of all states in which its Boolean expression is true, we use sets of states and state predicates interchangeably. Thus, conjunction, disjunction and negation of sets are the same as the conjunction, disjunction and negation of the respective state predicates. We say that the state predicate S is *closed* in p iff $(\forall s_0, s_1 :: s_0 \in S \wedge (s_0, s_1) \in \delta_p \ \Rightarrow \ s_1 \in S)$. In other words, S is closed in p iff every transition of p that begins in S also ends in S.

Let $\sigma{=}\langle s_0, s_1, ... \rangle$ be a sequence of states, then σ is a computation of $p(= \langle S_p, \delta_p \rangle)$ iff the following two conditions are satisfied:

- $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \in \delta_p$, where $length(\sigma)$ = the number of states in σ,
- if σ is finite and the last state in σ is s_l then there does not exist state s such that $(s_l, s) \in \delta_p$.

Notation. We call δ_p the transitions of p. When it is clear from context, we use p and δ_p interchangeably.

The safety specification, Sf_p, for program p is specified in terms of bad states, $SPEC_{bs}$, and bad transitions $SPEC_{bt}$. Thus, $\sigma{=}\langle s_0, s_1, ... \rangle$ satisfies the safety specification of p iff the following two conditions are satisfied.

- $\forall j : 0 \le j < length(\sigma) : s_j \notin SPEC_{bs}$, and
- $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \notin SPEC_{bt}$.

Let $\mathcal{F}$ and $\mathcal{T}$ be state predicates, then the liveness specification, Lv_p, of program p is specified in terms of one or more leads-to properties of the form $\mathcal{F} \leadsto \mathcal{T}$. A sequence $\sigma = \langle s_0, s_1, ... \rangle$ satisfies $\mathcal{F} \leadsto \mathcal{T}$ iff $\forall j : (\mathcal{F}$ is true in $s_j \Rightarrow \exists k : j \le k < length(\sigma) : \mathcal{T}$ is true in $s_k)$.

A specification, say *SPEC* is a tuple $\langle Sf_p \ , \ Lv_p \rangle$, where Sf_p is a safety specification and Lv_p is a liveness specification. A sequence σ satisfies *SPEC* iff it satisfies Sf_p and Lv_p. Hence, for brevity, we say that the program specification is an intersection of a safety specification and a liveness specification.

Let $I \subseteq S_p$ and $I \neq \{\}$ (i.e., the empty set). We say that a program p satisfies *SPEC* from I iff the following two conditions are satisfied.

- I is closed in p, and
- every computation of p that starts from a state in I satisfies *SPEC*.

If p satisfies $SPEC$ from I then we say that I is a legitimate state predicate of p for $SPEC$. We use the term "legitimate state predicate" and the corresponding "set of legitimate states" interchangeably.

Assumption 2.1 : For simplicity of subsequent definitions, if p satisfies $SPEC$ from I, we assume that p includes at least one transition from every state in I. If p does not include a transition from state s then we add the transition (s, s) to p. Note that this assumption is not restrictive in any way. It simplifies subsequent definitions, as one does not have to model terminating computations explicitly.

2.2 Faults and Fault-Tolerance

The faults, say f, that a program is subject to are systematically represented by transitions. Based on the classification of faults from [6], this representation suffices for physical faults, process faults, message faults, and improper initialization. It is not intended for program bugs (e.g. buffer overflow). However, if such bugs exhibit behavior such as component crash, it can be modeled using this approach. As an example, for the case considered in the Introduction, a sticky gas pedal can be modeled by a variable stuck.j; when this variable is false the gas pedal behaves normally. But a fault can set it to true thereby preventing the gas pedal from changing its status. Thus, a fault for $p(=\langle S_p, \delta_p\rangle)$ is a subset of S_pxS_p.

We use '$p[]f$' to mean 'p in the presence of f'. The transitions of $p[]f$ are obtained by taking the union of the transitions of p and the transitions of f. Just as we defined computations of a program in Section 2.1, we define the notion of program computations in the presence of faults. In particular, a sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$, is a computation of $p[]f$ (i.e., a computation of $p(= \langle S_p, \delta_p\rangle)$ in the presence of f) iff the following three conditions are satisfied:

- $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \in (\delta_p \cup f)$, and
- if $\langle s_0, s_1, ...\rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$,
- if σ is infinite then $\exists n : \forall j > n : (s_{j-1}, s_j) \in \delta_p$

Thus, if σ is a computation of p in the presence of f then in each step of σ, either a transition of p occurs or a transition of f occurs. Additionally, σ is finite only if it reaches a state from where the program has no outgoing transition. And, if σ is infinite then σ has a suffix where only program transitions execute. We note that the last requirement can be relaxed to require that σ has a sufficiently long subsequence where only program transitions execute. However, to avoid details such as the length of the subsequence, we require that σ has a suffix where only program transitions execute. (This assumption is not required for failsafe fault-tolerance described later in this section.) We use f-span (*fault-span*) to identify the set of states reachable by $p[]f$. In particular, a predicate T is an f-span of p from I iff $I \Rightarrow T$ and $(\forall (s_0, s_1) : (s_0, s_1) \in p[]f : (s_0 \in T \Rightarrow s_1 \in T))$.

Thus, at each state where I of p is true, f-span T of p from I is also true. Also, T, like I, is also closed in p. Moreover, if any action in f is executed in a state where T is true, the resulting state is also one where T is true. It follows that for all computations of p that start at states where I is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the actions in f.

Fault-Tolerance. In the absence of faults, a program, p, satisfies its specification and remains in its legitimate states. In the presence of faults, it may be perturbed to a state outside its legitimate states. By definition, when the program is perturbed by faults, its state will be one in the corresponding f-span. From such a state, it is desired that p does not violate its safety specification. Furthermore, p recovers to its legitimate states from where p subsequently satisfies both its safety and liveness specification.

Based on this intuition, we now define what it means for a program to be (masking) fault-tolerant. Let Sf_p and Lv_p be the safety and liveness specifications for program p. We say that p is masking fault-tolerant to Sf_p and Lv_p from I iff the following two conditions hold.

1. p satisfies Sf_p and Lv_p from I.
2. $\exists\, T$ ::
   (a) T is f-span of p from I.
   (b) $p[]f$ satisfies Sf_p from T.
   (c) Every computation of $p[]f$ that starts from a state in T has a state in I.

While masking fault-tolerance is ideal, for reasons of costs and feasibility, a weaker level of fault-tolerance is often required. Two commonly considered weaker levels of fault-tolerance include *failsafe* and *nonmasking*. In particular, we say that p is failsafe fault-tolerant [11] if the conditions 1, 2a, and 2b are satisfied in the above definition. And, we say that p is nonmasking fault-tolerant [12] if the conditions 1, 2a, and 2c are satisfied in the above definition.

3 Problem Statement

In this section, we formally define the problem of model revision with and without explicit legitimate states.

Model Revision *with* Explicit Legitimate States (Approach in Figure 1). We formally specify the problem of deriving a fault-tolerant program from a fault-intolerant program with explicit legitimate states I. The goal of the model revision is to modify p to p' by *only adding fault-tolerance*, i.e., without adding new behaviors in the absence of faults. Since the correctness of p is known only from its legitimate states, I, it is required that the legitimate states of p', say I', cannot include any states that are not in I. Additionally, inside the legitimate states, it cannot include transitions that were not transitions of p. Also, by Assumption 2.1, p cannot include new terminating states that were not terminating states of p. Finally, p' must be fault-tolerant. Thus, the problem statement (from [13]) for the case where the legitimate states are specified explicitly is as follows.

Problem Statement 3.1: Revision for Fault-Tolerance *with* Explicit Legitimate States.
Given p, I, Sf_p, Lv_p and f such that p satisfies Sf_p and Lv_p from I
Identify p' and I' such that: (Respectively, does there exist p' and I' such that)
A1: $I' \Rightarrow I$.
A2: $s_0 \in I' \;\Rightarrow\; \forall s_1 : s_1 \in I' : ((s_0, s_1) \in p' \Rightarrow (s_0, s_1) \in p)$.
A3: p' is f-tolerant to Sf_p and Lv_p from I'.

Note that this definition can be instantiated for each level of fault-tolerance (i.e., masking, failsafe, and nonmsaking). Also, the above problem statement can be used as a revision problem or a decision problem (with the comments inside parenthesis).

We call the above problem the problem of 'partial revision' because the transitions of p' that begin in I' are a subset of the transitions of p that begin in I'. An alternative formulation is that of 'total revision' where the transitions of p' that begin in I' are *equal to* the transitions of p that begin in I'. In other words, the problem of *total revision* is identical to the problem statement 3.1 except that A2 is changed to A2$'$ described next:

A2$'$: $s_0 \in I' \Rightarrow \forall s_1 : s_1 \in I' : ((s_0, s_1) \in p' \iff (s_0, s_1) \in p)$

Modeling Revision *without* Explicit Legitimate States (Approach in Figure 2). Now, we formally define the new problem of model revision without explicit legitimate states. The goal in this problem is to find a fault-tolerant program, say p_r. It is, also, required that there is some set of legitimate states for p, say I, such that p_r does not introduce new behaviors in I. Thus, the problem statement for partial revision for the case where the legitimate states are not specified explicitly is as follows.

Problem Statement 3.2: Revision for Fault-Tolerance *without* Explicit Legitimate States.
Given p, Sf_p and Lv_p, and f
Identify p_r such that: (Respectively, does there exist p_r such that)
($\exists I ::$
B1: $s_0 \in I \;\Rightarrow \forall s_1 : s_1 \in I : \; ((s_0, s_1) \in p_r \Rightarrow (s_0, s_1) \in p)$
B2: p_r is a f-tolerant to Sf_p and Lv_p from I.)

Just like problem statement 3.1, the problem of total revision is obtained from problem statement 3.2 by changing B1 with B1$'$ described next:

B1$'$: $s_0 \in I \;\Rightarrow \forall s_1 : s_1 \in I : \; ((s_0, s_1) \in p_r \iff (s_0, s_1) \in p)$

Existing algorithms for model revision [8,13,7,15] are based on Problem Statement 3.1. Also, the tool *SYCRAFT* [8] utilizes Problem Statement 3.1 for the addition of fault-tolerance. However, as stated in Section 1, this requires the users of *SYCRAFT* to identify the legitimate states explicitly. The goal of this paper is evaluate the effect of simplifying the task of the designers by permitting them to omit explicit identification of legitimate states.

4 Relative Completeness (Q. 1)

In this section, we show that if the problem of model revision can be solved with explicit legitimate states (Problem Statement 3.1) then it can also be solved without explicit legitimate states (Problem Statement 3.2). Since each problem statement can be instantiated with partial or total revision, this requires us to consider four combinations. We prove this result in Theorem 1.

Theorem 1. *If the answer to the decision problem 3.1 is affirmative (i.e., $\exists$ p' and I' that satisfy the constraints of the Problem 3.1) with input p, Sf_p, Lv_p, f, and I, then the answer to the decision problem 3.2 is affirmative (i.e., $\exists$ p_r that satisfies the constraints of the Problem 3.2) with input p, Sf_p, Lv_p, and f.*

Proof. Intuitively, a slightly revised version of the program that satisfies Problem 3.1 can be used to show that Problem 3.2 can be solved. Specifically, let the transitions of p_r to be $\{(s_0, s_1) | \, (s_0 \in I' \wedge s_1 \in I' \wedge (s_0, s_1) \in p) \vee (s_0 \notin I' \wedge (s_0, s_1) \in p')$ $\}$. For reasons of space we refer the reader to [1] for full proof. $\square$

Implication of Theorem 1 for Q. 1: From Theorem 1, it follows that answer to Q. 1 from Introduction is affirmative for both partial and total revision. Hence, the new formulation (c.f. Figure 2) is relatively complete.

5 Complexity Analysis (Q. 2)

In this section, we focus on the second question and compare the complexity class for Problem 3.1 with that of Problem 3.2. First, we note that the complexity of partial revision changes from P to NP-Complete if legitimate states are not specified explicitly. For reasons of space we refer the reader to [1] for full proof. Then in Section 5.1, we show that for total revision Problem 3.2 can be reduced to Problem 3.1 in polynomial time. In Section 5.2, we give a heuristic based approach for partial revision. Furthermore, we show that the heuristic is guaranteed to work when the answer to the corresponding Problem in Figure 1 is affirmative. Finally, we mention other complexity results in Section 5.3.

5.1 Complexity Comparison for Total Revision

Although the complexity of partial revision increases substantially when legitimate states are not available explicitly, we find that complexity of total revision effectively remains unchanged. We note that this is the first instance where complexity difference between partial and total revision has been identified. To show this result, we show that in the context of total revision Problem 3.2 is polynomial time reducible to Problem 3.1 Since the results in this section require the notion of weakest legitimate state predicate, we define it next. Recall that, we use the term legitimate state predicate and the corresponding set of legitimate states interchangeably. Hence, weakest legitimate state predicate corresponds to the largest set of legitimate states.

Definition. Let $I_w = wLsp(p, Sf_p, Lv_p))$ be the *weakest legitimate state predicate* of p for $SPEC (=\langle Sf_p \ , \ Lv_p \rangle)$ iff:
1: p satisfies $SPEC$ from I_w, and
2: $\forall \ I :: \ (\ p$ satisfies $SPEC$ from $I) \Rightarrow I_w$. □

Claim. Given p, Sf_p and Lv_p, it is possible to compute $wLsp(p, Sf_p, Lv_p)$ in polynomial time in the state space of p.

This claim was proved in [2] where we have identified an algorithm to compute weakest legitimate state predicate.

Theorem 2. *If the answer to the decision problem 3.2 (***with total revision***) is affirmative (i.e., $\exists \ p_r$ that satisfies the constraints of the Problem 3.2) with input p, Sf_p, Lv_p, and f, then the answer to the decision problem 3.1 (with total or partial revision) is affirmative (i.e., $\exists \ p'$ and I' that satisfy the constraints of the Problem 3.1) with input p, Sf_p, Lv_p, f, and $wLsp(p, Sf_p, Lv_p)$.*

Proof. Intuitively, the program p_r obtained for solving problem statement 3.2 can be used to show that problem 3.1 is satisfied. Specifically, let I_2 be the predicate used to show that p_r satisfies constraints of Problem 3.2. Then, let $p' = p_r$ and $I' = I_2$. For reasons of space we refer the reader to [1] for full proof. □

Theorem 3. *For total revision, the revision problem 3.2 is polynomial time reducible to the revision problem 3.1.*

Proof. Given an instance, say X, of the decision problem 3.2 that consists of p, Sf_p, Lv_p, and f, the corresponding instance, say Y, for the decision problem 3.1 is p, Sf_p, Lv_p, f and $wLsp(p, Sf_p, Lv_p)$. From Theorems 1 and 2 it follows that answer to X is affirmative iff answer to Y is affirmative. □

5.2 Heuristic for Polynomial Time Solution for Partial Revision

Theorem 2 utilizes the weakest legitimate state predicate to solve the problem of total revision without explicit legitimate states. In this section, we show that a similar approach can be utilized to develop a heuristic for solving the problem of partial revision in polynomial time. Moreover, if there is an affirmative answer to the revision problem with explicit legitimate states then this heuristic is guaranteed to find a revised program that satisfies constraints of Problem 3.2. Towards this end, we present Theorem 4.

Theorem 4. *For partial revision, the revision problem 3.2 consisting of* (p, Sf_p, Lv_p, f) *is polynomial time reducible to the revision problem 3.1 provided there exists a legitimate states predicate* I *such that the answer to the decision problem 3.1 for instance* (p, I, Sf_p, Lv_p, f) *is affirmative.*

Proof. Clearly, if an instance of Problem 3.1 has an affirmative answer then from Theorem 1, the corresponding instance of Problem 3.2 has an affirmative answer. Similar to the proof of Theorem 3, we map the instance of Problem 3.2 to an instance of Problem 3.1 where we use the weakest legitimate state predicate. Now, from Theorem 1 it follows that the answer to this revised instance of Problem 3.1 is also affirmative. □

5.3 Summary of Complexity Results

In summary, the results for complexity comparison are as shown in Table 1. Results marked with † follow from NP-completeness results from [1] . Results marked ‡ follow from Section 5.1 and5.2. Results marked △ are stated without proof due reasons of space. Results marked ? indicate that the, complexity of the corresponding problem is open. And, finally, results marked ∗ are from [13].

Table 1. The complexity of different types of automated revision (NP-C = NP-Complete). Results marked with asterisk are from [13]. Other results are from in this paper.

		Revision *Without* Explicit Legitimate States		**Revision *With* Explicit Legitimate States**	
		Partial	**Total**	**Partial**	**Total**
High Atomicity	Failsafe	?	$P^{\ddagger}$	P^*	P^*
	nonmasking	?	$P^{\ddagger}$	P^*	P^*
	masking	$NP-C^{\dagger}$	$P^{\ddagger}$	P^*	P^*
Distributed Programs	Failsafe	$NP-C^{\triangle}$	$NP-C^{\triangle}$	$NP-C^*$	$NP-C^*$
	nonmasking	?	?	?	?
	masking	$NP-C^{\triangle}$	$NP-C^{\triangle}$	$NP-C^*$	$NP-C^*$

6 Relative Computation Cost (Q. 3)

As mentioned in Section 1, the increased cost of model revision in the absence of explicit legitimate states needs to be studied in two parts: complexity class and relative increase in the execution time. We considered the former in Section 5. In this section, we consider the latter. The increased cost of model revision is essentially that of computing $wLsp(p, Sf_p, Lv_p)$. Hence, we analyze the cost of computing $wLsp(p, Sf_p, Lv_p)$ in the context of a case study. We choose a classic example from the literature namely Byzantine Agreement [14]. We explain this case study in detail and show the time required to generate the weakest legitimate state predicate for different numbers of processes. This case study illustrates that the increased cost when explicit legitimate states are unavailable is very small compared to the overall time required for the addition of fault-tolerance. In particular, we show that reducing the burden of the designer in terms of not requiring the explicit legitimate states increases the computation cost by approximately 1%.

Throughout this section, the experiments are run on a MacBook Pro with 2.6 Ghz Intel Core 2 Duo processor and 4 GB RAM. The OBDD representation of the Boolean formula has been done using the C++ interface to the CUDD package developed at the University of Colorado [16].

Byzantine Agreement Program. Now, we illustrate our algorithm in the context of the Byzantine agreement program. We start by specifying the fault-intolerant program. Then we provide the program specification. Finally we describe the weakest legitimate state predicate generated by our algorithm.

Program. The Byzantine agreement program consists of a "general" and three or more non-general processes. Each process copies the decision of the general and finalizes (outputs) that decision. The general process maintains two variables: the decision $d.g$ with domain $\{0, 1\}$ and the Byzantine $b.g$ with domain $\{true, false\}$, to indicate whether or not the general is Byzantine. Moreover, a byzantine process can change its decision arbitrarily. Each of the non-general processes has the following three variables: the decision d with domain $\{0, 1, \bot\}$, where $\bot$ denotes that the process did not yet receive any decision from the general, the Byzantine b with domain {true, false}, and the finalize f with the domain $\{true, false\}$ to denote whether or not the process has finalized (outputted) its decision. The following are the actions of the Byzantine agreement program. Of these, the first action allows a non-general to receive a decision from the general if it has not received it already. The second action allows the non-general to finalize its decision after it receives it. The third and fourth actions allow a Byzantine process to change its decision and finalized status. The last two actions are environment actions.

$$\begin{array}{l}
1 :: (d.j = \bot) \wedge (f.j = \mathit{false}) \quad \longrightarrow \quad d.j := d.g; \\
2 :: (d.j \neq \bot) \wedge (f.j = \mathit{false}) \quad \longrightarrow \quad f.j := \mathit{true}; \\
3 :: (b.j) \quad \longrightarrow \quad d.j := 1|0, \;\; f.j := \mathit{false}|\mathit{true}; \\
4 :: (b.g) \quad \longrightarrow \quad d.g := 1|0;
\end{array}$$

Where $j \in \{1 \dots n\}$ and n is the number of non-general processes.

Specification. The safety specification of the Byzantine agreement requires *validity* and *agreement*:

- *Validity* requires that if the general is non-Byzantine, then the final decision of a non-Byzantine process must be the same as that of the general. Thus, $validity(j)$ is defined as follows.
  $validity(j) = ((\neg b.j \;\wedge\; \neg b.g \;\wedge\; f.j) \;\Rightarrow\; (d.j = d.g))$
- *Agreement* means that the final decision of any two non-Byzantine processes must be equal. Thus, $agreement(j, k)$ is defined as follows.
  $agreement(j, k) = ((\neg b.j \;\wedge \neg b.k \;\wedge\; f.j \;\wedge\; f.k) \;\Rightarrow\; (d.j = d.k))$

– The final decision of a process must be either 0 or 1. Thus, $final(j)$ is defined as follows.
  $final(j) = f.j \Rightarrow (d.j = 0 \vee d.j = 1)$

We formally identify safety specification of the Byzantine agreement in the following set of bad states:

$$SPEC_{BA_{bs}} = (\ \exists j, k \in \{1..n\} :: (\ \neg(validity(j) \wedge agreement(j, k) \wedge final(j))\ )$$

Observe that $SPEC_{BA_{bs}}$ can be easily derived based on the specification of the Byzantine Agreement problem. The liveness specification of the Byzantine agreement requires that eventually every non-Byzantine process finalizes a decision. Thus the liveness specification is $\neg b.j \leadsto (f.j)$.

Application of Our Algorithm. The weakest legitimate state predicate computed (for 3 non-general processes) is as follows. If the general is non-Byzantine then it is necessary that $d.j$, where j is also a non-Byzantine, be either $d.g$ or $\perp$. Furthermore, a non-Byzantine process cannot finalize its decision if its decision equals $\perp$. Now, we consider the set of states where the general is Byzantine. In this case, the general can change its decision arbitrarily. Also, the predicate includes states where other processes are non-Byzantine and have the same value that is different from $\perp$. Thus, the generated weakest legitimate state predicate is as follows:

$$\begin{array}{l} I_{\mathcal{BA}} = \\ (\ \neg b.g \wedge (\ \forall p \in \{1..n\} :: ((\neg b.p \wedge f.p) \Rightarrow d.p \neq \perp)\ \wedge \\ (\neg b.p \Rightarrow (d.p = \perp \vee d.p = d.g)))\ )\ \vee \\ (\ b.g \wedge (\ \forall\ j, k \in \{1..n\} : j \neq k :: (d.j = d.k)\ \wedge\ (d.j \neq \perp)\ )\ ) \end{array}$$

Observe that $I_{\mathcal{BA}}$ cannot be easily derived based on the specification of the Byzantine Agreement problem. More specifically, the set of states where the general is Byzantine, are not reachable from the initial states of the program.

The amount of time required for performing the automated model revision and computing the set of legitimate states for a different number of processes is as shown in Table 2. Thus, the extra time required when legitimate states are unavailable is small (about 1%).

Finally, we use this case study to show that a typical algorithm for computing legitimate states to be reachable states from some initial state(s) does not work. In particular, we make the following claim:

Claim. An automated model revision approach where one uses legitimate states to be the set of states reached from the initial states is not relatively complete.

To validate this claim, observe that the initial states of the Byzantine Agreement program equal the states where all processes are non-byzantine and the decision of all non-general processes is $\perp$. States reached by the fault-intolerant program from these states do not include the states where the general is byzantine. Although, an agreement can be reached among non-general processes even though the general is Byzantine. And, utilizing these reachable states as the set of legitimate states is insufficient to obtain the fault-tolerant program.

Table 2. The time required to perform the automated model revision without explicit set of legitimate states for the Byzantine Agreement program

No.of Process	Reachable States	Leg. States Generation Time(Sec)	Total Revision Time(Sec)
10	10^9	0.57	6
20	10^{15}	1.34	199
30	10^{22}	4.38	1836
40	10^{30}	9.25	9366
50	10^{36}	26.34	> 10000
100	10^{71}	267.30	> 10000

7 Related Work

Our work is closely related to the work on controller synthesis (e.g. [9,4,5]) and game theory (e.g., [10]). In this work, supervisory control of real-time systems has been studied under the assumption that the existing program (called a *plant*) and/or the given specification is *deterministic*. These techniques require highly expressive specifications. Hence, the complexity is also high (Exptime-complete or higher). In addition, these approaches do not address some of the crucial concerns of fault-tolerance (e.g., providing recovery in the presence of faults) that are considered in our work.

Algorithms for automatic model revision and addition of fault-tolerance [7,8, 13,15] add fault-tolerance concerns to existing untimed or real-time programs in the presence of faults, and guarantee the addition of no new behaviors to the original program in the absence of faults. Kulkarni and Arora [13] introduce synthesis methods for automated addition of fault-tolerance to untimed centralized and distributed programs. In particular, they introduce polynomial-time sound and complete algorithms for adding all levels of fault-tolerance (failsafe, non-masking, and masking) to centralized programs. The input to these algorithms is a fault-intolerant centralized program, safety specification, and a set of fault transitions. The algorithms generate a fault-tolerant program along with the legitimate states predicate.

8 Conclusion and Future Work

The goal of this work is to simplify the task of model revision and, thereby, make it easier to apply in practice. Of the inputs required for model revision, existing model is clearly a must. Moreover, the task required in identifying it is easy, as model revision is expected to be used in contexts where designers already have an existing model. Another input, namely specification, is also already available to the designer when model revision is used in contexts where existing model fails to satisfy the desired specification. Yet another input is the new property that is to be added to the existing model. In the context of fault-tolerance, this requires the designers to identify the faults that need to be tolerated. Once again, identifying the fault is easy especially in contexts where the model needs to be revised due to newly identified faults (e.g., stuck-at gas pedal). While representing these faults may be somewhat difficult, it is possible to represent them easily for faults such as stuck-at faults, crash faults, Byzantine faults, transient faults, message faults, etc.

However, based on our experience, the hardest input to identify is the set of legitimate states from where the original model satisfies its specification. In part, it is because of the fact that identifying these legitimate states explicitly is often not required during the evaluation of the original model. Hence, our goal in this paper is to focus on the problem of model revision when these legitimate states are not specified explicitly. Moreover, as shown by the example in Section 6 typical algorithms for computing legitimate states based on initial states do not work.

We considered three questions in this context: (1) relative completeness, (2) qualitative complexity class comparison and (3) quantitative change the time for model revision. We illustrated that our approach for model revision without explicit legitimate states is relatively complete, i.e., if model revision can be solved with explicit legitimate states then it could also be solved without explicit legitimate states. This is important since it implies that the reduction in the human effort required for model revision does not reduce the class of the problems that could be solved.

Regarding the second question, we found some surprising and counterintuitive results. Specifically, for total revision, we found that the complexity class remains unchanged. However, for partial revision, the complexity class changes substantially. In particular, we showed that problems that could be solved in P when legitimate states are available explicitly become NP-complete if explicit legitimate states are unavailable. This result is especially surprising since this is the first instance where complexity levels for total and partial revision have been found to be different. Even though the general problem of partial revision becomes NP-complete without the explicit legitimate states, we found a subset of these problems that can be solved in P. Specifically, this subset included all instances where model revision was possible when legitimate states are specified explicitly.

Regarding the third question, we showed that the extra computation cost obtained by reducing the human effort for specifying the legitimate states is negligible (less than 1%).

References

1. Abujarad, F., Kulkarni, S.S.: Complexity issues in automated model revision without explicit legitimate state. Technical Report MSU-CSE-10-19, Computer Science and Engineering, Michigan State University, East Lansing, Michigan (July 2010), Available as Technical Report MSU-CSE-10-19 at, http://www.cse.msu.edu/cgi-user/web/tech/reports?Year=2010

2. Abujarad, F., Kulkarni, S.S.: Weakest Invariant Generation for Automated Addition of Fault-Tolerance. Electronic Notes in Theoretical Computer Science 258(2), 3–15 (2009), Available as Technical Report MSU-CSE-09-29 at, http://www.cse.msu.edu/cgi-user/web/tech/reports?Year=2009
3. Arora, A., Gouda, M.G.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering 19(11), 1015–1027 (1993)
4. Asarin, E., Maler, O.: As soon as possible: Time optimal control for timed automata. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 19–30. Springer, Heidelberg (1999)
5. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: IFAC Symposium on System Structure and Control, pp. 469–474 (1998)
6. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 11–33 (2004)
7. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 3–10 (2007)
8. Bonakdarpour, B., Kulkarni, S.S.: Sycraft: A tool for synthesizing distributed fault-tolerant programs. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 167–171. Springer, Heidelberg (2008)
9. Bouyer, P., D'Souza, D., Madhusudan, P., Petit, A.: Timed control with partial observability. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 180–192. Springer, Heidelberg (2003)
10. Faella, M., LaTorre, S., Murano, A.: Dense real-time games. In: Logic in Computer Science (LICS), pp. 167–176 (2002)
11. Gärtner, F.C., Jhumka, A.: Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. In: FORMATS/FTRTFT, pp. 183–198 (2004)
12. Gartner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Computing Surveys (CSUR) 31(1), 1–26 (1999)
13. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 82–93. Springer, Heidelberg (2000)
14. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
15. Mantel, H., Gärtner, F.C.: A case study in the mechanical verification of fault-tolerance. Technical Report TUD-BS-1999-08, Department of Computer Science, Darmstadt University of Technology (1999)
16. Somenzi, F.: CUDD: Colorado University Decision Diagram Package, http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

Algorithmic Verification of Population Protocols[⋆]

Ioannis Chatzigiannakis[1,2], Othon Michail[1,2], and Paul G. Spirakis[1,2]

[1] Research Academic Computer Technology Institute (RACTI), Patras, Greece
[2] Computer Engineering and Informatics Department (CEID), University of Patras
{ichatz,michailo,spirakis}@cti.gr

Abstract. In this work, we study the Population Protocol model of Angluin *et al.* from the perspective of protocol verification. In particular, we are interested in algorithmically solving the problem of determining whether a given population protocol conforms to its specifications. Since this is the first work on verification of population protocols, we redefine most notions of population protocols in order to make them suitable for algorithmic verification. Moreover, we formally define the general verification problem and some interesting special cases. All these problems are shown to be NP-hard. We next propose some first algorithmic solutions for a natural special case. Finally, we conduct experiments and algorithmic engineering in order to improve our verifiers' running times.

1 Introduction

Pervasive environments of tomorrow will consist of populations of tiny possibly mobile artifacts that will interact with each other. Such systems will play an important role in our everyday life and should be correct, reliable and robust. To achieve these goals, it is necessary to verify the correctness of future systems. Formal specification helps to obtain not only a better (more modular) description, but also a clear understanding and an abstract view of the system [4]. Given the increasing sophistication of algorithms for pervasive systems and the difficulty of modifying an algorithm once the network is deployed, there is a clear need to use formal methods to validate system performance and functionality prior to implementing such algorithms [20]. Formal analysis requires the use of models, trusted to behave like a real system. It is therefore critical to find the correct abstraction layer for the models and to verify the models.

Model checking is an exhaustive state space exploration technique that is used to validate formally specified system requirements with respect to a formal system description [14]. Such a system is verified for a fixed configuration; so, in most cases, no general system correctness can be obtained. Using some high-level formal modelling language, automatically an underlying state space can be derived, be it implicitly or symbolically. The system requirements are

⋆ This work has been partially supported by the ICT Programmes of the EU under contracts number ICT-2008-215270 (FRONTS) and ICT-2010-257245 (VITRO).

specified using some logical language, like LTL, CTL or extensions thereof [19]. Well-known and widely applied model checking tools are SPIN [18], Uppaal [6] (for timed systems), and PRISM [17] (for probabilistic systems). The system specification language can, e.g., be based on process algebra, automata or Petri nets. However, model checking suffers from the so-called state explosion problem, meaning that the state space of a specified system grows exponentially with respect to its number of components. The main challenge for model checking lies in modelling large-scale dynamic systems.

Towards providing a concrete and realistic model for future sensor networks, Angluin *et al.* [1] introduced the notion of a computation by a population protocol. In their model, individual agents are extremely limited and can be represented as finite-state machines. The computation is carried out by a collection of agents, each of which receives a piece of the input. Information can be exchanged between two agents whenever they come sufficiently close to each other. The goal is to ensure that every agent can eventually output the value that is to be computed. The critical assumption that diversifies the population protocol model from traditional distributed systems is that the protocol descriptions are independent of the population size, which is known as the *uniformity* property of population protocols. Moreover, population protocols are *anonymous* since there is no room in the state of an agent to store a unique identifier. See also [11,10,16,5,13,7] for population protocol relevant literature. For the interested reader, [3,12] constitute introductions to the area.

In this work, we provide a tool for computer-aided *verification* of population protocols. Our tool can detect errors in the design that are not so easily found using emulation or testing, and they can be used to establish the correctness of the design. A very interesting property of population protocols is protocol composition; one may reduce a protocol into two (or more) protocols of reduced state space that maintain the same correctness and efficiency properties.

Section 2 provides all necessary definitions. In particular, several population protocols' definitions are modified in order to become suitable for algorithmic verification. Then the verification problems that we study throughout this work are formally defined. In Section 3, we prove that all verification problems under consideration are NP-hard. In Section 4, we focus on a particular special case of the general population protocol verification problem, called $BPVER$, in which the population size on which the protocol runs is provided as part of the verifier's input. In particular, we devise three verifiers, two non-complete and one complete. The complete one is slower but provably guarantees to always provide the correct answer. We have implemented our verifiers in C++ by building a new tool named *bp-ver*. To the best of our knowledge, this is the first verification tool for population protocols. In Section 5, we conduct experiments concerning our verifiers' running times. It turns out that constructing the transition graph is a dominating factor. We then improve our verifiers by building all the reachable subgraphs of the transition graph one after the other and not all at once. In this manner, the running time is greatly improved most of the time and the new construction is easily parallelizable.

2 Basic Definitions

2.1 Population Protocols

A *Population Protocol* (PP) $\mathcal{A}$ is a 6-tuple (X, Y, Q, I, O, δ), where X, Y, and Q are all finite sets and X is the *input alphabet*, Y is the *output alphabet*, Q is the set of *states*, $I : X \to Q$ is the *input function*, $O : Q \to Y$ is the *output function*, and $\delta : Q \times Q \to Q \times Q$ is the *transition function.* If $\delta(q_i, q_j) = (q_l, q_t)$, then when an agent in state q_i interacts as the *initiator* with an agent in state q_j (which is the *responder* in this interaction) they update their states deterministically to $\delta_1(q_i, q_j) = q_l$ and $\delta_2(q_i, q_j) = q_t$, respectively. δ can also be treated as a relation $\Delta \subseteq Q^4$, defined as $(q_i, q_j, q_l, q_t) \in \Delta$ iff $\delta(q_i, q_j) = (q_l, q_t)$.

A population protocol runs on the nodes (also called *agents*) of a *communication graph* $G = (V, E)$. In this work, we always assume that the communication graph is a complete digraph, without self-loops and multiple edges (this corresponds to the *basic* population protocol model [1]). We denote by G^k the complete communication graph of k nodes.

Let $k \equiv |V|$ denote the *population size.* An *input assignment* x is a mapping from $V = [k]$ to X (where $[l]$, for $l \in \mathbb{Z}^{\geq 1}$, denotes the set $\{1, \ldots, l\}$), assigning an input symbol to each agent of the population. Since the communication graph is complete, due to symmetry, we can, equivalently, think of an input assignment as a $|X|$-vector of integers $x = (x_i)_{i \in [|X|]}$, where x_i is nonnegative and equal to the number of agents that receive the symbol $\sigma_i \in X$, assuming an ordering on the input symbols. We denote by $\mathcal{X}$ the set of all possible input assignments. Note that for all $x \in \mathcal{X}$ it holds that $\sum_{i=1}^{|X|} x_i = k$.

A state $q \in Q$ is called *initial* if $I(\sigma) = q$ for some $\sigma \in X$. A *configuration* c is a mapping from $[k]$ to Q, so, again, it is a $|Q|$-vector of nonnegative integers $c = (c_i)_{i \in [|Q|]}$ such that $\sum_{i=1}^{|Q|} c_i = k$ holds. Each input assignment corresponds to an *initial configuration* which is indicated by the input function I. In particular, input assignment x corresponds to the initial configuration $c(x) = (c_i(x))_{i \in [|Q|]}$, where $c_i(x)$ is equal to the number of agents that get some input symbols σ_j for which $I(\sigma_j) = q_i$ (q_i is the ith state in Q if we assume the existence of an ordering on the set of states Q). More formally, $c_i(x) = \sum_{j : I(\sigma_j) = q_i} x_j$ for all $i \in [|Q|]$. By extending I to a mapping from input assignments to configurations we can write $I(x) = c$ to denote that c is the initial configuration corresponding to input assignment x. Let $\mathcal{C} = \{(c_i)_{i \in [|Q|]} \mid c_i \in \mathbb{Z}^{\geq 0}$ and $\sum_{i=1}^{|Q|} c_i = k\}$ denote the set of all possible configurations given the population protocol $\mathcal{A}$ and G^k. Moreover, let $C_I = \{c \in \mathcal{C} \mid I(x) = c$ for some $x \in \mathcal{X}\}$ denote the set of all possible initial configurations. Any $r \in \Delta$ has four components which are elements from Q and we denote by r_i, where $i \in [4]$, the i-th component (i.e. state) of r. $r \in Q^4$ belongs to Δ iff $\delta(r_1, r_2) = (r_3, r_4)$. We say that a configuration c can go in one step to a configuration c' via transition $r \in \Delta$, and write $c \xrightarrow{r} c'$, if

- $c_i \geq r_{1,2}(i)$, for all $i \in [|Q|]$ for which $q_i \in \{r_1, r_2\}$,
- $c'_i = c_i - r_{1,2}(i) + r_{3,4}(i)$, for all $i \in [|Q|]$ for which $q_i \in \{r_1, r_2, r_3, r_4\}$, and
- $c'_j = c_j$, for all $j \in [|Q|]$ for which $q_j \in Q - \{r_1, r_2, r_3, r_4\}$,

where $r_{l,t}(i)$ denotes the number of times state q_i appears in (r_l, r_t). Moreover, we say that a configuration c can go in one step to a configuration c', and write $c \rightarrow c'$ if $c \xrightarrow{r} c'$ for some $r \in \Delta$. We say that a configuration c' is reachable from a configuration c, denoted $c \xrightarrow{*} c'$ if there is a sequence of configurations $c = c^0, c^1, \ldots, c^t = c'$, such that $c^i \rightarrow c^{i+1}$ for all i, $0 \leq i < t$, where c^i denotes the $(i+1)$th configuration of an execution (and not the ith component of configuration c which is denoted c_i). An *execution* is a finite or infinite sequence of configurations $c^0, c^1, \ldots$, so that $c^i \rightarrow c^{i+1}$. An execution is *fair* if for all configurations c, c' such that $c \rightarrow c'$, if c appears infinitely often then so does c'. A *computation* is an infinite fair execution. A predicate $p : \mathcal{X} \rightarrow \{0, 1\}$ is said to be *stably computable* by a PP $\mathcal{A}$ if, for any input assignment x, any computation of $\mathcal{A}$ contains an *output stable configuration* in which all agents output $p(x)$. A configuration c is called *output stable* if $O(c) = O(c')$, for all c' reachable from c (where O, here, is an extended version of the output function from configurations to output assignments in Y^k). We denote by $C_F = \{c \in \mathcal{C} \mid c \rightarrow c' \Rightarrow c' = c\}$ the set of all *final configurations*. We can further extend the output function O to a mapping from configurations to $\{-1, 0, 1\}$, defined as $O(c) = 0$ if $O(c(u)) = 0$ for all $u \in V$, $O(c) = 1$ if $O(c(u)) = 1$ for all $u \in V$, and $O(c) = -1$ if $\exists u, v \in V$ s.t. $O(c(u)) \neq O(c(v))$.

It is known [1,2] that a predicate is stably computable by the PP model iff it can be defined as a first-order logical formula in *Presburger arithmetic*. Let ϕ be such a formula. There exists some PP that stably computes ϕ, thus ϕ constitutes, in fact, the specifications of that protocol. For example, consider the formula $\phi = (N_a \geq 2N_b)$. ϕ partitions the set of all input assignments, $\mathcal{X}$, to those input assignments that satisfy the predicate (that is, the number of as assigned is at least two times the number of bs assigned) and to those that do not. Moreover, ϕ can be further extended to a mapping from C_I to $\{-1, 0, 1\}$. In this case, ϕ is defined as $\phi(c) = 0$ if $\phi(x) = 0$ for all $x \in I^{-1}(c)$, $\phi(c) = 1$ if $\phi(x) = 1$ for all $x \in I^{-1}(c)$, and $\phi(c) = -1$ if $\exists x, x' \in I^{-1}(c)$ s.t. $\phi(x) \neq \phi(x')$, where $I^{-1}(c)$ denotes the set of all $x \in \mathcal{X}$ for which $I(x) = c$ holds (the *preimage* of c).

We now define the *transition graph*, which is similar to that defined in [1], except for the fact that it contains only those configurations that are reachable from some initial configuration in C_I. Specifically, given a population protocol $\mathcal{A}$ and an integer $k \geq 2$ we can define the transition graph of the pair $(\mathcal{A}, k)$ as $G_{\mathcal{A},k} = (C_r, E_r)$, where the node set $C_r = C_I \cup \{c \in \mathcal{C} \mid c' \xrightarrow{*} c \text{ for some } c' \in C_I\}$ of G_r (we use G_r as a shorthand of $G_{\mathcal{A},k}$) is the subset of $\mathcal{C}$ containing all initial configurations and all configurations that are reachable from some initial one, and the edge (or arc) set $E_r = \{(c, c') \mid c, c' \in C_r \text{ and } c \rightarrow c'\}$ of G_r contains a directed edge (c, c') for any two (not necessarily distinct) configurations c and c' of C_r for which it holds that c can go in one step to c'. Note that G_r is a directed (weakly) connected graph with possible self-loops. It was shown in [1] that, given a computation Ξ, the configurations that appear infinitely often in Ξ form a *final strongly connected component* of G_r. We denote by S the collection of all strongly connected components of G_r. Note that each $B \in S$ is simply a set of configurations. Moreover, given $B, B' \in S$ we say that B can go in one step to B', and write $B \to B'$, if $c \to c'$ for $c \in B$ and $c' \in B'$. $B \xrightarrow{*} B'$ is defined as in the case of configurations. We denote by $I_S = \{B \in S \mid \text{such that } B \cap C_I \neq \emptyset\}$ those components that contain at least one initial configuration, and by $F_S = \{B \in S \mid \text{such that } B \to B' \Rightarrow B' = B\}$ the final ones. We can now extend ϕ to a mapping from I_S to $\{-1, 0, 1\}$ defined as $\phi(B) = 0$ if $\phi(c) = 0$ for all $c \in B \cap C_I$, $\phi(B) = 1$ if $\phi(c) = 1$ for all $c \in B \cap C_I$, and $\phi(B) = -1$ if $\exists c, c' \in B \cap C_I$ s.t. $\phi(c) \neq \phi(c')$, and O to a mapping from F_S to $\{-1, 0, 1\}$ defined as $O(B) = 0$ if $O(c) = 0$ for all $c \in B$, $O(B) = 1$ if $O(c) = 1$ for all $c \in B$, and $O(B) = -1$ otherwise.

2.2 Problems' Definitions

We begin by defining the most interesting and natural version of the problem of algorithmically verifying basic population protocols. We call it *GBPVER* ('G' standing for "General', 'B' for "Basic", and 'P' for "Predicate") and its complement $\overline{GBPVER}$ is defined as follows:

Problem 1 ($\overline{GBPVER}$). Given a population protocol $\mathcal{A}$ for the basic model whose output alphabet $Y_{\mathcal{A}}$ is binary (i.e. $Y_{\mathcal{A}} = \{0, 1\}$) and a first-order logical formula ϕ in Presburger arithmetic representing the specifications of $\mathcal{A}$, determine whether there exists some integer $k \geq 2$ and some legal input assignment x for the complete communication graph of k nodes, G^k, for which not all computations of $\mathcal{A}$ on G^k beginning from the initial configuration corresponding to x stabilize to the correct output w.r.t. ϕ.

A special case of *GBPVER* is *BPVER* (its non-general version as revealed by the missing 'G'), and is defined as follows.

Problem 2 (BPVER). Given a population protocol $\mathcal{A}$ for the basic model whose output alphabet $Y_{\mathcal{A}}$ is binary (i.e. $Y_{\mathcal{A}} = \{0, 1\}$), a first-order logical formula ϕ in Presburger arithmetic representing the specifications of $\mathcal{A}$, and an integer $k \geq 2$ (in binary) determine whether $\mathcal{A}$ conforms to its specifications on G^k.

"Conforms to ϕ" here means that for any legal input assignment x, which is a $|X_{\mathcal{A}}|$-vector with nonnegative integer entries that sum up to k, and any computation beginning from the initial configuration corresponding to x on G^k, the population stabilizes to a configuration in which all agents output the value $\phi(x) \in \{0, 1\}$.

Problem 3 (BBPVER). BBPVER (the additional 'B' is from "Binary input alphabet") is *BPVER* with $\mathcal{A}$'s input alphabet restricted to $\{0, 1\}$.

3 Hardness Results

Theorem 1. *BPVER is coNP-hard.*

Proof. We shall present a polynomial-time reduction from $HAMPATH = \{\langle D, s, t\rangle \mid D$ is a directed graph with a Hamiltonian path from s to t $\}$ to $\overline{BPVER}$. In other words, we will present a procedure that given an instance $\langle D, s, t\rangle$ of $HAMPATH$ returns in polynomial time an instance $\langle \mathcal{A}, \phi, k\rangle$ of $\overline{BPVER}$, such that $\langle D, s, t\rangle \in HAMPATH$ iff $\langle \mathcal{A}, \phi, k\rangle \in \overline{BPVER}$. If there is a hamiltonian path from s to t in D we will return a population protocol $\mathcal{A}$ that for some computation on the complete graph of k nodes fails to conform to its specification ϕ, and if there is no such path all computations will conform to ϕ.

We assume that all nodes in $V(D) - \{s, t\}$ are named $q_1, \ldots, q_{n-2}$, where n denotes the number of nodes of D (be careful: n does not denote the size of the population, but the number of nodes of the graph D in $HAMPATH$'s instance). We now construct the protocol $\mathcal{A} = (X, Y, Q, I, O, \delta)$. The output alphabet Y is $\{0, 1\}$ by definition. The input alphabet X is $E(D) - (\{(\cdot, s)\} \cup \{t, \cdot\})$, that is, consists of all edges of D except for those leading into s and those going out of t. The set of states Q is equal to $X \cup T \cup \{r\}$, where $T = \{(s, q_i, q_j, l) \mid 1 \leq i, j \leq n-2$ and $1 \leq l \leq n-1\}$ and its usefulness will be explained later. r can be thought of as being the "reject" state, since we will define it to be the only state giving the output value 0. Notice that $|Q| = \mathcal{O}(n^3)$. The input function $I : X \to Q$ is defined as $I(x) = x$, for all $x \in X$, and for the output function $O : Q \to \{0, 1\}$ we have $O(r) = 0$ and $O(q) = 1$ for all $q \in Q - \{r\}$. That is, all input symbols are mapped to themselves, while all states are mapped to the output value 1, except for r which is the only state giving 0 as output. Thinking of the transition function δ as a transition matrix Δ it is easy to see that Δ is a $|Q| \times |Q|$ matrix whose entries are elements from $Q \times Q$. Each entry $\Delta_{q,q'}$ corresponds to the rhs of a rule $(q, q') \to (z, z')$ in δ. Clearly, Δ consists of $\mathcal{O}(n^6)$ entries, which is again polynomial in n.

We shall postpone for a while the definition of Δ to first define the remaining parameters ϕ and k of $\overline{BPVER}$'s instance. We define formula ϕ to be a trivial first-order Presburger arithmetic logical formula that is always false. For example, in the natural nontrivial case where $X \neq \emptyset$ (that is, D has at least one edge that is not leading into s and not going out of t) we can pick any $x \in X$ and set $\phi = (N_x < 0)$ which, for N_x denoting the number of xs appearing in the input assignment, is obviously always false. It is useful to notice that the only configuration that gives the correct output w.r.t. ϕ is the one in which all agents are in state r. ϕ being always false means that in a correct protocol all computations must stabilize to the all-zero output, and r is the only state giving output 0. On the other hand for $\mathcal{A}$ not to be correct w.r.t. ϕ it suffices to show that there exists some computation in which r cannot appear. Moreover, we set k equal to $n - 1$, that is, the communication graph on which $\mathcal{A}$'s correctness has to be checked by the verifier is the complete digraph of $n - 1$ nodes (or, equivalently, agents).

To complete the reduction, it remains to construct the transition function δ:

- $(r, \cdot) \to (r, r)$ and $(\cdot, r) \to (r, r)$ (so r is a propagating state, meaning that once it appears it eventually becomes the state of every agent in the population)

- $((q_i, q_j), (q_i, q_j)) \rightarrow (r, r)$ (if two agents get the same edge of D then the protocol rejects)
- $((q_i, q_j), (q_i, q_l)) \rightarrow (r, r)$ (if two agents get edges of D with adjacent tails then the protocol rejects)
- $((q_j, q_i), (q_l, q_i)) \rightarrow (r, r)$ (if two agents get edges of D with adjacent heads then the protocol rejects - it also holds if one of q_j and q_l is s)
- $((q_i, t), (q_j, t) \rightarrow (r, r)$ (the latter also holds for the sink t)
- $((s, \cdots), (s, \cdots)) \rightarrow (r, r)$ (if two agents have both s as the first component of their states then the protocol rejects)
- $((s, q_i), (q_i, q_j)) \rightarrow ((s, q_i, q_j, 2), (q_i, q_j))$ (when s meets an agent v that contains a successor edge it keeps q_j to remember the head of v's successor edge and releases a counter set to 2 - it counts the number of edges encountered so far on the path trying to reach t from s)
- $((s, q_i, q_j, i), (q_j, q_l)) \rightarrow ((s, q_i, q_l, i+1), (q_j, q_l))$, for $i < n-2$
- $((s, q_i, q_j, i), (q_j, t)) \rightarrow (r, r)$, for $i < n-2$ (the protocol rejects if s is connected to t through a directed path with less than $n-1$ edges)
- All the transitions not appearing above are identity rules (i.e. they do nothing)

Now we prove that the above, obviously polynomial-time, construction is in fact the desired reduction. If D contains some hamiltonian path from s to t, then the $n-1$ edges of that path form a possible input assignment to protocol $\mathcal{A}$ (since its input symbols are the edges and the population consists of $n-1$ agents). When $\mathcal{A}$ gets that input it cannot reject (r cannot appear) for the following reasons:

- no two agents get the same edge of D
- no two agents get edges of D with adjacent tails
- no two agents get edges of D with adjacent heads
- only one $(s, \cdots)$ exists
- s cannot count less than $n-1$ edges from itself to t

So, when $\mathcal{A}$ gets the input alluded to above, it cannot reach state r, thus, it cannot reject, which implies that $\mathcal{A}$ for that input always stabilizes to the wrong output w.r.t. ϕ (which always requires the "reject" output) when runs on the G^{n-1}. So, in this case $\langle \mathcal{A}, \phi, k \rangle$ consists of a protocol $\mathcal{A}$ that, when runs on G^k, where $k = n-1$, for a specific input it does not conform to its specifications as described by ϕ, so clearly it belongs to $\overline{BPVER}$.

For the other direction, if $\langle \mathcal{A}, \phi, k \rangle \in \overline{BPVER}$ then obviously there exists some computation of $\mathcal{A}$ on the complete graph of $k = n-1$ nodes in which r does not appear at all (if it had appeared once then, due to fairness, the population would have stabilized to the all-r configuration, resulting to a computation conforming to ϕ). It is helpful to keep in mind that most arguments here hold because of the fairness condition. Since r cannot appear, every agent (of the $n-1$ in total) must have been assigned a different edge of D. Moreover, no two of them contain edges with common tails or common heads in D. Note that there is only one agent with state $(s, \cdots)$ because if there were two of them they would have rejected when interacted with each other, and if no $(s, \cdots)$ appeared then two agents would have edges with common tails because there are $n-1$ edges for $n-2$ candidate initiating points (we have not allowed t to be an initiating point) and the pigeonhole principle applies (and by symmetric arguments only one with state $(\cdots, t)$). So, in the induced graph formed by the edges that have been assigned to the agents, s has outdegree 1 and indegree 0, t has indegree 1 and outdegree 0 and all remaining nodes have indegree at most 1 and outdegree at most 1. This implies that all nodes except for s and t must have indegree equal to 1 and outdegree equal to 1. If, for example, some node had indegree 0, then the total indegree could not have been $n-1$ because $n-3$ other nodes have indegree at most 1, t has indegree 1, and s has 0 (the same holds for outdegrees). Additionally, there is some path initiating from s and ending to t. This holds because the path initiating from s (s has outdegree 1) cannot fold upon itself (this would result in a node with indegree greater than 1) and cannot end to any other node different from t because this would result to some node other than t with outdegree equal to 0. Finally, that path has at least $n-1$ edges (in fact, precisely $n-1$ edges), since if it had less the protocol would have rejected. Thus, it must be clear after the above discussion that in this case there must have been a hamiltonian path from s to t in D, implying that $\langle D, s, t\rangle \in HAMPATH$. □

Let us denote by $BPVER'$ the special case of $BPVER$ in which the protocol size is at least the size k of the communication graph. Clearly, the proof of Theorem 1 establishes that this problem is also coNP-hard. The following theorem captures the hardness of the other two main problems.

Theorem 2. *$BBPVER$ and $GBPVER$ are coNP-hard.*

Proof. The first statement can be proved by arguments similar to those used in the proof of Theorem 1. We will prove the second statement by presenting a polynomial-time reduction from $\overline{BPVER'}$ to $\overline{GBPVER}$. Keep in mind that the input to the machine computing the reduction is $\langle \mathcal{A}, \phi, k\rangle$. Let $X_{\mathcal{A}}$ be the input alphabet of $\mathcal{A}$. Clearly, $\phi'' = \neg(\sum_{x\in X_{\mathcal{A}}} N_x = k)$ is a semilinear predicate if k is treated as a constant (N_x denotes the number of agents with input x). Thus, there exists a population protocol $\mathcal{A}''$ for the basic model that stably computes ϕ''. The population protocol $\mathcal{A}''$ can be constructed efficiently. Its input alphabet $X_{\mathcal{A}''}$ is equal to $X_{\mathcal{A}}$. The construction of the protocol can be found in [1] (in fact they present there a more general protocol for any linear combination of variables corresponding to a semilinear predicate). When the number of nodes of the communication graph is equal to k, $\mathcal{A}''$ always stabilizes to the all-zero output (all agents output the value 0) and when it is not equal to k, then $\mathcal{A}''$ always stabilizes to the all-one output.

We want to construct an instance $\langle \mathcal{A}', \phi'\rangle$ of $\overline{GBPVER}$. We set $\phi' = \phi \vee \phi''$. Moreover, $\mathcal{A}'$ is constructed to be the composition of $\mathcal{A}$ and $\mathcal{A}''$. Obviously, $Q_{\mathcal{A}'} = Q_{\mathcal{A}} \times Q_{\mathcal{A}''}$. We define its output to be the union of its components' outputs, that is, $O(q_{\mathcal{A}}, q_{\mathcal{A}''}) = 1$ iff at least one of $O(q_{\mathcal{A}})$ and $O(q_{\mathcal{A}''})$ is equal to 1. It is easy to see that the above reduction can be computed in polynomial time.

We first prove that if $\langle \mathcal{A}, \phi, k\rangle \in \overline{BPVER'}$ then $\langle \mathcal{A}', \phi'\rangle \in \overline{GBPVER}$. When $\mathcal{A}'$ runs on the complete graph of k nodes, the components of its states corresponding to $\mathcal{A}''$ stabilize to the all-zero output, independently of the initial configuration. Clearly, $\mathcal{A}'$ in this case outputs whatever $\mathcal{A}$ outputs. Moreover, for this communication graph, ϕ' is true iff ϕ is true (because $\phi'' = \neg(\sum_{x \in X_{\mathcal{A}}} N_x = k)$ is false, and $\phi' = \phi \vee \phi''$). But there exists some input for which $\mathcal{A}$ does not give the correct output with respect to ϕ (e.g. ϕ is true for some input but $\mathcal{A}$ for some computation does not stabilize to the all-one output). Since ϕ' expects the same output as ϕ and $\mathcal{A}'$ gives the same output as $\mathcal{A}$ we conclude that there exists some erroneous computation of $\mathcal{A}'$ w.r.t. ϕ', and the first direction has been proven.

Now, for the other direction, assume that $\langle \mathcal{A}', \phi' \rangle \in \overline{GBPVER}$. For any communication graph having a number of nodes not equal to k, ϕ' is true and $\mathcal{A}'$ always stabilizes to the all-one output because of the $\mathcal{A}''$ component. This means that the erroneous computation of $\mathcal{A}'$ happens on the G^k. But for that graph, ϕ'' is always false and $\mathcal{A}''$ always stabilizes its corresponding component to the all-zero output. Now ϕ' is true iff ϕ is true and $\mathcal{A}'$ outputs whatever $\mathcal{A}$ outputs. But there exists some input and a computation for which $\mathcal{A}'$ does not stabilize to a configuration in which all agents give the output value that ϕ' requires which implies that $\mathcal{A}$ does not stabilize to a configuration in which all agents give the output value required by ϕ. Since the latter holds for G^k, the theorem follows. □

4 Algorithmic Solutions for *BPVER*

Our algorithms are search algorithms on the transition graph G_r. The general idea is that a protocol $\mathcal{A}$ does not conform to its specifications ϕ on k agents, if one of the following criteria is satisfied:

1. $\phi(c) = -1$ for some $c \in C_I$.
2. $\exists c, c' \in C_I$ such that $c \xrightarrow{*} c'$ and $\phi(c) \neq \phi(c')$.
3. $\exists c \in C_I$ and $c' \in C_F$ such that $c \xrightarrow{*} c'$ and $O(c') = -1$.
4. $\exists c \in C_I$ and $c' \in C_F$ such that $c \xrightarrow{*} c'$ and $\phi(c) \neq O(c')$.
5. $\exists B' \in F_S$ such that $O(B') = -1$.
6. $\exists B \in I_S$ and $B' \in F_S$ such that $B \xrightarrow{*} B'$ and $\phi(B) \neq O(B')$ (possibly $B = B'$).

Note that any algorithm that correctly checks some of the above criteria is a possibly *non-complete verifier*. Such a verifier guarantees that it can discover an error of a specific kind, thus, we can always trust its “reject” answer. On the other hand, an “accept” answer is a weaker guarantee, in the sense that it only informs that the protocol does not have some error of this specific kind. Of course, it is possible that the protocol has other errors, violating criteria that are indetectable by this verifier. However, this is a first sign of $BPVER$’s *parallelizability*.

Theorem 3. *Any algorithm checking criteria 1, 5, and 6 decides BPVER.*

Proof. Let $\mathcal{M}$ be such an algorithm. Let $\langle \mathcal{A}, \phi, k \rangle \in BPVER$. This implies that, for any input assignment x, any computation of $\mathcal{A}$ beginning from the initial configuration $I(x)$ reaches a final strongly connected component of the transition graph (consisting of those configurations that appear infinitely often in the computation, see Lemma 1 in [1] and Theorem 1 in [9]), in which all configurations have all agents giving the output $\phi(x)$. $\mathcal{M}$ correctly accepts in this case because none of the criteria 1, 5, and 6 is satisfied for the following reasons:

– $\phi(c) = -1$ for some $c \in C_I$: if it were, then there would be two input assignments x and x', such that $\phi(x) \neq \phi(x')$ both mapped to the same configuration c. But then any stable computation beginning from c would give the wrong output for at least one of x and x'. This would imply that $\mathcal{A}$ does not stably compute k.
– $\exists B' \in F_S$ such that $O(B') = -1$: B contains at least one initial configuration, e.g. c. Since B reaches B' there is a computation leading from c to B'. But B' is unstable.
– $\exists B \in I_S$ and $B' \in F_S$ such that $B \xrightarrow{*} B'$ and $\phi(B) \neq O(B')$ (possibly $B = B'$): Here, all initial configurations in B expect a different output from that given by the configurations in B'. Again, since $\mathcal{A}$ stably computes ϕ this cannot be the case.

Let, now $\langle \mathcal{A}, \phi, k \rangle \notin BPVER$. By definition, this implies that some initial configuration c is assigned two input assignments that expect different outputs or that each initial configuration c only expects a single output $\phi(c)$ but there exists some computation beginning from c that either does not stabilize or stabilizes to some output $y \neq \phi(c)$. We examine these cases mainly from a transition-graph perspective.

1. $\exists c \in C_I$ *s.t.* $\phi(c) = -1$: This is clearly handled by criterion 1, and $\mathcal{M}$ rejects.
2. *The computation does not stabilize*: Although it doesn't, c again either belongs to or reaches some final strongly connected component B' (Lemma 1 in [1]). But in this case, the component B' contains at least two configurations that give different outputs, or one configuration in which not all agents give the same output, which is written in our notation as $O(B') = -1$. In both cases, it holds that $B' \in F_S$, since it is final and reachable from the initial configuration c, thus, criterion 5 is clearly satisfied and $\mathcal{M}$ rejects.
3. *The computation stabilizes to* $y \neq \phi(c)$: In this case, a final strongly connected component B' is reached in which all configurations have all agents giving the output y. In our notation $O(B') = y \neq \phi(c)$. c either belongs to B' or belongs to some initial component B that reaches B'. In the former case $\phi(B') \neq O(B')$ and in the latter $\phi(B) \neq O(B')$. In both cases, criterion 6 is satisfied and $\mathcal{M}$ rejects.

□

Algorithm 1. *SinkVER*

Input: A PP $\mathcal{A}$, a Presburger arithmetic formula ϕ, and an integer $k \geq 2$.
Output: ACCEPT if $\mathcal{A}$ is correct w.r.t. its specifications and the criteria 1, 2, 3, and 4 on G^k and REJECT otherwise.

```
1: $C_I \leftarrow FindC_I(\mathcal{A}, k)$
2: if there exists $c \in C_I$ such that $\phi(c) = -1$ then
3:     return REJECT // Criterion 1 satisfied
4: end if
5: $G_r \leftarrow ConG_r(\mathcal{A}, k)$
6: for all $c \in C_I$ do
7:     Collect all $c'$ reachable from $c$ in $G_r$ by BFS or DFS.
8:     while searching do
9:         if one $c'$ is found such that $c' \in C_F$ and $(O(c') = -1$ or $\phi(c) \neq O(c'))$
           then
10:            return REJECT // Criterion 3 or 4 satisfied
11:        end if
12:        if one $c'$ is found such that $c' \in C_I$ and $\phi(c) \neq \phi(c')$ then
13:            return REJECT // Criterion 2 satisfied
14:        end if
15:    end while
16: end for
17: return ACCEPT // Tests for criteria 1,2,3, and 4 passed
```

4.1 Constructing the Transition Graph

Let $FindC_I(\mathcal{A}, k)$ be a function that given a PP $\mathcal{A}$ and an integer $k \geq 2$ returns the set C_I of all initial configurations. This is not so hard to be implemented. $FindC_I$ simply iterates over the set of all input assignments $\mathcal{X}$ and for each $x \in \mathcal{X}$ computes $I(x)$ and puts it in C_I. Alternatively, computing C_I is equivalent to finding all distributions of indistinguishable objects (agents) into distinguishable slots (initial states), and, thus, can be done by Fenichel's algorithm [15].

The transition graph G_r can be constructed by some procedure, call it $ConG_r$, which is a simple application of searching and, thus, we skip it. It takes as input a population protocol $\mathcal{A}$ and the population size k, and returns the transition graph G_r.

4.2 Non-complete Verifiers

We now present two non-complete verifiers, namely *SinkBFS* and *SinkDFS*, that check all criteria but the last two. Both are presented via procedure *SinkVER* (Algorithm 1) and the order in which configurations of G_r are visited determines whether BFS or DFS is used.

4.3 *SolveBPVER*: A Complete Verifier

We now construct the procedure *SolveBPVER* (Algorithm 2) that checks criteria 1, 5, and 6 (and also 2 for some speedup) presented in the beginning of this section, and, thus, according to Theorem 3, it correctly solves $BPVER$ (i.e. it is a complete verifier). In particular, *SolveBPVER* takes as input a PP $\mathcal{A}$, its specifications ϕ and an integer $k \geq 2$, as outlined in the $BPVER$ problem description, and returns "accept" if the protocol is correct w.r.t. its specifications on G^k and "reject" otherwise.

Algorithm 2. *SolveBPVER*

Input: A PP $\mathcal{A}$, a Presburger arithmetic formula ϕ, and an integer $k \geq 2$.
Output: ACCEPT if the protocol is correct w.r.t. its specifications on G^k and REJECT otherwise.

```
1: $C_I \leftarrow FindC_I(\mathcal{A}, k)$
2: if there exists $c \in C_I$ such that $\phi(c) = -1$ then
3:     return REJECT
4: end if
5: $G_r \leftarrow ConG_r(\mathcal{A}, k)$
6: Run one of Tarjan's or Gabow's algorithms to compute the collection $S$ of all
   strongly connected components of the transition graph $G_r$.
7: Compute the dag $D = (S, A)$, where $(B, B') \in A$ (where $B \neq B'$) if and only if
   $B \rightarrow B'$.
8: Compute the collection $I_S \subseteq S$ of all connected components $B \in S$ that contain
   some initial configuration $c \in C_I$ and the collection $F_S \subseteq S$ of all connected
   components $B \in S$ that have no outgoing edges in $A$, that is, all final strongly
   connected components of $G_r$.
9: for all $B \in F_S$ do
10:     if $O(B) = -1$ then
11:         return REJECT
12:     end if
13:     // Otherwise, all configurations $c \in B$ output the same value $O(B) \in \{0, 1\}$.
14: end for
15: for all $B \in I_S$ do
16:     if there exist initial configurations $c, c' \in B$ such that $\phi(c) \neq \phi(c')$ then
17:         return REJECT
18:     else
19:         // all initial configurations $c \in B$ expect the same output $\phi(B) \in \{0, 1\}$.

20:         Run BFS or DFS from $B$ in $D$ and collect all $B' \in F_S$ s.t. $B \overset{*}{\rightarrow} B'$
            (possibly including $B$ itself).
21:         if there exists some reachable $B' \in F_S$ for which $O(B') \neq \phi(B)$ then
22:             return REJECT
23:         end if
24:     end if
25: end for
26: return ACCEPT
```

The idea is to use Tarjan's [21] or Gabow's (or any other) algorithm for finding the strongly connected components of G_r. In this manner, we obtain a collection

S, where each $B \in S$ is a strongly connected component of G_r, that is, $B \subseteq C_r$. Given S we can easily compress G_r w.r.t. its strongly connected components as follows. The compression of G_r is a dag $D = (S, A)$, where $(B, B') \in A$ if and only if there exist $c \in B$ and $c' \in B'$ such that $c \rightarrow c'$ (that is, iff $B \rightarrow B'$). In words, the node set of D consists of the strongly connected components of G_r and there is a directed edge between two components of D if a configuration of the second component is reachable in one step from a configuration in the first one.

We have implemented our verifiers in C++. We have named our tool *bp-ver* and it can be downloaded from http://ru1.cti.gr/projects/BP-VER. Our implementation makes use of the *boost* graph library. In particular, we exploit boost to store and handle the transition graph and to find its strongly connected components. Boost uses Tarjan's algorithm [21] for the latter. We use Fenichel's algorithm [15] in order to find all possible initial configurations, and our implementation is based on Burkardt's FORTRAN code [8]. Protocols and formulas are stored in separate files and simple, natural syntax is used. Formulas are evaluated with Dijkstra's Shunting-yard algorithm for evaluating expressions.

5 Experiments and Algorithmic Engineering

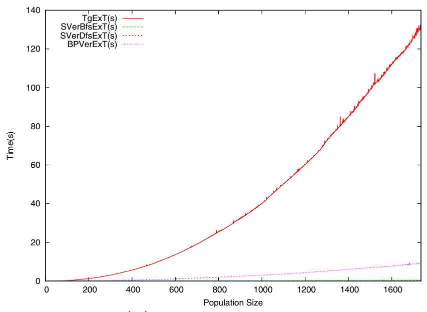
As presented so far, all verifiers first construct the transition graph and then start searching on it, each with its own method, in order to detect some error. Note also that all algorithms halt when the first error is found, otherwise they halt when there is nothing left to search.

Our first suspicion was that the time to construct the transition graph must be a dominating factor in the "reject" case but not in the "accept" case. To see this for the "reject" case imagine that all verifiers find some error near the first initial configuration that they examine. For example, they may reach in a few steps another initial configuration that expects different output. But, even if all of them reject in a few steps, they still will have paid the construction of the whole transition graph first, which is usually huge. On the other hand, this must not be a problem in the "accept" case, because all verifiers have, more or less, to search the whole transition graph before accepting.

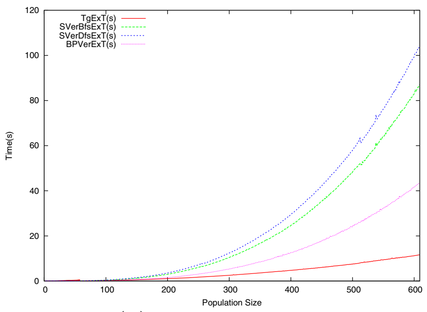
Protocol 3. *flock*$_i$

```
1: // $i$ must be at least 1
2: $X = Y = \{0, 1\}$, $Q = \{q_0, q_1, \dots, q_i\}$,
3: $I(0) = q_0$ and $I(1) = q_1$,
4: $O(q_l) = 0$, for $0 \le l \le i - 1$, and $O(q_i) = 1$,
5: $\delta$:
```

$$\begin{aligned}(q_k, q_j) &\rightarrow (q_{k+j}, q_0), \text{ if } k + j < i \\ &\rightarrow (q_i, q_i), \text{ otherwise.}\end{aligned}$$



(a) “reject” case.



(b) “accept” case.

Fig. 1. (a): Verifiers executed on erroneous version of *flock*$_2$ (see Protocol 3, where the corresponding stably computable predicate is $(N_1 \geq i)$) w.r.t. formula $(N_1 \geq 2)$. The dominating factor is the time needed to construct the transition graph. For all $n \geq 4$ all verifiers found some error. (b): Verifiers executed on the correct *flock*$_2$t w.r.t. formula $(N_1 \geq 2)$. *SinkDFS* and *SinkBFS* verifiers are clearly faster than *SolveBPVER* in this case. For all $n \geq 4$ all verifiers decided that the protocol is correct.

These speculations are confirmed by our first experiments whose findings are presented in Figure 1. In particular, we consider the “flock of birds” protocol that counts whether at least 2 birds in the flock were found infected. The corresponding Presburger formula is $(N_1 \geq 2)$. In Figure 1(a) we introduced a single error to the protocol’s code that all verifiers would detect. Then we counted and plotted the time to construct the transition graph and the execution (CPU) time of all verifiers (until they answer “reject”) for different population sizes. In Figure 1(b) we did the same for the correct version of the protocol. See Protocol 3 for the code of protocol *flock*$_i$.

The good news is that our verifiers’ running times can easily be improved. The idea is to take the initial configurations one after the other and search only in the subgraph of the transition graph that is reachable from the current initial configuration. In this manner, we usually avoid in the “reject” case to construct the whole transition graph. Especially in cases that we are lucky to detect an error in the first subgraph that we ever visit, and if this subgraph happens to be small, the running time is greatly improved.

References

1. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing 18(4), 235–253 (2006); Also in 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 290-299 (2004)
2. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)
3. Aspnes, J., Ruppert, E.: An introduction to population protocols. Bulletin of the European Association for Theoretical Computer Science 93, 98–117 (2007)

4. Bakhshi, R., Bonnet, F., Fokkink, W., Haverkort, B.: Formal analysis techniques for gossiping protocols. ACM SIGOPS Operating Systems Review, Special Issue on Gossip-Based Networking 41(5), 28–36 (2007)
5. Beauquier, J., Clement, J., Messika, S., Rosaz, L., Rozoy, B.: Self-stabilizing counting in mobile sensor networks. Technical Report 1470, LRI, Université Paris-Sud 11 (2007)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
7. Bournez, O., Chassaing, P., Cohen, J., Gerin, L., Koegler, X.: On the convergence of population protocols when population goes to infinity. Applied Mathematics and Computation 215(4), 1340–1350 (2009)
8. `http://people.sc.fsu.edu/~burkardt/f_src/combo/combo.f90`
9. Chatzigiannakis, I., Dolev, S., Fekete, S.P., Michail, O., Spirakis, P.G.: Not all fair probabilistic schedulers are equivalent. In: 13th International Conference on Principles of DIstributed Systems (OPODIS), pp. 33–47 (2009)
10. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Brief Announcement: Decidable graph languages by mediated population protocols. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 239–240. Springer, Heidelberg (2009)
11. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Mediated population protocols. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 363–374. Springer, Heidelberg (2009)
12. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Recent advances in population protocols. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 56–76. Springer, Heidelberg (2009)
13. Chatzigiannakis, I., Spirakis, P.G.: The dynamics of probabilistic population protocols. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 498–499. Springer, Heidelberg (2008)
14. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (2000)
15. Fenichel, R.: Distribution of indistinguishable objects into distinguishable slots. Communications of the ACM 11(6), 430 (1968)
16. Guerraoui, R., Ruppert, E.: Names trump malice: Tiny mobile agents can tolerate byzantine failures. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 484–495. Springer, Heidelberg (2009)
17. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
18. Holzmann, G.: The Spin model checker, primer and reference manual. Addison-Wesley, Reading (2003)
19. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge University Press, Cambridge (2004)
20. Olveczky, P.C., Thorvaldsen, S.: Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium International, p. 157 (2006)
21. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)

Energy Management for Time-Critical Energy Harvesting Wireless Sensor Networks

Bo Zhang, Robert Simon, and Hakan Aydin

Department of Computer Science
George Mason University, Fairfax, VA 22030
{bzhang3,simon,aydin}@cs.gmu.edu

Abstract. As Cyber-Physical Systems (CPSs) evolve they will be increasingly relied on to support time-critical monitoring and control activities. Further, many CPSs that utilize Wireless Sensor Networking (WSN) technologies require energy harvesting methods to extend their lifetimes. For this important system class, there are currently no effective approaches that balance system lifetime with system performance under both normal and emergency situations. To address this problem, we present a set of Harvesting Aware Speed Selection (HASS) algorithms. We use an epoch-based architecture to dynamically adjust CPU frequencies and radio transmit speeds of sensor nodes, hence regulate their power consumption. The objective is to maximize the minimum energy reserve over any node in the network, while meeting application's end-to-end deadlines. Through this objective we ensures highly resilient performance under emergency or fault-driven situation. Through extensive simulations, we show that our algorithms yield significantly higher energy reserves than the approaches without speed and power control.

1 Introduction

There is an increasing need to effectively support Wireless Sensor Network applications that have significant data collection and processing requirements. Examples range from Wireless Network Video Systems for surveillance [19] to Cyber-Physical Systems such as smart power grid using 802.15.4/Zigbee technology [14]. These types of systems often have strict timing and performance specifications. For instance, smart power grid systems need to provide real-time pricing information, while water distribution systems need to instantly react to a contamination. Further, many of these self$-^*$, unattended and deeply-embedded systems will be expected to last for several decades, and therefore must carefully manage available energy resources. The challenge faced by system designers is to balance the performance and system availability requirements with energy management policies that can maximize system lifetime.

One approach for maximizing system lifetime is to use energy harvesting [6]. By harvesting energy from environmental sources such as solar, wind or water flow, WSN nodes potentially have perpetual energy supply. However, given the large energy demands of the computational and communication intensive

WSN applications, and limited availability of environmental power, perpetual operation of WSN nodes cannot be realized without deliberate energy management. This problem is exacerbated if the application has unpredictable spikes in workload demand, such as a water distribution system reacting to a biological contamination. *The focus of this paper is a coordinated energy management policy for time-critical WSN applications that use energy harvesting and that must maintain required performance under emergency or fault-driven situations.*

Our approach is to make combined use of two energy saving techniques, Dynamic Voltage Scaling (DVS) [2] and Dynamic Modulation Scaling (DMS) [18]. The DVS technique saves computation energy by simultaneously reducing CPU supply voltage and frequency. The DMS technique saves communication energy by reducing radio modulation level and hence transmit speed. To take advantage of these methods we propose a set of *Harvesting Aware Speed Selection* (*HASS*) algorithms that use both DVS and DMS in conjunction with energy harvesting modules. The purpose of the *HASS* approach is to maximize energy reserves while meeting application performance requirements, therefore maximize the system's resilience to emergency situations.

One difficulty in managing energy for these systems is that nodes may have quite different workload demands and available energy sources. This may arise from natural factors such as differences in nodes energy harvesting opportunities, or unbalanced distribution of processing workloads or network traffic. Because of these conflicting design considerations, the *HASS* approach attempts to maximize the minimum energy reserve level over any node in the network, while guaranteeing required system performance. Specifically, *HASS* adjusts CPU processing speed and radio transmit speed, with a goal that the harvesting-rich nodes run faster to allow the harvesting-poor nodes to slow down and save energy, given tight end to end data collection latency constraint. The reduced energy consumption will secure higher energy reserves for the poor nodes.

Our specific contributions are summarized as follows: We first provide a basic architectural description for DVS and DMS nodes that use energy harvesting. We then propose a general network and performance model for time-critical WSN applications. Unlike the majority of existing works in energy harvesting WSN systems which mainly focus on individual nodes, we target a multi-hop sensor network with an end-to-end performance requirement. Next, we show how to formulate the problem of maximizing the minimum energy reserve while maintaining required performance as an optimization problem. We prove that this problem can be solved optimally and efficiently. We also propose and evaluate both centralized and distributed protocols to implement the *HASS* solution. We conducted extensive simulations to evaluate our methods under a variety of processing, communication and performance requirements. Unlike most existing works which assuming solar energy as the environmental sources in their simulations, we propose an experimental methodology to simulate energy harvesting WSN systems utilizing energy harvested from water flow in a water distribution system. Our results show that both centralized and distributed solutions significantly improve the capacity of time-critical WSN systems to deal with emergency situations, in addition to meeting performance requirements.

2 Background and Related Work

The joint use of DVS and DMS in wireless embedded systems has been explored in [7] and [17]. In [7], Kumar et al. addressed a resource allocation problem with the aim of minimizing energy consumption. They assume a system containing a mixed set of computation and communication tasks. In [17], the energy management problem is formulated as a convex optimization problem, which is then addressed through the use of genetic algorithms. In [18], Yu et al. proposed DMS-based approach for a multi-hop WSNs. They assume a data collection application in which a base station periodically collects sensed data from WSN nodes over a tree-based routing structure. In [21], we proposed a joint DVS-DMS energy management approach for individual energy harvesting WSN nodes with a goal of maximizing the minimum energy level over time. Unlike our work, [7], [17], [18] assume battery-powered system and target prolonging system lifetime by reducing energy consumption, without considering the need of ensuring perpetual operation through energy harvesting.

Many existing studies explored the design of energy harvesting WSNs. In [10], Moser et al. proposed the LSA algorithm (Lazy Scheduling Algorithm) for scheduling real-time tasks in the context of energy harvesting. LSA defers task execution and hence energy consumption as late as possible so as to reduce the amount of deadline misses. Liu et al. ([8]) proposed EA-DVFS (Energy-Aware Dynamic Voltage and Frequency Scaling) which improves the energy efficiency of LSA by using DVS. Both LSA and EA-DVFS manage only the CPU energy, while ignoring radio energy. Other related work includes [5] and [12] which aim at balancing energy supply and energy demand in energy harvesting WSN systems. Finally, [6], [11] proposed to maximally utilize harvested energy for maximizing the amount of completed works, and hence system performance. Neither of these works considered maximizing minimum energy level by using joint DVS-DMS techniques.

3 System Architecture

This section describes our architecture for energy harvesting WSN systems supporting time-critical applications. It consists of a basic node and device model, a task-based workload model and energy consumption analysis, and a performance model. This will provide a systematic methodology for modeling and analyzing the performance of this type of systems.

3.1 Device Model

Without loss of generality, we assume that each node has several functional units, including an energy harvester head, an energy storage unit, a DVS capable

CPU, a DMS capable radio, as well as required sensor suites. The harvester head is energy source-specific, such as solar panel or wind generator. The energy storage unit (e.g. rechargable battery or super-capacitor) has a maximum energy capacity of Γ^{max} joules. This unit receives power from the energy harvester, and delivers power to the sensor node. We take the commonly used approach that the amount of harvested power is uncontrollable, but reasonably predictable, based on the source type and harvesting history [6]. To capture the time-varying nature of environmental energy, time is divided into *epochs* of length S. Harvested power is modeled as an epoch-varying function denoted by P_i^h, where i is the epoch sequence number. P_i^h remains constant within the course of each epoch i, but changes for different epochs. To be precise, P_i^h is the actual power received by energy storage which incorporating the loss during power transfer from energy harvester to energy storage, and the power leakage of energy storage. The time unit used for harvesting prediction is therefore one epoch. The *prediction horizon*, H is an interval containing a number of epochs during which predictions can be reasonably made. Our approach needs to know the harvested power prediction of only the coming epoch, at the epoch start.

The node consumes power via either processing, radio communication or sensing. We now describe how to model energy consumption for an individual node. The basic time interval over which energy consumption is calculated is called a *frame*, defined precisely below in Section 3.2. Frames are invoked periodically. We assume the DVS-enabled CPU has m discrete frequencies $f_{min}{=}f_1{<}...{<}f_m{=}f_{max}$ in unit of cycles per second, and the DMS-enabled radio has n discrete modulation levels, $b_{min}{=}b_1{<}...{<}b_n{=}b_{max}$. We use the terms frequency and compute speed interchangeably. In practice, the modulation level represents the number of bits encoded in one signal symbol [18]. To understand this relationship, let R be the fixed symbol rate. Then modulation level b is associated with communicate speed d and expressed as:

$$d = R \cdot b \tag{1}$$

Let e^{sen} represents the energy required for each sensing event. Note that e^{sen} is a constant. The computation energy e_k^{cp} in the k^{th} frame is a function of compute speed f_k and supply voltage $V_{dd,k}$ [2]. The communication energy e_k^{cm} in the k^{th} frame is a function of communicate speed d_k [18]. Then we have:

$$e_k^{cp} = [\alpha f_k V_{dd,k}^2 + P^{ind,cp}] \cdot \frac{C}{f_k} \tag{2}$$

$$e_k^{cm} = [\beta R(2^{d_k/R} - 1) + P^{ind,cm}] \cdot \frac{M}{d_k} \tag{3}$$

Above, C and M are the computation and communication workloads in a frame. C is the number of CPU cycles to be processed, M is the number of bits to be transmitted. The α in Eq. (2) is the CPU switching capacitance which is a constant. The β in Eq. (3) is a constant determined by the transmission quality and noise level [18]. The terms $\alpha f_k V_{dd,k}^2$ and $\beta R(2^{d_k/R} - 1)$ give the *speed-dependent power* of CPU and radio which vary with f_k, $V_{dd,k}$, and d_k respectively.

$P^{ind,cp}$ and $P^{ind,cm}$ are two constants representing the *speed-independent power* of CPU and radio. By using DVS, the supply voltage $V_{dd,k}$ can be reduced linearly alongside with f_k to obtain energy saving (i.e. $f_k \propto V_{dd,k}$), making the speed-dependent CPU power a *cubic* function of f_k. Our model assumes a sufficient level of coordinated sleeping and transmission scheduling, so that the radio energy consumed by listening channel activities is not a significant factor. Finally, the total energy consumed in frame k, e_k^c equals:

$$e_k^c = e^{sen} + e_k^{cp} + e_k^{cm} \tag{4}$$

3.2 Network and Application Model

The system consists of N sensor nodes and the set of wireless links connecting them. A sensor node is denoted as V_i. Base stations or control points are denoted as BS. The N nodes are divided into two types: *source* nodes perform sensing, processing and communication operations, while *relay* nodes only perform processing and communication. Our data processing architecture is quite general, and supports systems that perform some levels of aggregation at each node, as well as systems that do not allow any aggregation. We represent a time-critical and performance sensitive WSN application by requiring all source nodes report their readings, which may or may not be aggregated into other readings, every π time units. The time interval π is the length of a data collection *frame*. Such frame-based data collection mechanism is quite common for WSN applications [7] [13] [18]. In other words, all sensed, processed or aggregated data must reach BS by the end of each frame. For example, at the start of the k^{th} frame (i.e. at time $(k-1)\cdot\pi$), each source node senses the environment and sends sensed data to BS. The data is routed by other nodes and must reach BS by the end of that frame, at time $k\cdot\pi$. We assume all nodes are time-synchronized so that they are aware of the same frame start and end times.

On a per-frame basis, energy consuming activities within each node are represented using a task-based model. In this way, frame-based energy consumption is determined by examining the energy demands of individual tasks (Eq. (4)). There are a total of three task types: *sensing*, *computation* and *communication*. Without loss of generality and in order to simplify the modeling process, we assume the three tasks are executed in the order of *sense*$\rightarrow$*compute*$\rightarrow$*communicate*. That is, in each frame, a node performs sensing first, then processes the sensor reading, then transmits the processed data. The workloads of the computation and communication tasks of any node V_i are fixed over any frame in a given epoch, and denoted as C_i and M_i, respectively.

We assume that each node uses standard WSN energy management techniques for transitioning to *sleep* states when there is no active task. We also assume that compute and communicate speeds only change at the start of an epoch. This design decision reduces the required level of control and synchronization overhead. For instance, the modulation level of a node must not change frequently, since each such change must be conveyed to its receiver in order to ensure correct demodulation of the transmitted data. Using this analysis we can calculate the time required by each node V_i to carry out all activities during frame k, referred as the *per-node latency*, $l_{i,k}$. The per-node latency depends upon the compute speed $f_{i,k}$ and the communicate speed $d_{i,k}$. Then $l_{i,k}$ is given by

$$l_{i,k} = t^{sen} + \frac{C_i}{f_{i,k}} + \frac{M_i}{d_{i,k}} \tag{5}$$

t^{sen} is the sensing time which is a constant. Note that t^{sen} equals zero for relay nodes. We make a common assumption that the effective data transmission time dominates the overall communication time while ignoring the carrier sense time [7], [17], [18]. Thus, the communication time is inversely proportional to $d_{i,k}$.

The system is organized into a data collection and processing tree rooted at BS, using tree construction algorithms such as [1]. In order to support time-critical operation we must define and calculate the *maximum data collection latency* and individual *path latency*. These two values are used in the optimization formulation in Sections 4 and 5 to ensure that all latency requirements are maintained. In each frame, a node V_i receives data from a set of child nodes denoted as $Children(V_i)$. V_i then forwards packets to its parent node, denoted as $Parent(V_i)$, after received data from all its children. Then the maximum data collection latency $L_{tot,k}$ in frame k is the time interval between the start of frame k, and when BS collects all sensed data, given by

$$L_{tot,k} = Max.\{L_{i,k} + l_{i,k} | V_i \in Children(BS)\} \tag{6}$$

Above, $L_{i,k}$ is the latency of the subtree rooted at node V_i, i.e. $L_{i,k} = Max.\{L_{j,k} + l_{j,k} | V_j \in Children(V_i)\}$. The subtree rooted at a leaf node contains only the leaf itself, and hence incurs zero latency.

Next, we define the *path* ρ_i from a node V_i to the root BS as the series of nodes and wireless links connecting V_i and BS. The notation $V_j \in \rho_i$ signifies that V_j is an intermediate node on path ρ_i. The latency $H_{i,k}$ of ρ_i is defined as:

$$H_{i,k} = \sum_{j:V_j \in \rho_i} l_{j,k} \tag{7}$$

Note that by resolving the recursion in Eq. (6), $L_{tot,k}$ actually equals to the latency of the longest path in the tree, i.e. $Max.\{H_{i,k} | \forall \rho_i\}$.

4 Harvesting Aware Speed Selection

Based on the node and network model presented in Section 3, we now formally define the *Harvesting Aware Speed Selection (HASS)* problem. Our goal is to maintain end-to-end performance while maximizing the system's resilience to abnormal or emergency situations. This is accomplished by maximizing the minimum energy level of any node.

The compute and communicate speeds at individual nodes are adjusted at the start of each epoch, and remain fixed throughout that epoch. As defined in

Section 3.1, an epoch is a time interval over which an energy harvesting prediction can be reasonably made. For an arbitrary epoch, the energy consumption $e^c_{i,k}$ and performance latencies $L_{i,k}$, $H_{i,k}$ of node V_i are fixed over any frame k. For simplicity we therefore rewrite them as e^c_i, L_i and H_i. Then the energy level Γ_i of a node V_i at the end of a given epoch is given as:

$$\Gamma_i = \Gamma_i^{init} + P_i^h \cdot S - \lfloor S/\pi \rfloor \cdot e_i^c \tag{8}$$

Γ_i^{init} is the starting energy level of V_i in the epoch. Recall that S is the epoch length. $\lfloor S/\pi \rfloor$ gives the number of frames in an epoch. Using this notation, we define Γ_{min} as

$$\Gamma_{min} = Min\{\Gamma_i | \forall V_i\} \tag{9}$$

Then the goal of our approach is to maximize Γ_{min}. The variables of the problem are the compute and communicate speeds f_i, d_i used by any node V_i in an epoch. Given N nodes in the tree, there are $2N$ unknowns in our problem. The optimal solution to this problem consists of N *speed configurations* (f_i, d_i), one for each node which maximize Γ_{min}. The problem *HASS* is given as:

$$\begin{aligned} Max\ \ & \Gamma_{min} && (10)\\ s.t.\ \ & \forall \rho_i, H_i \le \pi && (11)\\ & \forall V_i, f_i \in [f_{min}, f_{max}], d_i \in [d_{min}, d_{max}] && (12)\\ & \forall V_i, 0 < \Gamma_i \le \Gamma^{max} && (13) \end{aligned}$$

The constraint (11) ensures that the latency of any path ρ_i in the tree is smaller than the frame period π. As mentioned in Section 3.2, this is equivalent to ensuring that the latency of the entire tree is smaller than π. The constraint (12) gives the available ranges of f and d. The constraint (13) requires that the energy level of any node V_i must be confined to the range $(0, \Gamma^{max}]$. In [6], the authors introduced the *energy neutrality* condition, which essentially states that the energy consumed must be no larger than the energy available, such that Γ_i will never drop to zero. This is a necessary condition for an energy harvesting sensor node to operate non-interruptively and we therefore adopt it as a requirement. The left hand side of constraint (13) (called the *positivity constraint*) must hold in order to ensure energy neutrality, while the right hand side (called the *capacity constraint*) is used to model energy storage capacity. Given known and fixed harvested power, and fixed speeds and power consumption, the variation of energy level also fix throughout an epoch, i.e. either monotonically increase or decrease at a fixed rate. Therefore, ensuring a positive energy level at the end of an epoch also ensures positive energy level at the end of any frame in that epoch.

5 Centralized and Distributed Solutions

This section provides centralized and distributed solutions to problem *HASS*. The centralized version provides an optimal solution, while the distributed version is appropriate for systems that need to avoid single control point.

We first give Lemma 1 which states that solving problem *HASS* with full constraint set is equivalent to solving the same problem but without constraint (13). This enables us to remove constraint (13) and focus on a new problem obtained in this manner, denoted as *HASS-N*. Note that the objective function and all other constraints are retained in *HASS-N*.

Lemma 1. If in the optimal solution to HASS-N, Γ_{min} is strictly positive, then the solution to HASS is identical to that of HASS-N. Otherwise, HASS has no feasible solution.

The proof of Lemma 1 can be found in [20]. In the rest of this paper, we will focus on solving problem *HASS-N*. Solving *HASS-N* requires non-linear optimization methods, since it has a non-linear objective function (Eq. (10)). Such costly methods are difficult to implement on resource-constrained sensor nodes. We will show how to obtain an optimal solution efficiently.

A naive approach to solve *HASS-N* is to exhaustively search over all possible solutions. For a system with N nodes where each node has m compute speeds and n communicate speeds, there are $(mn)^N$ possible solutions, making brute force search impractical. However, we notice that many different solutions yield identical Γ_{min}. Using this observation we can simply enumerate each possible Γ_{min}, check if there exists a feasible solution that yields a minimum energy level (among any node) equaling the enumerated Γ_{min}, while satisfying constraints (11) and (12). The highest Γ_{min} that passes this check is by definition the maximum Γ_{min} that we are looking for.

For each node, mn speed configurations correspond to mn different power consumption levels. Since each node's power consumption is fixed throughout an epoch, a node has exactly mn energy consumption levels over an epoch. Thus, given a known starting energy level and a fixed prediction for how much energy can be harvested, a sensor node could end with at most mn possible energy levels in an epoch. Given N nodes, at the end of an epoch, there could be at most mnN different energy levels in the network, and Γ_{min} can be only of these possible values. The set of possible Γ_{min}s is referred as EL (Energy Level), and has a size of mnN.

5.1 Centralized Version

The centralized *HASS* algorithm is called *CHASS*, and is presented in Algorithm 1. It runs on the base station, and assumes that BS must collect Γ^{init} from each node in the system, and is aware of the available speed configurations of sensor nodes. *CHASS* first computes the possible energy levels of all the nodes using Eq. (8) to build the set EL, then sorts EL in non-increasing order (line 1). *CHASS* proceeds iteratively over the sorted EL starting from the first element (i.e. the highest energy level in EL) (line 2). In each iteration p, it solves a decision problem, called *Feasible Solution* denoted by FS_p, by calling algorithm *Is-Feasible* (line 3). The p^{th} element in EL, $EL[p]$ is input to *Is-Feasible*. The problem FS_p is specified as "Is there a solution which yields Γ_{min}=$EL[p]$, while satisfying constraints (11-12)?"

The loop in line 2-9 iterates through all the elements in EL. It continues if the answer to problem FS_p, ans_p is negative, and terminates once it met a FS_p with positive answer, i.e. in iteration z where FS_z is the first problem encountered with positive answer, $z = Min.\{p \in [1, |EL|] | ans_p = TRUE\}$ (line 4-8). By definition of problem *HASS-N* and *FS*, and the ordering of EL, $EL[z]$ is the maximum Γ_{min} that can be achieved (line 5), while satisfying all the constraints. If *CHASS* proceeds to the end of EL and never received a positive answer to any of the FS_p, this implies problem *HASS-N* has no feasible solution.

The algorithm *Is-Feasible* for solving problem FS_p is given in Algorithm 2. The algorithm has one input, the energy level enumerated in iteration p of *CHASS*, $EL[p]$. It has three returned values, the answer to problem FS_p, ans_p, and two speed sets of length N, F^*, D^* which contain f and d derived for all the nodes in the current iteration. F^*, D^* are returned only if ans is positive, otherwise they are empty.

Algorithm 1. CHASS

```
1. Compute and sort EL (in non-increasing order)
2. for $p = 1$ to $|EL|$ do
3.    $[ans_p, F_p^*, D_p^*]$ = call Is-Feasible($EL[p]$)
4.    if $ans_p ==$ TRUE then
5.       $Max\_\Gamma_{min} = EL[p]$
6.       $[F^{opt}, D^{opt}] = [F_p^*, D_p^*]$
7.       Break from for-loop
8.    end if
9. end for
```

Algorithm 2. Is-Feasible - Input: $EL[p]$

```
1. $\Gamma_{min} = EL[p]$
2. for $i = 1$ to N do
3.    $(F^*[i], D^*[i], l_i^{min})$ = call $find\_fastest(\Gamma_{min})$ on $V_i$
4. end for
5. Compute $H_i = \sum_{j:V_j \in \rho_i} l_j^{min}$ for any path $\rho_i$
6. if $\forall \rho_i, H_i \leq \pi$ then
7.    $ans = TRUE$
8. else
9.    $ans = FALSE$, $F^*, D^* = \emptyset$
10. end if
11. return $[ans, F^*, D^*]$
```

We now demonstrate that Algorithm 2 is correct. First, by making $\Gamma_{min} = EL[p]$ (line 1), $\Gamma_i \geq \Gamma_{min} = EL[p]$ must hold for any node V_i. Then the algorithm calls function *find_fastest* for each node (line 2-4) to search over all its *mn* speed configurations for the fastest one, while yielding $\Gamma_i \geq EL[p]$. Specifically, *find_fastest* returns a speed configuration for V_i, $(F^*[i], D^*[i])$ which satisfies:

$$F^*[i] \in [f_{min}, f_{max}], D^*[i] \in [d_{min}, d_{max}] \tag{14}$$

$$\Gamma_i(F^*[i], D^*[i]) \geq EL[p] \tag{15}$$

$$\forall(f,d), l_i(F^*[i], D^*[i]) \leq l_i(f,d) \tag{16}$$

$\Gamma_i(f,d)$ and $l_i(f,d)$ represent the energy level and per-node latency achieved using speed configuration (f,d). *find_fastest* also returns the per-node latency l_i^{min} at V_i achieved by using the derived $(F^*[i], D^*[i])$. Note that l_i^{min} is the least achievable latency according to Eq. (16). Next, for each path ρ_i, we compute its latency H_i by summing up any $l_j^{min}, V_j \in \rho_i$ (line 5). Since (F^*, D^*) minimizes the per-node latency at any node, it also minimizes the latency of any path H_i. Therefore, if $H_i \leq \pi, \forall \rho_i$, the constraint (11) is met, then the answer to problem FS_p is positive (line 6-7). Otherwise, constraint (11) can never be met, hence the answer is negative (line 8-9). Note that it is possible that function $find\_fastest$ does not return an answer, as there may exist some nodes having no possible energy level larger than the input $EL[p]$. In this case, the algorithm immediately rejects $EL[p]$. The speed sets F^*, D^* found in iteration z is set to be the optimal solution to problem *HASS-N* and also *HASS* (line 6 in Algorithm 1). $EL[z]$ is set to be the maximum achievable Γ_{min} (line 5 in Algorithm 1).

It is possible to reduce the runtime of *CHASS* by implementing the search for FS_z in a binary search fashion. This will reduce the number of iterations in *CHASS* from $O(|EL|)$ to $O(\log(|EL|))$. We describe a faster algorithm $CHASS^*$ implemented in this manner and give a complexity analysis in [20].

5.2 Distributed Version

We next describe the distributed *HASS* solution called *DHASS*. The purpose of the distributed version is to enable any node in the network to act as the base station, and therefore enable that node to make command and decisions.

The algorithm *DHASS* proceeds also in binary-search fashion. It requires one initialization round during which each sensor node sends an initialization message containing two pieces of information, its estimated lowest and highest energy levels at the end of the epoch, denoted as Γ^{low} and Γ^{high}. After the initialization round, all the nodes agree on the global lowest and highest achievable energy levels (among the entire tree). The continuous range between the two energy levels is the starting binary search space. Then, it runs for Y computation rounds, each of which corresponds to one iteration of binary search, and solves one problem FS using the distributed *Is-Feasible*. The distributed *Is-Feasible* requires accumulative collection of nodes' latency values, which are in turn used for the root to calculate the end-to-end latency L_{tot}. The root then compares L_{tot} to π in order to determine the answer to problem FS, and disseminates it to all the nodes. Note that any node in the network can be the root. The specification of *DHASS* is given in [20]. Though *DHASS* is only a heuristic-based solution, we demonstrate in [20] that it can closely match the performance of *CHASS*.

6 Performance Evaluation

We performed a series of simulations to evaluate the effectiveness of our *HASS* approaches. The goal of the evaluation is to determine how well both the *CHASS* and *DHASS* algorithm maximize the minimum energy level across the system. The evaluation examined a number of workload scenarios, including several emergency scenarios where there are sudden, unexpected peaks in the demand.

6.1 Experimental Methodology

We evaluated our approaches within a WSN system designed for residential monitoring of water usage and quality. Each customer (residence) is coupled to a supply pipe through a water meter. Each water meter is coupled with a DVS-DMS enabled node. Energy is harvested from the flow of water, using a device such as the one described in [15]. The amount of harvested energy is therefore dependent upon the rate at which the customer uses water. To our best knowledge, we are the first to simulate energy harvesting WSN systems utilizing water flow as the energy source.

We have developed simulation software combining TOSSIM, the standard WSN simulator, with EPANET [9], a public domain, water distribution system modeling program developed by the US Environmental Protection Agency. Our simulator can take as input a variety of WSN topologies, water distribution system configurations and customer usage patterns. Based on water utilization and water quality patterns, the software simulates energy harvesting, and various WSN processing and communication activities. The presented results are based upon a 100 node residential water distribution topology. The topology is derived from an existing suburban area of 100 houses. The 100 nodes installed in the water meters then form a WSN system. We use the *Collection Tree Protocol* [1] to organize the nodes into a data collection tree.

Due to standard repetitive water usage patterns we use a 6 hours cycle, as specified in EPANET. We fix the harvesting horizon at H=6 hours. A horizon is then divided into 24 epochs with equal length S=15 minutes. We run EPANET for 48 hours, containing 8 horizons or equivalently 192 epochs, and obtained hydraulic simulation reports. Using these reports we generate water energy harvesting profile for each node based on the observed water usage at the customer. Note that the difference in the amount of water used across residences will lead to quite different power harvesting profiles across nodes. In [20], we plotted the harvesting profile of one selected node. The frame period is set to π=240ms.

Both algorithm *CHASS* and *DHASS* were implemented in our simulation environment. Although there are no schemes that are directly comparable to our algorithms, we implemented a baseline scheme called No-Power-Management (*NPM*). Unlike the *HASS* approaches, *NPM* scheme is harvesting-unaware in the sense that it uses the highest frequency and modulation level for all the nodes in order to guarantee data collection timing constraint. Our experiments considered two basic application types: applications that support *complete-aggregation* and applications that do not require any aggregation (*non-aggregation*). By complete-aggregation, we mean that each node aggregates multiple packets received into one single packet, while in the non-aggregation case a sensor node forwards all packets to its parent without aggregation. The packet size is randomly selected between M=[64, 128] bytes, and the computational workload is randomly selected between C=[0, 3000000] cycles.

The hardware basis for a DVS-DMS capable platform is the widely available iMote-2 sensor node [16]. The iMote-2 platform has a Intel Xscale PXA27x CPU [4] and a ChipCon CC2420 radio [3]. PXA27x CPU has 6 frequency and power levels as specified in [4]. We derived the radio speed-independent power $P^{cm,ind} = 26.5$mW, radio symbol rate $R = 62.5k$ symbols/sec, and $\beta = 2.74 \times 10^{-8}$ based on CC2420 specification [3] and Eq. (3). We note that the CC2420 is not DMS-capable, so as in [18] we assume four modulation levels, $b = \{2, 4, 6, 8\}$ which give four communicate speeds: $d = \{125, 250, 375, 500\}$ kbps (Eq. (1)). The radio energy is calculated using Eq. (3). We assume a light sensor TSL2561 which takes 12ms to get one reading and consume 0.72mW. Each sensor node uses a rechargeable battery with capacity $\Gamma^{max} = 1000$ joules. All nodes start with the same initial energy level, $\Gamma^{init} = 600$ joules.

Nodes operate in either *normal* or *emergency* mode. We represent the emergency mode by increasing the frame-based workload by w times upon the normal mode, where w is a tunable parameter. This reflects the fact that nodes will need to perform additional duties during those times. We simulate emergency scenarios by introducing contaminant into the system at random time. This can be done by deteriorating the water quality at the water reservoir or a residence. As the contaminant spreads out, the water quality in the residences will decrease and finally been detected by sensor nodes. A sensor node then switches to emergency mode and perform additional workloads over a series of epochs, until the water quality returns to normal.

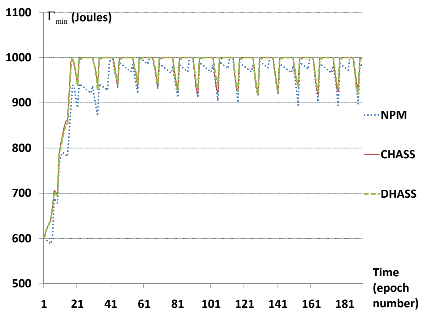
We consider three different types of emergency scenarios. The first type is *random* (RAND) attack. In this case, nodes fail according to a negative exponential distribution, and are picked according to a random uniform distribution from among all the nodes still operating in *normal* mode. The second mode is a *spreading attack* (SPRD). This represents an emergency that increases its area of impact over time. We introduce contaminant into the system from one randomly selected contamination source node. The contaminant spreads out of the system with the flow of water, and lasts a few epochs until water valves are shut off to stop further spreading. The third mode is *area instant* (INST) attack under which a large contiguous area of the network is affected. We simulated one emergency in each horizon, while an emergency started in one horizon may continue to affect multiple successive horizons.

6.2 Results

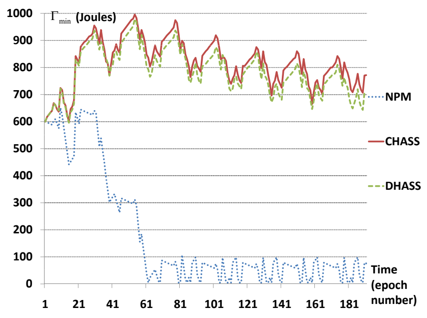
In *CHASS* scheme, the set *EL* contains 2400 elements, given 6 CPU frequencies, 4 modulation levels, and 100 nodes. The experiment setting of *DHASS* is given in [20]. We evaluated the performance of our algorithms under normal and all three emergency modes. We also varied emergency workload levels.

In Fig. 1a-2b, we compared different schemes in term of the achieved Γ_{min}, while assuming non-aggregating applications. We fix the emergency level w at 3.0 which means the emergency workload is three times the normal workload. In normal mode (Fig. 1a), the Γ_{min} value can be seen to vary semi-repetitively, i.e. though it stays close to full capacity at most time, however drops down twice in every horizon (24 epochs). This is because the workload and energy demand in normal mode is relatively low, such that the energy level is dominantly affected by the amount of harvested water energy which varies in repetitive pattern. However in Fig. 1b-2b, the significantly increased workload demand turns to have a dominant effect on energy level, therefore one emergency in each horizon leads to one drop of Γ_{min} in each horizon. This observation demonstrates the effects of harvested energy and workload demand over energy level when operating in different work modes.

As seen from all above figures, in normal and all emergency modes, the *CHASS* scheme achieves the highest Γ_{min}, followed by *DHASS* with slightly lower Γ_{min}. In normal mode (Fig. 1a), *NPM* scheme is able to support Γ_{min} close to full capacity, this is because the harvested energy is much larger than energy demand in normal mode which keeps the energy storage at high level. However, as workload demand increases in emergency mode (Fig. 1b-2b), the performance of *NPM* drops dramatically: its achieved Γ_{min} drops to zero after the 61^{th} epoch in all emergency modes. This implies that at least one node in the network fails to maintain non-empty energy storage and is forced to stop operation. The failure of these nodes will cause service interruption to the entire data collection application during the rest of the epochs. Such lasting service interruption is apparently unacceptable to the mission-critical applications. On the other hand, both *HASS* approaches achieve much higher Γ_{min} than *NPM*. In fact, *CHASS* and *DHASS* never drop to zero in RAND and SPRD modes, and only becomes zero during

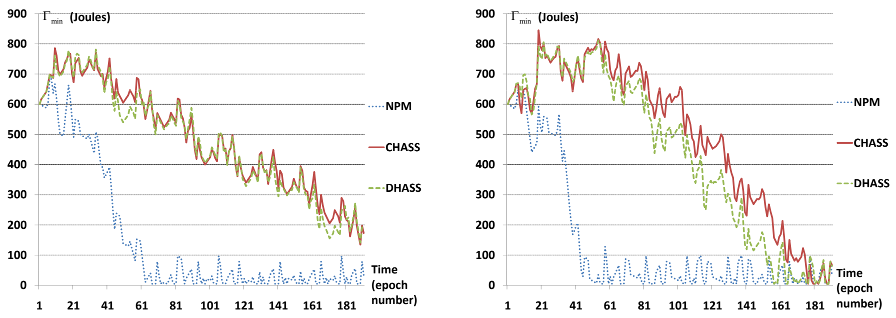


a. Normal mode



b. RAND mode

Fig. 1. Min. energy level Γ_{min}



a. SPRD mode

b. INST mode

Fig. 2. Min. energy level Γ_{min}

the last ten epochs in INST mode. This is because by using *HASS* approaches, the harvesting-rich nodes run at faster speeds to allow the harvesting-poor nodes to slow down, given tight end-to-end latency constraint. The reduced speeds allow the poor nodes to maintain a higher energy storage level, hence enhance the system's capacity to deal with emergencies. Although under extremely intensive emergency, zero Γ_{min} is inevitable even using the *HASS* approaches, it nevertheless demonstrates the importance of in-network data aggregation with regard to energy efficiency.

Another observation from our results is that *DHASS* closely matches the performance of *CHASS* in term of the value of Γ_{min}. In normal mode (Fig. 1a), the achieved Γ_{min} of *CHASS* and *DHASS* almost overlap. In emergency modes, their performance difference enlarges slightly due to increased influence of workload demands over energy level. This observation indicates that *DHASS* scheme can achieve near-optimal performance. Also, we observed that *DHASS* occasionally achieves higher performance than *CHASS*, e.g. between the 1^{st} and 21^{st} epoch in Fig. 2b, this is due to the energy overheads caused by packet collisions and retransmissions.

Table 1. Percent of depleted nodes: NPM

w	1	1.5	2.0	2.5	3.0
RAND	0%	0%	7%	10%	10%
SPRD	0%	0%	6%	9%	9%
INST	0%	0%	7%	10%	16%

We then conducted a *stress test* over the system while using different schemes. That is, we raise the intensity of emergency by increasing the value of w from 1.5 to 3.0 with an increment of 0.5. The aim of this stress test is to evaluate the resilience of different schemes to various emergency intensities. We measure the system resilience to emergency in term of the percentage of nodes that ran out of energy at the epoch which has the lowest Γ_{min} among all 192 epochs. The smaller the percentage of depleted nodes under the same emergency intensity, the higher resilience supported by a scheme compared to others. Table 1 gives the percentage of depleted nodes in all three emergency modes under various emergency intensities, using *NPM* scheme. As seen from Table 1, as emergency intensity increases, the percentage of depleted nodes increases noticeably in all modes when using *NPM*, which implying the low resilience of the harvesting-unaware NPM scheme to emergency situations. While using both *CHASS* and *DHASS*, the same increase in emergency intensity depletes almost no node in the network, except for the scenario when operating in INST mode with a intensity level $w = 3.0$. The results of the stress test demonstrates the benefit of our harvesting-aware approaches in mitigating the impact of emergencies over the system. We then repeated the same set of experiments above for aggregating applications, the results can be found in [20]. Due to in-network data aggregation, the network traffic pattern and workload demands across nodes are quite different to non-aggregating case, hence it would be very interesting to evaluate our solutions for this type of applications.

7 Conclusion

This paper presented an epoch-based approach for energy management in performance constrained WSNs that utilizing energy harvesting combined with DVS and DMS. We adjust radio modulation levels and CPU frequencies in order to satisfy performance requirement, while maximizing the minimum energy reserve over any node in the network. Through this objective, we ensure highly resilient performance under both normal and emergency situations. Through extensive simulations we demonstrated significant performance improvement by using both our solutions over a baseline scheme.

References

1. Gnawali, O., Fonseca, R., Jamieson, K., Moss, D., Levis, P.: Collection tree protocol. In: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys 2009, Berkeley, California, November 4-6, pp. 1–14. ACM, New York (2009)
2. Aydin, H., Melhem, R., Mosse, D., Alvarez, P.M.: Power-aware Scheduling for Periodic Real-time Tasks. IEEE Transactions on Computers 53(5), 584–600 (2004)
3. Texas Instrument. CC2420 Datasheet, `http://docs.tinyos.net/index.php/CC2420`
4. Marvell Technology, Xscale PXA27x Datasheet, `http://www.intel.com/design/intelxscale`
5. Gu, Y., Zhu, T., He, T.: ESC: Energy Synchronized Communication in Sustainable Sensor Networks. In: The 17th International Conference on Network Protocols, Princeton, NJ (October 2009)

6. Kansal, A., Hsu, J., Zahedi, S., Srivastava, M.B.: Power management in energy harvesting sensor networks. ACM Trans. Embed. Comput. Syst. 6(4), 32 (2007)
7. Kumar, G.S.A., Manimaran, G., Wang, Z.: End-to-End Energy Management in Networked Real-Time Embedded Systems. IEEE Transactions on Parallel and Distributed Systems, 1498–1510 (November 2008)
8. Liu, S., Qiu, Q., Wu, Q.: Energy aware dynamic voltage and frequency selection for real-time systems with energy harvesting. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, pp. 236–241. ACM, New York (2008)
9. EPANET 2.0. Water supply and water resources. US EPA (2010)
10. Moser, C., Thiele, L., Benini, L., Brunelli, D.: Real-Time Scheduling with Regenerative Energy. In: Proceedings of the 18th Euromicro Conference on Real-Time Systems, ECRTS, July 5-7, pp. 261–270. IEEE Computer Society, Washington (2006)
11. Moser, C., Thiele, L., Brunelli, D., Benini, L.: Robust and low complexity rate control for solar powered sensors. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, pp. 230–235. ACM, New York (2008)
12. Noh, D.K., Wang, L., Yang, Y., Le, H.K., Abdelzaher, T.: Minimum Variance Energy Allocation for a Solar-Powered Sensor System. In: Krishnamachari, B., Suri, S., Heinzelman, W., Mitra, U. (eds.) DCOSS 2009. LNCS, vol. 5516, pp. 44–57. Springer, Heidelberg (2009)
13. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: a Tiny AGgregation service for ad-hoc sensor networks. SIGOPS Oper. Syst. Rev. 36, 131–146 (2002)
14. Shah, P., Shaikh, T.H., Ghan, K.P., Shilaskar, S.N.: Power Management Using ZigBee Wireless Sensor Network. In: Proceedings of the 2008 First International Conference on Emerging Trends in Engineering and Technology, ICETET, July 16-18, pp. 242–245. IEEE Computer Society, Washington (2008)
15. Pitchford, et al.: Inventors, Systems and methods for generating power through the flow of water, US Patent 7,605,485 (issued October 20, 2009)
16. Crossbow Technology. iMote2 Datasheet, http://docs.tinyos.net/index.php/Imote2
17. Yeh, C., Fan, Z., Gao, R.X.: Energy-aware data acquisition in wireless sensor networks. In: IEEE Instrumentation and Measurement Technology Conference (2007)
18. Yu, Y., Krishnamachari, B., Prasanna, V.: Energy-latency tradeoffs for data gathering in wireless sensor networks. In: IEEE Infocom (2004)
19. Zamora, N.H., Kao, J., Marculescu, R.: Distributed power-management techniques for wireless network video systems. In: Proceedings of the Conference on Design, Automation and Test in Europe, Design, Automation, and Test in Europe. EDA Consortium, San Jose, CA, Nice, France, April 16-20, pp. 564–569 (2007)
20. Zhang, B., Simon, R., Aydin, H.: Energy management for time-critical energy harvesting wireless sensor networks, http://cs.gmu.edu/~simon/tr-2010-ehwsn.pdf
21. Zhang, B., Simon, R., Aydin, H.: Joint Voltage and Modulation Scaling for Energy Harvesting Sensor Networks. In: International Workshop on Energy Aware Design and Analysis of Cyber Physical Systems (WEA-CPS), Stockholm, Sweden (April 2010), http://cs.gmu.edu/~simon/weacps10.pdf

Stably Decidable Graph Languages by Mediated Population Protocols[⋆,⋆⋆]

Ioannis Chatzigiannakis[1,2], Othon Michail[1,2], and Paul G. Spirakis[1,2]

[1] Research Academic Computer Technology Institute (RACTI)
[2] Computer Engineering and Informatics Department (CEID), University of Patras, 26500, Patras, Greece
{ichatz,michailo,spirakis}@cti.gr

Abstract. We work on an extension of the Population Protocol model of Angluin *et al.* that allows edges of the communication graph, G, to have *states* that belong to a constant size set. In this extension, the so called Mediated Population Protocol model (MPP), both *uniformity* and *anonymity* are preserved. We study here a simplified version of MPP in order to capture MPP's ability to stably compute *graph properties*. To understand properties of the communication graph is an important step in almost any distributed system. We prove that any graph property is not computable if we allow disconnected communication graphs. As a result, we focus on studying (at least) *weakly connected* communication graphs only and give several examples of computable properties in this case. To do so, we also prove that the class of computable properties is *closed under complement*, *union* and *intersection* operations. Node and edge parity, bounded out-degree by a constant, existence of a node with more incoming than outgoing neighbors, and existence of some directed path of length at least $k = \mathcal{O}(1)$ are some examples of properties whose computability is proven. Finally, we prove the existence of symmetry in two specific communication graphs and, by exploiting this, we prove that there exists no protocol, whose states eventually stabilize, to determine whether G contains some directed cycle of length 2.

1 Introduction

Most recent advances in microprocessor, wireless communication and sensor/actuator-technologies envision a whole new era of computing, popularly referred to as pervasive computing. Autonomous, ad-hoc networked, wirelessly communicating and *spontaneously interacting* computing devices of *small size* appearing in *great number*, and embedded into environments, appliances and objects of everyday use will deliver services adapted to the person, the time, the place, or the context of their use. The nature and appearance of devices will change to be hidden in the fabric of everyday life and will be augmenting everyday environments to form a pervasive computing landscape.

[⋆] This work has been partially supported by the ICT Programme of the European Union under contract number ICT-2008-215270 (FRONTS).

[⋆⋆] A preliminary brief version of this work has appeared in [12].

In a seminal paper [2], Angluin *et al.* introduced the notion of a computation by a population to model such systems in which individual agents are extremely limited and can be represented as finite-state machines. In their model, finite-state, and complex behavior of the system as a whole emerges from simple rules governing pairwise interaction of the agents. The computation is carried out by a collection of agents, each of which receives a piece of the input. These agents move around and information can be exchanged between two agents whenever they come sufficiently close to each other. The most important innovations of the model are inarguably the *constant memory* constraint imposed to the agents and the *nondeterminism* inherent to the interaction pattern. These assumptions provide us with a concrete and realistic model for future systems.

A population protocol $\mathcal{A}$ consists of finite *input and output alphabets* X and Y, a finite set of *states* Q, an *input function* $I : X \to Q$ mapping inputs to states, an *output function* $O : Q \to Y$ mapping states to outputs, and a *transition function* $\delta : Q \times Q \to Q \times Q$. The model assumes a population of $n \equiv |V|$ agents and a protocol runs on a (simple) directed communication graph $G = (V, E)$. An *agent* has a *memory of constant size* (i.e., $\mathcal{O}(1)$ bits) and a *control unit* that updates the agent states according to the interactions taking place; the input and output of the agents may represent a *sensor* and/or an *actuator*. Each protocol has a constant-size description, i.e., independent of n, that can be stored in each agent of the population. This gives to population protocols two important properties: *uniformity* and *anonymity*; the transition function treats all agents in the same way and there is no room in the state of an agent to store a unique identifier.

The initial goal of the model was to study the *computational limitations* of cooperative systems consisting of many limited devices (agents), imposed to *passive* (but *fair*) communication by some *scheduler*. Much work showed that there exists an exact characterization of the computable predicates: they are precisely the *semilinear predicates* or equivalently the predicates definable by first-order logical formulas in *Presburger arithmetic* [2,3,5,6,7]. Some recent work has concentrated on performance, supported by a random scheduling assumption [4]. [10] proposed a generic definition of probabilistic schedulers and a collection of new fair schedulers, and revealed the need for the protocols to adapt when natural modifications of the mobility pattern occur. [9,14] considered a huge population hypothesis (population going to infinity), and studied the dynamics, stability and computational power of probabilistic population protocols by exploiting the tools of continuous nonlinear dynamics. In [9] it was also proven that there is a strong relationship between classical finite population protocols and models given by ordinary differential equations.

There exist a few extensions of the basic model in the relevant literature to more accurately reflect the requirements of practical systems. In [1] they studied what properties of restricted communication graphs are stably computable, gave protocols for some of them, and proposed the model extension with *stabilizing inputs*. The results of [5] show that again the semilinear predicates are all that can be computed by this model. Finally, some works incorporated agent failures [15] and gave to some agents slightly increased computational power [8] (heterogeneous systems). For an excellent introduction see [7].

Very recently, a natural variation of the basic model was proposed [13], where the *interactions* of the agents can be characterized by a state of constant size. Essentially the model is augmented to include a *Mediator*, i.e., a global storage capable of storing very limited information for each communication link (the state of the link). When pairs of agents interact, they can read and update the state of the link. Interestignly, although anonymity and uniformity are preserved, *the presence of a mediator allows us to obtain significant more computational power*; we can build systems with the ability of computing subgraphs and solve optimization problems concerning the communication graph. In [13] it was shown that the new model is capable of computing non-semilinear predicates and that any stably computable predicate belongs to $NSPACE(m)$, where m denotes the number of edges of the interaction graph. The latter inclusion was proven in [11] to hold with equality. Finally, [12] constitutes a preliminary brief version of this work.

In this work (as [1] did for population protocols), we consider a simplification of the above model in order to explore one of its most important capabilities: The computability of graph properties. To understand properties of the communication graph is an important step in almost any distributed system. In particular, we temporarily disregard the input notion of the population and assume that all agents simply start from a unique initial state (and the same holds for the edges). We are interested in protocols of the new model, that we call the GDMPP model, that when executed on any communication graph G, after a finite number of steps stabilize to a configuration where all agents give 1 as output if G belongs to a graph language L, and 0 otherwise. This is motivated by the idea of having protocols that eventually accept all communication graphs (on which they run) that satisfy a specific property, and eventually reject all remaining communication graphs. The reason for proposing a simplified model is that it enables us to study what graph properties are stably computable by the MPP model without the need to keep in mind its remaining parameters.

2 Our Results - Roadmap

In Section 3, we give a formal definition of the GDMPP model. In Section 4, we focus on weakly connected communication graphs. We prove that the class of computable graph properties is closed under complement, union, and intersection operations. Node and edge parity, bounded out-degree by a constant, existence of a node with more incoming than outgoing neighbors, and existence of some directed path of length at least $k = \mathcal{O}(1)$ are some examples of properties whose computability is proven. Moreover, the existence of symmetry in two specific communication graphs is revealed and is exploited to prove that there exists no GDMPP, whose states eventually stabilize, to compute the graph language $2C$, consisting of all weakly connected communication graphs that contain some 2-cycle. We leave as an interesting open problem whether $2C$ isn't computable in the general case. In Section 5, we focus on the universe of all possible communication graphs, containing also the disconnected ones. In this case (see Theorem 10) we prove that any nontrivial graph language (we exclude both the empty language and its complement) is not computable by the GDMPP model. As an interesting corollary we get that GDMPP cannot compute connectivity (Corollary 1). Finally, in Section 6 we discuss some future research directions.

3 The Model

A *Graph Decision Mediated Population Protocol* (GDMPP) $\mathcal{A}$ consists of a *binary output alphabet* $Y = \{0, 1\}$, a finite set of *agent states* Q, an *output function* $O : Q \rightarrow Y$ mapping agent states to outputs, a finite set of *edge states* S, and a *transition function* $\delta : Q \times Q \times S \rightarrow Q \times Q \times S$. If $\delta(a, b, s) = (a', b', s')$ we call $(a, b, s) \rightarrow (a', b', s')$ a *transition*, and we define $\delta_1(a, b, s) = a'$, $\delta_2(a, b, s) = b'$ and $\delta_3(a, b, s) = s'$.

We assume that all agents are initially in an *initial agent state* $q_0 \in Q$ and all edges in an *initial edge state* $s_0 \in S$. A *graph universe* (or *graph family*) is any set of communication graphs. We denote by $\mathcal{H}$ the graph universe consisting of all possible communication graphs of any finite number of nodes greater or equal to 2 (we do not allow the empty graph, the graph with a unique node and we neither allow infinite graphs) and by $\mathcal{G}$ the subset of $\mathcal{H}$ containing the weakly connected ones. All the following definitions hold w.r.t. some fixed graph universe $\mathcal{U}$. A *graph language* L is a subset of $\mathcal{U}$ containing communication graphs that possibly share some common property., e.g. $L = \{G \in \mathcal{U} \mid G$ contains a directed hamiltonian path$\}$. A graph language L is said to be *nontrivial* if $L \neq \emptyset$ and $L \neq \mathcal{H}$.

A GDMPP runs on a graph $G = (V, E)$, where V is a population of $|V| = n$ agents and E is an irreflexive binary relation on V. The graph on which the protocol runs is considered as the *input graph* of the protocol. The input graph of a GDMPP may be any $G \in \mathcal{U}$.

A *network configuration* (or simply *configuration*) is a mapping $C : V \cup E \rightarrow Q \cup S$ specifying the agent state of each agent in the population and the edge state of each edge in the communication graph. Let C and C' be network configurations, and let u, v be distinct agents. We say that C goes to C' via encounter $e = (u, v)$, denoted $C \xrightarrow{e} C'$, if $C'(u) = \delta_1(C(u), C(v), C(e))$, $C'(v) = \delta_2(C(u), C(v), C(e))$, $C'(e) = \delta_3(C(u), C(v), C(e))$, and $C'(z) = C(z)$ for all $z \in (V - \{u, v\}) \cup (E - \{e\})$. We say that C can go to C' in one step, denoted $C \rightarrow C'$, if $C \xrightarrow{e} C'$ for some encounter $e \in E$. We write $C \xrightarrow{*} C'$ if there is a sequence of configurations $C = C_0, C_1, \ldots, C_t = C'$, such that $C_i \rightarrow C_{i+1}$ for all i, $0 \leq i < t$, in which case we say that C' is *reachable* from C.

An *execution* is a finite or infinite sequence of network configurations $C_0, C_1, C_2, \ldots$, where C_0 is an initial configuration and $C_i \rightarrow C_{i+1}$, for all $i \geq 0$. An infinite execution is *fair* if for every pair of network configurations C and C' such that $C \rightarrow C'$, if C occurs infinitely often in the execution, then so does C'. A *computation* is an infinite fair execution.

At any point during the execution of a GDMPP, each agent's state determines its output at that time. The output of any agent u under configuration C is

$O(C(u))$. Note also that the *code* of any GDMPP is of *constant size* (independent of the population size) and, thus, can be stored in each agent (device) of the population.

Definition 1. *Let L be a graph language consisting of all $G \in \mathcal{U}$ for which, in any computation of a GDMPP $\mathcal{A}$ on G, all agents eventually output* 1*. Then L is* the language stably recognized by $\mathcal{A}$*. A graph language is said to be* stably recognizable *by the GDMPP model (also called GDMPP*-recognizable*) if some GDMPP stably recognizes it.*

Thus, any protocol *stably recognizes* the graph language consisting of those graphs on which the protocol always answers "accept", i.e. eventually all agents output the value 1 (possibly the empty language).

Definition 2. *We say that a GDMPP $\mathcal{A}$* stably decides *a graph language $L \subseteq \mathcal{U}$ (or equivalently a predicate $p_L : \mathcal{U} \rightarrow \{0,1\}$ defined as $p_L(G) = 1$ iff $G \in L$) if for any $G \in \mathcal{U}$ and any computation of $\mathcal{A}$ on G, all agents eventually output* 1 *if $G \in L$ and all agents eventually output* 0 *if $G \notin L$. A graph language is said to be* stably decidable *by the GDMPP model (also called GDMPP*-decidable*) if some GDMPP $\mathcal{A}$ stably decides it.*

A GDMPP $\mathcal{A}$ has *stabilizing states* if in any computation of $\mathcal{A}$, after a finite number of interactions, the states of all agents stop changing.

In some cases, a protocol, instead of stably deciding a language L, may provide some different sort of guarantee. For example, whenever runs on some $G \in L$, it may forever remain to configurations where at least one agent is in state a, and when $G' \notin L$ no agent will remain in state a. To formalize this, we say that a GDMPP $\mathcal{A}$ *guarantees $t : Q^* \rightarrow \{0,1\}$ w.r.t. $L \subseteq \mathcal{U}$* if, for any $G \in \mathcal{U}$, any computation of $\mathcal{A}$ on G eventually reaches a configuration C, s.t. for all C', where $C \xrightarrow{*} C'$, it holds that $t(C') = t(C) = 1$ if $G \in L$ and $t(C') = t(C) = 0$, otherwise.[1]

4 Weakly Connected Graphs

In this section, we study an interesting case in which the graph universe is not allowed to contain disconnected graphs. Thus, here the graph universe is $\mathcal{G}$ and, thus, a graph language can only be a subset of $\mathcal{G}$. The main reason for selecting this specific universe for devising our protocols is that, if we also allow disconnected graphs, then, as we shall see, it can be proven that no graph language is stably decidable.

4.1 Decidable Graph Languages

Our goal is to show the stable decidability of some interesting graph languages by providing protocols for them and proving their correctness. To begin, we prove some closure results to obtain a useful tool for our purpose.

[1] By assuming an ordering on V we can define configurations as strings from Q^*.

Theorem 1. *The class of stably decidable graph languages is closed under complement, union and intersection operations.*

Proof. We show that for any stably decidable graph languages L_1 and L_2, $L_3 = L_1 \cup L_2$ is also stably decidable. The remaining proofs are similar, so we ommit them. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be GDMPPs that stably decide L_1 and L_2, respectively (we know their existence). We let the two protocols operate in parallel, i.e. we devise a new protocol $\mathcal{A}_3$ whose agent and edge states consist of two components, one for protocol $\mathcal{A}_1$ and one for $\mathcal{A}_2$. Let O_1 and O_2 be the output maps of the two protocols. We define the output map O_3 of $\mathcal{A}_3$ as $O_3(q, q') = 1$ iff at least one of $O_1(q)$ and $O_2(q')$ equals to 1, for all $q \in Q_{\mathcal{A}_1}$ and $q' \in Q_{\mathcal{A}_2}$. If $G \in L_3$ then at least one of the two protocols has eventually all its agent components giving output 1, thus $\mathcal{A}_3$ correctly answers "accept", while if $G \notin L_3$ then both protocols have eventually all their agent components giving output 0, thus $\mathcal{A}_3$ correctly answers "reject". We conclude that $\mathcal{A}_3$ stably decides L_3 which proves that L_3 is stably decidable. □

In some cases it is not easy to devise a protocol that respects the *predicate output convention* (the predicate output convention was defined in [2] and simply requires all agents to eventually agree on the correct output value). In such cases, we can use the following variation of the Composition Theorem (Theorem 6) of [13] that facilitates the proof of existence of GDMPP protocols that stably decide a language.

Theorem 2. *If there exists a GDMPP $\mathcal{A}$ with stabilizing states that w.r.t. to a language L guarantees a semilinear predicate, then L is GDMPP-decidable.*

Proof. Immediate from the proof of the Theorem 6 of [13]. $\mathcal{A}$ can be composed with a provably existing GDMPP $\mathcal{B}$ whose stabilizing inputs are $\mathcal{A}$'s agent states to give a new GDMPP $\mathcal{C}$ that stably decides L w.r.t. the predicate output convention. Note that $\mathcal{B}$ is in fact a GDMPP, since its stabilizing inputs are not real inputs (GDMPPs do not have inputs). It simply updates its state components by taking also into account the eventually stabilizing state components of $\mathcal{A}$. Thus, their composition, $\mathcal{C}$, is also a GDMPP.

Theorem 3 (Node Parity). *The graph languages $N_{even} = \{G \in \mathcal{G} \mid |V(G)|$ is even$\}$ and $N_{odd} = \overline{N}_{even}$ are satbly decidable.*

Proof. Assume that the initial agent state is 1. Then there is a protocol in [2] that stably decides N_{even} in the case where G is complete. But, according to Theorem 4 in page 295 of [2], there must exist another protocol that stably decides N_{even} in the general case, i.e. in any graph $G \in \mathcal{G}$, by simply swapping states to ensure that any two states eventually meet. Thus, L is stably decidable by the population protocol model that does not use inputs and whose output alphabet is $\{0, 1\}$, and since this model is a special case of the GDMPP model, N_{even} is GDMPP-decidable. Moreover, since the class of GDMPP-decidable graph languages is closed under complement (see Theorem 1), it follows that $N_{odd} = \overline{N}_{even}$ is also GDMPP- decidable. □

Theorem 4 (Edge Parity). *The graph languages $E_{even} = \{G \in \mathcal{G} \mid |E(G)|$ is even$\}$ and $E_{odd} = \overline{E}_{even}$ are stably decidable.*

Proof. By exploiting the closure under complement, it suffices to prove that E_{even} is stably decidable by presenting a GDMPP that stably decides it. The initial agent state is $(0, 0)$ consisting of two components, where the first one is the *data bit* and the second the *live bit* following the idea of the parity protocol of [2]. An agent with live bit 0 is said to be *sleeping* and an agent with live bit equal to 1 is said to be *awake*. The initial edge state is 1, which similarly means that all edges are initially awake. We divide the possible interactions in the following four groups (we also present their effect):

1. Both agents are sleeping and the edge is awake:
   – The initiator wakes up, both agents update their data bit to 1, and the edge becomes sleeping.
2. Both agents are sleeping and the edge is sleeping:
   – Nothing happens.
3. One agent is awake and the other is sleeping:
   – The sleeping agent becomes awake and the awake sleeping. Both set their data bits to the modulo 2 sum of the data bit of the agent that was awake before the interaction and the edge's state, and if the edge was awake becomes sleeping.
4. Both agents are awake:
   – The responder becomes sleeping, they both set their data bits to the modulo 2 sum of their data bits and the edge's state, and if the edge was awake becomes sleeping.

It is easy to see that the initial modulo 2 sum of the edge bits (initially, they are all equal to 1) is preserved and is always equal to the modulo 2 sum of the bits of the awake agents and the awake edges. The first interaction creates the first awake agent and from that time there is always at least one awake agent and eventually remains only one. Moreover, all edges eventually become sleeping which simply means that eventually the one remaining awake agent contains the modulo 2 sum of the initial edge bits which is 0 iff the number of edges is even. All the other agents are sleeping, which means that they copy the data bit of the awake agent, thus eventually they all contain the correct data bit. The output map is defined as $O(0, \cdot) = 1$ (meaning even edge parity) and $O(1, \cdot) = 0$ (meaning odd edge parity). □

Theorem 5 (Constant Neighbors - Some Node). *The graph language $N_k^{out} = \{G \in \mathcal{G} \mid G$ has some node with at least k outgoing neighbors$\}$ is stably decidable for any $k = \mathcal{O}(1)$ (the same holds for $\overline{N}_k^{out}$).*

Proof. Initially all agents are in q_0 and all edges in 0. The set of agent states is $Q = \{q_0, \ldots, q_k\}$ the set of edge states is binary and the output function is defined as $O(q_k) = 1$ and $O(q_i) = 0$ for all $i \in \{0, \ldots, k-1\}$. We now describe the transition function. In any interaction through an edge in state 0, the initiator visits an unvisited outgoing edge, so it marks it by updating the edge's state to 1 and increases its own state index by one, e.g. initially $(q_0, q_0, 0)$ yields $(q_1, q_0, 1)$, and, generally, $(q_i, q_j, 0) \rightarrow (q_{i+1}, q_j, 1)$, if $i + 1 < k$ and $j < k$, and $(q_i, q_j, 0) \rightarrow (q_k, q_k, 1)$, otherwise. Whenever two agents meet through a marked edge they do nothing, except for the case where only one of them is in the special alert state q_k. If the latter holds, then both go to the alert state, since in this case the protocol has located an agent with at least k outgoing neighbors. To conclude, all agents count their outgoing edges and initially output 0. Iff one of them marks its k-th outgoing edge, both end points of that edge go to an alert state q_k that propagates to the whole population and whose output is 1, indicating that G belongs to N_k^{out}. □

Note that $\overline{N}_k^{out}$ contains all graphs that have no node with at least $k = \mathcal{O}(1)$ outgoing neighbors, in other words, all nodes have fewer than k outgoing edges, which is simply the well-known bounded by k out-degree predicate. The same statement for population protocols appears as Lemma 3 in [1].

Theorem 6 (Constant Neighbors - All Nodes). *The graph language $K_k^{out} = \{G \in \mathcal{G} \mid$ Any node in G has at least k outgoing neighbors$\}$ is stably decidable for any $k = \mathcal{O}(1)$ (the same holds for $\overline{K}_k^{out}$).*

Proof. Note, first of all, that another way to think of K_k^{out} is $K_k^{out} = \{G \in \mathcal{G}$ $\mid$ No node in G has less than k outgoing neighbors$\}$, for some $k = \mathcal{O}(1)$. The protocol we describe is similar to the one described in the proof of Theorem 5. The only difference is that when an agent counts its k-th outgoing neighbor as the initiator of an interaction, it goes to the special alert state q_k, but the alert state is not propagated (e.g. the responder of this interaction keeps its state). It follows that eventually any node that has marked at least k outgoing edges will be in the alert state, while any other node that has less than k outgoing edges will be in some state q_i, where $i < k$. Clearly the protocol has stabilizing states and provides the following semilinear guarantee:

- If $G \notin K_k^{out}$ then at least one agent remains in some state q_i, where $i < k$.
- If $G \in K_k^{out}$ no such state remains.

Thus, Theorem 2 applies, implying that there exists some GDMPP stably deciding K_k^{out} w.r.t. the predicate output convention. Thus, both K_k^{out} and $\overline{K}_k^{out}$ are stably decidable and the proof is complete. □

Theorem 7 (Compare Incoming and Outgoing Neighbors). *The graph language $M_{out} = \{G \in \mathcal{G} \mid G$ has some node with more outgoing than incoming neighbors$\}$ is stably decidable (the same holds for $\overline{M}_{out}$).*

Proof. Consider the following protocol: Initially all agents are in state 0 which is the *equality* state. An agent can also be in state 1 which is the *more-outgoing* state. Initially all edges are in state s_0 and S contains also o, i and b, where state o means that the edge has been used by the protocol only as outgoing so far, i

means only as incoming and b is for "both". Any agent always remembers if it has seen so far more outgoing edges or the same number of incoming and outgoing edges. So, if it is in equality state and is the initiator in an interaction where the edge has not been used at all (state s_0) or has been used only as an incoming edge (state i), which simply means that only the responder has counted it, then the agent goes to the more-outgoing state and updates the edge accordingly to remember that it has counted it. Similarly, if an agent in the more-outgoing state is the responder of an interaction and the edge is in one of the states s_0 or o, then it goes to the equality state and updates the edge accordingly. If we view the interaction from the edge's perspective, then we distinguish the following cases:

1. The edge is in state s_0. Both the initiator and the responder can use it. If only the initiator uses it (both initiator and responder in equality state), then the edge goes to state o. If only the responder uses it (both in more-outgoing state) then the edge goes to state i. If both use it (initiator in equality and responder in more-outgoing) then it goes to state b. If no one uses it it remains in s_0.
2. The edge is in state o. The initiator cannot use it, since it has already counted it. If the responder is in more-outgoing state, then it counts it, thus the edge goes to state b. If, instead, it is in the equality state, the edge remains in state o.
3. The edge is in state i. The responder cannot use it. If the initiator is in equality state, then it counts it, thus the edge goes to state b. If, instead, it is in the more-outgoing state, the edge remains in state i.
4. The edge is in state b. Both the initiator and the responder have used it, thus nothing happens.

The equality state outputs 0 and the more-outgoing state outputs 1. If there exists a node with more outgoing edges, then it will eventually remain in the more-outgoing state giving 1 as output, otherwise all nodes will eventually remain in equality state (although some of them may have more incoming edges), thus giving 0 as output. Computing that at least one more-outgoing state eventually remains is semilinear and the protocol, obviously, has stabilizing states, thus Theorem 2 applies and we conclude that M_{out} is stably decidable. Closure under complement implies that $\overline{M}_{out}$ is also stably decidable. □

Remark 1. By symmetry, the corresponding languages N_k^{in}, $\overline{N}_k^{in}$, K_k^{in} and $\overline{K}_k^{in}$ concerning incoming neighbors, $M_{in} = \{G \in \mathcal{G} \mid G$ has some node with more incoming than outgoing neighbors$\}$ and $\overline{M}_{in}$ are also stably decidable.

Theorem 8 (Directed Path of Constant Length). *The graph language $P_k = \{G \in \mathcal{G} \mid G$ has at least one directed path of at least k edges$\}$ is stably decidable for any $k = \mathcal{O}(1)$ (the same holds for $\overline{P}_k$).*

Proof. If $k = 1$ the protocol that stably decides P_1 is trivial, since it accepts iff at least one interaction happens (in fact it can always accept since all graphs have at least two nodes and they are weakly connected, and thus $P_1 = \mathcal{G}$). We give a general protocol, *DirPath* (Protocol 1), that stably decides P_k for any constant $k > 1$.

Protocol 1. *DirPath*

```
1: $Q = \{q_0, q_1, 1, \ldots, k\}$, $S = \{0, 1\}$,
2: $O(k) = 1$, $O(q) = 0$, for all $q \in Q - \{k\}$,
3: $\delta$:
```

$$\begin{aligned}
(q_0, q_0, 0) &\to (q_1, 1, 1) \\
(q_1, x, 1) &\to (x-1, q_0, 0), \text{ if } x \geq 2 \\
&\to (q_0, q_0, 0), \text{ if } x = 1 \\
(x, q_0, 0) &\to (q_1, x+1, 1), \text{ if } x + 1 < k \\
&\to (k, k, 0), \text{ if } x + 1 = k \\
(k, \cdot, \cdot) &\to (k, k, \cdot) \\
(\cdot, k, \cdot) &\to (k, k, \cdot)
\end{aligned}$$

Intuitively, the protocol expands non-communicating active paths (they can interact but the corresponding transitions do nothing, that's why they are not appearing in δ). The head of each path counts its length. If the length of an active path ever becomes equal to k, then a state giving 1 as output is propagated. Note that, to avoid getting stuck, the protocol keeps backtracking and even totally releasing the active paths. Fairness condition ensures that if a path of length at least k exists, then *DirPath* will eventually find it. □

4.2 Non Stably Decidable Languages

Now we are about to prove that a specific graph language cannot be stably decided by GDMPPs with stabilizing states. First we state and prove a useful lemma.

Lemma 1. *For any GDMPP $\mathcal{A}$ and any computation (infinite fair execution) $C_0, C_1, C_2, \ldots$ of $\mathcal{A}$ on G (Figure 1(a)) there exists a computation $C'_0, C'_1, C'_2, \ldots,$ $C'_i, \ldots$ of $\mathcal{A}$ on G' (Figure 1(b)) s.t.*

$$\begin{aligned}
C_i(v_1) &= C'_{2i}(u_1) = C'_{2i}(u_3) \\
C_i(v_2) &= C'_{2i}(u_2) = C'_{2i}(u_4) \\
C_i(e_1) &= C'_{2i}(t_1) = C'_{2i}(t_3) \\
C_i(e_2) &= C'_{2i}(t_2) = C'_{2i}(t_4)
\end{aligned}$$

for any finite $i \geq 0$.

Proof. The proof is by induction on i. We assume that initially all nodes are in q_0 and all edges in s_0 (initial states). So the base case (for $i = 0$) holds trivially. Now we make the following assumption: Whenever the scheduler of $\mathcal{A}$ on G (call it S_1) selects the edge e_1 we assume that the scheduler, S_2, of $\mathcal{A}$ on G' takes two steps; it first selects t_1 and then selects t_3. Whenever S_1 selects the edge e_2, S_2 first selects t_2 and then t_4. Formally, if $C_{i-1} \overset{e_1}{\rightarrow} C_i$ then $C'_{2(i-1)} \overset{t_1}{\rightarrow} C'_{2i-1} \overset{t_3}{\rightarrow} C'_{2i}$ and if $C_{i-1} \overset{e_2}{\rightarrow} C_i$ then $C'_{2(i-1)} \overset{t_2}{\rightarrow} C'_{2i-1} \overset{t_4}{\rightarrow} C'_{2i}$ for every finite $i \geq 1$. Obviously, S_2 is not a fair scheduler so to be able to talk about computation we only require this predetermined behavior to be followed by S_2 for a finite number of steps. After this finite number of steps, S_2 goes on arbitrarily but in a fair manner.

Now assume that all conditions are satisfied for some finite step i (inductive hypothesis). We will prove that the same holds for step $i + 1$ to complete the proof (inductive step). There are two cases:

1. $C_i \overset{e_1}{\rightarrow} C_{i+1}$ *(i.e. in step $i+1$ S_1 selects the edge e_1)*: Then we know that S_2 first selects t_1 and then t_3 (in its corresponding steps $2i+1$ and $2i+2$). That is, its first transition is $C'_{2i} \overset{t_1}{\rightarrow} C'_{2i+1}$ and its second is $C'_{2i+1} \overset{t_3}{\rightarrow} C'_{2(i+1)}$. But from the inductive hypothesis we know that $C'_{2i}(u_1) = C_i(v_1)$, $C'_{2i}(u_2) = C_i(v_2)$ and $C'_{2i}(t_1) = C_i(e_1)$ which simply means that interaction e_1 on G has the same effect as interaction t_1 on G' (u_1 has the same state as v_1, u_2 as v_2 and t_1 as e_1). Thus, $C'_{2i+1}(u_1) = C_{i+1}(v_1)$, $C'_{2i+1}(u_2) = C_{i+1}(v_2)$ and $C'_{2i+1}(t_1) = C_{i+1}(e_1)$. Moreover, in this step t_3 and both its endpoints do not change state (since the interaction concerned t_1), thus $C'_{2i+1}(u_3) = C'_{2i}(u_3) = C_i(v_1)$ (the last equation comes from the inductive hypothesis), $C'_{2i+1}(u_4) = C'_{2i}(u_4) = C_i(v_2)$ and $C'_{2i+1}(t_3) = C'_{2i}(t_3) = C_i(e_1)$. When in the next step S_2 selects t_3, t_1 and both its endpoints do not change state, thus $C'_{2(i+1)}(u_1) = C'_{2i+1}(u_1) = C_{i+1}(v_1)$, $C'_{2(i+1)}(u_2) = C'_{2i+1}(u_2) = C_{i+1}(v_2)$ and $C'_{2(i+1)}(t_1) = C'_{2i+1}(t_1) = C_{i+1}(e_1)$. Now let's see what happens to t_3 and its endpoints. Before the interaction the state of u_3 is $C_i(v_1)$, the state of u_4 is $C_i(v_2)$ and the state of t_3 is $C_i(e_1)$, which means that, in $C'_{2(i+1)}$, u_3 has gone to $C_{i+1}(v_1)$, u_4 to $C_{i+1}(v_2)$ and t_3 to $C_{i+1}(e_1)$. Finally, t_2 and t_4 have not participated in any of the two interactions of S_2 and thus they have maintained their states, that is $C'_{2(i+1)}(t_2) = C'_{2i}(t_2) = C_i(e_2) = C_{i+1}(e_2)$ (the last two equations follow from the inductive hypothesis and the fact that, in step $i + 1$, S_1 selects e_1 which means that e_2 maintains its state, respectively), and similarly $C'_{2(i+1)}(t_4) = C_{i+1}(e_2)$.
2. $C_i \overset{e_2}{\rightarrow} C_{i+1}$ *(i.e. in step $i+1$ S_1 selects the edge e_2)*: This case is symmetric to the previous one.

□

Let now $\mathcal{A}$ be a GDMPP that stably decides the graph language $2C = \{G \in \mathcal{G} \mid G$ has at least two nodes u, v s.t. both $(u, v), (v, u) \in E(G)$ (in other words, G has at least one 2-cycle)$\}$. So for any computation of $\mathcal{A}$ on G, after finitely many steps, both v_1 and v_2 go to some state that outputs 1, since $G \in 2C$, and do not change their output value in any subsequent step (call the corresponding output

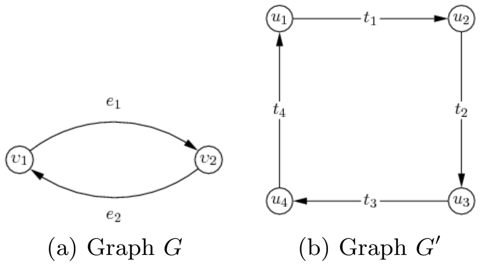


Fig. 1. $G \in 2C$ and $G' \notin 2C$

stable configuration C_i, where i is finite). But according to Lemma 1 there exists a computation of $\mathcal{A}$ on G' that under configuration C'_{2i} has u_1, u_2, u_3 and u_4 giving output 1. We use this fact to prove the following impossibility result.

Theorem 9. *There exists no GDMPP with stabilizing states to stably decide the graph language* $2C = \{G \in \mathcal{G} \mid G$ *has at least two nodes* u, v *s.t. both* $(u, v), (v, u) \in E(G)\}$.

Proof. Let $\mathcal{A}$ be a GDMPP with stablizing states that stably decides $2C$. It follows that when $\mathcal{A}$ runs on G (Figure 1(a)) after a finite number of steps v_1 and v_2 obtain two states w.l.o.g. q_1 and q_2, respectively, that output 1 (since $\mathcal{A}$ stably decides $2C$) and do not change in any subsequent step (since $\mathcal{A}$ has stabilizing states). Assume that at that point e_1 is in s_1 and e_2 in s'_1. Assume also that there exists a subset $S_1 = \{s_1, s_2, \ldots, s_k\}$ of S, of edge states that can be reached by subsequent interactions of the pair (v_1, v_2) and a subset $S_2 = \{s'_1, s'_2, \ldots, s'_l\}$ of S, of edge states that can be reached by subsequent interactions of the pair (v_2, v_1), where k and l are both constants independent of n (note that S_1 and S_2 are not necessarily disjoint). It follows that for all $s_i \in S_1$, $(q_1, q_2, s_i) \rightarrow (q_1, q_2, s_j)$, where $s_j \in S_1$, and for all $s'_i \in S_2$, $(q_2, q_1, s'_i) \rightarrow (q_2, q_1, s'_j)$, where $s'_j \in S_2$. In words, none of these reachable edge states can be responsible for a change in some agent's state. According to Lemma 1 there exists a computation of $\mathcal{A}$ on G' (Figure 1(b)) such that after a finite number of steps u_1, u_3 are in q_1, u_2, u_4 are in q_2, t_1, t_3 are in s_1 and t_2, t_4 are in s'_1. Since $\mathcal{A}$ stably decides $2C$, at some subsequent finite step (after we let the protocol run in a fair manner in G'), some agent obtains a new state q_3, since if it didn't then all agents would always remain to states q_1 and q_2 that output 1 (but in G' there is no 2-cycle and such a decision is wrong). This must happen through some interaction of the following two forms: (i) (q_1, q_2, s_i), where $s_i \in S_1$ and (ii) (q_2, q_1, s'_i), where $s'_i \in S_2$. But this is a contradiction, since we showed earlier that no such interaction can modify the state of any of its end points. Intuitively, if there exists some way for $\mathcal{A}$ to modify one of q_1 and q_2 in G' then there would also exist some way for $\mathcal{A}$ to modify one of q_1 and q_2 in G, after the system has obtained stabilizing states there, which is an obvious contradiction. □

5 Graphs Not Even Weakly Connected

In this section, our universe is $\mathcal{H}$ and, thus, a graph language can only be a subset of $\mathcal{H}$. Any disconnected graph G in $\mathcal{H}$ consists of (weakly or strongly connected) components $G_1, G_2, \ldots, G_t$, where $t \geq 2$ (note also that any component must have at least two nodes, to allow computation).

Lemma 2. *For any nontrivial graph language L, there exists some disconnected graph G in L where at least one component of G does not belong to L or there exists some disconnected graph G' in $\overline{L}$ where at least one component of G' does not belong to $\overline{L}$ (or both).*

Proof. Let L be such a nontrivial graph language and assume that the statement does not hold. Then for any disconnected graph in L, all of its components also belong to L and for any disconnected graph in $\overline{L}$, all of its components also belong to $\overline{L}$. There are two main cases:

1. L contains all connected graphs. But $\overline{L}$ is nontrivial which means that it must contain at least one disconnected graph. We know that for any disconnected graph in $\overline{L}$ all of its connected components belong to $\overline{L}$, but this is a contradiction, since all connected graphs belong to L.
2. L does not contain all connected graphs. There are now two possible subcases:
   (a) L contains at least one connected graph (but not all). This means that $\overline{L}$ contains also at least one connected graph. Let $G' = (V', E')$ be a connected graph from L and $G'' = (V'', E'')$ be a connected graph from $\overline{L}$. The disjoint union of G' and G'', $U = (V' \cup V^{''}, E' \cup E^{''})$ is a disconnected graph consisting of two connected components, one belonging to L and one to $\overline{L}$. U itself must belong in one of L and $\overline{L}$ implying that all of its components must belong to L or all to $\overline{L}$, which is a contradiction.
   (b) L contains no connected graph. Thus, since L is nontrivial, it contains at least one disconnected graph whose connected components belong to L. But all connected graphs belong to $\overline{L}$ which is a contradiction.

□

Theorem 10. *Any nontrivial graph language $L \subset \mathcal{H}$ is not stably decidable by the GDMPP model.*

Proof Idea. The proof of the result is based on the very simple observation that in disconnected graphs, the various components cannot communicate with each other. Then Lemma 2 can be used to argue that a language (or its complement) must contain at least one disconnected graph with a component not in the language, so any protocol making some decision on the whole graph would make the opposite decision on the component (since this component does not belong to the language and is isolated from the other components), which is contradictory. □

Proof. Let L be such a language and assume that there exists a GDMPP $\mathcal{A}_L$ that stably decides it. Thus, $\mathcal{A}_L$ has eventually all the agents of G giving output 1 if $G \in L$ and all giving output 0 if $G \notin L$. Moreover, the protocol $\mathcal{A}_{\overline{L}}$ that has the output map of $\mathcal{A}_L$ complemented stably decides $\overline{L}$. Those GDMPPs (and in fact any GDMPP) have no way to transmit data between agents of different components when run on disconnected graphs. In fact it is trivial to see that, when run on disconnected graphs, those protocols essentially run individually on the different components of those graphs. This means that when, for example, $\mathcal{A}_L$ runs on a disconnected graph G, where G has at least two components $G_1, G_2, \ldots, G_t$, then $\mathcal{A}_L$ runs in t different copies, one for each component, and each such copy stably decides the membership of the corresponding component (on which it runs on) in L. The same holds for $\mathcal{A}_{\overline{L}}$. By Lemma 2 there exists at least one disconnected graph in L with at least one component in $\overline{L}$ or at least one disconnected graph in $\overline{L}$ with at least one component in L. If L contains such a disconnected graph then, obviously, $\mathcal{A}_L$ when run on this graph, call it G, has eventually all the nodes of the component(s) in $\overline{L}$ giving 0 as output. This is a contradiction, because $G \in L$ and $\mathcal{A}_L$ stably decides L, which means that all agents should eventually output 1. If $\overline{L}$ contains such a disconnected graph then the contradiction is symmetric. □

As an immediate consequence we get the following corollary:

Corollary 1. *The graph language* $C = \{G \in \mathcal{H} \mid G \text{ is (weakly) connected}\}$ *is not GDMPP-decidable.*

Proof. C is a nontrivial graph language and Theorem 10 applies. □

6 Future Work

Whether the graph language $2C$ (Theorem 9) is not stably decidable by the GDMPP model in the general case remains an interesting open problem. If it were, we believe that proving the undecidability of many other properties, like kC (all graphs with at least one directed cycle of length k) and k-transivity, would become an easy next step. Our primary focus is to eventually provide a complete characterization of the class of stably decidable graph languages in the weakly-connected case. Moreover, consider the variant of the GDMPP model in which the communication graph G is always complete and an *edge initialization function* $\iota : E \rightarrow \{0, 1\}$ indicates a subgraph of G whose membership in a language is sought. What are the stably decidable graph languages here?

Acknowledgements

We would like to thank Elias Koutsoupias for bringing to our attention the importance of the connectivity property.

References

1. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In: Proc. Distributed Computing in Sensor Systems: 1st IEEE International Conference, pp. 63–74 (2005)
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: 23rd Annual ACM Symposium on Principles of Distributed Computing, PODC, pp. 290–299. ACM, New York (2004)
3. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing 18(4), 235–253 (2006)
4. Angluin, D., Aspnes, J., Eisenstat, D.: Fast computation by population protocols with a leader. Distributed Computing 21(3), 183–199 (2008)
5. Angluin, D., Aspnes, J., Eisenstat, D.: Stably computable predicates are semilinear. In: Proc. 25th Annual ACM Symposium on Principles of Distributed Computing, pp. 292–299 (2006)
6. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)
7. Aspnes, J., Ruppert, E.: An introduction to population protocols. Bulletin of the European Association for Theoretical Computer Science 93, 98–117 (2007); Mavronicolas, M. (ed.): Columns: Distributed Computing
8. Beauquier, J., Clement, J., Messika, S., Rosaz, L., Rozoy, B.: Self-stabilizing counting in mobile sensor networks. Technical Report 1470, LRI, Université Paris-Sud 11 (2007)
9. Bournez, O., Chassaing, P., Cohen, J., Gerin, L., Koegler, X.: On the convergence of population protocols when population goes to infinity. Applied Mathematics and Computation (2009) (to appear)
10. Chatzigiannakis, I., Dolev, S., Fekete, S.P., Michail, O., Spirakis, P.G.: Not all fair probabilistic schedulers are equivalent. In: 13th International Conference on Principles of DIstributed Systems (OPODIS), pp. 33–47 (2009)
11. Chatzigiannakis, I., Michail, O., Nikolaou, S., Pavlogiannis, A., Spirakis, P.G.: All symmetric predicates in NSPACE(n^2) are stably computable by the mediated population protocol model. FRONTS Technical Report FRONTS-TR-2010-17 (April 2010), http://fronts.cti.gr/aigaion/?TR=155
12. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Brief announcement: Decidable graph languages by mediated population protocols. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 239–240. Springer, Heidelberg (2009)
13. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Mediated population protocols. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 363–374. Springer, Heidelberg (2009)
14. Chatzigiannakis, I., Spirakis, P.G.: The dynamics of probabilistic population protocols. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 498–499. Springer, Heidelberg (2008)
15. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Ruppert, E.: When birds die: Making population protocols fault-tolerant. In: Proc. 2nd IEEE International Conference on Distributed Computing in Sensor Systems, pp. 51–66 (2006)

Broadcasting in Sensor Networks of Unknown Topology in the Presence of Swamping

Evangelos Kranakis and Michel Paquette

School of Computer Science, Carleton University, Ottawa, Ontario, Canada
kranakis@scs.carleton.ca, mic.paquette@gmail.com

Abstract. In this paper, we address the problem of broadcasting in a wireless sensor network under a novel communication model: the *swamping* communication model. In this model, nodes communicate only with those nodes at distance greater than s and at most r from them. We consider networks of unknown topology of diameter D, with a lower bound α on the geometric distance between nodes and a parameter $g = 1/\alpha$ (*granularity*). We present broadcast algorithms for networks of nodes placed on the line and on the plane with respective time complexities $O(D/l + g^2)$ and $O(Dg/l + g^4)$, where $l \in \Theta(\max\{(1-s), \alpha\})$.

Keywords: sensor network, broadcasting, unknown topology, faults, swamping.

1 Introduction

One of the known problems commonly faced by radio transceivers is that of *swamping* (cf., e.g., [1,2,10]). When two wireless nodes are at close proximity, their receivers cannot adapt to strong incoming signals; communication becomes difficult, even impossible. In contrast to traditional radio communication models, nodes at close proximity are not able to communicate directly; as many as $\Theta(D)$ intermediate nodes may be needed to relay their messages. Therefore, communication between nearby nodes can be very time intensive in the presence of swamping.

In this paper, we study analytically the problem of broadcasting in networks where nodes may be suffering from swamping. We propose broadcasting algorithms for this novel communication model which successfully broadcast in networks of unknown topology.

The paper is structured as follows: in Section 2, we define the model and the problem addressed; in Section 3, we discuss related work. We present fast broadcasting algorithms for unknown topology networks of nodes placed on the line (Section 4) and on the plane (Section 5). Because of space restrictions, most of the proofs are deferred to the full version of this paper.

2 The Model and Problem Definition

Typical wireless receivers are built from a radio-frequency amplifier, a demodulator and a decoder. The amplifier adapts the strength of the received signal such that it becomes usable for the demodulator stage. However, this amplifier is not ideal.

When the received signal strength is too low its output is either too weak or too noisy to be usable; the first situation occurs when the communication range of a receiver is exceeded, for instance. When the received signal strength is too high, its input stage becomes saturated leading to a distorted signal (cf., e.g., [11]); in this case, we say that the receiver is *swamped* (cf., e.g., [1,2,10]). This occurs when there is a radio transmitter which is too close to a receiver. We now propose our model for this fault phenomenon. In what follows, whenever we speak of the distance, it is meant in its geometric sense, unless otherwise mentioned.

We work in the swamping communication model. Our graphs are built from a set V of $|V| = n$ nodes, placed on the line (Section 4) or on the plane (Section 5). Nodes are equipped with communication range r and limited by a minimum distance requirement of s (the swamping distance). Two nodes $u, v \in V$ located at distance $dist(u, v)$ greater than s and at most r from one-another are neighbors and share an undirected link $(u, v) \in E$ in the graph G; no other links exist in G. In each round, each node is either a sender or a receiver. A node u which is a transmitter in a given round sends a message to the entire set of its neighbors $\Gamma(u)$ within the same round; this transmission also makes the receiving of messages impossible for all nodes within distance s. More formally, for each round when a node within distance s of it transmits, a node v receives no message; in this case, only noise is heard by v, indistinguishable from the background noise heard when no messages are sent. In a fixed round, a node v receives a message if and only if it is a receiver, exactly one of its neighbors is a sender, and no node within distance s sends a message. If no neighbor of v is a sender, then there is no message on the channel which v can receive. If more than one neighbor of v sends a message, we say that a *collision* occurs at v and v can only perceive noise on the channel. Nodes do not have collision detection abilities, i.e., they cannot distinguish collision noise from background noise (which is apparent when no messages are heard). For simplicity of presentation, we will let $r = 1$ throughout this paper.

The swamping communication model can be viewed as a GRN on which radio communication is implemented with additional transient reception faults on all nodes at close proximity of a transmitter, i.e., a node cannot receive messages at each round when some node within distance s from it transmits. Alternately, we can say that all *incoming* links of nodes at close proximity to a transmitter fail.

Throughout this paper, we study networks of *unknown topology* where nodes are placed on the line and on the plane. Nodes are *location-aware*, i.e., each node knows its own location with respect to a global reference, but all nodes are unaware of the location of any other node. We restrict attention to connected networks where nodes are positioned with some minimum distance $\alpha > 0$ between each other. This minimum distance may be due to physical constraints, such as the size of nodes. Let the parameter $g = 1/\alpha$ be called *granularity*.

Nodes are also aware of the parameter α, the *swamping distance* s, and the communication distance 1.

We consider the process of *broadcasting* in a network which can be described as follows: the process starts with a distinguished node, called *source*, which has a message to be sent to all other nodes. The message is sent on the network links such that it is eventually received by all nodes of the network. We consider the *spontaneous wake up* model in which all nodes are considered to be functional when the source begins transmission. Thus nodes may contribute to the broadcasting process even before receiving the source message, by exchanging control messages. The nodes have synchronized individual clocks that measure time steps; we call these time steps *rounds*. In the sequel, we consider that nodes execute algorithms in a synchronous way.

We consider deterministic algorithms without global knowledge (Section 4) and with some knowledge about messages received by nodes close by (Section 5). The algorithm is known to all nodes and its execution is based solely on the location of nodes in the network, the history known to each node, and the granularity $g = 1/\alpha$. In Section 5, the algorithm execution is also based on some information about messages received by nodes surrounding each node.

3 Related Work

Few results are known about fault-tolerant communication in geometric radio networks. In [8], the authors consider the problem of broadcasting in a fault-free connected component of a radio network whose nodes are located at grid points of square grids and can communicate within a square of size r. For an upper bound t on the number of faulty nodes, in worst-case location, the authors propose a $\Theta(D + t)$-time oblivious broadcast algorithm and a $\Theta(D + \log(\min(r, t)))$-time adaptive broadcast algorithm, both operating on a connected fault-free component of diameter D. More recently, in [9], an algorithm was demonstrated to broadcast correctly with probability $1 - \epsilon$ in faulty random geometric radio networks of diameter D, in time $O(D + \log 1/\epsilon)$.

The question of communication in networks of unknown topology has been widely studied in recent years. In [3], the authors state that broadcasting algorithms which function in unknown GRNs also function in the resulting fault-free connected components of faulty GRNs. A basic performance criterion of broadcasting algorithms is the time necessary for the algorithm to terminate; in synchronous networks, this time is measured as the number of communication rounds. For networks whose fault-free part has a diameter D, $\Omega(D)$ is a trivial lower bound on broadcast time, but optimal running time is a function of the information available to the algorithms (cf., e.g., [4]). For instance, in [4], an algorithm was obtained which accomplishes broadcast in arbitrary GRNs in time $O(D)$ under the assumption that nodes have a large amount of knowledge about the network, i.e. given that all nodes have a *knowledge radius* larger than R, the largest communication radius. The authors also show that algorithms broadcasting in time $O(D + \log n)$ are asymptotically optimal, for unknown GRNs when nodes communicate spontaneously and either can detect collisions or have knowledge of node locations at some positive distance ϵ, arbitrarily small.

In the present paper, we assume that nodes communicate spontaneously, but know nothing of the network, other than their own location, and cannot detect collisions. We propose algorithms to broadcast in networks of nodes placed on the line and on the plane under the swamping communication model. Contrary to the traditional radio communication model, it is not possible for nodes under the swamping communication model to directly receive messages from nodes located at close proximity to them. This contrasts sharply to the SINR model (studied by Wattenhoffer et al., in [5,6]) whereby a given station can communicate its message by increasing the power of the transmitted signal.

4 Highway Model

In this section, we analyze the problem of broadcasting along a line segment of length L where nodes are placed by an adversary. The adversary designs the network such that it is *connected* and the distance between any pair of nodes u, v is at least α. We say that a network is connected if, for any node pair u, v, there exists a path in the network from node u to node v. Observe that the network is connected only if $\alpha \leq 1$.

In this section, we present a broadcasting algorithm $\mathcal{B}$ and show the following result.

Theorem 1. *Algorithm $\mathcal{B}$ broadcasts a message m in a network of diameter D in time $O(D/l + g^2)$, where $l = \max\{(1-s), \alpha\}$.*

Before we are ready to prove the main result of this section, we need several preparatory lemmas.

Fact 1. *For any node u in a connected network, there is at least one node v within the set Γ_u.*

4.1 Partition $\mathcal{P}$ of the Line

We now define a partition, called $\mathcal{P}$, on which our communication algorithm will operate. For each line segment in the partition below, the segment includes its leftmost point and excludes its rightmost point so that there will be no intersection between adjacent segments. We provide a graphical representation of the partition in Figure 1 and describe it below in detail.

Partition the line into line segments of length 3, called *regions*. The line contains $\lceil L/3 \rceil$ regions, where $\lfloor L/3 \rfloor$ are of length 3 and at most one (the rightmost) is shorter, and even may consist of a single point.

Further, partition each region into smaller line segments, called *blocks*, of length $l = \max\{(1-s), \alpha\}$. Here, $l \leq 1$ since both $\alpha \leq 1$ and $1 - s \leq 1$. Each region contains $\mu = \lceil 3/l \rceil$ blocks, where $\lfloor 3/(1-s) \rfloor$ are of length l and at most

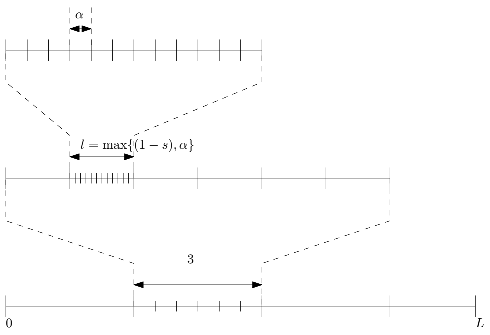


Fig. 1. Partition $\mathcal{P}$

one (the rightmost) is shorter, and even may consist of a single point. For each region, label blocks $1, 2, \ldots, \mu$, from left to right.

Partition also each block into line segments of length α, called *homes*. Each block contains $\nu = \lceil l/\alpha \rceil$ homes, where $\lfloor l/\alpha \rfloor$ are of length α and at most one (the rightmost) is shorter, and even may consist of a single point. For each block, label homes $1, 2, \ldots, \nu$, from left to right.

Partition Properties. We now show communication properties related to the partition defined above. We first show that transmissions in distinct regions do not collide, if properly scheduled. We then show that transmissions by a few distinguished nodes in a part or all of a block can reach all neighbors of nodes on this block or part of a block. However, before showing these properties, we observe that since homes are of length at most α, at most one node can occupy each home. Hence, we have the following lemma.

Lemma 1. *Each home contains at most one node.*

Lemma 2. *Transmissions from unique nodes inside identically labeled blocks in distinct regions do not collide.*

Proof (of Lemma 2). Consider nodes u, v in different regions and identically labeled blocks. Each region has length 3 and each block has length $l \leq 1$. Because the block labels are identical in each region, the minimum distance between two identically-labeled blocks (that contain the nodes u, v) is $3 - l \geq 2$. Since each line segment of the partition excludes its rightmost point, there is no point within distance 1 of both u and v. ∎

Lemma 3. *Consider any pair u, v of nodes within distance $1 - s$. Also consider the set $\mathcal{U}$ of all nodes inclusively located between u and v. We have that $\Gamma_{\mathcal{U}} = \Gamma_u \cup \Gamma_v$.*

Proof. Consider two nodes u, v at distance $d \leq 1 - s$ from one-another; u is to the left of v and u is at coordinate 0. Consider the right part of the range of u

and v (the argument for the left is symmetric). Then, the range of u to the right covers the interval $(s, 1]$. Similarly, the range of v to the right covers the interval $(s + d, 1 + d]$. Since $d \leq 1 - s$, we have that $s + d \leq s + 1 - s = 1$. Hence, the ranges of u and v overlap and cover the interval $(s, 1 + d]$.

Consider any node w between u and v, i.e., at distance d_w from u, with $0 < d_w < d$. The range of w to the right covers the interval $(s + d_w, 1 + d_w]$. Since $0 < d_w < d$, the range of w to the right is completely included in the ranges of u and v to the right. The argument is symmetric for the left. ■

In the following sections, we describe communication procedures that will enable nodes to broadcast messages to all nodes of their networks.

4.2 Neighborhood Discovery Procedure $\mathcal{D}^*$

We now define procedures used to communicate once from each node to all other nodes within distance 1 of them. We refer to this process as *Neighborhood Discovery.*

Procedure $\mathcal{D}$. We now present Procedure $\mathcal{D}$, in which nodes in distinct homes inside a region sequentially send a message while other nodes listen. This procedure is executed in parallel over all regions and for all $(block, home)$ labels sequentially. All homes with some $(block, home)$ label transmit a message while all other nodes listen for incoming messages. More formally, refer to the code for Procedure $\mathcal{D}$. By Lemma 2, no collision occurs in this procedure. Hence we claim

```
Procedure $\mathcal{D}$
  In parallel for all regions
  $N_u \leftarrow \emptyset$ // the set of nodes known to $u$
  $H_u \leftarrow$ the (block, home) label of $u$
  for $block = 1..\mu$ do
    for $home = 1..\nu$ do
      if $H_u = (block, home)$ then
        Transmit $hello$
      else if a $hello$ is heard then
        $N_u \leftarrow N_u \cup (block, home)$
```

that each node will gain knowledge of all nodes located within its communication range as a result of Procedure $\mathcal{D}$. We have the following lemma:

Lemma 4. *After one execution of Procedure $\mathcal{D}$, nodes know of all nodes within distance* 1 *and greater than s of them.*

By the previous lemma, since the graph is connected, each node will discover at least one node within its communication range by the end of procedure $\mathcal{D}$. We will use this fact in the following subsection to allow all nodes to discover all other nodes within distance s of them.

Procedure $\mathcal{D}^*$. Recall that, in our model, it is not possible for any node to hear messages from nodes within distance s of them. Observe that the length of the path between two nodes within distance s is only bounded above by the diameter D, in many cases. We now concentrate on a time-guaranteed procedure for discovery of nodes within distance 1.

We now consider Procedure $\mathcal{D}_{(b,h)}$ by which nodes use the absence of distinguishable messages from collisions to discover nodes within distance s of them; this collision-driven detection scheme was used in Procedure Echo from [7].

```
Procedure $\mathcal{D}_{(b,h)}$
  // $N_u$ is the set of nodes known to $u$
  // $H_u$ is the (block, home) label of $u$
  In parallel for all nodes $u \in V$
  for $block = 1..\mu$ do
    for $home = 1..\nu$ do
      if $H_u = (block, home)$ OR $H_u = (b, h)$ then
        Transmit hello
      else if no hello is heard AND $(b, h) \in N_u$ then
        $N_u \leftarrow N_u \cup (block, home)$
```

Lemma 5. *By Procedure $\mathcal{D}_{(b,h)}$, nodes neighbor to (b, h) know all other nodes within geometric distance 1 of them in time $\Theta(g)$.*

Proof. The time complexity of Procedure $\mathcal{D}_{(b,h)}$ is in $\Theta(\mu\nu)$. With $\alpha \leq l \leq 1$, we have that $\mu = \lceil 3/l \rceil \in \Theta(1/l)$ and $\nu = \lceil l/\alpha \rceil \in \Theta(l/\alpha)$. Hence,

$$\mu\nu \in \Theta((1/l)(l/\alpha)) = \Theta(1/\alpha) = \Theta(g).$$

We now prove correctness. Consider the execution of Procedure $\mathcal{D}_{(b,h)}$, during which the node (b, h) will transmit messages at every step. A message from (b, h) will be heard by u at every step when no collision occurs at u. Furthermore, when no message can be distinguished, another node within distance 1 of u must be transmitting from the home with label $(block, home)$ (as defined in the procedure). Since Procedure $\mathcal{D}_{(b,h)}$ schedules all nodes to transmit in pairs with (b, h), upon completion of this procedure, the node u will have discovered all nodes w for which the geometric distance $d_{u,w}$ from u is at most 1. ■

Now consider Procedure $\mathcal{D}^*$ consisting of one execution of Procedure $\mathcal{D}$ followed by the execution of Procedure $\mathcal{D}_{(b,h)}$ for all $(b, h) \in \{1, 2, \ldots, \mu\} \times \{1, 2, \ldots, \nu)\}$. In plain words, Procedure $\mathcal{D}^*$ schedules colliding transmissions for all $(block, home)$ couple pairs

$$((b, h), (b', h')) \in \{\{1, 2, \ldots, \mu\} \times \{1, 2, \ldots, \nu)\}\}^2.$$

More formally, refer to the pseudo code for Procedure $\mathcal{D}^*$.

```
**Procedure** $\mathcal{D}^*$
  **Call** Procedure $\mathcal{D}$
  **for** $b = 1..\mu$ **do**
    **for** $h = 1..\nu$ **do**
      **Call** Procedure $\mathcal{D}_{(b,h)}$
```

Lemma 6. *By Procedure $\mathcal{D}^*$, nodes know all other nodes within geometric distance* 1 *of them in time $\Theta(g^2)$.*

Proof. The time complexity of Procedure $\mathcal{D}_{(b,h)}$ is in $\Theta(\mu\nu)$. Hence, the time complexity of Procedure $\mathcal{D}^*$ is in $\Theta(\mu^2\nu^2)$. By the above and by Lemma 5, the time complexity of Procedure $\mathcal{D}^*$ is therefore in $\Theta(g^2)$.

We now prove correctness. By Fact 1, for any node u, since the graph is connected, there exists a node (b, h) such that Lemma 5 will hold. By the above and by Lemma 5, all nodes know all other nodes that are within distance 1 of them. ■

With knowledge of all nodes within distance 1, nodes have the basic tools to select distinguished nodes to relay messages for all nodes of a block. We discuss such a procedure in the following subsection.

4.3 Selection of Spokesman Nodes

We now describe a procedure for the selection of distinguished nodes for each block known as *spokesmen*. We wish to select these spokesmen in order to avoid collisions and speed up the broadcasting process. Before we concentrate on the different cases, we present the following fact.

Fact 2. *Given location-awareness, if a sender includes its location inside a message, then a receiver can determine all points where the message may be received.*

Proof. The sender knows its own location and therefore can incorporate this as part of his message. The receiver then knows the origin of the received message and hence can determine the covered region.

Consider the spokesman selection procedure that elects, for each block,

1. *right (left) boundary spokesmen*: the node in the rightmost (leftmost) home known to be completely contained within the transmission range of a sender, if this home is the rightmost (leftmost) home on the block;
2. *right (left) range spokesmen*: the node in the rightmost (leftmost) home known to be completely contained in the transmission range of a sender, if this home is not the rightmost (leftmost) home on the block;
3. *right (left) potential spokesmen*: the node in the rightmost (leftmost) home known to be partially contained within the transmission range of a sender.

We now show that the spokesman selection procedure making the above selections selects unique spokesmen for each type.

Lemma 7. *The spokesman selection procedure selects at most one node for each spokesman type.*

Proof. Given that right and left boundary spokesmen are unique by definition (those nodes in the home that is closest to the block boundaries), we prove the lemma for right and left range and boundary spokesmen. In the case when $l = \alpha$, there is only one home per block, hence the lemma holds in this case. We now prove the lemma for the case when $l = 1 - s$.

Given that the range of a node is of size $1 - s = l$, the range of a transmitter always encloses at least one of the homes that is closest to the block boundaries; call this home a boundary home. For any set $\mathcal{S}$ of transmitters whose ranges enclose a same boundary home, the intersection of their communication ranges with the block t defines a set $\mathcal{I}$ of intervals for which there is a largest interval. This largest interval is the communication range of a node $u \in \mathcal{S}$ that includes all other communication ranges inside of the set $\mathcal{I}$. By Fact 2, all nodes located inside this interval know the limits of the communication range of u. It follows that the potential and range spokesmen for the set of nodes $\mathcal{S}$ are unique. These spokesmen are right (left) potential and range spokesmen if the leftmost (rightmost) home of t is completely included in the range of u and not the rightmost (leftmost) home of t.

It also follows from the above discussion that for any pair of transmitters u and v whose ranges do not enclose a same boundary home, the spokesmen types defined will be different (right ws. left spokesmen).

4.4 Broadcasting Algorithm $\mathcal{B}$

We now describe Procedure $\mathcal{T}$ which uses the spokesmen to relay the messages and complete the broadcasting process. In the pseudo code for Procedure $\mathcal{T}$, time is reserved for all spokesmen, even those potential spokesmen that may not be actual spokesmen.

Procedure $\mathcal{T}$

```
$S_u$ ←the label of the block containing $u$
In parallel for all regions
repeat
   for $block = 1..\mu$ do
      if $S_u = block$ then
         update spokesman status
         Spokesmen transmit the message $m$ sequentially
      else
         Listen to incoming messages for 6 rounds
until no node has transmitted in an iteration
```

Lemma 8. *Procedure $\mathcal{T}$ broadcasts the message correctly through the network in time $O(D/l)$.*

Proof. Consider a network G of diameter D built by the adversary under the swamping model. Consider also the network G' with the same nodes and links as G, but where nodes may receive messages from multiple neighbors in one round without collisions. Let the nodes of G' execute the broadcasting algorithm $\mathcal{F}$: when a node receives a message m the first time, it transmits this message to all its neighbors the next turn. For the network G', the algorithm $\mathcal{F}$ executes in $\Theta(D)$ steps. We prove the lemma statement by comparing the execution of Procedure $\mathcal{T}$ on G to the execution of Algorithm $\mathcal{F}$ on G'.

Consider G and the partition $\mathcal{P}$. Since each region has $\lceil 3/l \rceil$ blocks, where $l = \max\{\alpha, (1-s)\}$ and since each block has a constant number of spokesmen, the broadcast algorithm sequentially makes all spokesmen of a region communicate every $\Theta(1/l)$ turns. From Lemma 2, the process is collision-free. From Lemma 3, the spokesmen of a block reach all the nodes that can be reached by any node on their block that do know the message m. It then follows that the message m being relayed through the network may be slowed down by a factor $O(1/l)$ with respect to the execution of Algorithm $\mathcal{F}$ in G'. Hence, for any network G of diameter D, the total transmission time is in $O(D/l)$. ■

```
Algorithm $\mathcal{B}$
  In parallel for all nodes
  Call Procedure $\mathcal{D}^*$
  Call Procedure $\mathcal{T}$
```

Proof (of Theorem 1). From Lemma 8, the time of execution of Procedure $\mathcal{T}$ is $O(D/l)$. From Lemma 6, the time of execution of Procedure $\mathcal{D}^*$ is $\Theta(g^2)$. Adding these times together, we get a total time of $O(D/l + g^2)$. ■

5 City Model

We now consider the task of broadcasting in a connected network of unknown topology. In particular, we consider networks of nodes placed on the plane, in which all nodes are located at least at some distance α from each-other. Each node u is equipped to communicate with all nodes that are both within distance 1 and at distance greater than some s from it. Hence, in this section, we assume that $r = 1$ for simplicity. More formally, the communication range of a node u is the annulus centered at u with radii s and 1. The adversary designs the network such that it is *connected.* The lower bound on the distance between any pair of nodes u, v is α. We say that a network is connected if, for any node pair u, v, there exists a path in the network from node u to node v. Observe that the network is connected only if $\alpha \leq 1$.

We wish to complete broadcasting in a collision avoidance scheme. We will use the assumption of spontaneous wake-up of the nodes, i.e., we assume that all nodes are awake at the beginning of the broadcasting process and may execute some preprocessing in order to speed up the broadcasting process.

In this section, we will assume that nodes know about the transmissions made within close proximity. We will show the following result.

Theorem 2. *Algorithm $\mathcal{B}^2$ broadcasts a message m in a network of diameter D in time $O(Dg/l + g^4)$, where $l = \max\{(1-s)/(3\sqrt{2}), \alpha/\sqrt{2}\}$.*

5.1 Partition $\mathcal{P}^2$ of the Plane

We now define a partition, called $\mathcal{P}^2$, on which our communication algorithm will operate.

Each square in the partition below includes its North border, its West border, and both its North vertices; it excludes its East border, its South border and both its South vertices. We provide a graphical representation of the partition in Figure 2 and now describe it below.

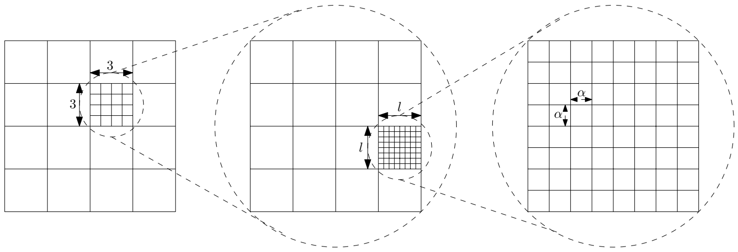


Fig. 2. Partition $\mathcal{P}^2$: from left to right, the plane is partitioned into 3×3 squares called regions; for $l = \max\{(1-s)/(3\sqrt{2}), \alpha/\sqrt{2}\}$ regions are partitioned into $l \times l$ squares called blocks; blocks are partitioned into $\alpha/\sqrt{2} \times \alpha/\sqrt{2}$ squares called homes

Partition the plane into a mesh of 3×3 squares called *regions*.

Further partition each region into a mesh of $l \times l$ squares, called *blocks*, with length $l = \max\{(1-s)/(3\sqrt{2}), \alpha/\sqrt{2}\}$. Here, $l \le 1/\sqrt{2}$ since both $\alpha \le 1$ and $(1-s)/3 \le 1$. Each region contains $\mu = \lceil 3/l \rceil^2$ blocks, where $\lfloor 3/l \rfloor^2$ are of area l^2 and at most $2\lfloor 3/l \rfloor + 1$ are smaller, and even may consist of a single line or point. For each region, label blocks $1, 2, \ldots, \mu$, from West-East row by row, North to South.

Partition also each block into a mesh of $\alpha/\sqrt{2} \times \alpha/\sqrt{2}$ squares, called *homes*. Each block contains $\nu = \lceil \sqrt{2}l/\alpha \rceil^2$ homes, where $\lfloor \sqrt{2}l/\alpha \rfloor^2$ are of area $\alpha^2/2$ and at most $2\lfloor \sqrt{2}l/\alpha \rfloor + 1$ are smaller, and even may consist of a single line or point. For each block, label homes sequentially $1, 2, \ldots, \nu$, from West-East row by row, North to South.

Partition Properties. In section 4, we showed properties for the partition $\mathcal{P}$. We now show the validity of lemmas 1 and 2 for the partition $\mathcal{P}^2$. For ease of reading, we now repeat these lemmas.

Lemma 1. *Each home contains at most one node.*

Observe that since homes have diameter at most α, at most one node can occupy each home. Hence, Lemma 1 holds for partition $\mathcal{P}^2$.

Lemma 2. *Transmissions from unique nodes inside identically labeled blocks in distinct regions do not collide.*

Consider nodes u, v in different regions and identically labeled blocks. Since each region has side length 3 and each block has side length $l \leq 1$, Lemma 2 holds for $\mathcal{P}^2$.

In the following sections, we describe communication procedures that will enable nodes to broadcast messages to all nodes of their networks.

5.2 Procedure $\mathcal{D}$ for Nodes in Range

Recall Procedure $\mathcal{D}$ in which nodes send a message sequentially, based on the value of their home label. Since Lemma 1 and Lemma 2 both hold for $\mathcal{P}^2$, we have that Lemma 4 also still holds for $\mathcal{P}^2$.

Lemma 4. *Upon completion of Procedure $\mathcal{D}$, nodes know of all nodes within distance 1 and greater than s of them.*

5.3 Procedure $\mathcal{D}^*$ for Neighborhood Discovery

Recall Procedure $\mathcal{D}_{(b,h)}$ by which nodes use collisions to discover nodes within distance s of them.

By a proof similar to that of Lemma 5, we can show Lemma 9. The difference in time complexity is caused by the dimensionality of the environment.

Lemma 9. *By Procedure $\mathcal{D}_{(b,h)}$, nodes neighbor to (b, h) know all other nodes within geometric distance 1 of them in time $\Theta(g^2)$.*

Recall Procedure $\mathcal{D}^*$ consisting of one execution of Procedure $\mathcal{D}$ followed by the execution of Procedure $\mathcal{D}_{(b,h)}$ for all $(b, h) \in \{1, 2, \ldots, \mu\} \times \{1, 2, \ldots, \nu)\}$. For the plane, Procedure $\mathcal{D}^*$ allows the discovery of nodes within distance 1.

By a proof similar to that of Lemma 6, we can show Lemma 10. The difference in time complexity is caused by the dimensionality of the environment.

Lemma 10. *By Procedure $\mathcal{D}^*$, nodes know all other nodes within geometric distance 1 of them in time $\Theta(g^4)$.*

With knowledge of all nodes within distance 1, nodes have the basic tools to select distinguished nodes to relay messages for all nodes of a block. We discuss such a procedure in the following section.

5.4 Selection of Spokesman Nodes

In this section, we assume that nodes know which nodes of their own block possess the source message m. The spokesmen nodes are those nodes in each row, column and diagonal of homes within a block which possess the message and which are located in the home which is closest to either end of that row, column or diagonal. We state Lemma 11, the main result of this section.

Lemma 11. *If all spokesmen of a block b transmit in a collision-avoidance scheme, then all nodes neighbor to any node in b will receive the source message.*

The proof will be given following some preliminary facts and discussion. More formally, the rules for deciding which nodes are spokesmen are as follows: For a row (column) i of homes of partition $\mathcal{P}^2$, among nodes possessing the message, those two nodes in homes closest to the West and East (North and South) borders of a block in $\mathcal{P}^2$ are spokesmen, respectively labeled W_i and E_i (N_i and S_i). For a Southeast-Northwest (Southwest-Northeast) diagonal i of homes of partition $\mathcal{P}^2$, among nodes possessing the message, those two nodes in homes closest to the borders of a block in $\mathcal{P}^2$ are spokesmen, respectively labeled SE_i and NW_i (SW_i and NE_i). Spokesmen can be assigned more than one label. We now claim that only these spokesmen are necessary to broadcast.

First, recall Fact 3.

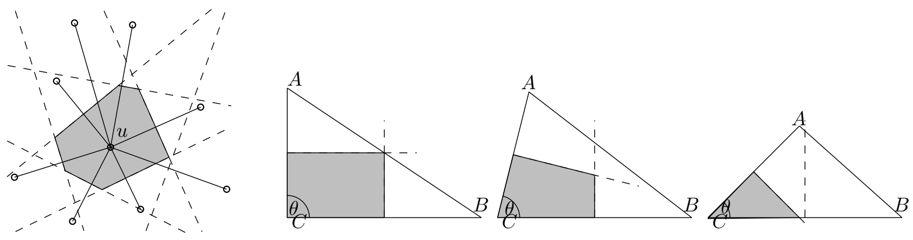
Fact 3. *Consider two vertices A and B and the line $\overline{AB}$ joining them. The line l perpendicular to $\overline{AB}$ and through its center defines two halfplanes $H_{A,B}$ and $H_{B,A}$. The halfplane $H_{A,B}$ (resp. $H_{B,A}$) contains A (B) and has all points closer to A (B) than to B (A).*

We now proceed to the presentation of Lemma 12 in preparation of the proof to Lemma 11.

Lemma 12. *For any point p and any non-spokesman node u, there is always a spokesman node v that is farther from p than u. The set of spokesmen of a block is closer to any point p outside the block than any non-spokesman node.*

Proof. We prove the first statement. Consider a non-spokesman node u and the set of all spokesmen in its row, column and diagonals of homes within its block. For u not to be a spokesman, it must have one spokesman on each side of itself for its row, column and diagonals. Let these spokesmen be labeled sequentially $u_1, u_2, \ldots, u_8$ in a clockwise order around the node u. Recall Fact 3. Consider all halfplanes H_{u,u_i}, $i = 1, 2, \ldots, 8$. These halfplanes contain all those points to which u is closer than u_i, or those from which u_i is farther than u. Since the distance between nodes is at least α and because of the geometry of the partition, we have that the angle of each sector $u_i\, u\, u_{(i+1) \bmod 8}$ is less than $\pi/2$. Therefore, the union of these halfplanes covers the entire plane. This proves the statement. See Figure 3a.

We prove the second statement. Consider the halfplanes defined by the vertex pairs u, u_i and u, u_{i+1} as described in Fact 3. If the node u is closer than u_i and u_{i+1} to a point p, then p is in the intersection of H_{u,u_i} and $H_{u,u_{i+1}}$. Moreover, if $\theta = \pi/2$, then $S_i \cap H_{u,u_i} \cap H_{u,u_{i+1}}$ is a rectangle contained within the triangle $u_i u u_{i+1}$. As θ decreases, the region $S_i \cap H_{u,u_i} \cap H_{u,u_{i+1}}$ remains contained within the triangle $u_i u u_{i+1}$. See Figure 3b. It follows that all other points of S_i are closer to either u_i or u_{i+1}. Hence, the node u is farther from any point in a S_i, and outside the triangle $u_i u u_{(i+1) \bmod 8}$, than the spokesmen u_i and $u_{(i+1) \bmod 8}$. Since the spokesmen are part of the block, the triangle is contained within the block, proving the statement. See Figure 3a. ■



(a) sectors $u_i\, u\, u_{i+1}$ **(b)** triangle $ACB = u_i u u_{i+1}$

Fig. 3. For all points outside of the gray region, there is always a spokesman that is closer than u. For all points, there is always a spokesman that is farther than u.

We are now ready to prove the main lemma of this section.

Proof (of Lemma 11). Fix a node u inside block b. Fix a neighbor v of u. If u is a spokesman, we are done. Otherwise, show that there is a spokesman w that shares a link with v.

Let $d_{u,v}$ be the distance from u to v. If u is not a spokesman, then from Lemma 12 there is a spokesman w that is closer to v than u and there is a spokesman w' that is farther. For some δ, w is at distance $d_{u,v} - \delta < d_{v,w} < d_{u,v}$ of v and w' is at distance $d_{u,v} < d_{v,w'} < d_{u,v} + \delta$ from v. Since u shares a link with v, we know that $s < d_{u,v} < 1$. Moreover, for $\delta < (1-s)/2$, either $s < d_{u,v} - \delta < 1$ or $s < d_{u,v} + \delta < 1$. Since the diameter of a block is $(1-s)/3 < (1-s)/2$, at least one of w and w' shares a link with v. ■

5.5 Broadcasting Algorithm $\mathcal{B}$

Recall Procedure $\mathcal{T}$ and Algorithm $\mathcal{B}$. Procedure $\mathcal{T}$ is executed in parallel for all regions. Sequentially for all blocks, we have the set of spokesmen transmit the message m on a turn basis. Spokesmen send the message only once each and the procedure ends implicitly when the last message is sent.

By a proof similar to that of Lemma 8, we can show Lemma 13. The difference in time complexity is caused by the dimensionality of the environment.

Lemma 13. *Procedure $\mathcal{T}$ broadcasts the message correctly through the network in time $O(Dg/l)$.*

Proof (of Theorem 2). From Lemma 13, the time of execution of Procedure $\mathcal{T}$ is $O(Dg/l)$. From Lemma 6, the time of execution of Procedure $\mathcal{D}^*$ is $\Theta(g^4)$. Adding these times together, we get a total time of $O(Dg/l + g^4)$. ■

Acknowledgements

Many thanks to Ioannis Lambadaris for suggesting the swamping paradigm and useful conversations on this topic as well as to Andrzej Pelc for useful conversations on broadcasting in networks. Research supported in part by NSERC and MITACS grants.

References

1. Berg, P.: Dual Conversion Receivers Are Better Than Single Conversion Receivers ...Fact or Fiction? (2002), http://www.bergent.net/SC-DC.pdf (accessed 19/03/2010)
2. Industry Canada. Spectrum management and telecommunications: Report on the national antenna tower policy review (July 2009), http://ic.gc.ca/eic/site/smt-gst.nsf/eng/sf08347.html (accessed 19/03/2010)
3. Clementi, A.E.F., Monti, A., Silvestri, R.: Round robin is optimal for fault-tolerant broadcasting on wireless networks. J. Par. Distrib. Comp. 64, 89–96 (2004)
4. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. Journal of Discrete Algorithms 5, 187–201 (2007)
5. Goussevskaia, O., Moscibroda, T., Wattenhofer, R.: Local broadcasting in the physical interference model. In: DIALM-POMC 2008: Proceedings of the Fifth International Workshop on Foundations of Mobile Computing, pp. 35–44. ACM, New York (2008)
6. Goussevskaia, O., Oswald, Y.A., Wattenhofer, R.: Complexity in geometric SINR. In: MobiHoc 2007: Proceedings of the 8th ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 100–109. ACM, New York (2007)
7. Kowalski, D., Pelc, A.: Deterministic broadcasting time in radio networks of unknown topology. In: Proc. The 43rd Annual IEEE Symposium on Foundations of Computer Science, pp. 63–72 (2002)
8. Kranakis, E., Krizanc, D., Pelc, A.: Fault-tolerant broadcasting in radio networks. Journal of Algorithms 39, 47–67 (2001)
9. Kranakis, E., Paquette, M., Pelc, A.: Communication in random geometric radio networks with positively correlated random faults. In: Coudert, D., Simplot-Ryl, D., Stojmenovic, I. (eds.) ADHOC-NOW 2008. LNCS, vol. 5198, pp. 108–121. Springer, Heidelberg (2008)
10. Radiocontact Ltd. Wireless transmission product installation guide cct2240, http://www.radcon.com/pdfs/m_cct2440.pdf (accessed 19/03/2010)
11. Sedra, A.S., Smith, K.C.: Microelectronic Circuits, 4th edn. Oxford University Press, Oxford (1998)

Brief Announcement: Configuration of Actuated Camera Networks for Multi-target Coverage

Matthew P. Johnson[1], Amotz Bar-Noy[1], and Mani Srivastava[2]

[1] City University of New York Graduate Center
[2] University of California at Los Angeles

In various domains, including public safety, first-responder, and security applications, an important task is monitoring a public space for events of interest, using sensors of various types, including e.g. networks of cameras installed on the ground or unmanned aerial vehicles (UAVs), which may be either autonomous or not. In the settings we consider here, cameras are characterized by a family of parameters, some fixed, some settable, including location, viewing range, and so on. The task is to deploy the sensors, i.e., set the applicable parameters, in order to optimize an objective function capturing the quality with which we observe a set of targets. In a dynamic scenario, we may wish to observe a large set of targets (or a continuous region) with observation quality passing some low threshold, sufficient for surveillance; upon request (i.e., when events are detected), we may then be obliged to observe a small number of distinguished targets to a higher level of quality, sufficient for identification and localization. An example objective function might maximize the total observation quality of all targets, conditioned on the hard constraint that each target is observed to the appropriate minimum quality threshold.

In full generality, the configuration of a camera will be characterized by the following tuple: $< x, y, z, \theta, \phi, f >$. The first three entries indicate the camera's location in space. The targets will lie in the plane, but in some cases, as with UAVs or cameras deployed atop buildings, cameras may look down from on high. The fourth and fifth are angles indicating orientation. Cameras are directional, and may swivel (pan ϕ); in the case of cameras positioned above the ground, there is a second orientation degree of freedom (tilt θ). Finally, the last indicates the camera's focal length or zoom factor. By shrinking a camera's focal length (and thus enlarging its field of view), we allow the camera to observe more target locations, at the cost of correspondingly reducing the quality of the resulting image; conversely, by zooming in on a small area, the camera will record better quality images of the targets therein, while sacrificing targets lying outside this area. (Ideally, these targets will be covered by other cameras.) We now enumerate some example settings lying within this framework, in order of increasing generality:

– $x, y, z > 0, \theta, \phi$ fixed; f variable: UAVs hovering at fixed locations, facing downwards (predepoyed disc-shaped sensors); for each sensor, we choose the radius.

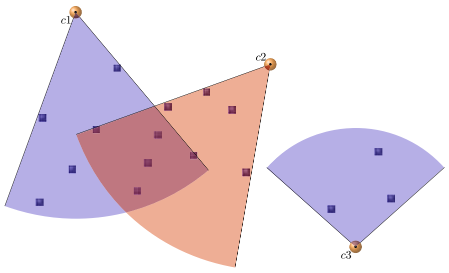


Fig. 1. Three cameras configured to observe many targets

- $x, y, z = 0, \theta = 90^o$ fixed; ϕ, f variable: ground-based cameras looking horizontally (predepoyed cone-shaped sensors); for each sensor, we choose the angle ϕ and focal length f.
- x, y, z fixed; θ, ϕ, f variable: For each sensor, we also now choose orientation. Viewing regions are now conic sections.
- x, y, z, θ, ϕ, f variable: We now may also choose the sensor locations.

There are orthogonal options vis-a-vis mobility in the case of UAVs. Such vehicles may be deployable at will to arbitrary locations, or, if the UAVs are obliged to fulfill other responsibilities or, say, to maintain a certain topology, they may be limited to certain regions. Alternatively, the UAVs may be cycling over certain trajectories autonomously over time.

In the remainder of this announcement we describe a specific geometric coverage problem inspired by an application involving cameras and targets, corresponding to the first special case listed above: cameras deployed to observe targets in the field may both swivel and adjust their focal length in order to observe one or more targets. That is, for each camera we choose an orientation and a zoom factor (x, y, z, θ are fixed, with $z = 0$). The objective, informally speaking, is then configure the cameras in order to observe as many targets as possible, with the highest possible precision.

Motivation and model. More formally, given are n camera locations in the field and m target locations to observe. (See Figure 1.) For each camera there are two parameters to set: the viewing direction ϕ and the viewing angle ψ, which assignment will allow the camera to observe all targets in the cone defined by angles $\phi - \psi$ and $\phi + \psi$ and the camera position P. The quality of the a camera c's observation of a visible target t depends on both the distance between c and t and on ψ. The specific deterioration function may vary by camera hardware, but for a camera producing rectilinear images, the field of view is proportional to the distance between camera and scene, specifically: $\frac{o}{d} = \frac{i}{f}$. Here f is the focal length, i is the (constant) image size, and d is the distance. In this case, the object dimension, i.e., the field of view, is $o = di/f$. An equation of this form holds in both horizontal and vertical dimensions. We are interested in the image quality, which then (for a camera s and a target t) depends on both d and f: $u(s, t) = f^2/d(s, t)^2$. That is, the observation quality varies inversely with the distance squared and directly with focal length squared. Increasing the focal length, however, decreases field of view $o = di/f$. Thus there is a tradeoff between the angle of the viewing cone and the quality. Narrowing the cone thus extends the range of imaging at a given quality farther out in the cone.

With this tradeoff between imaging quality and scope in place, we can now define an optimization problem. Given a set of target locations and a set of placed cameras, the goal is to configure cameras so as to maximize the total sensing quality. How to encode the assignments? Each possible direction/angle assignment for a camera will cover some *contiguous* subset of the targets. Although there are infinitely many possible angle/focal length choices, only a finite, polynomial number of such choices need be considered. For each camera, we can restrict our attention to its $O(m^2)$ possible *pinned cones*, which are camera configurations in which a camera's cone-shaped viewing range intersects a target on both its boundaries (possibly the same target), which allows us to obtain discrete algorithms that space limitations do not permit us to describe.

Related work. The camera configuration problem can be understood as a variant of the Maximum Coverage problem, a dual problem to Set Cover, in which, given a set system and an integer bound k, the goal is to choose at most k sets to cover a maximum-weight set of elements. A hardness of approximation $1 - 1/e$ lower bound is known for this problem [3]. Other related problems include the Multiple-Choice Knapsack problem, the Budgeted Maximum Coverage problem, and the Generalized Maximum Coverage problem [2]. Recent work treating related but *importantly different* coverage problems include [1,4,5].

Acknowledgements. This research was sponsored by US Army Research Laboratory and the UK Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the US Government, the UK Ministry of Defence, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

1. Berman, P., Jeong, J.K., Kasiviswanathan, S.P., Urgaonkar, B.: Packing to angles and sectors. In: SPAA, pp. 171–180 (2007)
2. Cohen, R., Katzir, L.: The generalized maximum coverage problem. Inf. Process. Lett. 108(1), 15–22 (2008)
3. Feige, U.: A threshold of ln for approximating set cover. J. ACM 45(4), 634–652 (1998)
4. Fusco, G., Gupta, H.: Selection and orientation of directional sensors for coverage maximization. In: IEEE SECON, pp. 1–9 (2009)
5. Fusco, G., Gupta, H.: Placement and orientation of rotating directional sensors. In: IEEE SECON (2010)

Brief Announcement: On the Hardness of Topology Inference

H.B. Acharya[1] and Mohamed Gouda[1,2]

[1] Department of Computer Science
University of Texas at Austin
[2] National Science Foundation
{acharya,gouda}@cs.utexas.edu

Abstract. Many systems require information about the topology of networks on the Internet, for purposes like management, efficiency, testing of new protocols and so on. However, ISPs usually do not share the actual topology maps with outsiders. Consequently, many systems have been devised to reconstruct the topology of networks on the Internet from publicly observable data. Such systems rely on traceroute to provide path information, and attempt to compute the network topology from these paths. However, traceroute has the problem that some routers refuse to reveal their addresses, and appear as anonymous nodes in traces. Previous research on the problem of topology inference with anonymous nodes has demonstrated that it is at best NP-complete. We prove a stronger result. There exists no algorithm that, given an arbitrary trace set with anonymous nodes, can determine the topology of the network that generated the trace set. Even the weak version of the problem, which allows an algorithm to output a "small" set of topologies such that the correct topology is included in the solution set, is not solvable: there exist trace sets such that any algorithm guaranteed to output the correct topology outputs at least an exponential number of networks. We show how to construct such a pathological case even when the network is known to have exactly two anonymous nodes.

1 Introduction

Several systems have been developed to reconstruct the topology of networks in the Internet from publicly available data - [4], [3]. In such systems, Traceroute [2] is executed on a node, called the source, by specifying the address of a destination node. This execution produces a sequence of identifiers, called a *trace*, corresponding to the route taken by packets traveling from the source to the destination. For example, a trace may be (a, b, c, d, e) where a, b, c, d, e are the unique identifiers (IP addresses) of nodes in the network. We assume a trace is undirected, that is, traces (a, b) and (b, a) are equivalent.

A trace set T is generated by repeatedly executing Traceroute over a network N, varying the source and destination. The problem of reconstructing the topology of the network which generated a trace set, given the trace set, is the *network tracing problem.*

In our earlier work [1], we studied network tracing for the special case where no node is consistently anonymous, but nodes may be *irregular* - anonymous in some traces, but not in others. In this paper, we extend the theory of network tracing to networks with consistently anonymous nodes.

Clearly, a trace set is generable from a network only if every trace in the trace set corresponds to a path in the network. However, if we define this as sufficient, it is trivial to see that a trace set is generable from many different networks. For example, $T = \{(a, *_1, b), (a, *_2, b)\}$ is generable from any network with nodes a, b, and at least one anonymous node between a and b.

To mitigate this problem, we add two conditions:

– Complete coverage. Every node and every edge must appear in the trace set.
– Consistent routing. For every two distinct nodes x and y, if x and y occur in two or more traces in T, then the exact same set of nodes occur between x and y in every trace in T where both x and y occur.

However, if we modify the trace set slightly to $T' = \{(a, *_1, b), (a, *_2, c)\}$, we see that T' can still be generated from two networks - one where $*_1$ and $*_2$ are the same anonymous node, and one where they are distinct. To solve this problem, we add the assumption that the network that generated a trace set is *minimal*: it is the smallest network, measured by node count, from which the trace set is generable. This forces $*_1$ and $*_2$ to be the same node.

Clearly, these are strong assumptions; it may be argued that in practical cases, both consistent routing and the assumption of minimality may fail. However, even under these assumptions, is the network tracing problem solvable?

Unfortunately, we show in the following section that the answer is negative.

2 The Hardness of Network Tracing

We demonstrate how to construct a trace set which is generable from an exponential number of networks with two anonymous nodes, and no networks with one or fewer anonymous nodes.

It is of interest to note that our results are derived under a network model with multiple strong assumptions (stable and symmetric routing, unique identifiers, and complete coverage). As no algorithm can solve the minimal network tracing problem, even under the friendly conditions of our model, we conclude the problem resists the strongest known network tracing techniques (such as Paris Traceroute to detect artifact paths, and inference of missing links).

We begin by constructing a very simple trace set with only one trace,

$$T_{0,0} = \{(a, *_1, b_1)\}$$

Next, we introduce a new b-node b_2, which is connected to a through an anonymous node $*_2$. To ensure that $*_2$ is not a previously seen anonymous node, we add the trace $(b_1, *_3, a, *_4, b_2)$. By consistent routing, $*_1 = *_3$ and $*_2 = *_4$, but $*_1 \neq *_2$. (Note that consistent routing precludes routing loops. As $*_3$ and $*_4$ occur in the same trace, they cannot be the same node.)

$$T_{1,0} = \{(a, *_1, b_1), (a, *_2, b_2), (b_1, *_3, a, *_4, b_2)\}$$

We now define operation "*Op*2". In *Op*2, we introduce a new non-anonymous node (c_i). We add traces such that c_i is connected to a through an anonymous node, and is directly connected to all b and c nodes.

A single application of *Op*2 to the trace set $T_{1,0}$ produces the trace set $T_{1,1}$ given below.

$$T_{1,1} = \{(a, *_1, b_1), (a, *_2, b_2), (b_1, *_3, a, *_4, b_2), \\ (a, *_5, c_1), (b_1, c_1), (b_2, c_1)\}$$

Now, we apply *Op*2 l times. Every time we apply *Op*2, we get a new anonymous identifier. This new identifier can correspond to a new node or to a previously-seen anonymous node. As we are considering only minimal networks, we know that this is a previously-seen anonymous node. But there are 2 such nodes to choose from ($*_1$ and $*_2$), and no information in the trace set to decide which one to choose. Furthermore, each of these nodes is connected to a different (non-anonymous) b-node, and is therefore distinct from all other anonymous nodes; in other words, each choice will produce a distinct topology.

Hence the number of minimal networks from which the trace set $T_{1,l}$ is generable, is 2^l. As the total number of nodes in a minimal network is n, we also have $n = l + 4$. Thus the number of networks from which $T_{1,l}$ is generable, is exponential in n. An algorithm given this trace set must necessarily output this exponential-sized solution set to ensure it reports the correct topology. Exactly which topology actually generated the trace set cannot be determined.

References

1. Acharya, H.B., Gouda, M.G.: A theory of network tracing. In: 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (November 2009)
2. Cheswick, B., Burch, H., Branigan, S.: Mapping and visualizing the internet. In: Proceedings of the USENIX Annual Technical Conference, Berkeley, CA, USA, pp. 1–12. USENIX Association (2000)
3. Jin, X., Yiu, W.-P.K., Chan, S.-H.G., Wang, Y.: Network topology inference based on end-to-end measurements. IEEE Journal on Selected Areas in Communications 24(12), 2182–2195 (2006)
4. Yao, B., Viswanathan, R., Chang, F., Waddington, D.: Topology inference in the presence of anonymous routers. In: Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies, March 3, vol. 1, pp. 353–363. IEEE, Los Alamitos (April 2003)

Self-stabilizing Algorithm of Two-Hop Conflict Resolution

Stéphane Pomportes[1], Joanna Tomasik[2], Anthony Busson[1],
and Véronique Vèque[1]

[1] Université de Paris-Sud, Institut d'Électronique Fondamentale, Bâtiment 220,
91405 Orsay Cedex, France
{stephane.pomportes,anthony.busson,veronique.veque}@u-psud.fr
[2] SUPELEC Systems Sciences (E3S)
Computer Science Department
91192 Gif sur Yvette, France
joanna.tomasik@supelec.fr

Abstract. Ad hoc networks are increasingly used by the civil protection forces to coordinate their actions in emergency situations. To enable them to work properly, the satisfaction of the demanded quality of service (QoS) is crucial. One of the most effective methods of assuring the QoS is to use multiplexing modes based on a resource reservation like TDMA or FDMA. The principal demands in terms of QoS concern a guarantee of connectivity and resource availability. Our idea consists in the separation of the interference detection mechanism in order to make it independent of the *pure* resource allocation algorithm. We present an algorithm which detects and corrects conflicts of assignment. Our algorithm is proved to be self-stabilizing and reaches a stable state in up to five rounds.

1 Introduction

Ad hoc networks are increasingly used, particularly by the civil protection forces (medical staff, firemen, policemen, etc.) to coordinate their actions on emergency response. To enable them to work properly, the satisfaction of the demanded quality of service (QoS) is crucial. One of the most effective methods of assuring the QoS is to use multiplexing modes based on a resource reservation like TDMA or FDMA. However, the problems related to the QoS such as traffic management, interference management, and assignment, remain open problems.

We are interested in resource allocation in wireless networks within the context of the RAF project which is also one of the projects of SYSTEM@TIC PARIS-REGION[1] competitiveness cluster. This project focuses on ad hoc networks deployed in disaster scenes. The nodes of such networks are rescue workers. The principal demands in terms of quality of service concerns a guarantee

[1] SYSTEM@TIC PARIS-REGION and the RAF project are supported by the French Ministry of Industry, the department Essonne, the department Haut-de-Seine and the Paris General Council.

of connectivity and resource availability. The robustness of the networks is a crucial aspect of the allocation mechanism. At the same time the mechanism of resource allocation has also to cope with a possible interference (a resource cannot be assigned if it interferes with another resources). An assignment may be in an incoherent state, i.e. in which some resources interfere one with another due to arriving nodes, leaving nodes, nodes' mobility or allocation errors. These conflicts must be detected and corrected, and may lead to a reallocation.

Our idea consists in the separation of the interference detection mechanism in order to make it independent of the *pure* resource allocation algorithm. We provide the justification of the proposed approach in Section 2. In Section 3 we formalize our approach in terms of models of connectivity, interference and communication, and present our algorithm of conflict detection in detail. We also deal with the definition of the self-stabilization property. Section 4 is the main section of our paper, where we describe a state automata model together with its requirements (safety, liveliness). The last subsection is dedicated entirely to proofs of self-stabilization, safety, and liveliness of the proposed mechanism. Section 5 contains conclusions and outlines perspectives.

2 Related Work and Justification of Our Approach

Allocation strategies are generally classified according to the underlying multiplexing methods. These methods, in terms of resource allocation, consist in cutting the radio medium into elementary resources. Whether these resources are frequencies (as in FDMA), or timeslots (as in TDMA) does not change the principle of resource allocation which can be seen as a classical graph coloring problem in many cases. Therefore, also in terms of resource allocation, many multiplexing methods can be grouped under a common model. Consequently, as a criterion, multiplexing is not adapted for classification of allocation strategies. In contrast, the modeling of the network has a significant impact on the design of algorithms which allocate resources. Most models found in the existing publications can be defined by: the entity in which the resource allocation has place, the nature of the links, and the size of the interference area compared to the transmission area. The resources can be assigned to nodes [1,8,9,12,14] or to links [13,15,10,11,5]. In the former case, a node can use its resources to communicate with all or a subset of its neighbors. Such an assignment is especially well suited for a broadcast communication. In the latter case, each resource is assigned to a specific link. The authors of [7] show that this assignment achieves a higher spatial reuse of resources than a node assignment. Communication between two nodes can be unidirectional or bidirectional. It is modeled either by directed links [15,10,11,5] or undirected links [13,1,8,9,12,14]. In directed links, the transmitters and receivers are clearly identified while in undirected links each node of a link can be both a transmitter or a receiver. A directed link also allows for a better modeling of asymmetric communications. The definition of the area of interference is also important. If it assumed that the interference zone its equals to the transmission zone, as in [13,1,9,8,12,14], then each transmitter can communicate with the receiver with which it interferes. But if we consider that a signal may be too weak to be received, but still strong enough to interfere with another signal, we must consider that the interference area is bigger than the communication area, see [15,10,11,5].

Some previous studies proposed to separate problems of conflict detection and allocation of resources. The authors of [1] have tried this solution for a resource assignment in a node with a conflict area and a transmission area of the same size. In their paper, all nodes have a synchronized execution, which is split into two parts. In the first one, each node tries to allocate resources concurrently and then sends a message to its neighbors announcing its new state. In the second part, which begins only when all messages from the neighborhood have been received, each node reads concurrently its own allocation and the allocations of its neighbors. If a conflict is detected, it will be corrected during the next execution. The article [2] offers an improvement of the described approach by taking a conflict area twice as large as the transmission area. Unlike the article [1] where the conflicts occurs only between two neighboring nodes, the article [2] takes into account a mechanism for relaying information for the detection of a two-hop conflict.

We agree with the authors of these two articles and believe that the detection of conflicts should be separated from resource allocation. Based on this separation we can study and optimize each of these problems separately. We can also study the efficiency of allocation heuristics which are built according to the same acting mechanism to detect conflicts. Nevertheless, contrary to the articles [1,2], we are interested here in an allocation of resources on the unidirectional links. The communications are modeled by unidirectional links. In our model, a conflict may occur between a transmitter and any receiver which is one or two hops away. To detect and correct conflicts of allocation, our algorithm broadcasts information concerning the presence of transmitter or receiver within two hop distance. Specifically, each node has a set of variables, one per slot, called a *channelState*. Each of these variables may take one of the eight different values. Each of these eight states will be described in detail in Section 4. Due to the self-stabilizing property which we will prove in Section 6, our algorithm allows the network to achieve a steady state regardless of the initial state of the network nodes.

3 Model and Assumptions

Network modeling: We assume that there is a contention-based channel like CSMA/CA allowing nodes to communicate even when TDMA or FDMA channels are unavailable.

The network is modeled by a directed graph $G = (V, E)$ where V is a set of nodes and E is a set of directed edges. This graph is called the *connectivity graph* in the rest of the paper. Each node models a communication unit and each arc, (t, r), represents a unidirectional link between a transmitter, $t \in V$ and its associated receiver, $r \in V$. In this paper, we consider only connected graphs. We consider that there is a set of channels H attributed to each node from V. In our model a channel can represent either a frequency or a timeslot.

Interference modeling: Assuming that a signal may be too weak to be received, but still strong enough to interfere with another signal, we consider that a transmitter can interfere with all its two-hop receivers. We say that two nodes are two hops away if they are not neighbors but have at least one neighbor in common. To model interference, we define an undirected graph $G_C = (V_C, E_C)$ which consists of set of vertices V_C and edges E_C. The set V_C has a one to one relation with the set edge E of the graph $G = (V, E)$; i.e. for each edge e $\in$ E, there exists a corresponding vertex $v_c \in V_C$. We call this graph a *conflict graph.* An edge (e_1,e_2) from E_C exists only if the transmitter from an arc associated with one end of this edge and the receiver from the arc of the other end are nodes which are two hops away. Since each resource is supposed to be independent, we defined a set of conflict graphs $G_{Ci} = (V_{Ci}, E_{Ci})$, one graph for each channel i of H. The set V_{C_i} represents the subset of elements of V_C which use the channel i. The set E_{C_i} is obviously the subset of edges from E_C between two elements of V_{C_i}.

State model: The state model which is used to write the algorithm presented in the next subsection, is a shared memory system model where all communications are abstract. It means that there is no direct message exchange between nodes. Each node executes the same program defined as a set of variables and guarded actions. In our algorithm, there is one variable per channel for each node. Each of these variables represents the current state of the associated channel. Each node can read its own variables and those of its one-hop neighbors. However, it can only change the value of its own variables.

A guarded action has the form $<$ *label* $>::<$ *guard* $>\rightarrow<$ *statement* $>$. A label is used to identify a guarded action. A guard is a predicate involving both the local variables and the variables of neighbors. A statement is a set of instructions operating only on the local variables. When a guard holds, the corresponding action is said to be *enabled.* When a node has, at least, one enabled action it is said to be *enabled.* An *activation* of a node consists in executing simultaneously instructions of all enabled actions of the node. In this paper, we focus on an assignment of resources on the links, but since the nodes execute the algorithm, the variables representing the resources need to be located on nodes.

The *state* of a node is defined by the value of all its variables. The set of the states of all nodes of a system at a given time is called *a configuration.* An execution of a system is the sequence of configurations c_1,..., c_n where c_{i+1} is obtained from configuration c_i after the activation of at least one node. To evaluate the time complexity of the algorithm, we use the notion of a *round*, defined in [3,6] as the minimal subset of configurations in which each enabled node is activated at least once.

Self-stabilization: As most systems, a network may be subject to faults like message loss or node breakdown. When such events occur, the algorithms must allow the system to continue operating properly. This property is called *fault-tolerance*. For distributed systems, Dijkstra introduced in [4] the notion of self-stabilization as the ability of a system to converge in a finite time to a legitimate state when faults occurred.

4 Conflict Resolution Algorithm

4.1 Presentation

The algorithm presented below (Algorithm 1) focuses on the correction of conflicts for one channel (for instance, a timeslot for a TDMA multiplexing, a frequency for FDMA). If we assume that all channels are independent, the execution of the algorithm for each resource corrects all conflicts in a network. This independence of resources is generally verified in the case of TDMA when the synchronization is sufficiently strong to avoid any overlap between consecutive slots. In the FDMA context, the independence of resources is assured when the frequencies are orthogonal.

Table 1. Description of variable *channelState*

State	Scope	Meaning
T	local	slot locally used for transmission
R	local	slot locally used for reception
N_R	one hop	at least one neighbor uses the slot for reception
N_{T1}	one hop	at least one neighbor uses the slot for transmission
N_{TR}	one hop	slot used by at least one transmitter neighbor and by exactly one receiver neighbor
N_{TR+}	one hop	Slot used by at least one transmitter neighbor and by more than two receiver neighbors
N_{T2}	two hops	Slot used by a transmitter which is two hops away
F	—	There is no transmitter at distance two and no receiver in neighborhood

Our algorithm uses a node variable called *channelState* which represents the state of the considered channel. This variable can be in one of the eight different states whose meaning is detailed in Table 1. In Figure 1 we illustrate the way in which a value is chosen for a given variable state. This choice is made according to the state of the neighbors' variables. In this figure, the solid lines indicate a condition which has to exist in the neighborhood. The crossed lines indicate a condition which is forbidden in the neighborhood. The dotted lines indicate a condition that is not necessary but which is not conflictual. The arrows indicate that a transmitter and a receiver have to be associated.

For example, consider the case of a transmitter (Fig. 1, upper left corner). In the corresponding scheme in Figure 1, an arrow, starting from the transmitter and ending in the receiver, indicates that the transmitter has to be associated with a receiver present in its neighborhood. To avoid interference, the considered

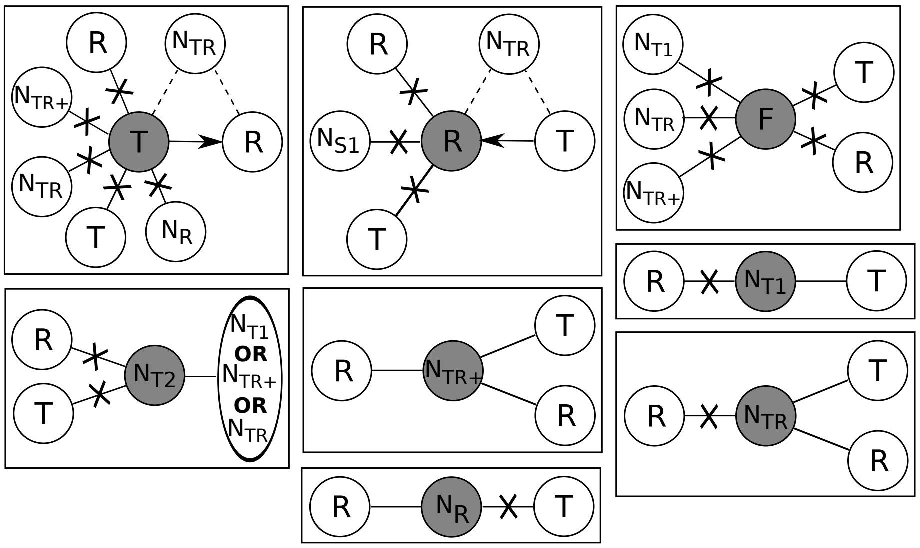


Fig. 1. Interactions between neighbors

transmitter and any other receiver should not be one or two hops away. Thus the neighborhood of the transmitter should be free of nodes in state R (one-hop conflict), T (two-hop conflict between the studied transmitter and the receiver associated with the node in the state T), N$_R$ and N$_{TR+}$ (indicating the presence of receivers two hop aways). The case of N$_{TR}$ is special, this state indicates that there is a single receiver in the neighborhood of the node in N$_{TR}$ state. This single receiver may be the associated receiver of the considered transmitter, in which case there is no conflict. This situation is represented by dotted lines between the node in N$_{TR}$ state and both, the studied transmitter and its associated receiver. But the single receiver may be another receiver in which case there is a conflict. Thus the neighborhood of the transmitter should also be free of nodes in N_{TR} state. All these forbidden situations are represented by crossed lines between the transmitters and the nodes in states R, T, N$_R$, N$_{TR}$, N$_{TR+}$. There are six different actions in the algorithm which can be use to update variables of a node(see Algorithm 1). Each of these actions toggles the values of a *channelState* to $\{N_R, N_{TR}, N_{TR+}, N_{T1}, N_{T2}, F\}$. Since the allocation should be realized by another algorithm, none of the actions of our algorithm is allowed to switch the value of *channelState* to T or R.

If we follow only the definitions given in Table 1, we may enable several values of *channelState* simultaneously. For example, a node may be adjacent to a receiver and two hops away from a transmitter, yet the state of the slot cannot be N$_R$ and N$_{T2}$ simultaneously. This kind of situation is managed by the predicates of the algorithm designed to introduce a priority between the different *channel-State* values. This priority follows two simple rules. The information about the neighborhood ($\{N_R, N_{TR}, N_{TR+}, N_{T1}\}$) as a higher priority over other (N_{T2}, F)

and the priority increases with the amount of information carried by the value of the state ($N_{TR+} > N_{TR} > N_{T1} \geq N_R$ et $N_{T2} > F$).

4.2 Specifications

Here we consider that a system is steady if there is no enabled node in the system. Our algorithm must verify the following properties:

Safety:

1. There is no conflict in a steady system.
2. In a steady system, any node P in V ::$S_P \in \{$F,N$_{T2}\}$ and without neighbour in state N_R, can establish a link to any neighbor in state F without creating conflict.

Liveliness:

1. The steady state of any system is reached in up to five rounds whatever the connectivity graph, the value of *channelStates*, and the size of the network.

4.3 Proof of Specifications

In this section, we show that our algorithm detects and corrects all conflicts in up to five rounds. Due to the self-stabilizing property of our algorithm, the variables may be chosen arbitrarily. We assume that the topology of the network does not change. We also assume there is no resource allocation before the end of the five rounds. Let us not forget that when a transmitter communicates with a receiver we consider that they are *associated.* A transmitter and a receiver which do not communicate with each other are called *dissociated.* An one-hop conflict occurs when a transmitter and a receiver, which are dissociated, are neighbors. A two-hop conflict occurs when a transmitter and a receiver, which are dissociated, are not neighbors but have at least one neighbor in common.

Lemma 1. *At least one of the following predicates* − Pre-NeighRcv$_p$, Pre-1HopNeighTrans$_p$, Pre-2HopNeighTrans$_p$, Pre-NeighTR$_p$, Pre-NeighTR+$_p$, Pre-Free$_p$ − *holds*

Proof. Let us assume the contrary, i.e. there exists a node A for which none of these predicates holds.

Since the connectivity graph $G = (V, E)$ is connected, A has at least one neighbor B. If B is a receiver and A has neither other neighbors nor neighbor whose *channelState* is in T state, predicate *Pre-NeighRcv$_p$* is satisfied. If B is a receiver and A has at least one neighbor whose *channelState* is in T state but no other receiving neighbor (*channelState*$= R$), the predicate *Pre-NeighTR$_p$* is satisfied. If B is a receiver, and A has at least one transmitting neighbor (*channelState*$= T$) and at least one receiving neighbor (*channelState*$= R$), the predicate *Pre-NeighTR+$_p$* is satisfied.

Algorithm 1. Conflict detection and correction

```
Variables :
$StatesSet = \{T, R, N_R, N_{TR}, N_{TR+}, N_{T1}, N_{T2}, F\}$
 $S_p \in StatesSet$ : $channelState$ of node p.
 $Neigh_p$            : set of neighbors of p.
 $OtherEnd(p)$        : the node associated to p.

Predicates :
R-Isolated$_p$          $\equiv S_p = R \wedge (\nexists q \in Neigh_p :: (S_q = T) \wedge (q = OtherEnd(p))$
T-Isolated$_p$          $\equiv S_p = T \wedge (\nexists q \in Neigh_p :: (S_q = R) \wedge (q = OtherEnd(p))$
T-1HopInterf$_p$        $\equiv S_p = T \wedge (\exists q \in Neigh_p :: S_q = R \wedge q \neq OtherEnd(p)))$
R-1HopInterf$_p$        $\equiv S_p = R \wedge (\exists q \in Neigh_p :: S_q = T \wedge q \neq OtherEnd(p)))$
T-2HopInterf$_p$        $\equiv S_p = T \wedge (\exists q \in Neigh_p :: (S_q \in \{T, N_{TR+}, N_R\})$
                                   $\vee(S_q = N_{TR} \wedge OtherEnd(p) \notin Neigh_q))$
R-2HopInterf$_p$        $\equiv S_p = R \wedge \exists q \in Neigh_p :: (S_q = R)$
IncorrectT$_p$          $\equiv S_p = T \wedge (T\text{-}Isolated_p \vee T\text{-}1HopInterf_p \vee T\text{-}2HopInterf_p)$
IncorrectR$_p$          $\equiv (R\text{-}Isolated_p \vee R\text{-}1HopInterf_p \vee R - 2HopInterf_p)$

Pre-NeighRcv$_p$        $\equiv ((\exists q \in Neigh_p :: S_q = R) \wedge (\nexists q' \in Neigh_p :: S_{q'} = T))$
Pre-1HopNeighTrans$_p$  $\equiv ((\exists q \in Neigh_p :: S_q = T) \wedge (\nexists q' \in Neigh_p :: S_{q'} = R))$
Pre-2HopNeighTrans$_p$  $\equiv ((\exists q \in Neigh_p :: S_q \in \{N_{T1}, N_{TR}, N_{TR+})$
                          $\wedge(\nexists q' \in Neigh_p :: S_{q'} \in \{T, R\}))$
Pre-NeighTR$_p$         $\equiv ((\exists! q_R \in Neigh_p :: S_{q_R} = R) \wedge (\exists q_T \in Neigh_p :: S_{q_T} = T))$
Pre-NeighTR+$_p$        $\equiv (\exists (q_T, q_{R1}, q_{R2}) \in Neigh_p^3 :: (S_{q_{R1}} = S_{q_{R2}} = R) \wedge (S_{q_T} = T))$
Pre-Free$_p$            $\equiv (\nexists q \in Neigh_p :: S_q \in States \backslash \{N_R, N_{T2}, F\})$

NeighRcv$_p$            $\equiv Pre\text{-}NeighRcv_p \wedge (IncorrectT_p \vee IncorrectR_p$
                                          $\vee S_p \in States \backslash \{T, R, N_R\})$
1HopNeighTrans$_p$      $\equiv Pre\text{-}1HopNeighTrans_p \wedge (IncorrectT \vee IncorrectR_p$
                                          $\vee S_p \in States \backslash \{T, R, N_{T1}\})$
2HopNeighTrans$_p$      $\equiv Pre\text{-}2HopNeighTrans_p \wedge (IncorrectT \vee IncorrectR_p$
                                          $\vee S_p \in States \backslash \{T, R, N_{T2}\})$
NeighTR$_p$             $\equiv Pre\text{-}NeighTR_p \wedge (IncorrectT \vee IncorrectR_p$
                                          $\vee S_p \in States \backslash \{T, R, N_{TR}\})$
NeighTR+$_p$            $\equiv Pre\text{-}NeighTR+_p \wedge (IncorrectT \vee IncorrectR_p$
                                          $\vee S_p \in States \backslash \{T, R, N_{TR+}\})$
Free$_p$                $\equiv Pre\text{-}Free_p \wedge (IncorrectT \vee IncorrectR_p$
                                          $\vee S_p \in States \backslash \{T, R, F\})$

Actions :
$N_R\text{-}Action_p$     :: $NeighRcv_p$          $\longrightarrow$   $S_p := N_R$
$N_{T1}\text{-}Action_p$  :: $1HopNeighTrans_p$    $\longrightarrow$   $S_p := N_{T1}$
$N_{T2}\text{-}Action_p$  :: $2HopNeighTrans_p$    $\longrightarrow$   $S_p := N_{T2}$
$N_{TR}\text{-}Action_p$  :: $NeighTR_p$           $\longrightarrow$   $S_p := N_{TR}$
$N_{TR+}\text{-}Action_p$ :: $NeighTR+_p$          $\longrightarrow$   $S_p := N_{TR+}$
$F\text{-}Action_p$       :: $Free$                $\longrightarrow_p$ $S_p := F$
```

If B is a receiver and A has no other receiving neighbor (*channelState*$= T$), the predicate *Pre-1HopNeighTrans*$_p$ is verified. If B is a receiver and A has exactly one receiving neighbor, the predicate *Pre-NeighTR*$_p$ is verified. If B is a receiver and A has at least two receiving neighbors, the predicate *Pre-NeighTR+*$_p$ is verified. If B is one of the following N_{T1}, N_{TR} or N_{TR+} states and A has no neighbor which is either a transmitter or a receiver, *Pre-2HopNeighTrans*$_p$ holds. If the channel State of all the neighbors of A is either N_{T2} or F, the predicate*Pre-Free*$_p$ holds. So whatever the neighborhood of A, there is always at least one of the predicates − *Pre-NeighRcv*$_p$, *Pre-1HopNeighTrans*$_p$, *Pre-2HopNeighTrans*$_p$, *Pre-NeighTR*$_p$, *Pre-NeighTR+*$_p$, *Pre-Free*$_p$− holds. □

Lemma 2. *There is at most one enabled action in any node.*

Proof. There are at least two concurrent enabled action iff their guards hold concurrently. In our algorithm, at least two guards hold concurrently if at least two predicates from the set P ={*Pre-NeighRcv*$_p$, *Pre-1HopNeighTrans*$_p$, *Pre-2HopNeighTrans*$_p$, *Pre-NeighTR*$_p$, *Pre-NeighTR+*$_p$, *Pre-Free*$_p$} hold concurrently. Let us assume the contrary, i.e. at least two predicates of the previous set hold concurrently.

We show here explicitly the first two steps (for the predicates *Pre-NeighRcv* and *Pre-1HopNeighTrans*) we leave the remaining three to be completed by a reader following the schema we outline.

Let us assume the predicate *Pre-NeighRcv* is one of the predicates which belongs to P and which holds concurrently with the other predicates from P. A receiving neighbor is required in order to satisfy the predicate *Pre-NeighRcv*. This, however, is forbidden in the predicates *Pre-1HopNeighTrans*, *Pre-2Hop NeighTrans*, and *Pre-Free*. Conversely, a transmitting neighbor is forbidden in the predicate *Pre-NeighRcv*, but is required to satisfy the predicates *Pre-NeighTR* and *Pre-NeighTR+*. Thus *Pre-NeighRcv* cannot be true if one of the other predicates holds.

Let us assume the *Pre-1HopNeighTrans* is a predicate from P which may be enabled concurrently with other predicates from P. A transmitting neighbor is required in the predicate *Pre-1HopNeighTrans* but is forbidden in the predicates *Pre-NeighRcv*, *Pre-2HopNeighTrans*, and *Pre-Free*. Conversely, a receiving neighbor is forbidden in the predicate *Pre-1HopNeighTrans*, but is required in the predicates *Pre-NeighTR* and *Pre-NeighTR+*. Thus *Pre-NeighRcv* cannot hold if one of the other predicates hold. □

Proposition 1. *All one-hop conflicts are corrected at the end of the first round.*

Proof. Let us assume the contrary, i.e. at the end of the first round a conflict exists between a transmitter A_T and one of its neighbor, a receiver B_R (Fig. 2). According to the algorithm, there is no action which allows us to change a *channelState* to R or T. Consequently, these two nodes are already in the state T for A_T and R for B_R at the beginning of the first round. Since nodes A_T and B_R are dissociated, the predicates *T-1HopInterf*$_{A_T}$ and *R-1HopInterf*$_{B_R}$ hold. If these predicates hold, the predicates *IncorrectT*$_{A_T}$ and *IncorrectR*$_{B_R}$ will be valid too. According to Lemma 1 and 2, nodes A_T and B_T have one and only one enabled action

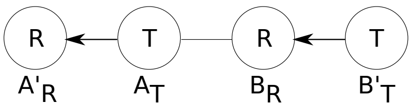


Fig. 2. One-hop conflict

at the beginning of the first round. Since during a round, each enabled node is activated, the *channelState* of node A_T (respectively B_R) cannot be T (respectively R) at the end of the first round. □

Proposition 2. *All two-hop conflicts are corrected at the end of the second round.*

Proof. Let us assume the contrary, i.e. at the end of the second round a two-hop conflict exists between a transmitter A_T and a receiver B_R. Since this is a two-hop conflict, there exists at least one node, C_i, which is a neighbor of A_T and B_R (Fig. 3). We will show that the presence of C_i, regardless of the value of its *channelState* variable, allows us to detect and correct the conflict between A and B before the end of the second round.

Let us assume that C_i is a receiver or a transmitter and it is dissociated from A_T or B_R. In this case, there is a one-hop conflict between C_i and A_T or B_R, which should have been corrected at the end of the first round (see proposition 1).

Let us assume that C_i is the receiver associated with A_T (i.e. C_i is in R state). In this case, the receivers B_R and C_i are neighbors. We know that in a correct state each receiver has an associated transmitter. Consequently, when two receivers are neighbors they are in conflict with each other's associated transmitter. When this occurs the predicate *R-2HopInterf*, and therefore the predicate *IncorrectR* , of each receiver hold. Since *IncorrectR* is valid for both receivers there is one and only one enabled action for these two nodes at the beginning of the first round. Consequently, none of the *channelState* values of these two nodes is in R state at the end of the first round.

Let us assume that C_i is the transmitter associated with B_R (i.e. C_i is in T state). For the same reasons as previously mentioned, there is a conflict between C_i and A_T which is detected by the predicate *T-2HopInterf* of each transmitter. Using the same mechanism, these two transmitters are enabled at the beginning of the first round. Consequently, neither of the *channelState* variables of these two nodes is in T state at the end of the first round.

Let us now consider the other cases in which $S_{C_i} \notin \{T, R\}$.

Let us assume that C_i has no other receiver than B_R in its neighborhood. Since C_i has a single receiver and at least one transmitter in its neighborhood, the predicate *Pre-NeighTR*$_{C_i}$ holds. Consequently, the *channelState* of C_i is in N_{TR} state at the end of the first round. At the beginning of the second round, since the node A_T has a neighbor whose *channelState* is in N_{TR} state and the receiver associated with A_T is not in C_i neighborhood, the predicate *T-2HopInterf*$_{A_T}$ holds. Therefore, *IncorrectT*$_{A_T}$ holds and the node A_T is enabled. At the end of the second round, A_T cannot be in state T.

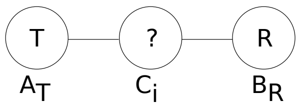


Fig. 3. Two-hop conflict

Let us assume that C_i has another receiver than B_R in its neighborhood. Since C_i has at least two receivers and one transmitter in its neighborhood, its predicate *Pre-NeighTR+* holds. Consequently, the *channelState* of C_i is in N_{TR+} state at the end of the first round. Since the node A_T has a neighbor whose *channelState* is in N_{TR+} state, the predicate *T-2HopInterf*$_{A_T}$ holds. Therefore, *IncorrectT*$_{A_T}$ holds and the node A_T is enabled at the beginning of the second round. The node A_T cannot be in the state T at the end of the second round.

We have shown that whatever the value of the *channelState* variable of C_i, the conflict between A_T and B_R is detected and corrected before the second round ends. □

Proposition 3. *The sets of nodes whose channelState is R or T is steady at the end of the second round.*

Proof. We consider here that a set is steady iff no element can join or leave this set. We call E_X a set of nodes whose *channelState* is in X state.

We consider that if a transmitter or a receiver does not have an associated node, they are *isolated*. For example, let us assume a transmitter A associated with the associated receiver B. If the *channelState* of B changes, due to a conflict detection, A will become a transmitter without an associated receiver.

Thanks to Proposition 1 we know that all one-hop conflicts are corrected before the end of the first round. To correct a one-hop conflict, the algorithm changes the *channelState* of one or both nodes which are in conflict. These corrected nodes may be associated with other nodes, which find themselves isolated. Such a situation is detected by the predicates *R-Isolate*, in the isolated receiver, and/or *T-Isolate*, in the isolated transmitter. Since these predicates hold, *IncorrectR* or *IncorrectS* hold too. Consequently, these isolated nodes are enabled at the beginning of the second round. These isolated nodes in turn cannot be in state R nor T at the end of the second round. Proposition 2 shows that two-hop conflicts are resolved at the end of the second round. The correction of these two-hop conflicts brings up isolated nodes. For the same reasons as one-hop conflicts, these isolated nodes are corrected before the end of the third round.

Since all conflicts are corrected at the end of the second round, the receivers and transmitters which are not isolated cannot be enabled after this second round. Since all isolated receivers and transmitters are corrected before the end of the third round, all the receivers or transmitters in the network are associated. At the end of the third round, there is no receiver and transmitter that can be enabled. The sets E_R and E_T are steady. □

Proposition 4. *All nodes in state N_R, N_{TR}, N_{TR+} or N_{T1} are steady, at the end of the fourth round.*

Proof. First, we consider the set E_{N_R} of nodes whose *channelState* is N_R. We know that a node can join this group during the fifth round, only if the action N_R-*Action* is enabled during the fourth one. For this, it is necessary that the predicate *Pre-1HopNeighRcv* holds. This predicate depends only on nodes belonging to sets E_T and E_R. According to Proposition 3, these sets are steady at the end of the third round. Consequently, if the *Pre-1HopNeighRcv* holds during the fourth round, it already held at the beginning of this fourth round. In such a case, the node must change its *channelState* value before the end of the fourth round.

The proof is the same for the other sets $E_{N_{TR}}, E_{N_{TR+}}$ and $E_{N_{T1}}$. Consequently, a node cannot join the sets $E_{N_R}, E_{N_{TR}}, E_{N_{TR+}}$ and $E_{N_{T1}}$ after the end of the fourth round.

Let us now assume that a node A leaves the set E_{N_R} during the fifth round. In such a situation A would have had to be activated during the fourth round. We have just shown that A cannot join the sets $E_{N_{TR}}, E_{N_{TR+}}$ and $E_{N_{T1}}$. Neither can it join the sets E_T and E_R, which have been already steady since the end at the third round. The node A can join only the sets E_F of EN_{T2}. Let us assume that the node A can leave the set E_{N_R} and join the sets EN_{T2} or E_F. This can be done only if the receivers in neighborhood of A disappear during the fourth round. According to Proposition 3 this is impossible because the set E_R has been steady since the third round. The node A can join neither the set EN_{T2} nor the set E_F.

The node A cannot join another set and consequently, it cannot leave the set E_{N_R} after the end of the fourth round. The proof is the same for the other sets $E_{N_{TR}}, E_{N_{TR+}}$ and $E_{N_{T1}}$. The nodes can neither leave nor join the sets $E_{N_R}, E_{N_{TR}}, E_{N_{TR+}}$ and $E_{N_{T1}}$ after the end of the fourth round, as a result these sets are steady. □

Proposition 5. *All nodes in state N_{T2} or F are steady at the end of the fifth round.*

Proof. Let us assume the contrary, i.e. a node A can join or leave the sets E_F or $E_{N_{T2}}$ during the sixth round. If the *channelState* of A changes during the sixth round, one of its actions must be enabled during the fifth one. According to Propositions 3 and 4, the sets of nodes E_T, E_R, E_{N_R}, $E_{N_{TR}}$, $E_{N_{TR+}}$ and $E_{N_{T1}}$ are steady at the end of the fourth round. Consequently, node A cannot join or leave these sets during the sixth round. The node A can only leave the set E_F to join the set $E_{N_{T2}}$ and vice versa.

Let us assume A leaves the set E_F to join the set $E_{N_{T2}}$ during the sixth round. In such a case, the predicate *Pre-2HopNeighTransp*$_A$ must hold during the fifth round. Since this predicate depends only on nodes in the sets which are already steady at the end of the fourth round, *Pre-2HopNeighTransp*$_A$ holds at the beginning of the fifth round. Consequently, A can only join the set $E_{N_{T2}}$ before the end of the fifth round. Now, let us assume A leaves the set $E_{N_{T2}}$ and joins the set E_F. Since the predicate *Pre-Free*$_A$ depends only on nodes in the sets that are already steady at the end of the fourth round, the proof is the same as in the previous case. Finally, the nodes can neither leave nor join the sets E_F

and $E_{N_{T2}}$ during the sixth round. These sets are steady at the end of the fifth round. □

Proof of liveliness: A steady state of any network is reached in up to five rounds whatever the connectivity graph, values of* channelState *and the size of the network.
According to Propositions 3, 4 and 5, all sets of nodes are steady at the end of the fifth round. Consequently by the end of the fifth round, the all network is steady.

Proof of safety 1: there is no conflict in a steady network.
If a network is steady, the *channelState* of any node cannot be changed. According to Propositions 1 and 2, if there is a conflict it will be detected and corrected. Since the correction needs to change the *channelState* of nodes, a network cannot be steady if there is a conflict.

Proof of safety 2: In a steady system, any node P, in V::$S_P \in \{$F,N$_{T2}\}$ and without a neighbour in state N_R, can establish a link to any neighbor in state F without creating conflict.
Let the two nodes A_T and B_R belong to the set E_F. The neighborhood of A_T is free of nodes whose *channelState* is N_R. Let us assume that a conflict occurs when a channel is assigned from A_T to B_R in a steady system.

Let us assume this is a one-hop conflict. In such a case, B_R is a neighbor of a transmitter or A_T is a neighbor of a receiver. Since the system is steady the *channelState* variables of this node cannot be in F state. A one-hop conflict cannot occur.

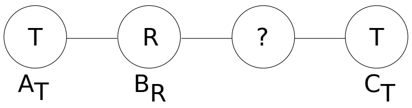


Fig. 4. A_T is responsible for the conflict

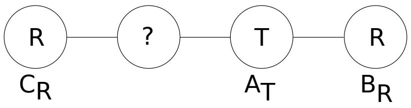


Fig. 5. B_R undergoes the conflict

Let us assume there is a two-hop conflict:

– If A$_T$ is responsible for the conflict, then there is a neighbor C$_R$ in R state two hops away from A$_T$ (Fig. 4). In this case, the neighbors of these two nodes would be in the state N$_R$. According to our assumptions this is impossible.

– If B_R undergoes the conflict, then there is a neighbor C_T, in T state, two hops away from B_R (Fig. 5). In this case, the neighbors of these two nodes would be in the state N_{T1}. This is impossible because in a steady network, the *channelState* variable of B_R cannot be in state F if one of its neighbor has the *channelState* variable in the N_{T1} state(*Action* N_{T2}-*Action*$_p$ should be enabled).

One-hop and two-hop conflicts are impossible, the second safety rule applies.

5 Conclusion

We propose a conflict resolution algorithm for ad hoc networks. The algorithm works with networks with unidirectional links, assuming that the interference range is twice as large as the communication range. We prove this algorithm reaches a steady state in which there is no conflict, in a maximum of five rounds. We also prove that each node, which wants to be a transmitter, can identify the subset of the neighbors with which it can establish a connection. This property is only true if the system is stable. Since we have created a conflict correction algorithm, our future work will focus on creating resource assignments algorithms. These algorithms are expected to be simpler than those proposed in existing studies where assignment and resolution of conflicts are treated simultaneously. We expect that we will be able to evaluate the performance of the strategies of each of these resource allocating algorithms, regardless of the conflict detection and correction mechanism. Another problem arises when there is a change in a topology (for instance, agents' movements). It would be interesting to prove or validate by simulations that despite the agents' mobility, our algorithm resolves conflicts locally, in a two-hop neighborhood. For certain applications which require frequent broadcast communications one might recommend the resource allocation in a node. In this case it would be interesting to propose an appropriate algorithm of two-hop conflict resolution and to verify whether it is self-stabilizing.

References

1. Boman, E., Bozdağ, D., Catalyurek, U., Gebremedhin, A., Manne, F.: A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 241–251. Springer, Heidelberg (2005)
2. Bozdağ, D., Catalyurek, U., Gebremedhin, A., Manne, F., Boman, E., Özgüner, F.: A Parallel Distance-2 Graph Coloring Algorithm for Distributed Memory Computers. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 796–806. Springer, Heidelberg (2005)
3. Bui, A., Datta, A., Petit, F., Villain, V.: Snap-Stabilization and PIF in Tree Networks. Distributed Computing 20(1), 3–19 (2007)
4. Dijkstra, E.: Self-Stabilizing Systems in Spite of Distributed Control. Communications of the ACM 17(11), 644 (1974)

5. Djukic, P., Valaee, S.: Link Scheduling for Minimum Delay in Spatial Re-Use TDMA. In: Proc. of IEEE INFOCOM 2007, pp. 28–36 (May 2007)
6. Dolev, S., Israeli, A., Moran, S.: Uniform Dynamic Self-Stabilizing Leader Election. Distributed Algorithms, 167–180 (1997)
7. Grönkvist, J.: Assignment Methods for Spatial Reuse TDMA. In: MobiHoc 2000: Proc. of the 1st ACM International Symposium on Mobile Ad Hoc Networking & Computing, pp. 119–124. IEEE Press, Piscataway (2000)
8. Herman, T., Tixeuil, S.: A Distributed TDMA Slot Assignment Algorithm for Wireless Sensor Networks. Algorithmic Aspects of Wireless Sensor Networks, 45–58 (2004)
9. Kanzaki, A., Uemukai, T., Hara, T., Nishio, S.: Dynamic TDMA Slot Assignment in ad hoc Networks. In: Proc. of the 17th International Conference on Advanced Information Networking and Applications, p. 330. IEEE Computer Society, Los Alamitos (2003)
10. Kodialam, M., Nandagopal, T.: Characterizing the Capacity Region in Multi-Radio Multi-Channel Wireless Mesh Networks. In: Proceedings of the 11th Annual International Conference on Mobile Computing and Networking, pp. 73–87. ACM, New York (2005)
11. Kumar, V., Marathe, M., Parthasarathy, S., Srinivasan, A.: Algorithmic Aspects of Capacity in Wireless Networks. In: Proc. of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, p. 144. ACM, New York (2005)
12. Rhee, I., Warrier, A., Min, J., Xu, L.: DRAND: Distributed Randomized TDMA Scheduling for Wireless ad-hoc Networks. In: Proc. of the 7th ACM International Symposium on Mobile Ad Hoc Networking and Computing, p. 201. ACM, New York (2006)
13. Salonidis, T., Tassiulas, L., Tassiulas, R.: Distributed on-Line Schedule Adaptation for Balanced Slot Allocation in Wireless Ad Hoc Networks. Tech. Rep. In: Proc. IEEE International Workshop on Quality of Service, Montreal,Canada (2002)
14. Sivrikaya, F., Busch, C., Magdon-Ismail, M., Yener, B.: ASAND: Asynchronous Slot Assignment and Neighbor Discovery Protocol for Wireless Networks. In: OPNETWORK 2007, Washington DC, pp. 27–31 (2007)
15. Wang, W., Wang, Y., Li, X., Song, W., Frieder, O.: Efficient Interference-Aware TDMA Link Scheduling for Static Wireless Networks. In: Proceedings of the 12th Annual International Conference on Mobile Computing and Networking, pp. 262–273. ACM, New York (2006)

Low Memory Distributed Protocols for 2-Coloring

Amos Israeli[1], Mathew D. McCubbins[3], Ramamohan Paturi[2,⋆], and Andrea Vattani[2]

[1] Netanya Academic College
amos.israeli@netanya.ac.il
[2] University of California, San Diego
{paturi,avattani}@cs.ucsd.edu
[3] University of Southern California
mmccubbins@marshall.usc.edu

Abstract. In this paper we present new distributed protocols to color even rings and general bipartite graphs. Our motivation is to provide algorithmic explanation for human subject experiments that show human subjects can achieve distributed coordination in the form of 2-coloring over networks with a simple communication protocol. All our protocols use low (often constant) memory and reach a solution in feasible (polynomial rounds) and sometimes optimal time. All the protocols also have short message length and use a broadcast communication strategy. Our contributions include two simple protocols RingGuess and GraphCoalescing for rings and general bipartite graphs, which can be viewed as candidates for natural human strategies. We present two other protocols RingElect and GraphElect which are optimal or nearly optimal in terms of the number of rounds (proportional to the diameter of the graph) but require somewhat more complex strategies. The question of finding simple protocols in the style of RingGuess and GraphCoalescing that run in time proportional to diameter is open.

1 Introduction

In this paper we present new and simple distributed algorithms for 2-coloring in a basic model of distributed computing where each node has only local information about the network and requires very low memory. Our work is motivated by recent studies of social networks in which human subject experiments for achieving distributed coordination were conducted [14,11,12,15,8]. In particular, the experiments in [11,15,8] focus on the 2-coloring problem. In these experiments, each subject is a node of the underlying network, which is 2-colorable, and subjects are motivated (in the form of cash payment) to achieve a legal

⋆ This research is supported by NSF grant CCF-0905645 from the Division of Computing and Communication Foundations. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

2-coloring of the network. Each subject communicates with its neighbors using colored flags, and can change his/her color as many times as he/she wants. The network topologies considered are bipartite (2-colorable) graphs. Kearns, Suri and Montfort [11] show that subjects in an experimental setting can indeed solve a distributed version of the graph coloring problem using only local information when they are incentivized to work collectively. McCubbins, Paturi and Weller [15] extend the work of Kearns et. al. by considering asymmetric incentives: if a proper coloring is achieved, the participants ending with a specic color are paid more than the participants ending with the other color. The effect during the game is that participants are reluctant to leave the special color for the other one and therefore proper colorings are reached in longer times.

The fact that human subjects are able to achieve proper colorings is somewhat surprising given the limited resources they have at their disposal during the experiments: in terms of memory, they can only rely on their individual memorization skills; in terms of knowledge of the network and communication, they just have a local perspective and can only interact with their immediate neighbors in the network.

The goal of this paper is to provide an algorithmic interpertation for the success of the human agents in solving the 2-coloring problem over bipartite networks. We ask whether there exist protocols that are natural in the sense of closely representing the experimental conditions. In particular, we ask whether there exist protocols that rely solely on local information, use small amount of memory, have no knowledge of the size of the underlying graph, broadcast messages, and employ simple logic so they can be viewed as candidates for human strategies.

1.1 Model

We consider a distributed message-passing model which is synchronous, anonymous and uniform. That is, nodes in the network do not have distinct identifiers and do not have knowledge of the size of the network (or other parameters such as diameter). Moreover, they all run the same protocol.

Given our assumptions of anonymity and uniformity of the network, there exists no (even randomized) protocol that computes a coloring of the network and *terminates* (this impossibility result about termination is famous for the leader election problem, e.g. [3], and it extends easily to the coloring problem). However, we observe that the experiments in [11,15] do not require the human subjects to terminate but only to obtain a proper coloring within the allocated time. The low memory protocols in [16] for the consensus problem are also not concerned with guaranteeing termination.

We recall that for synchronous networks the running time of a protocol is given by the number of communication rounds. In each round a processor receives messages from its neighbors, performs a local computation, and sends messages to its neighbors. Finally, we remark that all our protocols work in a simple broadcast model, where a node sends the same message to all its neighbors.

The broadcast model is more suitable for capturing the setting of the coloring experiments in [11,15,8]. Indeed, in coloring experiments, the subjects are not allowed to communicate with each other except that they are allowed to change their color and observe the changes in the color of a neighbor. Thus, a subject can only communicate with the neighbors by signaling or broadcasting a color change.

1.2 Results

Our results consist of constant/low memory protocols to solve the 2-coloring problem on (even) rings as well as on general bipartite graphs, and to compute optimal colorings of preferential attachment graphs. We remark that these graphs encompass most of the topologies considered in the experiments of [11,15,8]. For the ring, we present two constant-memory protocols. The first one RingGuess is extremely simple and converges in time quadratic in the size of the network. RingGuess can be viewed as a candidate for a natural human strategy. The second protocol RingElect achieves optimality in terms of time (linear number of rounds) while only requiring constant memory. In addition, this protocol can elect a leader in optimal time[1].

For general bipartite graphs with n nodes and m edges, we present a protocol GraphCoalescing which can be viewed as a generalization of RingGuess,that computes a 2-coloring in $O(nm^2 \log n)$ (or $O(\Delta n^2)$ for Δ-regular graphs) rounds with ea ch node u using $O(\log \delta_u)$ bits of memory[2]. GraphCoalescing is also very simple and is a candidate for a human strategy. We also present the protocol GraphElect which employs a leader election strategy. GraphElect requires up to $O(\log n)$ memory without requiring the knowledge of n. We show that GraphElect computes 2-coloring in $O(\log n + D)$ (essentially optimal) rounds where D is the diameter of the graph. Finally, for bipartite graphs, we discuss how asymmetric incentives can slow down the convergence of the protocols.

Recently, Mossel and Schoenebeck [16] have presented low memory protocols for solving the consensus problem and a variation of it called the majority coordination problem. They analyze the consensus problem in social networks and parameterize social network computation by the amount of memory allocated to each node. Although their work is similar in spirit to ours in terms of the focus on low memory algorithms, their model diverges from ours in several aspects (see Section 1.1). Our focus on natural strategies calls for simplest possible protocols.

The work by Chaudhuri, Graham and Jamall [6] is also motivated by the coloring experiments in [11], but their setting is entirely different in that nodes never change their color. They show that a greedy strategy properly colors a network with high probability if the number of colors is at least $\Delta + 2$, where Δ is the maximum degree of the network.

[1] To the best of our knowledge, optimal leader election protocols are not known for our setting. (See section 6.3 of [18] for a summary of leader election protocols. Also see subsection **Model** for details about our setting.)

[2] Here δ_u denotes the degree of u.

2 Rings

Rings are among the most studied topologies in distributed coloring experiments as well as in distributed computing. In this section we analyze two constant-memory protocols for the ring topology. The first protocol, RingGuess, is natural and is a plausible candidate for subject strategies in rings. A slight variant of this protocol is also used in [11] as a comparative tool with respect to human subject performance[3]. We show that RingGuess converges to a 2-coloring in $\Theta(n^2)$ rounds (in expectation) in a ring with n nodes. The protocol does not involve any explicit message passing except that each node has access to the color of its neighbors. Its message complexity defined as the total number of color changes by all the nodes is bounded by $O(n^2 \log n)$.

Our analysis of RingGuess raises the question whether there exists a constant memory protocol that converges in linear number of rounds, which is clearly optimal for the ring. We present a new protocol, RingElect, to 2-color a ring which uses constant memory and converges in $O(n)$ rounds. RingElect employs a leader election strategy, and also elects a leader within the same resource bounds.

At the end of this section, we discuss how asymmetric incentives will slow down the protocols.

2.1 A Natural Protocol

Consider a situation that frequently occurs in the experiments of [11,15,8] when at some point during the game a subject sees a neighbor choosing the same color as his/hers. In this situation the subject may either change its color or wait for its neighbor to change its color. One could conceivably use timing strategies to make the decision. However it is not possible to implement timing strategies in bounded memory and without the knowledge of the size of the ring. As such, the most natural action is probably to change color with some (constant) probability.

With this motivation, we introduce protocol RingGuess: Initially, each node has an arbitrary color. Any node which has the same color as one of its neighbors repeatedly executes the following 2-round protocol:

1. Change your color with probability $p = \frac{1}{2}$, while memorizing your old color and the colors of your two neighbors.
2. If any of your neighbors changes its color during the first round, restore the previous color.

We now present the analysis of protocol RingGuess. Let a *conflict* be an edge with nodes of the same color, and the distance between two conflicts be the

[3] In [11] the protocol bears the name *distributed heuristic* and works for any number c of colors. When restricted to the case $c = 2$ is essentially RingGuess but is used asynchronously. They simulate this algorithm on the networks used in human subject experiments to compare the steps required by the algorithm with the time to solve the problem by the subjects. No algorithmic analysis is provided in [11].

minimum number of edges that separates them. We observe that since a node with no conflicts does not change its color, and one with conflicts ends the 2-round protocol with a different color only if both its neighbors did not change their color, the total number of conflicts on the ring never increases. The 2-round protocol RingGuess 'moves' the conflicts (clockwise or counterclockwise) with some probability. Also, when two conflicts have a node in common (i.e., 3 consecutive nodes have the same color), there is a probability of $p^3 = \frac{1}{8}$ that the two conflicts vanish – this happens when the middle node is the only one flipping its color.

The convergence proof of the protocol will make use of random walks. The following lemma bounds the number of steps for a random walk to terminate.

Lemma 1. *Let $\mathcal{W}_k = (X_0, X_1, \ldots)$ be a unidimensional random walk on a path of nodes $1, 2, \ldots, k, \ldots$ starting on the the first node (i.e. $X_0 = 1$) and terminating when the k-th node is reached or surpassed. For $\delta \in \{1, 2\}$ and $j \geq 1$, consider the following transition probabilities:*

$$\begin{aligned}
\mathbf{P}_{j \to j-\delta} &= \begin{cases} q_\delta & \text{if } j - \delta > 0 \\ 0 & \text{if } j - \delta \leq 0 \end{cases} \\
\mathbf{P}_{j \to j+\delta} &= q_\delta \\
\mathbf{P}_{j \to j} &= 1 - \sum_{\delta \in \{1,2\}} (\mathbf{P}_{j \to j-\delta} + \mathbf{P}_{j \to j+\delta})
\end{aligned}$$

where $\mathbf{P}_{i \to j} = \Pr[X_{t+1} = j | X_t = i]$, $q_1, q_2 > 0$ and $2(q_1 + q_2) \leq 1$. Then the expected time for $\mathcal{W}_k$ to terminate is at most $\frac{k^2}{q_1 + 4q_2}$.

Proof (sketch). Let h_i be the expected value of the random variable representing the number of steps to reach (or surpass) state k from state i. Then the following system $\mathcal{S}^*$ of equations holds

$$\begin{aligned}
&h_1 = c_1 + q_1 h_2 + q_2 h_3 + (1 - q_1 - q_2) h_1 \\
&h_2 = c_2 + q_1(h_1 + h_3) + q_2 h_4 + (1 - 2q_1 - q_2) h_2 \\
&h_k = 0 \\
&h_{k+1} = 0 \\
&\text{For } 3 \leq j \leq k - 1, \\
&h_j = c_j + q_1(h_{j-1} + h_{j+1}) + q_2(h_{j-2} + h_{j+2}) + (1 - 2q_1 - 2q_2) h_j
\end{aligned}$$

with $c_j = 1$, for $1 \leq j \leq k - 1$.

Recall that our goal is to show that $h_1 \leq \frac{k^2}{q_1 + 4q_2}$. Let $h_1 = x_1^*, h_2 = x_2^*, \ldots, h_{k+1} = x_{k+1}^*$ be the solution of this system $\mathcal{S}^*$, and let $h_1 = \tilde{x}_1, h_2 = \tilde{x}_2, \ldots, h_{k+1} = \tilde{x}_{k+1}$ be the solution of the system $\tilde{\mathcal{S}}$ obtained by setting $c_1 = 1$, $c_j = 2$, for $2 \leq j \leq k - 1$, and replacing equation $h_k = 0$ with $h_k = \frac{k^2 - (k-1)^2}{q_1 + 4q_2}$. We observe that it has to be $\tilde{x}_j \geq x_j^*$ for any $1 \leq j \leq k + 1$. Using induction, we show $\tilde{x}_j = \frac{k^2 - (j-1)^2}{q_1 + 4q_2}$ for $1 \leq j \leq k + 1$. From this, we conclude that $h_1 = x_1^* \leq \tilde{x}_1 = \frac{k^2}{q_1 + 4q_2}$. □

We apply Lemma 1 to bound the number of rounds required for two conflicts in a ring to come close to each other. This will be the main ingredient to establish the main theorem.

Lemma 2. *Consider any* 2*-coloring with* m *conflicts such that no conflicts are at distance less than* 2*. Then, after at most* $\frac{n^2}{2p^2m^2}$ *expected number of rounds, there will be two conflicts at a distance less than* 2*.*

Proof. We observe that after an execution of the 2-round protocol, any conflict (independently of the others) will move one edge clockwise with probability p^2, one edge counterclockwise with probability p^2, and it will not change position with probability $1 - 2p^2$. Fix two consecutive conflicts and let D be a random variable representing the distance between them. After an execution of the 2-round protocol, D will (a) increase by 2, as well as decrease by 2, with probability $(p^2)^2 = p^4$; (b) increase by 1, as well as decrease by 1 with probability $2p^2(1 - 2p^2) = 2p^2 - 4p^4$; (c) not change with probability $1 - 2(p^4 + (2p^2 - 4p^4)) = 1 - (4p^2 - 6p^4)$.

We observe that the behavior of D can be interpreted as the random walk $\mathcal{W}_n$ in Lemma 1 using $q_1 = 2p^2 - 4p^4$ and $q_2 = p^4$ (We use $\mathcal{W}_n$ because the distance between two conflicts is always less than n.) Lemma 1 assures D will be less than 2 in at most $\frac{n^2}{q_1+4q_2} = \frac{n^2}{2p^2}$ expected number of rounds.

Now, in order to prove the lemma, we will show that the expected time for two conflicts out of the total m conflicts to be at a distance less than 2 is no larger than the expected time for a random walk $\mathcal{W}_{\lfloor n/m \rfloor}$ to terminate. Consider the Markov chain $\bar{D}_0, \bar{D}_1, \ldots$, with $\bar{D}_t = (D_t^{(1)}, D_t^{(2)}, \ldots, D_t^{(m)})$, where $D_t^{(i)}$ is the random variable representing the distance between the i-th and $(i+1)$-st conflict on the ring at time t. $D_t^{(m)}$ represents the distance between the last and the first conflict at time t. We couple this Markov chain with another one, $M_t = \min_i D_t^{(i)}$, that keeps track of the distance between the closest pair of conflicts for $t \geq 0$. Now we observe that $M_t \leq \left\lfloor \frac{n}{m} \right\rfloor$, and that M_t approaches a value less than 2 at least as fast as the random walk $\mathcal{W}_{\lfloor n/m \rfloor}$ terminates. This observation along with Lemma 1 concludes the proof. □

The main theorem of this section follows easily by Lemma 2.

Theorem 1. *Protocol* RingGuess *computes a* 2*-coloring of the ring in* $\Theta(n^2)$ *expected number of rounds and* $O(n^2 \log n)$ *bit complexity.*

Proof. By Lemma 2, starting with a configuration with m conflicts, it takes at most $\frac{n^2}{m^2}$ expected number of rounds for two conflicts to be at a distance less than 2 since $p = \frac{1}{2}$. We observe that when two conflicts are at a distance less than 2, there is a constant probability that the two conflicts vanish, decreasing the total number of conflicts by two. Therefore, the expected number of rounds for two conflicts to vanish is bounded by $O(\frac{n^2}{m^2})$. We conclude that the expected number of rounds to resolve all the conflicts is bounded by

$\sum_{m=1}^{n} c\frac{n^2}{m^2} = O(n^2)$. Analogously the number of messages (or color changes) is bounded by $\sum_{m=1}^{n} 2cm\frac{n^2}{m^2} = O(n^2 \log n)$ since the expected number of color changes in a configuration with m conflicts is $2m$. The bound on the number of rounds is tight since if started with two conflicts at $\Omega(n)$ distance, it takes $\Theta(n^2)$ expected number of rounds for the conflicts to be at a distance less than 2. □

2.2 An Optimal Protocol for Rings

In this section we present an optimal protocol, RingElect, for ring networks: it uses constant memory and converges in $\Theta(n)$ expected number of rounds. The protocol elects a leader from which a 2-coloring of the ring follows. In its current form, RingElect is not self-stabilizing. We postpone the discussion on stabilizing RingElect to the final paper.

We describe RingElect in a more restrictive model, where local orientation is assumed. Specifically, a node is capable of sending a message only to a specific neighbor, and on reception of a message can distinguish which of its neighbors sent that message. This assumption is relaxed later.

The intuition behind the protocol RingElect is the following. We begin in a configuration where all nodes are leaders. Each leader plays two local leader elections on the segments of the ring connecting it to its clockwise-next and counterclockwise-next leaders. A leader losing in both its segments becomes slave and sends a message notifying that it is conceding the election to the leaders at the end of its segments. A concede message has the extra function of aborting all the election messages encountered whilst traversing the segment. A detailed description of the protocol follows. As mentioned before, we consider two types of messages, *concede* and *contest*. Each message `msg` has a field `msg.type` to denote its type. Concede messages are sent by nodes who became slaves. Contest messages are generated by leaders (a) during the first round, (b) on reception of a concede message, and (c) on reception of a contest message. A contest message `msg` carries a bit `msg.b` indicating the position of the leader who sent it and a "history" `msg.h` $\in \{\star, 0, 1\}$ indicating what generated it: in cases (a) and (b) the history is set to $\star$; in case (c) the history is set to the election bit contained in the received contest message.

Each node has the following local variables: `status`$\in \{start, leader, slave\}$ (initialized with *start*); for $i \in \{l, r\}$, $\mathtt{msg}_i$ to remember the latest message received from direction i; $\mathtt{b}_i$ to store the random bit used for the election on the segment in direction i; $\mathtt{losing}_i$ to remember whether it is "losing" the election on the segment in direction i. Each node runs the protocol in figure 1 in each round.

A portion of the ring between nodes u and v form a *segment* at round t if at round t, u and v are leaders and all other nodes between u and v are slaves. At the beginning we have exactly n segments. As the protocol progresses, the number of segments goes down and the segments get larger. At the point when there is only one leader, all segments vanish.

```
Receive messages from both the neighbors and update variables $\mathtt{msg}_i$
If $\mathtt{status} = \mathit{start}$ then
   For both $i \in \{l, r\}$ do $\mathit{Election}(i, \star)$
   $\mathtt{status} := \mathit{leader}$
Elsif $\mathtt{status} = \mathit{leader}$ then
   For both $i \in \{l, r\}$ do
      If $\mathtt{msg}_i$ is received in the current round then
         Set $\mathtt{losing}_i$ to the value of the predicate:
         $(\mathtt{msg}_i.\mathtt{type} = \mathit{contest}$ and $\mathtt{msg}_i.\mathtt{b} = 1$ and $\mathtt{msg}_i.\mathtt{h} = 0$ and $\mathtt{b}_i = 0)$
   If $\mathtt{losing}_i = \mathit{true}$ for both $i \in \{l, r\}$ then
      Send concede messages to both neighbors, and $\mathtt{status} := \mathit{slave}$
   Else for both $i \in \{l, r\}$ do
      If $\mathtt{msg}_i$ is received in the current round then
         If $\mathtt{msg}_i.\mathtt{type} = \mathit{contest}$ then $\mathit{Election}(i, \mathtt{msg}_i.\mathtt{b})$ else $\mathit{Election}(i, \star)$
Elsif $\mathtt{status} = \mathit{slave}$ then
   Forward message $\mathtt{msg}_l$ to neighbor $r$ unless $\mathtt{msg}_l.\mathtt{type} = \mathit{contest}$
   and $r$ sent a concede message in the current or in the previous round
   Forward message $\mathtt{msg}_r$ to neighbor $l$ unless $\mathtt{msg}_r.\mathtt{type} = \mathit{contest}$
   and $l$ sent a concede message in the current or in the previous round
```

```
$\mathit{Election}(i, h)$:
   $\mathtt{b}_i :=$ random bit
   Send a contest message $\mathtt{msg}$ with $\mathtt{msg.b} := \mathtt{b}_i$
   and with $\mathtt{msg.h} = h$ to neighbor $i$
```

Fig. 1. RingElect protocol

Analysis of RingElect

The analysis of the protocol is not trivial. We proceed as follows: first, we observe some properties of the protocol and give useful definitions; then we give a careful characterization of the possible configurations of any segment at any round. The properties of these configurations will imply that the protocol is consistent, that is, there is always at least a leader at any round. Finally, we prove that the convergence of the protocol is optimal.

We begin by observing the following properties of the protocol RingElect. During any round if a node becomes a slave, it will remain a slave for all subsequent rounds. For every node and for all $i \in \{l, r\}$, $\mathtt{b}_i$ is the same as the bit in the most recent contest message sent in the direction i. Also u receives contest messages only from v from the direction of v, and v only from u from the direction of u for the entire duration u and v are the leaders of a segment. When one of the two nodes in a segment becomes a slave a concede message will be sent by the slave toward the other leader of the segment. Any contest message received by a slave node which has received a concede message in the current or previous round going in the opposite direction will not be forwarded further.

We say that a message $\mathtt{msg}$ is *crucial* if $\mathtt{msg.type} = \mathit{contest}$, $\mathtt{msg.b} = 1$ and $\mathtt{msg.h} = 0$. Similarly, a message variable $\mathtt{msg}_i$ of a leader node is *crucial* at the end of round t if it holds a crucial message at the end of round t. For a segment with left-leader u and right-leader v, the variables of the segment are the variable $\texttt{msg}_r$ of u and the variable $\texttt{msg}_l$ of v.

Henceforth, we will say that a message m is on a segment at the end of round t, if during round t a node u of that segment sent m to another node v of that segment. The direction of a message is defined by who sent the message and who received it. We say that two messages m from u to v and m' from u' to v' on a segment are *converging* (resp. *diverging*) if a path $u, v, \ldots, v', u'$ (resp. $v, u, \ldots, u', v'$) is in the segment. Finally, for a segment of leaders u and v, and for a message m on the segment directed toward u, any message in between m and v is said to be *behind* m.

Definition 1. *A segment is in a safe configuration at the end of round t if the following properties hold at the end of round t.*

(i) *There are one or two concede messages on the segment.*
(ii) *No variable of the segment is crucial.*
(iii) *Every crucial contest message on the segment is converging to a concede message on the segment.*
(iv) *If there are two concede messages on the segment then they are diverging and no other message is in between them. If there is only one concede message on the segment, then there can be only one message behind it. This message is non-crucial and is traveling in the same direction as the concede message.*

Definition 2. *A segment is in a contest configuration at the end of round t if the following properties hold at the end of round t.*

(a) *There are exactly two contest messages and no concede messages on the segment.*
(b) *At most one variable of the segment is crucial.*
(c) *Crucial messages on the segment travel in the same direction. Also if a variable of the segment is crucial, crucial messages travel toward the leader holding that variable.*

Lemma 3. *At the end of any round $t \geq 1$, any segment is either in a safe configuration or in a contest configuration.*

Proof. We prove the lemma by induction on t. During round $t = 1$ every node sends a non-crucial contest message per segment, therefore at the end of the round there will be two non-crucial contest messages (history of messages is set to $\star$) per segment (also no leader has crucial variables). Therefore, at the end of round $t = 1$ each segment is in a contest configuration.

By induction suppose that the lemma holds at the end of round t, and consider any segment. First suppose that the segment is in a safe configuration at the end of round t. If no concede messages are received by any of the two leaders at the beginning of round $t+1$, it is easy to check that there will be a safe configuration at the end of round $t+1$. Otherwise a leader receiving a concede message at the beginning of round $t+1$, will send a non-crucial contest message on the segment (the history of the message is set to $\star$). Therefore at the end of round $t+1$ the segment will be either in a safe configuration (if at least a concede message is present) or in a contest configuration (if no concede messages are left).

Now suppose that the segment is in a contest configuration at the end of round t. Consider first the case when no leader of the segment becomes a slave during round $t+1$. We want to prove that at the end of round $t+1$ there will be a contest configuration. Note that property (a) holds at the end of round $t+1$ because a slave receiving a contest message will forward it, while a leader receiving a contest message will send another contest message on the segment (recall that we are assuming that leaders do not become slaves during round $t+1$). Property (b) holds at the end of round $t+1$ because property (c) guarantees that only one leader can receive a crucial message at the beginning of round $t+1$ (and always by (c) the other leader cannot have a crucial variable). Property (c) holds at the end of round $t+1$ because a leader that receives a crucial message at the beginning of round $t+1$, will send a non-crucial contest message on the segment (the history of the message will be 1 since the received message is crucial). Now consider the case when a leader of the segment becomes a slave during round $t+1$. A leader becomes a slave only if both its variables are crucial. Therefore property (b) implies that only one leader of the segment can become a slave during round $t+1$. Let u be this node, and let v and w be the other leaders of the two segments of u. Note that since u becomes a slave during round $t+1$, it must be the case that both segments of u are in a contest configuration at the beginning of round $t+1$ (if not, one of u's variable would not be crucial). We want to prove that at the end of round $t+1$ the new segment defined by the leaders v and w will be in a safe configuration. Property (i) and (iv) are trivial since u will send two concede messages on its two sides. By property (b) both v and w have non-crucial variables at the end of round t, and since all crucial contest messages are traveling toward u by (c), properties (ii) and (iii) will hold at the end of round $t+1$. □

From Lemma 3, property (ii) of safe configurations and properties (b)-(c) of contest configurations, it follows that there is always at least one leader.

Corollary 1. *At any round t there exists at least one leader.*

We can now prove that the protocol converges in optimal time.

Theorem 2. *Protocol* RingElect *elects a leader and computes a* 2*-coloring in* $O(n)$ *expected number of rounds and* $O(n \log n)$ *bit-complexity.*

Proof. Fix a round t. For a leader u, let the *scope* of u be the union of the two segments of u (notice that scopes are not disjoint and each scope contains exactly 3 leaders.) Let $l^{\langle t\rangle}$ be the numbers of leaders at round t. Let T be a maximal set of node-disjoint scopes. Then it has to be $|T| \geq \lfloor \frac{l^{\langle t\rangle}}{3} \rfloor$. We observe that at least $(\lfloor \frac{T}{2} \rfloor - 1)$ scopes of T have length at most $\frac{6n}{l^{\langle t\rangle}}$ (otherwise the remaining scopes of T would contain at least $(\frac{|T|}{2}+1)\frac{6n}{l^{\langle t\rangle}} > n$ distinct nodes.) Consider any of these "short" scopes. Observe that at most every $\frac{12n}{l^{\langle t\rangle}}$ rounds either at least one of the three leaders has become a slave or all the nodes have drawn new bits for the elections on the two segments of the scope. In each of these phases there is a constant probability that the central leader of the scope becomes a slave (note that the two segments are a in contest configuration assuming the two non-central leaders do not become slaves). Therefore, we can conclude that in expectation after $O(\frac{n}{l^{\langle t\rangle}})$ rounds all the short scopes will have lost at least one leader. Thus, in expectation $l^{\langle t+O(n/l^{\langle t\rangle})\rangle} \leq l^{\langle t\rangle} - (\frac{|T|}{2} - 1) \leq \frac{l^{\langle t\rangle}}{c}$ for some constant $c > 1$. That is, the number $l^{\langle t\rangle}$ of leaders at some round t decreases by a constant factor of c after a phase of $c'\frac{n}{l^{\langle t\rangle}}$ rounds (for some constant c'). Note that given $c' \geq 12$ we can assume that the election bits in a phase are independent from the bits of the previous phase because the leaders we consider are playing in short scopes, and therefore draw new bits every $\frac{12n}{l^{\langle t\rangle}}$ rounds. By iterating this argument, the expected number of rounds in order to reduce the number of leaders from l to 1 is proportional to $\frac{n}{l}\sum_{i=0}^{\log_c(l)} c^i = O(n)$. With a similar analysis it is possible to show that the expected number of messages is $O(n \log l) = O(n \log n)$. The existence of at least one leader is established in Corollary 1.

We will now explain how a 2-coloring can be achieved. In the first round every node chooses a color for itself. Every time that a leader receives a concede message, it will start to propagate its coloring. A slave who receives two non-compatible colorings by its two neighbors will not propagate any of the two. When only one leader is left, the coloring of this leader will be propagated through the entire network (in linear number of rounds). □

Relaxation of the model. We will briefly describe how to modify the protocol RingElect so that only broadcast is used. We will still assume that when a node receives a message, it can distinguish which neighbor broadcast it[4]. The key property that will use is the fact that a slave never receives two messages coming from the same neighbor in two consecutive rounds. (This property can be shown to hold inductively.) Using this property we can modify the protocol as follows. A slave node u will accept (and therefore broadcast) a message m broadcast from a slave neighbor v iff at least one of these conditions holds: (i) in the previous round u did not accept m from the other neighbor; (ii) v is broadcasting two messages (this happens only if in the previous round v accepted the message m from u and a message m' from its other neighbor; in this case, if $m \neq m'$, u knows what message to ignore, otherwise u will accept any of the two). A slave node u will accept a message m broadcast from a leader neighbor w iff in the previous round u broadcast a message received by the other neighbor. Similar rules can be used for leader nodes. The only major modification is the following: when a leader accepts two messages in the same round (coming from the two different segments) and does not become a slave, it will draw only one bit and use (broadcast) it for both segments. This modification of the election process does not affect the performance of the protocol which will still converge in linear time.

[4] Note that this is a natural assumption. For example, this assumption holds for the coloring experiments in [11,15,8].

Asymmetric incentives. As mentioned in the introduction, the experiments in [15] introduce asymmetric incentives: if a proper coloring is achieved, the participants ending with a specific color are paid more than the participants ending with the other color. Longer convergence time has been observed in this setting.

We wish to quantify the influence of these asymmetric incentives on our protocols. We model "selfish" participants in the following way. We say that a node is Byzantine if, when supposed to give up the special color or become a slave, it does so with probability q strictly less than one. Now consider a ring with (at least) two Byzantine nodes at odd distance $\Omega(n)$. (Note that if we place two Byzantine nodes at random on the ring, this situation will happen with constant probability.) Then, with proofs similar to the ones presented, it is possible to show that the convergence time of the protocols gets slower in the following way: RingGuess will converge in $\Theta(\frac{n^2}{q^2} + \frac{1}{q^3})$ time; and RingElect will converge in time $O(\frac{n}{q})$.

An interesting aspect caused by the requirement on constant memory, is that detection of Byzantine nodes is impossible for the other nodes.

3 General Bipartite Graphs

We now turn our attention to general bipartite graphs. First we present a simple protocol that computes a 2-coloring of any bipartite graph in poly(n) time. Each node v uses an amount of memory proportional to the logarithm of its degree. Secondly, we show that using a little more memory per node, namely $O(\log n)$ bits, we develop a protocol whose convergence time is essentially optimal.

3.1 A Coalescing Particles Protocol

Without loss of generality let the color palette be $\{0, 1\}$. Consider the following simple protocol that we call GraphCoalescing.

In the first round every node chooses a random color b and sends a "suggestion" $\bar{b}$, the complement of b, to a random neighbor. In every round $t \geq 2$, each node u receiving at least one suggestion:

(a) randomly chooses a suggestion b among the received ones;
(b) colors itself with b; and
(c) randomly chooses a node w in the set composed of its neighbors and itself; if w is a neighbor sends suggestion $\bar{b}$ to w, otherwise re-suggests b to itself for the next round.

Observe that each node u uses $O(\log \delta_u)$ bits of memory to select a random neighbor.

The idea of this protocol is that every node proposes a coloring. Each proposal takes a random walk on the graph, and when two (or more) proposals meet, only one of them will survive (the proposals will *coalesce* into one) and continue its random walk. Now suppose that at some point only one proposal is left: then it will walk randomly through the network, and will lead to a proper coloring once all nodes have been visited. Viewing proposals as tokens, it follows that the protocol can also be used to provide a token management scheme (See [10] for a similar approach). The following theorem borrows heavily from the literature.

Theorem 3. *Protocol* GraphCoalescing 2*-colors any bipartite graph with m edges in $O(m^2 n \log n)$ expected number of rounds. If the graph is Δ-regular the expected number of rounds is $O(\Delta n^2)$.*

Proof. In the proof we refer to each proposal as a particle. Let G be a (bipartite) graph. We observe that part (c) of the protocol implies that each particle is performing a random walk on the graph G' that is obtained from G by adding self-loops to each node. Therefore, since G' is not bipartite, the random walk of each particle is aperiodic. The expected number of rounds required for coloring the graph is bounded by the expected number T_{coalesce} of rounds for the particles to coalesce to a single particle, plus the cover time T_{cover} (that is, the expected number of rounds for a single particle to visit all the nodes). A classic result in [1] shows that the cover time of a graph is $T_{\text{cover}} = O(mn)$. By [2, Section 14.3], we have that $T_{\text{coalesce}} = O(\Delta n^2)$ for Δ-regular graphs, and $T_{\text{coalesce}} = O(T_{\text{cat\&mouse}} \log n)$ for general graphs, where $T_{\text{cat\&mouse}}$ is the time required for two random walks to meet. For non-bipartite graphs it is well-known that $T_{\text{cat\&mouse}} = O(m^2 n)$. The theorem follows. □

We observe that the protocol (as it is) is not suitable for a broadcast model because nodes must be able to send messages to a specific neighbor. We now argue that this issue can be addressed if we observe the isomorphism between the coalescing particles process and the voter model. In the voter model each node starts with an opinion (a proposal in our case). As time passes, nodes modify their opinions in the following way. At each step, each node changes its opinion to the opinion of a random neighbor or stick to its opinion where all the options are equally probable. It is known that the expected time for only one opinion to survive (the number of opinions can only decrease with time) is the same as the expected time for all the particles to coalesce (e.g. see [7]). This observation easily leads to a broadcast protocol with the same guarantees.

3.2 A Time-Optimal Protocol

In this section we present GraphElect, a protocol that uses $O(\log n)$ memory in expectation and computes a 2-coloring of any bipartite graph in $O(D + \log n)$ expected number of rounds, where n and D are size and diameter of the graph respectively. Any distributed protocol that 2-colors general bipartite graphs requires $\Omega(D)$ rounds: therefore GraphElect is time-optimal in graphs of diameter at least $\Omega(\log n)$.

We now describe the protocol GraphElect. At any given stage, a processor can be either a leader or a slave. At the beginning of the protocol all processors are leaders. Each processor presents to its neighbors a variable `Leading-Rank` (initial value 0), and its color (initial value either 0 or 1). In addition, each processor keeps locally a variable `Rank` (initial value 0). At the beginning of each round, a processor reads the state variables of all its neighbors and computes the maximal leading rank among all its neighbors. The processor holding that maximal leading rank is the processor's leading neighbor. If there is more than a single processor holding the maximal leading rank, the leading neighbor is elected arbitrarily from among the leading processors. If the maximal leading rank is larger than the processor's own rank, the processor adjusts its color to be the opposite color of its leading neighbor, and becomes a slave (if it was a leader). A slave keeps doing this simple routine forever and never gets a chance to become a leader again.

In addition to all aforementioned variables, a leader also holds a timer whose initial value is zero. The nodes counts down from timer value to zero. When the count goes to 0, if the processor is still a leader, it increments its rank and leading rank by 1. Then it updates its timer value to be twice the old timer value, plus a random value in $\{0,1\}$.

The following lemma is pivotal for the analysis of the protocol.

Lemma 4. *Let u and v two nodes in the graph. If, at the beginning of a certain round, u is still a leader and its rank is greater than the rank of v, then for the rest of the computation there will be some node (possibly u) with rank greater than the rank of v.*

Proof. Let $W_k^{\langle u\rangle}$ be the value of the timer of u right after the k-th update. $W_k^{\langle u\rangle} = 2\cdot W_{k-1}^{\langle u\rangle} + B_k^{\langle u\rangle}$ (as long as u is a leader), where the $B_k^{\langle u\rangle}$'s are i.i.d. random variables taking values from $\{0,1\}$. We observe that $W_k^{\langle u\rangle} = \sum_{i=0}^{k} 2^{k-i} B_k^{\langle u\rangle}$. Now consider the first round t^* when the rank of u is greater than the rank of v. During round t^*, u must have updated its timer, and let this one be its k-th update. Note that it must be the case that $W_j^{\langle u\rangle} = W_j^{\langle v\rangle}$ (and therefore $B_j^{\langle u\rangle} = B_j^{\langle v\rangle}$) for all $j < k-1$, and $W_{k-1}^{\langle u\rangle} < W_{k-1}^{\langle v\rangle}$ (and therefore $B_{k-1}^{\langle u\rangle} < B_{k-1}^{\langle v\rangle}$). Now consider the k-th update for u and v. We have that $W_k^{\langle u\rangle} < W_k^{\langle v\rangle}$ since

$$2W_{k-1}^{\langle u\rangle} + B_k^{\langle u\rangle} \le 2W_{k-1}^{\langle u\rangle} + 1 \le 2(W_{k-1}^{\langle v\rangle} - 1) + 1 < 2W_{k-1}^{\langle v\rangle} \le 2W_{k-1}^{\langle v\rangle} + B_k^{\langle v\rangle}.$$

Therefore, $W_k^{\langle u\rangle}$ rounds after t^*, u will increase its rank to $k+1$, while the rank of v can only increase it to $k+1$ after at least $1 + W_k^{\langle v\rangle} > W_k^{\langle u\rangle}$ many rounds from t^*.

By induction, as long as u is a leader, u will have a rank greater than the rank of v. If at some point u loses its leadership status, it must be that another node u' has a rank greater than the rank of u, and thus greater than the rank of v. □

The main theorem of this section is the following.

Theorem 4. *Protocol* GraphElect *elects a leader and computes a 2-coloring of the graph in $O(D + \log n)$ expected number of rounds, where D is the diameter of the graph.*

Proof. We will compute the expected time to have only one leader remaining in the network. Once this event happens, the coloring propagated by the leader will be adapted by each node eventually thus producing a 2-coloring. Let $t(k)$ be the minimum round such that there exists a node that is updating its timer for the k-th time. In other words $t(k)$ is the round during which one or more nodes achieve the rank k for the first time and k is the largest rank at round $t(k)$. Also let L_k be the set of nodes with the largest rank k at round $t(k)$. By Lemma 4, we have that $L_{k+1} \subseteq L_k$ for any k. We want to compute the expected k^* such that $|L_{k^*}| = 1$. At the beginning L_0 contains all the nodes in the graph, so $|L_0| = n$. At round $t(k)$ we have L_k nodes which will select i.i.d. random numbers in $\{0, 1\}$: we observe that only the nodes that select 0 will be in L_{k+1}. Therefore, in expectation, $|L_{k+1}| = \frac{1}{2}|L_k|$. We conclude that in $O(\log n)$ expected rounds there will be only one node with the largest rank.

At this point we have only one node of largest rank, call it u^*. However we can still have multiple leaders: for example u^* might be very far from some other leader w, and by the time the leading rank is propagated from u^* to w, w might have increased its own rank. Note that this cannot happen when the timer length of u^* (and therefore of all the other leaders) is at least D. Since, after $O(\log n)$ rounds the u^*'s timer value will be more than 0 with high probability, and the timer value doubles at each update, we have that after at most $O(\log n + D)$ rounds from round $t(k^*)$ the u^*'s timer value will be at least D. Thus, after $O(\log n + D)$ expected number of rounds there will be only one leader. □

References

1. Aleliunas, R., Karp, R.M., Lipton, R.J., Lovász, L., Rackoff, C.: Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems. In: FOCS 1979, pp. 218–223 (1979)
2. Aldous, D., Fill, J.: Reversible Markov Chains and Random Walks on Graphs, http://www.stat.berkeley.edu/~aldous/RWG/book.html
3. Attiya, H., Welch, J.: Distributed Computing; Fundamentals, Simulations and Advanced Topics, 2nd edn. John Wiley & Sons, Chichester (2004)
4. Barabási, A.L., Albert, R.: Emergence of Scaling in Random Networks. Science 286(5439), 509–512 (1999)
5. Bollobás, B., Riordan, O., Spencer, J., Tusnády, G.: The degree sequence of a scale-free random graph process. Random Structures & Algorithms 18(3) (May 2001)
6. Chaudhuri, K., Chung Graham, F., Jamall, M.S.: A network coloring game. In: Papadimitriou, C., Zhang, S. (eds.) WINE 2008. LNCS, vol. 5385, pp. 522–530. Springer, Heidelberg (2008)
7. Cooper, C., Frieze, A., Radzik, T.: Multiple random walks in random regular graphs. SIAM Journal on Discrete Mathematics 23(4), 1738–1761 (2009)
8. Enemark, D., McCubbins, M., Paturi, R., Weller, N.: Good edge, bad edge: How network structure affects a group's ability to coordinate. In: ESORICS (March 2009)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)

10. Israeli, A., Jalfon, M.: Token Management Schemes and Random Walks Yield Self-Stabilizing Mutual Exclusion. In: PODC 1990, pp. 119–131 (1990)
11. Kearns, M., Suri, S., Montfort, N.: An experimental study of the coloring problem on human subject networks. Science 313(5788), 824–827 (2006)
12. Kearns, M., Judd, S., Tan, J., Wortman, J.: Behavioral experiments on biased voting in networks. National Academy of Science (January 2009)
13. Khot, S.: Improved inapproximability results for maxclique, chromatic number and approximate graph coloring. In: FOCS 2001, pp. 600–609 (2001)
14. Latané, B., L'Herrou, T.: Spatial clustering in the conformity game: Dynamic social impact in electronic groups. Journal of Personality and Social Psychology 70(6), 1218–1230 (1996)
15. McCubbins, M.D., Paturi, R., Weller, N.: Connected Coordination: Network Structure and Group Coordination. American Politics Research 37, 899–920 (2009)
16. Mossel, E., Schoenebeck, G.: Reaching Consensus on Social Networks. In: Innovations in Computer Science, ICS (2009)
17. Peleg, D.: Distributed Computing: A Locally-Sensitive Approach. SIAM Monographs, Philadelphia (2000)
18. Santoro, N.: Design and Analysis of Distributed Algorithms. John Wiley & Sons, Inc., Chichester (2007)

Connectivity-Preserving Scattering of Mobile Robots with Limited Visibility*

Taisuke Izumi[1], Maria Gradinariu Potop-Butucaru[2], and Sébastien Tixeuil[2]

[1] Graduate School of Engineering, Nagoya Institute of Technology
t-izumi@nitech.ac.jp
[2] Université Pierre et Marie Curie - Paris 6, LIP6 CNRS 7606, France
{maria.gradinariu,sebastien.tixeuil}@lip6@fr

Abstract. The *scattering* problem is a fundamental task for mobile robots, which requires that no two robots share the same position. We investigate the scattering problem in the limited-visibility model. In particular, we focus on *connectivity-preservation* property. That is, the scattering must be achieved so that the disconnection of the visibility graph never occurs (in the visibility graph robots are the nodes of the graph and the edges are their visibility relationship). The algorithm we propose assumes ATOM (*i.e.* semi-synchronous) model. In these settings our algorithm guarantees the connectivity-preserving property, and reaches a scattered configuration within $O(\min\{n, D^2 + \log n\})$ asynchronous rounds in expectation, where D is the diameter of the initial visibility graph. Note that the complexity analysis is *adaptive* since it depends on D. This implies that our algorithm quickly scatters all robots crowded in a small-diameter visibility graph. We also provide a lower bound of $\Omega(n)$ for connectivity-preserving scattering. It follows that our algorithm is optimal in the sense of the non-adaptive analysis.

1 Introduction

Background Algorithmic studies about autonomous mobile robots recently emerged in the distributed computing community. In most of those studies, a robot is modelled as a point in a Euclidean plane, and its abilities are quite limited: It is usually assumed that robots are *oblivious* (*i.e.* no memory is used to record past situations), *anonymous* (*i.e.* no ID is available to distinguish two robots), and *uniform* (*i.e.* all robots run the same identical algorithm). In addition, it is also assumed that each robot has no direct means of communication. The communication between two robots is done in an implicit way by having each robot observe its environment, which includes the positions of the other robots.

The *scattering* problem is a fundamental tasks in mobile robotics. In this task, starting from an arbitrary configuration (*i.e.* arbitrary initial positions for the participating robots), eventually no two robots share the same position. The

* This work is supported in part by Grand-in-Aid for Young Scientists ((B)22700010) of JSPS. Additional support from ANR projects R-Discover, SHAMAN, and ALADDIN.

scattering algorithm can be considered as the dual problem of gathering (making all robots reach a common point), and plays an important role to make pattern formation algorithms self-stabilizing: The pattern formation problem requires that all robots in the system organize in some geometric shape such as a line or a circle. If all robots are initially located at the same point, we have to divide them into a number of different locations to form the geometric pattern. Thus, the scattering can be seen as the "preprocessing" part of a pattern formation protocol.

However, because of the hardness of symmetry breaking, it is trivially impossible to construct deterministic scattering algorithms. That is, there is no deterministic way to separate two robots on the same position into different positions if both of them execute synchronously. Thus, most of previous approaches to the scattering problem exploit *randomization*.

Our contribution. This paper investigates the randomized scattering with connectivity preservation in the ATOM model with restricted visibility. Robots are anonymous and oblivious and have a *limited visibility* [7,9,1]. That is, each robot can see only the robots within the unit visibility range (*a.k.a.* the unit distance range). The limited visibility is a practical assumption but makes the design of algorithms quite difficult because it prevents each robot from obtaining global position information about all other robots. Furthermore, it also brings another design issue, called *connectivity preservation*: Oblivious robots cannot use the previous history of their execution. Hence, once some robot r_1 disappears from the visibility range of another robot r_2, r_2 can behave as if r_1 does not exist in the system and vice versa. Since the cooperation between r_1 and r_2 becomes impossible, it follows that completing any task starting from those situations is also impossible. This phenomenon can be formally described by using a *visibility graph*, which is the graph induced by the robots (as nodes) and their visibility relationship (as edges). The requirement we have to guarantee in the limited visibility model is that any task or sub-task in a protocol must be achieved in a manner that preserves the connectivity of the visibility graph.

Our contribution is to propose a probabilistic scattering algorithm with the connectivity-preserving property. To the best of our knowledge, this is the first attempt and result considering scattering problem in the limited visibility model. We also give the complexity analysis of the proposed algorithm. Interestingly, the analysis is adaptive in the sense that it depends on the diameter D of the visibility graph in the initial configuration. Our algorithm achieves scattering within $O(\min\{n, D^2 + \log n\})$ asynchronous rounds in expectation. This implies that our algorithm quickly scatters all robots initially crowded within a small diameter visibility graph (*i.e.*, loss of connectivity due to robot movement is unlikely).

Another interesting point is that we were able to compute the lower bound for the round complexity of the scattering problem in our model. In spite of highly-concurrent behaviour of distributed mobile robots, any scattering algorithm can be as slow as $\Omega(n)$ (where n is the number of robots) rounds in the worst case if they have to preserve the connectivity of the visibility graph.

Related work. While the scattering problem is mentioned in the seminal paper that originated algorithmic for mobile robots [10], only a limited number of contributions considered this problem.

The initiating paper by Suzuki and Yamashita [10] introduced the scattering problem and proposed a deterministic algorithm under the assumption that *clones* do not exist. Two robots are considered to be clones of each other if they have the same local $x - y$ coordinate system and the same initial position, and they always become active simultaneously. In [5], the authors formalize the scattering problem, and propose a probabilistic algorithm based on the concept of Voronoi diagrams (this technique typically requires that robots are endowed with full system visibility at all times). They also show how the scattering can be used in solving the pattern formation problem. Furthermore, [4] investigates the time complexity of scattering, and exhibit a relation between the time complexity and the robots capability to detect the multiplicity (the number of robots sharing the same position).

Flocchini *et al.* consider a stronger variant of the scattering problem that requires all robots to reach different positions uniformly distributed over some discrete space [6]. This direction is also recently explored by Barrière *et al.* [2].

The limited visibility model was also considered in a number of previous papers. The first paper using this model considers the convergence problem [1]. Our algorithm builds on ideas shown in this paper. Two papers by Flocchini *et al.* [7] and Souissi *et al.* [9] follow up research in this model and consider the gathering problem with different variations of the limited-visibility model.

Structure of the paper. The remainder of this paper is organized as follows. In Section 2, we describe the system model and the basic terminology. Section 3 provides our scattering algorithm, its correctness and analysis. Section 4 shows a complexity lower bound. Finally, Section 5 concludes the paper.

2 Preliminaries

2.1 Models

The system consists of n robots, denoted by $r_0, r_1, r_2, \cdots, r_{n-1}$. Robots are *anonymous*, *oblivious* and *uniform*. That is, each robot has no identifier distinguishing itself and others, cannot explicitly remember the history of its execution, and works following a common algorithm independent of the value of n. In addition, no device for direct communication is equipped. The cooperation of robots is done in an implicit manner: Each robot has a sensor device to observe the environment (i.e., the positions of other robots). One robot is modelled as a point located on a two-dimensional space. Observing environment, each robot can see the positions of other robots transcripted in its *local coordinate system*. We assume *limited visibility* and *local-weak multiplicity detection*: Each robot can see only the robots located within unit distance, and can detect whether some point is occupied by one or more robots but cannot know the exact number of robots. These two assumptions imply that any knowledge about the number of robots n is not available to the robots in the system. Each robot executes the deployed algorithm in *computational cycles* (or briefly *cycles*). At the beginning of a cycle, a robot observes the current environment (i.e., the positions of other robots) and determines the destination point based on the deployed algorithm. Then, the robot moves toward the computed destination. It is guaranteed that each robot necessarily reaches the computed destination at the end of the cycle.

As the timing model, we adopt ATOM. Any execution follows a discrete time $1, 2, 3 \cdots$. At the beginning of each time unit, each robot is active or inactive. Active ones perform (and complete) one cycle within one time unit. We assume that any execution is *fair*, where all robots become active infinitely often. Since we consider randomized algorithms, the computation is allowed to use random-bits. We also assume that the number of random bits each robot can use in a cycle is one[1].

Throughout this paper, we use the following notations and terminology: to specify the location of each robot consistently, we use the global coordinate system. Notice that the global coordinate system is introduced only for ease the explanation, and thus robots are not aware of it. Without explicitly stated, the sentence "coordinate of a point p" implies p's global coordinate. A *location* is the point where at least one robot exists. We define $\mathbf{r}_j(t)$ to be the coordinate of r_j at t. We say that a location is *single* if exactly one robot stays on it. All other locations are called *multiple*. For any two coordinates $\mathbf{A}$ and $\mathbf{B}$, $\overline{\mathbf{AB}}$ denotes the segment whose endpoints are $\mathbf{A}$ and $\mathbf{B}$, and $|\mathbf{AB}|$ denotes its length. A *configuration* is the multiset consisting of all robot locations. We define $C(t)$ as the configuration at t. We also define the point set $P(C)$ of a configuration C to be the set of all locations without multiplicity. For short, we call $P(C(t))$ the point set at t, and it is denoted by $P(t)$.

A visibility graph $G(t)$ is the graph where nodes represent robots and an edge between two robots implies the visibility between two robots. More formally, the visibility graph at t consists of n nodes $\{v_0, v_1, v_2, \cdots v_{n-1}\}$. Nodes v_i and v_j are connected if and only if r_i and r_j are visible to each other.

Scheduler and Asynchronous Round. In this paper, we adopt *asynchronous rounds* (or *rounds* for short) term to evaluate time complexity of algorithms, which is a standard criterion for asynchronous distributed systems. An asynchronous round is defined as the shortest fragment of an execution in which each robot in the system executes at least once its cycle.

2.2 Connectivity-Preserving Scattering

The scattering problem requires the system to reach the configuration where all robots have different locations and their positions never change in the following execution. In addition, we also require the connectivity-preserving property. That is, the visibility graph must be connected during the scattering process.

[1] This restriction is introduced only for simplicity. It is easy to extend our result on the model that k random bits are usable.

3 Scattering Algorithm

3.1 Blocked Locations

We start the explanation of the algorithm by introducing the notion of *blocked* locations. Intuitively, a blocked location is such that deletion of edges in visibility graph can occur if some robots move.

Definition 1. *Let $\mathbf{p_c}$ be a location, C be the circle centred at $\mathbf{p_c}$ with diameter one, and $B = \{\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, \cdots \mathbf{p_j}\}$ be the set of all locations on the boundary of C. The location $\mathbf{p_c}$ is* blocked *if no arc of C with a center angle less than π can contain all locations in B.*

Examples illustrating the notion of blocked locations are shown in Fig. 1 (a) and (b). Intuitively, for a robot r_i to be movable while preserving edges of the visibility graph, its destination must be within distance one from the robots that r_i sees before the movement. For a robot at a blocked location there is no such destination. Assume the contrary, let r_i be blocked and move to some other point $\mathbf{p}(\neq \mathbf{r_i})$. Then, we take the line l which is orthogonal to the vector $\mathbf{p} - \mathbf{r_i}$ and passes through $\mathbf{r_i}$. This line cuts the circle C into two arcs with center angle π. From the definition of blocked points, both arcs have at least one robot. However, the arc in the opposite side of $\mathbf{p}$ (about l) is out of r_i's visibility after the movement to $\mathbf{p}$ (see Fig. 1 (c). Thus, if a robot r_j is on a blocked location, it cannot move anywhere without deletion of edges.

3.2 Algorithm *CPS*

The pseudo-code of our scattering algorithm *CPS* is shown in Algorithm 3.2. The key idea of our algorithm is to move all robots at non-blocked locations. If they move once, at least one blocked location newly becomes non-blocked. Thus, repeating this process, all robots will eventually become non-blocked. Then, by dividing the robots on the same location into single ones in a probabilistic way, scattering is achieved.

Let some non-blocked robot r_i be activated at t, and $R_i(t) = \{r'_0, r'_1, \cdots r'_k\}$ be the set of robots in its visibility range at t. As we mentioned, our algorithm allows only non-blocked robots to move. In the algorithm, each of non-blocked robot first determines a vector $\mathbf{p}$ which represents the direction and the maximum distance of destinations, and takes two destination candidates on the segment $\overline{\mathbf{pr_i}}$, Finally, one of two candidates is chosen by using random bits.There are two cases for deciding the vector $\mathbf{p}$:

- No robot in $R_i(t)$ is on the boundary: r_i first calculates the possible travel length d. It is bounded by the distance between the boundary of r_i's visibility and the robots in $R_i(t)$, and the distance up to r_i's nearest neighbour. That is, the value of d is the "safety margin" for preserving connectivity and avoiding conflict (i.e., two nodes on different locations have the same destination). Thus, r_i takes two points within distance $d/3$ from its current positions, and chooses one of them as the destination (by using the random bit).

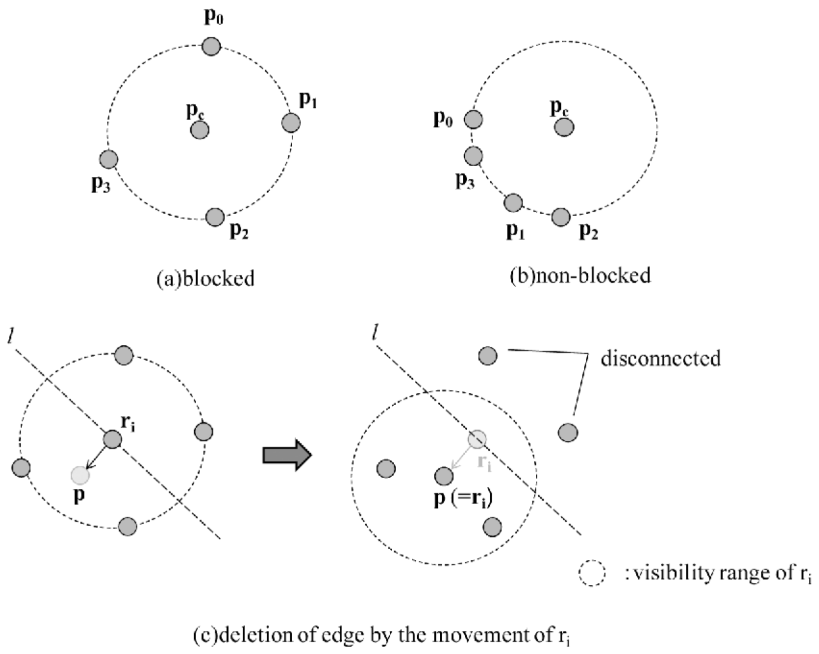


Fig. 1. The illustration of blocked locations

– Some robot in $R_i(T)$ is on the boundary: It takes the minimal arc A of the visibility border that contains all robots on the boundary. Let C be the chord of A and L be the half line bisecting the center angle of A, we set $\mathbf{p}$ by the intersection of L and the segment C. Then, its length is reduced by d, where d is computed in the same way as the first case.

It should be noted that even robots on a single location must move to make other locations non-blocked. Since robots on the visibility border is a necessary condition to block the observer robot, the robot on a single point can stop its movement only when no robot stays on its visibility border.

3.3 Correctness

We show the correctness of the algorithm *CPS*. First, we prove the connectivity-preserving property.

Lemma 1. *Let r_i and r_j be two robots that are adjacent in $G(t)$. If either r_i or r_j changes its position at t, $|\mathbf{r_i}(t+1) - \mathbf{r_j}(t+1)| < 1$ holds.*

Proof. Let $b = |\mathbf{r_i}(t) - \mathbf{r_j}(t)|$ for short. Since it is trivial if activated robots are blocked, we assume they are non-blocked.

Algorithm 1. Algorithm *CPS*

```
1:  define:
2:      $R_i$ : the set of all visible robots
3:      $R'_i$ : the set of all visible robots on the boundary of visible range
4:      $m$ : The flag indicating the multiplicity of the current location
5:      $Rand()$ : random oracle

6:  if $r_i$ is not blocked then
7:      $d = \min\{\min\{1 - |\mathbf{r}|, |\mathbf{r}|\} \mid r \in (R_i - R'_i) \cup \{r_i\}\}$
8:      $\mathbf{p} \leftarrow$ arbitrary vector with unit length
9:      if $R'_i \neq \emptyset$ then
10:         Compute the shortest arc $A$ containing all robots in $R'_i$ and its chord $C$
11:         Compute the half line $L$ bisecting the center angle of $A$
12:         $\mathbf{p} \leftarrow (C \cap L)$
13:     endif
14:     $\mathbf{p} \leftarrow d\mathbf{p}$
15:     if $R_i \neq \emptyset$ or $m =$ MULTIPLE then
16:         if $Rand() = 1$ then
17:             move($\mathbf{p}/4$)
18:         else
19:             move($\mathbf{p}/2$)
20:         endif
21:     endif
22: endif
```

- $b < 1$: If r_i is activated at t, the value of d computed by r_i does not exceed $1 - b$. Thus, the length travelled by r_i during $[t, t+1]$ is less than or equal to $(1 - b)/3$. It follows $|\mathbf{r_i}(t+1) - \mathbf{r_i}(t)| \leq (1 - b)/3$ (notice that this holds even if r_i is not activated). The same argument also holds for r_j, and thus $|\mathbf{r_j}(t+1) - \mathbf{r_j}(t)| \leq (1-b)/3$ holds. Consequently, we have $|\mathbf{r_j}(t+1) - \mathbf{r_i}(t+1)| \leq |\mathbf{r_j}(t+1) - \mathbf{r_j}(t)| + |\mathbf{r_j}(t) - \mathbf{r_i}(t)| + |\mathbf{r_i}(t) - \mathbf{r_i}(t+1)| \leq (1-b)/3 + b + (1-b)/3) < 1$, which implies r_i and r_j is adjacent at $t+1$.
- $b = 1$: Let D be the disk whose diameter corresponds to the segment $\overline{\mathbf{r_i}(t)\mathbf{r_j}(t)}$. We show both $\mathbf{r_j}(t+1)$ and $\mathbf{r_i}(t+1)$ are contained in D. By symmetry, it suffice to show $\mathbf{r_i}(t+1) \in D$ for r_i's activation. For ease of explanation, we introduce the coordinate system such that the origin is $\mathbf{p}$ computed by r_i and L corresponds to its y-axis (the direction to $r_i(t)$ is the positive side). Then, A is in the lower half-space of the x-axis because its center angle is less than π. It implies that any point on A has a polar angle larger than π. Since $r_i(t+1)$ is on the segment $\overline{\mathbf{p}\mathbf{r_i}(t)}$, its is $\pi/2$. Thus, the angle formed by $\mathbf{r_i}(t+1)$ and $\mathbf{r_j}(t)$ about $\mathbf{p}$ is larger than $\pi/2$. Consequently, we obtain $(\mathbf{r_j}(t) - \mathbf{r_i}(t+1)) = \pi/2$ and $(\mathbf{r_i}(t) - \mathbf{r_i}(t+1)) > \pi$, and thus $(\mathbf{r_j}(t) - \mathbf{r_i}(t+1)) \cdot (\mathbf{r_i}(t) - \mathbf{r_i}(t+1)) < 0$ holds[2]. This implies $\mathbf{r_i}(t+1)$ is in D and not on the boundary.

[2] Notice that the the inside of the disk having the segment $\overline{\mathbf{d_1}\mathbf{d_2}}$ as a diameter is the set of coordinates $\mathbf{v}$ satisfying $(\mathbf{d_1} - \mathbf{v}) \cdot (\mathbf{d_2} - \mathbf{v}) < 0$

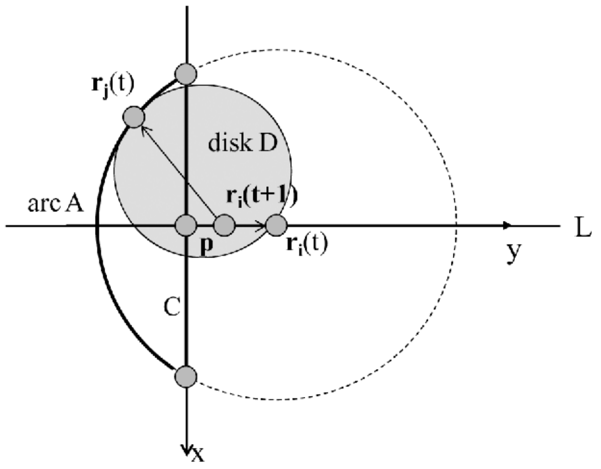


Fig. 2. The illustration used in the proof of Lemma [1]

The lemma is proven. □

Importantly, the above lemma does not only state connectivity preservation, but also shows that if a non-blocked robot r_i is activated at t, any robot in r_i's visibility range at t is not on r_i's visibility border at $t+1$. Furthermore, this fact holds for any t' after t. That is, if a robot becomes non-blocked, it remains non-blocked in the following execution (recall that robots on the boundary is a necessary condition to block the observing robot). Thus, we can obtain two following corollaries.

Corollary 1. *If two robots r_i and r_j are adjacent to each other in $G(t)$, they are also adjacent in $G(t+1)$.*

Corollary 2. *If a robot r_i is non-blocked at t, it remains non-blocked at $t+1$.*

Using the above corollaries we show the main theorem.

Theorem 1. *The algorithm CPS achieves the connectivity-preserving scattering within $O(n)$ expected rounds.*

Proof. Connectivity-preservation is guaranteed from Corollary [1]. Thus, we show that the following two properties: (1) All locations are non-blocked within $O(n)$ expected time. (2) After all robots are non-blocked, they are scattered within $O(\log n)$ expected time.

Let t_1 and t_2 be the beginnings of any two consecutive rounds. From corollary [2], it is sufficient to show that at least one blocked robot becomes non-blocked during $[t_1, t_2]$. We consider the convex hull H of all blocked locations at t_1, and take its corner location $\mathbf{p}$ (arbitrarily chosen). Let S_1 and S_2 be the two border segments of H connecting to $\mathbf{p}$, and A and $\bar{A}$ be the arcs of $\mathbf{p}$'s visibility border cut by S_1 and S_2. Without loss of generality, we assume A is contained in H.

From the definition of convex hulls, A has a center angle less than π. It implies that any robot on $\bar{A}$ is non-blocked. From the lemma 1, any non-blocked robot on $\bar{A}$ is not on $\bar{A}$ after its activation, and thus we can conclude $\mathbf{p}$ becomes non-blocked.

Next, we show the following corollary. Let k be the first (asynchronous) round when all robots become non-blocked. It is clear that two robots at different locations never have the same destination for their simultaneous activations. That is, if two robots r_i and r_j are once scattered, they never gather again. Thus, two robots can have the same position only if they have the same random input. Conversely, if two robots receive different random inputs, they necessarily stay at different locations. Let s_i be the infinite sequence of random inputs that r_i receives. The prefix of s_i with length T is denoted by $s_i(T)$. In what follows, we bound the expectation of T such that $s_i(T) \neq s_j(T)$ holds for any i and j, which implies the bound for the expected number of rounds taken to achieve scattering because any robot can change its position at each round after k.

For any h, we calculate the probability $P(h)$ such that $s_i(h \log n) \neq s_j(h \log n)$ holds for any i and j:

$$\begin{aligned}
P(h) &= \frac{2^{h\log n}}{2^{h\log n}} \cdot \frac{2^{h\log n}-1}{2^h \log n} \cdot \frac{2^{h\log n}-2}{2^{h\log n}} \cdots \frac{2^{h\log n}-n}{2^{h\log n}} \\
&= 1 \cdot \left(1 - \frac{1}{n^h}\right)\left(1 - \frac{2}{n^h}\right) \cdots \left(1 - \frac{1}{n^{h-1}}\right) \\
&\geq \left(1 - \frac{1}{n^{h-1}}\right)^{n-1} \\
&\geq \left(1 - \frac{1}{n^{h-2}}\right)
\end{aligned}$$

Thus, we obtain $\Pr[T \geq h \log n] \leq 1/n^{h-2}$. Using it, the expectation of T is bounded as follows:

$$\begin{aligned}
E[T] &\leq \sum_{t=1}^{\infty} t \Pr[T = t] \\
&\leq \sum_{t=1}^{\infty} \Pr[T \geq t] \\
&\leq \sum_{h=1}^{\infty} \sum_{k=1}^{\log n} \Pr[T \geq (h-1)\log n + k] \\
&\leq 3 \log n + \sum_{h=4}^{\infty} \log n \Pr[T \geq (h-1)\log n] \\
&\leq 3 \log n + \log n \sum_{h=4}^{\infty} 1/n^{h-2} \\
&= O(\log n)
\end{aligned}$$

The lemma is proved. $\square$

3.4 Diameter-Sensitive Analysis

The complexity analysis of the previous subsection depends only on the number of robots n. This section provides another complexity analysis of *CPS* which is parametrized by D, the diameter of $G(0)$. The result we show is that $O(D^2)$ rounds are sufficient to make all robots non-blocked.

Lemma 2. *All robots are non-blocked at the beginning of the round* $D^2 + 2$.

Proof. Let t_k be the beginning of round k. Suppose for contradiction that a robot r_0 is still blocked at t_{D^2+2}. From Lemma 1, if all robots within distance one from r_0 move once, r_0 becomes non-blocked. Thus, there exists one robot r_1 blocked at t_{D^2+1} (because all non-blocked robots move once during $[t_{D^2+1}, t_{D^2+2}]$). Considering r_1 in the same way as r_0, we can conclude that some robot r_2 ($\neq r_1, r_2$) is blocked at t_{D^2}. It follows that there exists at least one chain of robots $r_0, r_1, r_2, \cdots r_{D^2+1}$ such that each robot r_k is blocked by the end of round $D^2 + 1 - k$.

Without loss of generality, we assume that $\mathbf{r_0}$ is the origin of the global coordinate system. Let $\mathbf{w_k} = \mathbf{r_{k+1}} - \mathbf{r_k}$. The proof idea is that we can choose a chain satisfying (1) $|\mathbf{w_k}| = 1$ and (2) $\cos(\arg(\mathbf{w_k}) - \arg(\mathbf{r_k})) \geq 0$ for any k ($0 < k \leq D^2 + 1$)[3]. If the above conditions hold, it is easy to show $|\mathbf{r_{k+1}}|^2 \geq |\mathbf{r_k}|^2 + 1$ (see Fig. 3). This implies $|\mathbf{r_{D^2+1}}|^2 \geq (D^2 + 1)$, which contradicts the fact of $G(0)$'s diameter D.

The remaining part of the proof is to show the existence of the chain of length k satisfying (1) and (2) for $k = D^2 + 1$. We prove it by induction on k.

Basis: Since r_0 is blocked, the arc of r_0's visibility border with center angle $[-\pi/2, \pi/2]$ necessarily includes one robot. Thus, we can choose it as r_1, which satisfies the condition (1) and (2).

Inductive step: Suppose as induction hypothesis that we have the chain of length k. Then, we introduce the coordinate system originated at $\mathbf{r_k}$ and its x-axis is directed to the same angle as $\mathbf{r_k}$. Then by the same way as the basis, we can obtain r_{k+1}.

Consequently the lemma is proved. □

From this lemma, we can obtain a better upper bound for time complexity of the algorithm *CPS*:

Theorem 2. *The algorithm CPS achieves the connectivity-preserving scattering within* $O(\min\{n, D^2 + \log n\})$ *expected rounds.*

4 Lower Bound

In this section, we show that our $O(n)$-upper bound is tight: we identify an initial configuration for which any connectivity-preserving scattering algorithm takes $\Omega(n)$ rounds. We start the proof from the following proposition.

[3] This condition implies that the polar angle of $\mathbf{w_k}$ about $\mathbf{r_k}$ is in the range of $[-\pi/2, \pi/2]$.

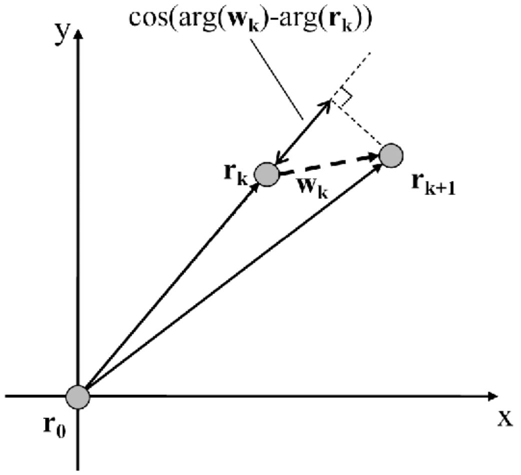


Fig. 3. The illustration used in the proof of Lemma 2

Proposition 1. *Consider any connectivity-scattering algorithm and the configuration of three robots* r_0, r_1, r_2 *located at* $(-1, 0), (0, 0), (1, 0)$. *Then if* r_1 *is activated, its destination is* $(0, 0)$ *(i.e.,* r_1 *does not move for its activation). Similarly, consider the configuration of four robots* r_0, r_1, r_2, r_3 *located at* $(-1, 0)$, $(0, 0)$, $(0, 0)$, $(1, 0)$. *Then if either* r_1 *or* r_2 *is activated, its destination is* $(0, 0)$.

The above proposition trivially holds because of connectivity-preserving requirement will be violated with non-zero probability if r_1 (or r_2) changes its position. Using this observation, we construct the worst-case configuration:

Theorem 3. *Let* C *be the configuration of* $2n$ *robots where* r_k *is located as follows:*

$$\mathbf{r_k} = \begin{cases} (k, 0) & (0, \leq k \leq n-1) \\ (k-1, 0) & (n \leq k \leq 2n) \end{cases}$$

Then, any scattering algorithm takes $\Omega(n)$ *rounds if the initial configuration is* C.

Proof. From Proposition 1, two robots on $(n-1, 0)$ never move unless either r_{n-2} or r_{n+1} changes its position. Similarly, to make r_k (r_{2n-k}) move, r_{k-1} (r_{2n-k-1}) must move in advance. Thus if we consider the round-robin schedule where each robot is activated in the order

$$r_n, r_{n-1}, r_{n+1}, r_{n-2}, \cdots, r_{n+j}, r_{n-j-1}, \cdots, r_{2n}, r_0$$

only two robots r_k and r_{2n-k} can change their positions during round k. This implies that it takes $\Omega(n)$ rounds that r_n and r_{n-1} are scattered. □

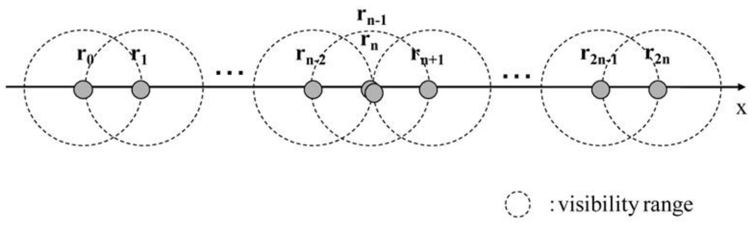


Fig. 4. The configuration C used in the proof of Theorem 3

5 Concluding Remarks

In this paper, we presented a probabilistic solution to the scattering problem in the limited visibility model. The proposed algorithm assumes ATOM (*i.e.* semi-synchronous) model, and guarantees the connectivity-preserving property. We also analysed its time complexity. The algorithm achieves scattering within $O(\min\{n, D^2 + \log n\})$ asynchronous rounds in expectation, where D is the diameter of the initial visibility graph. Because of the dependency on D, our algorithm quickly scatters robots located closely at the initial configuration. We also presented an initial configuration for which any scattering algorithm would take $\Omega(n)$ rounds in the worst case. In the sense of non-adaptive analysis, our algorithm is optimal for connectivity-preserving scattering.

There are several interesting open question raised by our work:

1. We only focused on the ATOM semi-synchronous model. Extending our result to the fully asynchronous CORDA model [8] looks challenging.
2. Combining the constraint of a maximal distance (for connectivity preservation) *and* a minimal distance (for evenly scattering the robots as in [6]) is an intriguing question with respect to the feasibility aspects.
3. The possibility that some robots misbehave (*i.e.* exhibit Byzantine behaviour [3]) has significant impact on connectivity preservation, since a Byzantine node may unilaterally choose to disconnect the visibility graph when finding it is an articulation point in this graph. We wish to pursue research in designing protocols that can cope with such behaviours.

References

1. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed memoryless point convergence algorithm for mobilerobots with limited visibility. IEEE Transactions on Robotics and Automation 15(5), 818–828 (1999)
2. Barriere, L., Flocchini, P., Mesa-Barrameda, E., Santoro, N.: Uniform scattering of autonomous mobile robots in a grid. In: International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–8 (2009)
3. Bouzid, Z., Potop-Butucaru, M.G., Tixeuil, S.: Optimal byzantine-resilient convergence in unidimensional robot networks. In: Theoretical Computer Science, TCS (2010)

4. Clement, J., Défago, X., Potop-Butucaru, M.G., Izumi, T., Messika, S.: The cost of probabilistic agreement in oblivious robot networks. Information Processing Letters 110(11), 431–438 (2010)
5. Dieudonné, Y., Petit, F.: Scatter of robots. Parallel Processing Letters 19(1), 175–184 (2009)
6. Flocchini, P., Prencipe, G., Santoro, N.: Self-deployment of mobile sensors on a ring. Theor. Comput. Sci. 402(1), 67–80 (2008)
7. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous robots with limited visibility. Theoreical Computer Science 337(1-3), 147–168 (2005)
8. Prencipe, G.: Instantaneous actions vs. full asynchronicity: Controlling and coordinating a set of autonomous mobile robots. In: Restivo, A., Ronchi Della Rocca, S., Roversi, L. (eds.) ICTCS 2001. LNCS, vol. 2202, pp. 154–171. Springer, Heidelberg (2001)
9. Souissi, S., Défago, X., Yamashita, M.: Using eventually consistent compasses to gather memory-less mobile robots with limited visibility. ACM Transactions on Autonomous and Adaptive Systems 4(1), 1–27 (2009)
10. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM Journal of Computing 28(4), 1347–1363 (1999)

Computing in Social Networks

Andrei Giurgiu[1], Rachid Guerraoui[1], Kévin Huguenin[2], and Anne-Marie Kermarrec[3]

[1] EPFL
[2] Université de Rennes 1 / IRISA
[3] INRIA Rennes - Bretagne Atlantique

Abstract. This paper defines the problem of Scalable Secure Computing in a Social network: we call it the S^3 problem. In short, nodes, directly reflecting on associated users, need to compute a function $f : V \rightarrow U$ of their inputs in a set of constant size, in a *scalable* and *secure* way. Scalability means that the message and computational complexity of the distributed computation is at most $\mathcal{O}(\sqrt{n} \cdot \operatorname{polylog} n)$. Security encompasses (1) accuracy and (2) privacy: accuracy holds when the distance from the output to the ideal result is negligible with respect to the maximum distance between any two possible results; privacy is characterized by how the information disclosed by the computation helps faulty nodes infer inputs of non-faulty nodes.

We present AG-S3, a protocol that S^3-computes a class of aggregation functions, that is that can be expressed as a commutative monoid operation on U: $f(x_1, \ldots, x_n) = x_1 \oplus \cdots \oplus x_n$, assuming the number of faulty participants is at most $\sqrt{n}/\log^2 n$. Key to our protocol is a dedicated overlay structure that enables secret sharing and distributed verifications which leverage the social aspect of the network: nodes care about their reputation and do not want to be tagged as misbehaving.

1 Introduction

The past few years have witnessed an explosion of online social networks and the number of users of such networks is still growing regularly by the day, e.g. Facebook boasts by now more than 400 millions users. These networks constitute huge live platforms that are exploited in many ways, from conducting polls about political tendencies to gathering thousands of students around an evening drink. It is clearly appealing to perform large-scale general purpose computations on such platforms and one might be tempted to use a central authority for that, namely one provided by the company orchestrating the social network. Yet, this poses several privacy problems, besides scalability. For instance, there is no guarantee that Facebook will not make any commercial usage of the personal information of its users. In 2009, Facebook tried to change its privacy policy to impose new terms of use, granting the company a perpetual ownership of personal contents – even if the users decide to delete their account. The new policy was not adopted eventually, but highlighted the eagerness of such companies to use personal and sensitive information.

We argue for a decentralized approach where the participants in the social network keep their own data and perform computations in a distributed fashion without any central authority. A natural question that arises then is what distributed computations can be performed in such a decentralized setting. Our primary contribution is to lay the ground for precisely expressing the question. We refer to the underlying problem as the S^3 problem: *Scalable Secure Computing in a Social network*. Whereas *scalability* characterizes the message and computational complexity of the computation, the *secure* aspect of S^3 encompasses accuracy and privacy. *Accuracy* refers to the robustness of the computation and aims at ensuring accurate results in the presence of dishonest participants. This is crucial in a distributed scheme where dishonest participants might, besides disrupting their own input, also disrupt any intermediary result for which they are responsible. The main challenge is to limit the amount of bias caused by dishonest participants. *Privacy* is characterized by the amount of information on the inputs disclosed to other nodes by the computation. Intuitively, achieving all three requirements seem impossible. Clearly, tolerating dishonest players and ensuring privacy calls for cryptographic primitives. Yet, cryptographic schemes, typically used for multi-party computations, involve too high a computation overhead and rely on higher mathematics and the intractability of certain computations [1,2,3]. Instead, we leverage users' concern for reputation using a information theoretical approach and alleviate the need for cryptographic primitives. A characteristic of the social network context is indeed that the nodes are in fact users who might not want to reveal their input, nor expose any misbehavior. This reputation concern determines the extent to which dishonest nodes act: up to the point that their misbehavior remains discrete enough not to be discovered.

Solving the S^3 problem is challenging, despite leveraging this reputation concern: to ensure privacy, an algorithm must ensure that even when all the nodes except one have the same inputs, the information obtained by the coalition of faulty nodes cannot know which non-faulty node had a different input. This requires the existence of two configurations of inputs that differ for two nodes, which with high probability lead to the same sequence of messages received by the faulty nodes. In turn, this boils down to *swapping* two nodes' inputs transparently (from the standpoint of the faulty nodes), which is challenging when the protocol needs to be also scalable and accurate. The scalability requirement (i.e., each node communicates with a limited number of nodes) makes it difficult to find a chain of messages that can be swapped transparently between two nodes in the system. The trade-off between privacy and accuracy can be illustrated by the following paradox: on the one hand verifying that nodes do not corrupt the messages they receive (without digital signature) requires the verifier to gather some information about what the verified node received; on the other hand the more the nodes know about the messages exchanged the more the privacy of the nodes is compromised.

Our contributions are twofold. Firstly, we define the Scalable Secure Computing problem in a Social network, namely the S^3 problem. Secondly, we present a distributed protocol, we call AG-S3 (i.e., S^3 for AGgregation), that solves the problem for a class of aggregation functions that derive from a monoid operation on U: $f(x_1, ..., x_n) = x_1 \oplus \cdots \oplus x_n$, under the assumption that the number of faulty nodes is upper-bounded by $\sqrt{n}/\log^2 n$. At the core of our protocol lie (1) a structured overlay where nodes are clustered into groups, (2) a secret sharing scheme that allows the nodes to obfuscate their inputs, and (3) a verification procedure which potentially tags the profiles of suspected nodes. Beyond these contributions, our paper can be viewed as a first step toward characterizing what can be computed in a large scale social network while accounting for the human nature of its users.

2 Problem

This section defines the problem of *Scalable Secure* Computing in a *Social network*: the S^3 problem. The problem involves a S^3 candidate, namely the function to be computed, and a set of nodes $\Pi = \{p_1, \ldots, p_n\}$.

2.1 Candidates

Definition 1 (S^3 candidate). *A S^3 candidate is a quadruple (f, V, U, d), where V is an arbitrary set, f is a function $f : V^* \to U$ such that $f(v_1, \ldots, v_n) = f(v_{\sigma(1)}, \ldots, v_{\sigma(n)})$ for any permutation σ of the inputs, and (U, d) is a metric space.*

Each node in Π has an input value in the set V, and a S^3 candidate maps the inputs of the nodes to a value in a metric space. The function f is assumed to be symmetric in the sense that the output depends on the multiset of inputs but not on their assignation to nodes. For example, a binary poll over Π can be modeled by the S^3 candidate $((v_1, v_2, \ldots, v_n) \mapsto v_1 + \cdots + v_n, \{-1, +1\}, \mathbb{Z}, (z_1, z_2) \mapsto |z_1 - z_2|)$. Consider also a component-wise addition on $U = \mathbb{Z}^d$, where V is the set of all vectors with exactly one nonzero component, which is either $+1$ or -1. The distance function is then just the L_1 (or Manhattan distance).

The nodes considered in the S^3 problem are users of a social network, able to (1) communicate with private message passing and (2) tag the public profile of each other. As such, every node directly reflects on the associated user. Nodes care about their privacy and their reputation: a user wants neither the private information contained in her input, nor her misbehavior, if any, to be disclosed. This reputation concern is crucial to make the problem tractable. To ensure security, part of the computation consists in checking the correctness of other nodes' behavior. The output of a node p is a value in U plus a set $\mathcal{F}_p$ of nodes that p detected as faulty. This information is eventually reported on the public profile of the involved nodes by means of tags of the form "p detected nodes in $\mathcal{F}_p$ as faulty".

Faulty nodes are considered rational: their goal is only to bias the output of the computation and infer the inputs of the users taking part in the computation. As such, their behavior is more restricted than that of Byzantine users [4]. To achieve their goal, faulty nodes may collude.

In our context, a distributed computation $\mathcal{D}$ on the set of nodes Π, is a sequence of message exchanges and local computations such that any non-faulty node p eventually outputs a value o_p. The content of the message and the nodes' outputs are random variables whose value is determined by the random choices made by the nodes during the computation. In the following, we define the desirable properties of a distributed computation in a social network, namely scalability and security, itself encompassing privacy and accuracy.

2.2 Scalability

Scalability means that the computation is able to handle a large number of inputs (i.e., large values of n): consequently, the properties are expressed in the form of asymptotic bounds.

Definition 2 ($\sqrt{\ }$-Scalability). *A distributed computation is said to be $\sqrt{\ }$-scalable if the message, spatial and computational complexities at each node are $\mathcal{O}(\sqrt{n} \cdot \mathrm{polylog}\, n)$.*

The intuition behind the logarithmic factor in the asymptotic bound is that operations with the nodes' identifiers and the memory needed to store such identifiers remain within $\mathcal{O}(\log n)$.

2.3 Accuracy

The definition of the accuracy of a computation relies on the metric space structure of the output space U: accuracy is given by the distance between the output of the computation and the actual value of the output of f. To render it meaningful, we normalize this distance by the diameter of $f(V^n)$ for a distributed computation over n nodes.

Definition 3 ($\sqrt{\ }$-Accuracy). *A distributed computation $\mathcal{D}$ is said to $\sqrt{\ }$-accurately compute a S^3 candidate (f, U, V, d) if:*

$$\frac{1}{\Delta(n)} \cdot \max_{p \text{ non-faulty}} d(o_p, f(v_1, \ldots, v_n)) = \mathcal{O}\left(\frac{1}{\sqrt{n}}\right),$$

where v_i is the input of the i-th node and

$$\Delta(n) = \max_{\substack{(x_1, \ldots, x_n) \\ (y_1, \ldots, y_n)}} d(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n)).$$

This definition highlights the importance of specifying the distance measure of a S^3 candidate: providing the output space with the coarse grain distance $d(x, y) = 0$ if $x = y$, and 1 otherwise, will restrict the class of S^3 computations to those that output the exact value of f. Meanwhile, for binary polling for instance Dpol [5], considering the natural distance on relative numbers includes computations for which the error on the tally is negligible when compared to the sample size n (i.e., $\Delta(n) = 2n$).

2.4 Privacy

Privacy characterizes how the information gained by curious nodes taking part in the distributed computation enables them to recover the input of a particular non-faulty node. Clearly, the cases where an input can be inferred from only the output and the inputs of the faulty nodes are ignored when looking at the privacy leaks of a computation. In a perfectly private distributed computation, a coalition of faulty nodes should be able to recover the input of a non-faulty node if and only if its input can be inferred from the output of the computation and the inputs of the faulty nodes. Such configurations of inputs are captured by the notion of *trivial* inputs. An example of such configuration of inputs is the case where all non-faulty nodes taking part in a binary poll have the same input, be it -1 or 1. Since S^3 candidates are symmetric by definition, a trivial input is a configuration where all nodes start with the same input.

Definition 4 (Trivial input). *An element v of V^* is said to be a trivial input for a coalition B if there is a node $p \notin B$ such that for all input configuration v' that coincides with v for all nodes in B, $f(v) = f(v')$ implies $v_p = v'_p$.*

We say in our context that a distributed computation is *private* if the probability of recovering the input of a particular non-faulty node (assuming that it cannot be inferred from the output alone, i.e., the configuration of inputs is non-trivial) decreases as $1/n^\alpha$ for some positive α. We capture this notion more formally through the notion of *probabilistic anonymity*, itself based on the very notion of *message trace.*

Definition 5 (Message trace). *A* message trace *(or* trace *for short) of a distributed computation is the collection of messages sent in a possible execution of a program. A trace is said to be* compatible *with an input configuration v if the trace can be obtained from v with a nonzero probability. We say that two traces are equivalent with respect to a coalition of faulty nodes B if each node in B receives the exact same messages in both traces.*

We are ready now to introduce the concept of probabilistic anonymity, which encapsulates the degree of privacy we require.

Definition 6 (Probabilistic anonymity). *A distributed computation $\mathcal{D}$ is said to be probabilistically anonymous if for any coalition of faulty nodes B, for any non-faulty node p, and for any trace D compatible with a non-trivial (w.r.t. B) input configuration v, there exists with high probability a trace D' compatible with an input configuration v' such that (1) D and D' are equivalent w.r.t. B and (2) v and v' differ on the input value of node p.*

The intuition behind this definition is that a coalition of faulty nodes cannot distinguish, with high probability, different executions of a computation in which non-faulty nodes had different inputs.

Definition 7 (S^3 computation). *A distributed computation is said to S^3-compute a S^3 candidate if it is $\sqrt{\ }$-scalable, $\sqrt{\ }$-accurate and probabilistically anonymous with respect to the candidate.*

3 Protocol

In this section, we focus on a class of aggregation functions and propose a protocol, namely AG-S3 (S^3 for AGgregation), which S^3-computes such functions for $|B| \leq \sqrt{n}/\log^2 n$ faulty nodes.

3.1 Assumptions

We consider S^3 candidates for which the function f is an aggregation function, i.e. deriving from an associative binary operation on U: $f(v_1 \ldots, v_n) = v_1 \oplus \cdots \oplus v_n$. Because a S^3 candidate must be symmetric, the '$\oplus$' operation is commutative. This induces a commutative monoid structure on $(U, \oplus)$ and it implies that V is a subset of U. We further assume that the '$\oplus$' operation is *compatible* with the distance measure d in the sense that

$$d(v_1 \oplus v_2, v_1' \oplus v_2') \leq d(v_1, v_1') + d(v_2, v_2') \ . \tag{1}$$

As an example, note that the S^3 candidate $((v_1, v_2, \ldots, v_n) \mapsto v_1 + \cdots + v_n, \{-1, +1\}, \mathbb{Z}, (z_1, z_2) \mapsto |z_1 - z_2|)$, introduced in the previous section, satisfies the compatibility condition described above. A simple example of S^3 candidate which cannot be expressed as an aggregation is the one given by the sum of products of pairs of inputs, i.e. $f(x_1, \ldots, x_n) = x_1 \cdot x_2 + x_1 \cdot x_3 + x_2 \cdot x_3 + \ldots$. This function is symmetric, and choosing $U = \mathbb{Z}$ turns this function into a valid S^3 candidate, but it is clearly not an aggregation function.

We assume the size of the set of possible inputs to be constant and the size of the output space to be polynomial in n implying that any input or output can be represented by $\mathcal{O}(\log n)$ bits. In addition, we assume that the diameter $\Delta(n)$ of the output space is $\Omega(n)$. Due to this assumption, bit operators do not fall into our definition. Finally, we assume that V is closed with respect to inverses: if v is in the input set V then $\ominus v$ is in V as well, where $\ominus v$ denotes the inverse of v with respect to the '$\oplus$' operation. We denote by δ_V the diameter of V: $\delta_V = \max_{v,v' \in V} d(v, v')$.

3.2 Design Rationale

The main challenge of S^3 computing is the trade-off between scalability and accuracy on the one hand and privacy on the other hand. We describe below this trade-off and how we address it before describing the protocol in details.

To ensure scalability, we cluster the nodes into groups of size $\sqrt{n}$, and require that a node sends messages only to other nodes in a small set of neighboring groups. We introduce two parameters of the protocol, κ and l. A non-faulty node p is allowed to send messages to any other node in its own group, and to exactly l nodes in each of κ other groups. For scalability, l and κ need to be low, since they are directly proportional to message complexity. The same for accuracy: intuitively, the larger l and κ, the more opportunities a node has to cheat (i.e., corrupt the unique pieces of information it receives before forwarding them), which entails a higher impact on the output. To preserve privacy (i.e. probabilistic anonymity), we need a mechanism which, for any node p, transforms any trace (i.e. input values and messages) into another trace, in such a manner that all messages received by the coalition of faulty nodes are preserved, and p has a different input in the two traces. This prevents the coalition from determining the input value of p. It will become apparent in our proof of privacy that both κ and l need to be large in order to obtain reasonable privacy requirements. To summarize, accuracy and scalability require the parameters κ and l to be small, whereas privacy requires them to be large. As a trade-off, we pick them both to be $\Theta(\log n)$, which reasonably ensure the S^3 requirements.

3.3 Protocol

We describe AG-S3 which computes general aggregation in a S^3 manner: the protocol is composed of two interleaved components: one computes the aggregation function while the other checks the behavior of users. The pseudo-code of all is given in Algorithms 1-4.

Structure. AG-S3 uses a special structure inspired from [6], where the n nodes are distributed into groups of size $\sqrt{n}$. Such an overlay can be obtained in a distributed fashion with strong guarantees on the randomness of nodes placement in the groups even in the presence of malicious users [7]. The groups (or *offices*) are placed in a ring, with nodes from a particular group sending messages to either nodes from the same office (called *officemates*) or to selected nodes from the next offices on the ring (called *proxies*). More specifically, a node is connected to its $\sqrt{n}$ officemates and to l proxies in each of the next κ groups on the ring. If a node p' is a proxy of p, then p is said to be a *client* of p'. The partitioning into groups and their placement on the ring are chosen uniformly at random. We further assume a perfect client-proxy matching that ensures that a proxy has exactly $\kappa \cdot l$ clients. For example, we can index the nodes inside each group and assign to the i-th node of a group the nodes $i+1, \ldots, i+l \mod \sqrt{n}$ as proxies in each of the next κ groups on the ring. We set $\kappa = 3/2 \cdot \lfloor \log n \rfloor$ and $l = 5 \cdot |V| \cdot \lfloor \log n \rfloor + 1$. These choices are motivated in the next section.

Aggregation. In the first phase, each participant splits its input into $\kappa \cdot l$ *shares* in V and sends them randomly to its assigned proxies. The randomized scheme ensures that the aggregate of the shares is the input value. The shares are generated as follows: $(\kappa \cdot l - 1)/2$ are chosen uniformly at random, $(\kappa \cdot l - 1)/2$ are the inverses of the randomly chosen shares, and one is the actual input of the node.

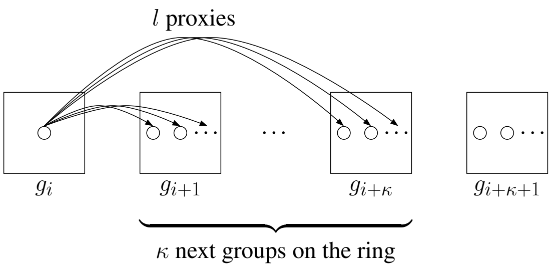


Fig. 1. Overview of the overlay

```
1 procedure share_input( $v$ );
2   for $i \leftarrow 1$ to $(l \cdot \kappa - 1)/2$ do
3       $s_i \leftarrow_{\text{rand}} V$   # random values in $V$;
4       $s_{i+(l\cdot\kappa-1)/2} \leftarrow \ominus s_i$;
5   $s_{l\cdot\kappa} \leftarrow v$   # the actual input;
6   $\sigma \leftarrow_{\text{rand}} \mathcal{S}_{l\cdot\kappa}$   # random permutation to distribute the shares;
7   for $i_{\text{group}} \leftarrow 1$ to $\kappa$ do
8       for $i_{\text{proxy}} \leftarrow 1$ to $l$ do
9           send (SHARE, $p_{i_{\text{group}},i_{\text{proxy}}}$, $s_{\sigma(i_{\text{group}}\cdot l+i_{\text{proxy}})}$);
```

Algorithm 1. Input sharing

In the counting phase, each proxy aggregates the shares received in the previous phase to obtain an *individual aggregate*. Each node then broadcasts its individual aggregate to all its officemates. Each node computes the aggregate of the individual aggregates of its officemates and obtains a *local aggregate*. If all nodes are non-faulty, then all local aggregates computed in an office are identical.

```
1 upon event receive (SHARE, $c$, $s$) do
2   Verify $c$ is a client;
3   Verify $s$ is a valid input in $V$   # $s \in V$;
4   $u_{\text{ind}} = u_{\text{ind}} \oplus s$;
```

Algorithm 2. Individual aggregation

In the *forwarding phase*, the local aggregates are disseminated to other nodes thanks to tokens forwarded along the ring, as explained below. The forwarding phase is bootstrapped by a special group (that can be determined by the social networking infrastructure at random). The nodes in this special group send a token containing the local aggregate computed in their group to their proxies from the next group. The tokens are further forwarded along the ring. The first time a token reaches a node in a particular group, this node aggregates the local aggregate to the token and forwards it to its proxies in the next group. When a

```
1 procedure local_count();
2   foreach officemate $o$ do
3       send (INDIVIDUAL_AGG, $o$, $u_{\text{ind}}$);
```

Algorithm 3. Local aggregate broadcast

```
1 upon event receive (INDIVIDUAL_AGG, $o$, $u$) do
2   Verify $u$ is a valid aggregate of $\kappa \cdot l$ shares;
3     # $d(u, v_1 \oplus \cdots \oplus v_{\kappa \cdot l}) \leq \kappa \cdot l \cdot \delta_V$ where $v_1 \oplus \cdots \oplus v_{\kappa \cdot l}$ are random values in $V$;
4   $u_{\text{local}} \leftarrow u_{\text{local}} \oplus u$ ;
```

Algorithm 4. Local aggregation

node receives a token for the second time, the node sets its own output to the value of the token and forwards it. The third time a node receives a token, it discards it.

Verifications. The purpose of verifications is to track nodes that deviate from the protocol. This is achieved by leveraging the value attached by the nodes to their reputation. The basic mechanism is that misbehaviors are reported by the participants who discover a faulty node and subsequently tag the latter's profile. The verifications are performed in each phase of the protocol. In the sharing phase, each proxy verifies that the shares received are valid input values. In the second phase, each node checks whether the distance between the individual aggregates sent and some random valid individual aggregate is at most $\kappa \cdot l \cdot \delta_V$. The reason for this is that due to the compatibility of the distance function with the monoid operation, for any $v_1, \ldots, v_k, v'_1, \ldots, v'_k \in V$, we have that

$$d(v_1 \oplus \cdots \oplus v_k, v'_1 \oplus \cdots \oplus v'_k) \leq d(v_1, v'_1) + \cdots + d(v_k, v'_k) \leq k \cdot \delta_V.$$

The verification in the third phase works as follows: if all the tokens received by a node in a given round (remember that tokens circulate up to three times around the ring) are not the same, then an alarm is raised and the profiles of the involved nodes are tagged. Otherwise, the node broadcasts the unique value of the tokens it received to its officemates. If it is not the case that all values broadcast are equal, again an alarm is raised.

3.4 Correctness

We prove here that AG-S3 satisfies the S^3 conditions for $|B| \leq \sqrt{n}/\log^2 n$.

Theorem 1 (Scalability). *The AG-S3 protocol is $\sqrt{\ }$-scalable.*

Proof. The nodes need to maintain a list of officemates, a list of proxies, and a list of clients. This amounts to $\mathcal{O}(\sqrt{n} \cdot \log n)$ space complexity as nodes' identifiers can be represented using $\mathcal{O}(\log n)$ bits. The message complexity is similarly $\mathcal{O}(\sqrt{n})$ arising from the following components: a node sends $\kappa \cdot l = \mathcal{O}(\log^2 n)$ shares during the sharing phase, $\mathcal{O}(\sqrt{n})$ copies of its individual aggregate in the counting phase, and $\mathcal{O}(\sqrt{n})$ in the forwarding phase. $\square$

Theorem 2 (Accuracy). *The AG-S3 protocol is $\sqrt{}$-accurate.*

Proof. A faulty node can bias the output of the computation by either sending an invalid set of shares, changing the value of its individual aggregate, or corrupt the aggregate during the forwarding phase. However, a node never misbehaves in a way that this is exposed with certainty (by the verifications presented in the previous section).

Sharing: Not to be detected, a node must send shares in V. Therefore, the distance between the sum of a node's shares and a valid input is at most $\kappa \cdot l \cdot \delta_V$.

Counting: Suppose that a faulty node changes its individual aggregate from $v = v_1 \oplus \cdots \oplus v_{\kappa \cdot l}$ to some value u. When its officemates receive its individual aggregate u they compute the distance between this aggregate and an arbitrary aggregate $w = w_1 \oplus \cdots \oplus w_{\kappa \cdot l}$. If this distance is larger than $\kappa \cdot l \cdot \delta_V$ then the misbehavior is reported. If the distance is within the bound, the triangular inequality yields an upper-bound on the maximum impact: $d(u, v) \leq d(u, w) + d(w, v) \leq 2\kappa \cdot l \cdot \delta_V$.

Forwarding: To corrupt a token without being detected, the coalition of faulty nodes must *fool* (i.e., make a node decide and forward a corrupted token without raising an alarm) all the non-faulty nodes of a group. Otherwise the corruption is detected by the verification consisting in a node broadcasting the token received to its officemates. To fool a single non-faulty node, all the l tokens it received from its clients (remember that nodes forward tokens only to their proxies in the next group) must be equal. Since nodes have l proxies in the next group, f faulty nodes can fool up to f non-faulty nodes. Assuming that a group contains f non-faulty nodes (and $\sqrt{n} - f$ faulty nodes), then corrupting a token without being detected requires another f faulty nodes in preceding groups. That is a total of $\sqrt{n}$ faulty nodes which cannot happen under the assumption $|B| \leq \sqrt{n}/\log^2 n$. To conclude, the local aggregates cannot be corrupted during the forwarding phase.

The impact of a faulty node on the output of the computation is bounded by $3\kappa \cdot l \cdot \delta_v$. We have $|B| \leq \sqrt{n}/\log^2 n$, $\kappa = \mathcal{O}(\log n)$, $l = \mathcal{O}(\log n)$ and $\Delta(n) = \Omega(n)$. Putting everything together, we get that the accuracy of definition 3 is $\mathcal{O}(\sqrt{n}/\log^2 n \cdot \log n \cdot \log n/n) = \mathcal{O}(1/\sqrt{n})$, which concludes the proof. $\square$

Theorem 3 (Probabilistic anonymity). *The AG-S3 protocol is probabilistically anonymous.*

Proof. We need to show that, with high probability, there exists a mechanism that for any node p, transforms any trace in such a way that the coalition of faulty nodes receives the same messages, but p has a different input. We first give an outline of the proof.

The transformation mechanism consists of changing the values transmitted between non-faulty nodes, in such a way that any subsequent message sent by non-faulty nodes to the nodes in the coalition does not change. As a result, the coalition receives the same information. The basic idea of this mechanism is to *swap* the inputs of two nodes p_1 and p_2, provided that there is a non-compromised group g (a group with no faulty nodes) that contains proxies of both p_1 and p_2. In this case, we can modify the shares sent by p_1 and p_2 to proxies in g, in such a way that the local aggregate of g is maintained. Since we assume that all nodes in g are non-faulty, the coalition does not have access to information exchanged in g during the counting phase. The coalition only sees what the nodes in g decide to broadcast in the forwarding phase, but that is identical to what is sent in the original trace. To modify the shares of p_1 and p_2, we assume that both send a share containing their own input to some proxies in g. Each of p_1 and p_2 has l proxies in g, so the larger l is, the larger the probability that our assumption is true. Then the aforementioned shares of p_1 and p_2 are swapped, resulting a consistent trace, where p_1 and p_2 swapped input values.

In case there is no such common non-compromised group g for p_1 and p_2, we may still find a chain of nodes with endpoints p_1 and p_2, such that two consecutive nodes in the chain can swap input values. The larger κ, the larger the probability that such a chain exists. Afterwards, the nodes can swap shares along the chain, resulting in a consistent configuration where p_1 has as input the old input value of p_2. The rest of the proof is concerned with making our outline description precise.

Let D be a trace of AG-S3 compatible with a non-trivial input v, B be a coalition of faulty nodes ($|B| \leq \sqrt{n}/\log^2 n$) and p be a non-faulty node. Since the input is non-trivial, there exists a node p' whose input is different from the input of p in v, and we prove that with high probability there exists a trace equivalent to D compatible with an input configuration v' which is the same as v, except that the inputs of p and p' have been swapped.

We say that a group compromised if it contains at least one faulty node. The coalition of faulty nodes knows the local aggregates of all the groups, the individual aggregates of the proxies in the compromised groups, the shares they received and their own inputs.

We first prove the following lemma.

Lemma 1. *The probability that in any sequence of* $\kappa - 1$ *consecutive groups there is at least one non-compromised group, is at least* $1 - \sqrt{n}\left(\frac{|B|}{\sqrt{n}}\right)^{\kappa-1}$.

Proof. This probability is minimized if no two faulty nodes lie in the same group, i.e. there are $|B|$ compromised groups. Fix $\kappa - 1$ consecutive groups. The number of configurations in which these groups are compromised is $\binom{\sqrt{n}-\kappa+1}{|B|-\kappa+1}$. The total number of configurations is $\binom{\sqrt{n}}{|B|}$, so the probability that all the fixed k consecutive groups are compromised is given by the ratio of the two binomial coefficients, which is upper-bounded by $(|B|/\sqrt{n})^{\kappa-1}$. We use the union bound to upper-bound the probability that there is at least one such sequence of $\kappa - 1$ consecutive compromised groups. There are $\sqrt{n}$ sequences of $\kappa - 1$ consecutive groups, which proves the lemma. □

Since $\kappa = 3/2 \cdot \lfloor \log n \rfloor$ and $|B| \leq \sqrt{n}/\log^2 n$, we get that the probability of having κ consecutive compromised groups is at most $1/n$.

Lemma 2. *Given $x \in V$, the probability that a node sends at least one share of value x to a proxy situated in a given group, assuming this node has proxies in that group, is at least $1 - 1/n^3$.*

Proof. The l shares sent to a group by a node are randomly picked from a set of $\kappa \cdot l$ shares in which $(\kappa \cdot l - 1)/2$ are random, $(\kappa \cdot l - 1)/2$ are the inverses of the random shares, and one is the actual input of the node. At least $(l-1)/2$ of them are independent, and drawn uniformly at random from V. Thus, the probability that a is not one of them is at most $(1 - 1/|V|)^{(l-1)/2}$. Since $(l - 1)/2 = 5/2 \cdot |V| \cdot \lfloor \log n \rfloor$, this probability is upper-bounded by $1/n^{5/2}$, which proves the lemma. □

Let $g(\cdot)$ denote the index of a group in which a node lies. Without loss of generality, we assume that $g(p) = 0$. Since we assume that the input v is not trivial, let p' be a node such that its input v'_p is different from the input of p, i.e., v_p. Let $i_1, \ldots, i_M$ be a sequence of indexes such that: (1) group g_{i_m} is non-compromised for all m, (2) $0 < i_1 < \kappa$, (3) $0 < i_{m+1} - i_m < \kappa$ for all m, and (4) $0 < i_M - g(p') < \kappa$. Such a sequence exists with high probability according to Lemma 1. For all $1 \leq m < M$, we define p_m as an arbitrary non-faulty node in group $g_{i_m - 1}$. Additionally, we set $p_0 = p$ and $p_M = p'$. Since all nodes have proxies in the κ groups succeeding them, we have that for all $1 \leq m \leq M$, p_{m-1} and p_m both have proxies in g_{i_m} as depicted in Figure 2.

Using Lemma 2 and using an union bound on the $1 \leq m \leq M$, we get that the probability that for all $1 \leq m \leq M$, p_{m-1} sends a share of value v_p to a proxy in g_m and p_m sends a share of value v_p to a proxy in g_m, is at least $1 - 2M/n^{5/2}$. Since M is bounded by the number of groups, namely $\sqrt{n}$, this probability is lower-bounded by $1 - 2/n^2$.

Assuming that this event occurs, we exhibit a trace compatible with a configuration of inputs where the inputs of p and p' are swapped: for all $1 \leq m \leq M$, the v_p share sent by p_{m-1} to g_{i_m} is replaced by v'_p and the v'_p share sent by p_m to g_{i_m} is replaced by v_p, as illustrated in Figure 2. This trace is equivalent to D with respect to the coalition B as no share sent to a compromised group is changed and all local aggregates remain the same.

We complete the proof by showing that this trace is indeed compatible with the modified configuration of inputs. In the case of AG-S3, compatible means that the set of shares sent by a node is composed of $(\kappa \cdot l - 1)/2$ values of V, their inverses, and the actual input of the node. For p and p', we only change the value of one share equal to their inputs. Therefore, their set of shares remains compatible with their new inputs. For the other nodes p_m, $0 < m < M$, two of their shares are simply swapped.

We proved that the privacy of *a given* non-faulty node p is preserved with probability at least $1 - 2/n^2$, given that the event of Lemma 1 occurs. Since the probability of this event is large (according to Lemma 1), using Bayes rule it is clear that $1 - 3/n^2$ is an upper bound on the probability that privacy of a particular node is preserved. Using a union bound over the whole set of at most n non-faulty node nodes, we obtain that probabilistic anonymity as defined in Definition 6 is preserved with probability $1 - 2/n$. □

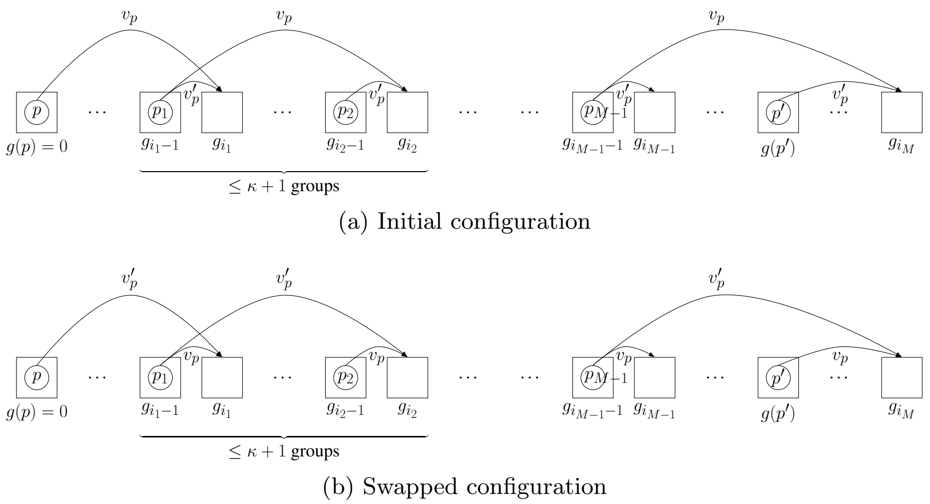


Fig. 2. Illustration of the proof of privacy: pairs of shares sent in the same group can be swapped ((a) → (b)) leading to an equivalent trace compatible with a different configuration of inputs.

4 Related Work

Cryptographic primitives and secure multi-party computation [1,2,3] allow to compute aggregation functions in a secure way. This comes however at the price of non-scalability. Assuming trust relationships between users of a social network, Vu *et al.* [8] proposed an improved secret sharing scheme to protect privacy. In that scheme, the actual relationships between nodes are used to determine the trustworthy participants, and the shares are only distributed to those. In contrast, AG-S3 exploits solely the human nature of social networks without making any assumptions on the social relationships themselves.

The population protocol model of *et al.* [9] provides a theoretical framework of mobile devices with limited memory, which relates to the scalability requirement of the S^3 problem. The model however can only compute first order formulas in Presburger arithmetic [10] and can tolerate only a constant number of benign failures [11]. The community protocol model [12] relax the scalability requirements on the memory sizes of tiny agents which enables powerful computations and Byzantine fault-tolerance. Yet, the model breaks anonymity as agents are assigned unique ids. This illustrates the trade-off between the power and security of a model on one hand and privacy on the other hand. The problem of privacy in population protocols was also tackled in [13]. The sharing scheme of AG-S3 is inspired by the obfuscation mechanism proposed in that paper, namely adding unit noise (+1 or -1) to their inputs, upon a state exchange. Dpol [5], itself also inspired by [13], can be be viewed as a restricted form of AG-S3. Dpol is restricted to binary polling: it aggregates values in $\{-1, +1\}$ and it uses a rudimentary secret sharing scheme and overly structure that assume *(i)* a uniform distribution of inputs, and *(ii)* a built-in anonymous overlay: these are the two main difficulties of the privacy challenge as defined in the S^3 problem.

Differential privacy [14] and k-anonymity [15] are two common ways to express privacy in the context of distributed computations on sensitive databases. Contrary to AG-S3, where faulty nodes take part in the computation, those techniques aim at protecting the privacy of inputs from an external attacker that queries the database. Differential privacy characterizes the amount of information disclosed by the output by bounding the impact of a single input on the output. It is typically achieved by adding noise to the output. However, as pointed out in [16], differential privacy does not capture the cases of *rare input configurations* due to the multiplicative bounds in its formulation, which is precisely the difficult case we need to address in the S^3 problem, i.e., the case where everybody but one node have the same inputs. The obfuscating technique consisting in adding noise to intermediate results cannot be used in the context of S^3 computing. The granularity of noise may indeed by high if elements of V are far away. In addition, it gives more opportunities to faulty nodes to bias the output of the computation. On the other hand, k-anonymity guarantees that any input value maps to at least k input nodes. In the S^3 problem, privacy can be seen as 2-anonymity with high probability, expressed in a distributed setting. With AG-S3, faulty nodes cannot map any input to a restricted subset of nodes as any two nonfaulty nodes can swap their inputs transparently. It thus ensures $n - B$-anonymity with high probability.

5 Conclusion

Social networks constitute now huge platforms on which it is very tempting to perform large scale computations. Yet, such computations are challenging as one needs to ensure privacy, scalability and accuracy. We leverage the very fact that, in such platforms, behind every node lies a respectable user who cares about his reputation, in order to make the problem tractable. We define what the notion of computation means in that context and propose a protocol that computes a class of aggregation functions. This is a first step toward understanding what can be computed in a social network and many open questions are left open such as what is the maximum number of faulty nodes a S^3 protocol can tolerate and what else besides aggregation functions can be computed in a S^3 manner?

References

1. Benaloh, J.: 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. Rivest, R., Shamir, A., Tauman, Y.: How to Share a Secret. CACM 22, 612–613 (1979)
3. Yao, A.C.: Protocols for Secure Computations. In: FOCS, pp. 160–164 (1982)
4. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM TPLS 4(3), 382–401 (1982)
5. Guerraoui, R., Huguenin, K., Kermarrec, A.M., Monod, M.: Decentralized Polling with Respectable Participants. In: OPODIS, pp. 144–158 (2009)
6. Galil, Z., Yung, M.: Partitioned Encryption and Achieving Simultaneity by Partitioning. Information Processing Letters 26(2), 81–88 (1987)
7. Gupta, I., Birman, K., Linga, P., Demers, A., van Renesse, R.: Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 160–169. Springer, Heidelberg (2003)
8. Vu, L.H., Aberer, K., Buchegger, S., Datta, A.: Enabling secure secret sharing in distributed online social networks. In: ACSAC, pp. 419–428 (2009)
9. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in Networks of Passively Mobile Finite-state Sensors. Distributed Computing 4, 235–253 (2006)
10. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The Computational Power of Population Protocols. Distributed Computing 20, 279–304 (2007)
11. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Ruppert, E.: When Birds Die: Making Population Protocols Fault-tolerant. In: Gibbons, P.B., Abdelzaher, T., Aspnes, J., Rao, R. (eds.) DCOSS 2006. LNCS, vol. 4026, pp. 51–66. Springer, Heidelberg (2006)
12. Guerraoui, R., Ruppert, E.: Names Trump Malice: Tiny Mobile Agents Can Tolerate Byzantine Failures. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 484–495. Springer, Heidelberg (2009)
13. 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)
14. Dwork, C.: Differential Privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006)
15. Samarati, P.: Protecting Respondents' Identities in Microdata Release. TKDE 13, 1010–1027 (2001)
16. Roy, I., Setty, S.T., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: Security and Privacy for MapReduce. In: NSDI (2010)

On Transactional Scheduling in Distributed Transactional Memory Systems

Junwhan Kim and Binoy Ravindran

ECE Department, Virginia Tech, Blacksburg, VA, 24061
{junwhan,binoy}@vt.edu

Abstract. We present a distributed transactional memory (TM) scheduler called *Bi-interval* that optimizes the execution order of transactional operations to minimize conflicts. *Bi-interval* categorizes concurrent requests for a shared object into read and write intervals to maximize the parallelism of reading transactions. This allows an object to be simultaneously sent to nodes of reading transactions (in a data flow TM model), improving transactional makespan. We show that *Bi-interval* improves the makespan competitive ratio of the Relay distributed TM cache coherence protocol to $O(\log(n))$ for the worst-case and $\Theta \log(n-k)$ for the average-case, for n nodes and k reading transactions. Our implementation studies confirm *Bi-interval*'s throughput improvement by as much as 200% $\sim$ 30%, over cache-coherence protocol-only distributed TM.

Keywords: Transactional Memory, Transactional Scheduling, Distributed Systems, Distributed Cache-Coherence.

1 Introduction

Transactional memory (TM) is an alternative synchronization model for shared in-memory data objects that promises to alleviate difficulties with lock-based synchronization (e.g., lack of compositionality, deadlocks, lock convoying). A transaction is a sequence of operations, performed by a single thread, for reading and writing shared objects. Two transactions *conflict* if they access the same object and one access is a write. When that happens, a contention manager (CM) is typically used to resolve the conflict. The CM resolves conflicts by deciding which transactions to abort and aborting them, allowing only one transaction to proceed, and thereby ensures atomicity. Aborted transactions are retried, often immediately. Thus, in the contention management model, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). Efficient contention management ensures transactional *progress*—i.e., at any given time, there exists at least one transaction that proceeds to commit without interruption [19]. TM for multiprocessors has been proposed in hardware [11], in software [12], and in hardware/software combination [16].

A complimentary approach for dealing with transactional conflicts is *transactional scheduling*. Broadly, a transactional scheduler determines the ordering of transactions so that conflicts are either avoided altogether or minimized. This includes serializing transaction executions to avoid conflicts based on transactions' predicted read/write access sets [8] or collision probability [7]. In addition, conflicts can be minimized by carefully deciding when a transaction that is aborted due to a conflict is resumed [1,7], or when a transaction that is stalled due to potential for an immediate conflict is later dispatched [21]. Note that, while contention management is oblivious to transactional operations, scheduling is operation-aware, and uses that information to avoid/minimize conflicts. Scheduling is not intended as a replacement for contention management; a CM is (often) needed and scheduling seeks to enhance TM performance.

Distributed TM promises to alleviate difficulties with lock-based distributed synchronization [13,4,17,15,23]. Several distributed TM models are possible. In the data-flow model [13], which we also consider, object performance bottlenecks can be reduced by migrating objects to the invoking transactional node and exploiting locality. Moreover, if an object is shared by a group of geographically-close clients that are far from the object's home, moving the object to the clients can reduce communication costs. Such a data flow model requires a *distributed cache-coherence protocol*, which locates an object's latest cached copy, and moves a copy to the requesting transaction, while guaranteeing one writable copy. Of course, CM is also needed. When an object is attempted to be migrated, it may be in use. Thus, a CM must mediate object access conflicts. Past distributed TM efforts present cache coherence protocols (e.g., Ballistic [13], LAC [23], Relay [22]) and often use a globally consistent CM (e.g., Greedy [9]).

We consider distributed transactional scheduling to enhance distributed TM performance. We present a novel distributed transactional scheduler called *Bi-interval* that optimizes the execution order of transactional operations to minimize conflicts. We focus on read-only and read-dominated workloads (i.e., those with only early-write operations), which are common transactional workloads [10]. *Bi-interval* categorizes concurrent requests for a shared object into read and write intervals to maximize the parallelism of reading transactions. This reduces conflicts between reading transactions, reducing transactional execution times. Further, it allows an object to be simultaneously sent to nodes of reading transactions, thereby reducing the total object traveling time.

We evaluate *Bi-interval* by its makespan competitive ratio—i.e., the ratio of *Bi-interval*'s makespan (the last completion time for a given set of transactions) to the makespan of an optimal transactional scheduler. We show that *Bi-interval* improves the makespan competitive ratio of the Relay cache coherence protocol with the Greedy CM from $O(n)$ [22] to $O(log(n))$, for n nodes. Also, *Bi-interval* yields an average-case makespan competitive ratio of $\Theta(log(n-k))$, for k reading transactions.

We implement *Bi-interval* in a distributed TM implementation constructed using the RSTM package [5]. Our experimental studies reveal that *Bi-interval* improves transactional throughput of Relay by as much as 188% and that of LAC protocols by as much as 200%. In the worst-case (i.e., without any reading transaction), *Bi-interval* improves throughput of Relay and LAC protocols (with the Greedy CM) by as much as 30%. Thus, the paper's contribution is the *Bi-interval* transactional scheduler. To the best of our knowledge, this is the first ever transactional scheduler for distributed TM.

The rest of the paper is organized as follows. We review past and related work in Section 2. We describe our system model and definitions in Section 3. Section 4 describes the *Bi-interval* scheduler, analyzes its performance, and gives a procedural description. We discuss *Bi-interval*'s implementation and report experimental evaluation in Section 5. The paper concludes in Section 6.

2 Related Work

Past works on distributed transactional memory include [4,13,17,15,23]. In [17], the authors present a page-level distributed concurrency control algorithm, which maintains several distributed versions of the same data item. In [4], the authors decompose a set of existing cache-coherent TM designs into a set of design choices, and select a combination of such choices to support TM for commodity clusters. Three distributed cache-coherence protocols are compared in [15] based on benchmarks for clusters.

In [13], Herlihy and Sun present a distributed cache-coherence protocol, called Ballistic, for metric-space networks, where the communication cost between nodes form a metric. Ballistic models the cache-coherence problem as a distributed queuing problem, due to the fundamental similarities between the two problems, and directly uses an existing distributed queuing protocol, the Arrow protocol [6], for managing transactional contention. Since distributed queuing protocols, including Arrow, do not consider contention between transactions, Ballistic suffers from a worst-case queue length of $O(n^2)$ for n transactions requesting the same object. Further, its hierarchical structure degrades its scalability—e.g., whenever a node joins or departs the network, the whole structure has to be rebuilt. These drawbacks are overcome in the Relay protocol [22], which reduces the worst-case queue length by considering transactional contention, and improves scalability by using a peer-to-peer structure.

Zhang and Ravindran present a class of location-aware distributed cache-coherence (or LAC) protocols in [23]. For LAC protocols, the node which is "closer" to the object (in terms of the communication cost) always locates the object earlier. When working with the Greedy CM, LAC protocols improve the makespan competitive ratio.

None of these efforts consider transactional scheduling. However, scheduling has been explored in a number of multiprocessor TM efforts [8,1,21,7,3]. In [8], Dragojević *et. al.* describe an approach that schedules transactions based on their predicted read/write access sets. They show that such a scheduler can be 2-competitive with an optimal scheduler, and design a prediction-based scheduler that dynamically serializes transactions based on the predicted access sets. In [1], Ansari *et. al.* discuss the Steal-On-Abort transaction scheduler, which queues an aborted transaction behind the non-aborted transaction, and thereby prevent the two transactions from conflicting again (which they likely would, if the aborted transaction is immediately restarted).

Yoo and Lee present the Adaptive Transaction Scheduler (ATS) [21] that adaptively controls the number of concurrent transactions based on the contention intensity: when the intensity is below a threshold, the transaction begins normally; otherwise, the transaction stalls and do not begin until dispatched by the scheduler. Dolev *et. al.* present the CAR-STM scheduling approach [7], which uses per-core transaction queues and serializes conflicting transactions by aborting one and queueing it on the other's queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, and thereby minimizes conflicts.

Attiya and Milani present the BIMODAL scheduler [3], targeting read-dominated and bimodal (i.e., those with only early-write and read-only) workloads. BIMODAL alternates between "writing epochs" and "reading epochs" during which writing and reading transactions are given priority, respectively, ensuring greater concurrency for reading transactions. BIMODAL is shown to significantly outperform its makespan competitive ratio in read-dominated workloads, and has an $O(s)$ competitive ratio.

Our work is inspired by the BIMODAL scheduler. The main idea of our work is also to build a read interval, which is an ordered set of reading transactions to simultaneously visit requesting nodes of those reading transactions. However, there is a fundamental trade-off between building a read interval and moving an object. If an object visits only read-requesting nodes, the object moving time may become larger. On the other hand, if an object visits in the order of the nearest node, we may not fully exploit the concurrency of reading transactions. Thus, we focus on how to build the read interval, exploiting this trade-off. Note that this tradeoff does not occur for BIMODAL.

3 Preliminaries

We consider Herlihy and Sun's data-flow TM model [13]. In this model, transactions are immobile, but objects move from node to node. A CM module is responsible for mediating between conflicting accesses to avoid deadlocks and livelocks. We use the Greedy CM which satisfies the work conserving [2] and pending commit [9] properties.

When a transaction attempts to access an object, the cache-coherence protocol locates the current cached copy of the object, moves it to the requesting node's cache, and invalidates the old copy (e.g., [22,23]). If no conflict occurs, the protocol is responsible for locating the object for the requesting nodes. Whenever a conflict occurs, the CM aborts the transaction with the lower priority. The aborted transaction is enqueued to prevent it from concurrently executing again. When the current transaction commits, the transactional scheduler should decide in what order the enqueued transactions should execute. We assume that the same scheduler is embedded in all nodes for consistent scheduling. We only consider read and write operations in transactions: a transaction that only reads objects is called a reading transaction; otherwise, it is a writing transaction.

Similar to [13], we consider a metric-space network where the communication costs between nodes form a metric. We assume a complete undirected graph $G = (V, E)$, where $\mid V \mid = n$. The cost of an edge $e(i, j)$ is measured by the communication delay of the shortest path between two nodes i and j. We use $d_G(i, j)$ to denote the cost of $e(i, j)$ in G. Thus, $d_G(i, j)$ forms the metric of G.

A node v has to execute a transaction T, which is a sequence of operations on the objects R_1, R_2,... R_s, where $s \geq 1$. Since each transaction is invoked on an individual node, we use v_{T_j} to denote the node that invokes the transaction T_j. We define $V_T = \{ v_{T_1}, v_{T_2}, \ldots v_{T_n} \}$ indicating the set of nodes requesting the same object. We use $T_i \prec T_j$ to represent that transaction T_j is issued a higher priority than T_i by the Greedy CM. We use τ_j to denote the duration of a transaction's execution on node j.

Definition 1. *A scheduler A is conservative if it aborts at least one transaction in every conflict.*

Definition 2 (Makespan). *Given a scheduler A, $makespan_i(A)$ is the time that A needs to complete all the transactions in $V_{T_n}^{R_i}$ which require accesses to an object R_i.*

We define two types of makespans: (1) *traveling makespan*($\mathrm{makespan}_{\mathrm{i}}^{\mathrm{d}}(\mathrm{A})$), which is the total communication delay to move an object; and (2) *execution makespan* ($\mathrm{makespan}_{\mathrm{A}}^{\tau}(\mathrm{A})$), which is the time duration of transactions' executions including all aborted transactions.

Definition 3 (Competitive Ratio). *The competitive ratio (CR) of a scheduler A for $V_{T_n}^{R_i}$ is $\frac{makespan_i(A)}{makespan_i(OPT)}$, where OPT is the optimal scheduler.*

Definition 4 (Object Moving Time). *In a given graph G, the object moving cost $\eta_G^A(u, V)$ is the total communication delay for visiting each node from node u holding an object to all nodes in V, under scheduler A.*

We now present bounds on transactional aborts.

Lemma 1. *Given n transactions, in the worst-case, the number of aborts of a CM in $V_{T_n}^{R_i}$ is n^2.*

Proof. Since the CM is assumed to be work-conserving, there exists at least one transaction in $V_{T_n}^{R_i}$ that will execute uninterruptedly until it commits. A transaction T can be aborted by another transaction in $V_{T_n}^{R_i}$. In the worst-case, $\lambda_{CM} = \sum_{m=1}^{n-1} m \leq n^2$.

Lemma 2. *Given n transactions, in the worst-case, the number of aborts of a conservative scheduler in $V_{T_n}^{R_i}$ is $n-1$.*

Proof. By Lemma 1, we know that a transaction T can be aborted as many times as the number of elements of $V_{T_n}^{R_i}$. In the worst-case, the number of aborts of T is $n-1$. Thus, a scheduler enqueues the requests that are aborted and an object moves along the requesting nodes. Hence, in the worst-case, $\lambda_{Scheduler} = \sum_{m=1}^{n-1} 1 = n-1$.

We now investigate the makespan for all requests for object R_i by an optimal off-line scheduler.

Lemma 3. *The makespans of the optimal off-line scheduler are bounded as:*

$$makespan_i^d(OPT) \geq \min d_G(v_T, V_{T_{n-1}}^{R_i}), makespan_i^\tau(OPT) \geq \sum_{m=1}^{n-1} \tau_m$$

Proof. Suppose that n concurrent requests are invoked for the same object R_i and n-1 transactions are aborted in the worst-case. The optimal moving makespan is the summation of the minimum paths for R_i to visit each requested nodes. R_i starts moving from v_T to each node that belongs to $V_{T_{n-1}}^{R_i}$ according to the shortest paths. Once R_i visits a node, the node holds it during τ_m for processing.

Lemma 4. *The makespans of scheduler A are bounded as:*

$$makespan_i^d(A) \leq \max \eta_G^A(v_T, V_{T_{n-1}}^{R_i}), makespan_i^\tau(A) \leq \sum_{m=1}^{n-1} \tau_m$$

Proof. If $n-1$ requests arrive before the completion of a transaction at v_T, conflicts occur at v_T, which currently is R_i's holding node. The scheduler builds a list of requested nodes to visit. Once a transaction is aborted, object moving and execution times are determined using the number of nodes that would be used in the worst case scenario.

4 The *Bi-interval* Scheduler

Bi-interval is similar to the BIMODAL scheduler [3] in that it categorizes requests into read and write intervals. When a request of transaction T_2 on node 2 arrives at node 1, there are two possible cases: 1) if the transaction T_1 on node 1 has committed, then the object will be moved to node 2; 2) if T_1 has not committed yet, then a conflict occurs. For the latter case, two sub-cases are possible. If the queue of aborted transactions is empty, the (Greedy) CM aborts the newer transaction [9]. If $T_1 \prec T_2$, T_1 is aborted. To handle a conflict between a reading and a writing transaction, a reading transaction is aborted to concurrently process it with other reading transactions in the future. On the other hand, if there are aborted transactions in the queue, T_1 and the aborted transactions will be scheduled by node 1. If T_1 has been previously scheduled, we use a pessimistic concurrency control strategy [20]: T_2 is aborted and waits until the aborted transactions are completed. Otherwise, we use the Greedy CM.

When a transaction commits and the aborted transactions wait in the queue, *Bi-interval* starts building the read and write intervals.

Write Interval: The scheduler finds the nearest requesting node. If the node has requested an object for a writing transaction, a write interval starts and the scheduler keeps searching for the next nearest node. When a requesting node for a reading transaction is found, the write interval ends and a read interval starts. The object is visited according to the chain of writing transactions in serial order.

Read Interval: When the scheduler finds the nearest node that has requested an object for a reading transaction, a read interval starts. The scheduler keeps searching for the next nearest node to build the read interval. If a requesting node for a writing transaction is found, the scheduler keeps checking the nearest node until a node requesting the object for a reading transaction appears again. If it appears, the read requesting node is joined to the read interval, meaning that the previously found requesting node(s) for a writing transaction is(are) scheduled behind the read interval. Instead of giving up the benefit of shorter object traveling times by visiting the nearest node, we achieve the alternative benefit of increased concurrency between reading transactions. Before the joining procedure to extend the read interval, the scheduler computes the trade-off between these benefits. If scheduling is completed, the object is simultaneously sent to all requesting nodes involved in the read interval. If no node for a reading transaction appears, another write interval is created.

There are two purposes for building a read interval through scheduling. First, the total execution time decreases due to the concurrent execution of reading transactions. Second, an object is simultaneously sent to some requesting nodes for reading transactions. Thus, the total traveling time in the network decreases. We now illustrate *Bi-interval*'s scheduling process with an example.

Figure 1 shows an example of a five-node network. Node 3, where the conflicts occurs, is responsible for scheduling four requests from nodes 1, 2, 4, and 5. If all requests involve write operations, node 3 schedules them as the following scheduled list: ④ $\rightarrow$ ⑤ $\rightarrow$ ① $\rightarrow$ ②. If only nodes 2 and 4 have reading transactions, node 3 yields the following scheduled list: ④ $\rightarrow$ ② $\rightarrow$ ① $\rightarrow$ ⑤. Node 3 simultaneously sends a copy of the object to nodes 4 and 2. Once the copy arrives, nodes 4 and 2 process it. After processing the object, node 4 sends a signal to node 2 letting it know about

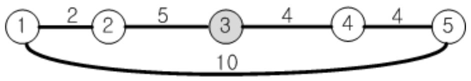


Fig. 1. A Five-Node Network Example for Bi-interval's Illustration

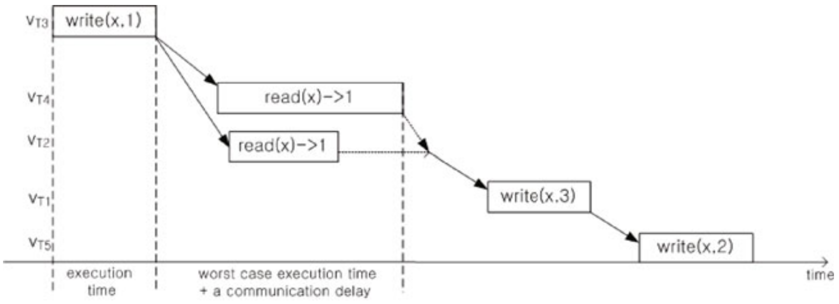


Fig. 2. An Example of Exploiting Parallelism in a Read Interval

the object availability. At this time, node 2 is processing or has finished processing the object. After processing it, node 2 sends the object to node 1. The makespan is improved only when $min(\tau_3, \tau_4) > \eta_G(3, 2)$. In the meantime, if other conflicts occur while the object is being processed along the scheduled list, aborted transactions (due to the conflicts) are scheduled behind the list to ensure *progress* based on pessimistic concurrency control. This means that those aborted transactions will be handled after processing the scheduled list.

Figure 2 shows an example of the parallelism in a read interval. Even though the object is simultaneously sent to nodes 4 and 2, it may not arrive at the same time. Due to different communication delays of the object and different execution times of each transaction, nodes 4 and 2 may complete their transactions at different times. According to the scheduled order, node 4 sends a signal to node 2 and node 2 immediately sends the object to node 1. Thus, the total makespan at nodes 4 and 2 includes only the worst-case execution time plus the object moving time. However, the communication delay between nodes 4 and 1 takes longer because node 2 is not the nearest node of node 4.

4.1 Algorithm Description

Bi-interval starts finding the set of consecutive nearest nodes using a nearest neighbor algorithm. $V_{T_m}^{R_i}$ denotes the set of nodes for reading transactions to obtain an object R_i, where $m \geq 1$. $V_{T_w}^{R_i}$ denotes the set of nodes for writing transactions to obtain R_i. Suppose that the scheduler found the ordered set of $V_{T_m}^{R_i}$ and $V_{T_w}^{R_i}$ as nearest nodes.

$$\eta_G(V_{T_m}^{R_i}, v_{m+1}) - \eta_G(V_{T_m}^{R_i}, V_{T_w}^{R_i}) < \tau_\omega \qquad (1)$$

When a request for a reading transaction from node v_{m+1} appears after $V_{T_w}^{R_i}$ is found, $V_{T_w}^{R_i}$ is switched to v_{m+1} in the condition of Equation 1 to extend the size of $V_{T_m}^{R_i}$. Equation 1 shows the condition for a reading request v_{m+1} to move to the previous read interval if the difference between the delay from $V_{T_m}^{R_i}$ to v_{m+1} and from $V_{T_m}^{R_i}$ to $V_{T_w}^{R_i}$ is less than or equal to τ_ω, where τ_ω is the worst-case execution time of a reading transaction, and $1 \leq m \leq k$.

$$\min_{1 \leq I_r \leq k} (\tau_\omega \cdot I_r + \sum_{m=1}^{n-k} \tau_m + \eta_G(v_T, \bar{V}_T^{R_i})) \tag{2}$$

Here, I_r is the number of read intervals, k is the number of reading transactions, $v_T \in V_T^{R_i}$, and $\bar{V}_T^{R_i} = \{ v_{T_1}, v_{T_2}, \ldots\ v_{T_{n+I_r-k}} \}$.

Equation 2 expresses the minimization of makespan for the execution and object traveling time, which is *Bi-interval*'s main objective:

Algorithm 1. Algorithm of *Bi-interval*

```
   Input: $V_{T_n}^{R_i} = \{ v_{T_1}, v_{T_2}, \cdots\ v_{T_n} \}$
   Output: $L_T^{R_i}$ /* Scheduled List */
1  $W_T^{R_i} \leftarrow \varnothing$; $L_T^{R_i} \leftarrow \varnothing$;
2  $p \leftarrow$NULL; $q \leftarrow$NULL
3  repeat
4      p=FindNearestNode($V_{T_n}^{R_i}$);
5      if p is a reading request then
6          if q is a writing request and
           DetermineTotal(p,$W_T^{R_i}$) is not OK
           then
7              A write interval is confirmed;
8              $L_T^{R_i} \leftarrow L_T^{R_i} \cup W_T^{R_i}$;
9              $W_T^{R_i} \leftarrow \varnothing$;
10         else
11             $V_{T_n}^{R_i} \leftarrow V_{T_n}^{R_i} \setminus \{ p \}$;
12             $L_T^{R_i} \leftarrow L_T^{R_i} \cup \{ p \}$;
13     else
14         $V_{T_n}^{R_i} \leftarrow V_{T_n}^{R_i} \setminus \{ p \}$;
15         $W_T^{R_i} \leftarrow W_T^{R_i} \cup \{ p \}$;
16     q = p;
17 until $V_{T_n}^{R_i}$ is $\varnothing$ ;
18 Boolean DetermineTotal(p,$W_T^{R_i}$)
19 if delay($L_T^{R_i}$, p)-delay($L_T^{R_i}$, $W_T^{R_i}$)< $\tau_{worst}$
   then
20     return OK;
21 return Not OK;
```

Algorithm 1 shows a detailed description of *Bi-interval* based on the nearest neighbor problem [14], which is known to be an NP-complete problem. Algorithm 1 is invoked when a transaction is committed and aborted requests are accumulated to be scheduled. In order to solve Equation 2, we consider a greedy approach, where at each stage of the algorithm, the link with the shortest delay is taken.

In order to visit k requesting nodes, the path from a starting node to visit k nodes in $V_{T_k}^{R_i}$ is selected. The set of $L_T^{R_i}$ is initiated, and the last remaining element is returned as a result. If the nearest node is found and it is a request for a read operation, the algorithm checks if a read interval has been started. If a read interval was previously started, the *DetermineTotal* function is called. If it returns *OK*, the read requesting node is joined to the previous read interval. Otherwise, a new read interval is created. Note that if a new read interval is started, it means that a write interval is confirmed because the *DetermineTotal* function is called only if a read requesting node is found as the nearest node right after a write requesting node is found in a queue.

The $FindNearestNode$ function finds the smallest delay from node v_T to a node in the set of $V_{T_k}^{R_i}$. Whenever the $FindNearestNode$ function returns a requesting node p for a reading transaction after finding a requesting node q for a writing transaction, Algorithm 1 has to check whether a write interval is created (i.e., $L_T^{R_i} \leftarrow L_T^{R_i} \cup W_T^{R_i}$) by comparing the delay corresponding to the total execution times and communication delay. The time and message complexity of Algorithm 1 is $O(n^2)$.

4.2 Competitive Ratio Analysis

We focus on the analysis of execution and traveling makespan competitive ratios.

Theorem 1. *Bi-interval's execution makespan competitive ratio is* $1+\frac{I_r}{n-k+1}$.

Proof. The optimal off-line algorithm concurrently executes all reading transactions. So, *Bi-interval*'s optimal execution makespan ($\text{makespan}_{\text{i}}^{\tau}(\text{OPT})$) is $\sum_{m=1}^{n-k+1} \tau_m$.

$$CR_{Biinterval}^{\tau} \leq \frac{\tau_\omega \cdot I_r + \sum_{m=1}^{n-k+1} \tau_m}{\sum_{m=1}^{n-k+1} \tau_m} \approx \frac{I_r + n - k + 1}{n-k+1}$$

Theorem 2. *Bi-interval's traveling makespan competitive ratio is* $\log(n + I_r - k - 1)$.

Proof. *Bi-interval* follows the nearest neighbor path to visit each node in the scheduling list. We define the *stretch* of a transactional scheduler as the maximum ratio of the moving time to the communication delay—i.e., $Stretch_\eta(v_T, V_{T_{n-1}}^{R_i}) = \max \frac{\eta_G(v_T, V_{T_{n-1}}^{R_i})}{d_G(v_T, V_{T_{n-1}}^{R_i})}$ $\leq \frac{1}{2}\log(n-1) + \frac{1}{2}$ from [18]. Hence, $CR_{Biinterval}^{d} \leq \log(n + I_r - k - 1)$.

Theorem 3. *The total worst-case competitive ratio* $CR_{Biinterval}^{Worst}$ *of Bi-interval for multiple objects is* $O(\log(n))$.

Proof. In the worst-case, $I_r = k$. This means that there are no consecutive read intervals. Thus, $\text{makespan}_{\text{OPT}}$ and $\text{makespan}_{\text{Biinterval}}$ satisfy the following, respectively:

$$makespan_{OPT} = \sum_{m=1}^{n-k+1} \tau_m + \min d_G(v_T, V_{T_{n-k+1}}^{R_i}) \tag{3}$$

$$makespan_{Biinterval} = \sum_{m=1}^{n-1} \tau_m + \log(n-1) \max d_G(v_T, V_{T_{n-1}}^{R_i}) \tag{4}$$

Hence, $CR_{Biinterval}^{Worst} \leq \log(n-1)$.

We now focus on the case $I_r < k$.

Theorem 4. *When* $I_r < k$, Bi-interval *improves the traveling makespan* $(\mathrm{makespan}_\mathrm{i}^\mathrm{d}(\mathrm{Biinterval}))$ *as much as* $O(|\log(1-(\frac{k-I_r}{n-1})|)$.

Proof.

$$\max \frac{\eta_G(v_T, V^{R_i}_{T_{n+I_r-k-1}})}{d_G(v_T, V^{R_i}_{T_{n-1}})} = \max \Big(\frac{\eta_G(v_T, V^{R_i}_{T_{n-1}})}{d_G(v_T, V^{R_i}_{T_{n-1}})} + \frac{\varepsilon}{d_G(v_T, V_{T_{n-1}})}\Big) \tag{5}$$
$$\leq \frac{1}{2}\log(n-k+I_r-1) + \frac{1}{2}$$

When $I_r < k$, a read interval has at least two reading transactions. We are interested in the difference between $\eta_G(v_T, V^{R_i}_{T_{n-1}})$ and $\eta_G(v_T, V^{R_i}_{T_{n+I_r-k-1}})$. Thus, we define ε as the difference between two η_G values.

$$\max \frac{\varepsilon}{d_G(v_T, V_{T_{n-1}})} \leq \frac{1}{2}\log(\frac{n-k+I_r-1}{n-1}) \tag{6}$$

In (6), due to $I_r < k$, $\frac{n-k+I_r-1}{n-1} < 1$. *Bi-interval* is invoked after conflicts occur, so $n \neq k$. Hence, ε is a negative value, improving the traveling makespan.

The average-case analysis (or, probabilistic analysis) is largely a way to avoid some of the pessimistic predictions of complexity theory. *Bi-interval* improves the competitive ratio when $I_r < k$. This improvement depends on the size of I_r—i.e., how many reading transactions are consecutively arranged. We are interested in the size of I_r when there are k reading transactions. We analyze the expected size of I_r using probabilistic analysis. We assume that k reading transactions are not consecutively arranged (i.e., $k \geq 2$) when n requests are arranged according to the nearest neighbor algorithm. We define a probability of actions taken for a given distance and execution time. The action indicates the satisfaction for the inequality of Equation 1. We focus on the analysis of the average-case competitive ratios of *Bi-interval*.

Theorem 5. *The expected number of read intervals* $E(I_r)$ *of Bi-interval is* $\log(k)$.

Proof. The distribution used in the proof of Theorem 5 is an independent uniform distribution. p denotes the probability for k reading transactions to be consecutively arranged.

$$E(I_r) = \int_{p=0}^{1} \sum_{I_r=1}^{k} \binom{k}{I_r} \cdot p^k(1-p)^{k-I_r}\,dp$$
$$= \sum_{I_r=1}^{k} \Big(\frac{k!}{I_r! \cdot (k-I_r)!}\int_{p=0}^{1} p^k(1-p)^{k-I_r}dp\Big)$$
$$\approx \sum_{I_r=1}^{k} \frac{k!}{I_r!} \cdot \frac{k!}{(2k-I_r+1)!} \approx \log(k) \tag{7}$$

We derive Equation 7 using the beta integral.

Theorem 6. *Bi-interval 's total average-case competitive ratio ($CR^{Average}_{Biinterval}$) is* $\Theta(\log(n-k))$.

Proof. We define $CR^{m}_{Biinterval}$ as the competitive ratio of node m. $CR^{Average}_{Biinterval}$ is defined as the sum of $CR^{m}_{Biinterval}$ of $n + E(I_r) - k + 1$ nodes.

$$\begin{aligned} CR^{Average}_{Biinterval} &\le \sum_{m=1}^{n+E(I_r)-k+1} CR^{m}_{Biinterval} \\ &\le \log(n + E(I_r) - k + 1) \approx \log(n-k) \end{aligned}$$

Since $E(I_r)$ is smaller than k, $CR^{Average}_{Biinterval} = \Theta(\log(n-k))$.

5 Implementation and Experimental Evaluation

We implemented *Bi-interval* in an experimental distributed TM implementation, which was built using the RSTM package [5]. Figure 3 shows the architecture of our distributed TM implementation. As a C++ TM library, RSTM provides a template that returns a transaction-enabled wrapper object. We implemented a distributed database repository that is connected to the template for handling transactional objects. The architecture consists of two parts: local TM and remote TM. Algorithm 2 gives detailed descriptions of these parts. When an object is needed, the local TM is invoked in the requesting node and the remote TM is invoked in the object holding node.

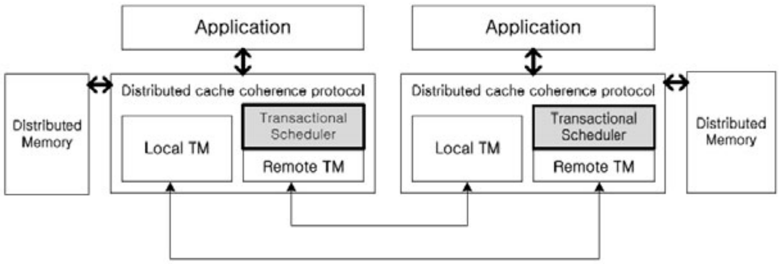
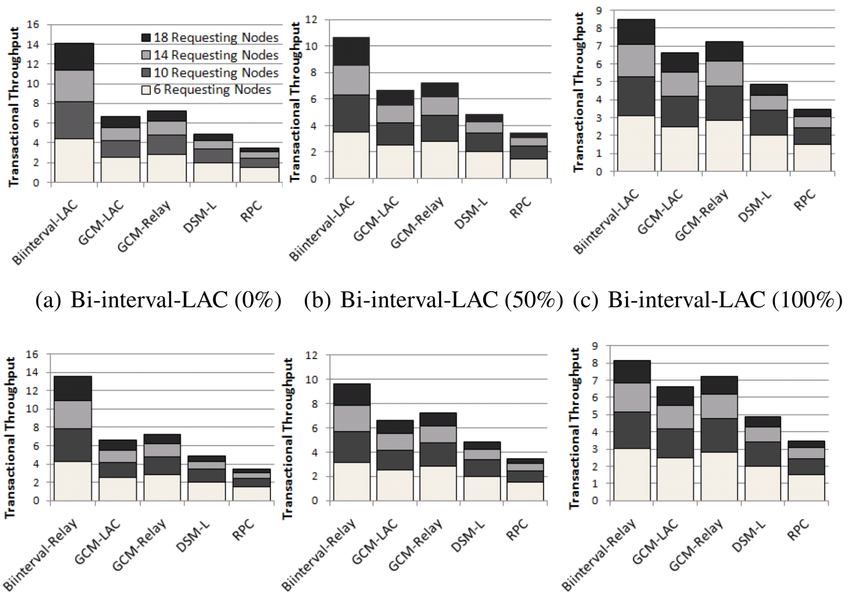


Fig. 3. Architecture of Experimental Distributed TM System

Experimental Evaluation. The purpose of our experimental evaluation is to measure the transactional throughput when the *Bi-interval* scheduler is used to augment a distributed cache-coherence-based TM. We implemented the LAC and Relay cache coherence protocols, which can be augmented with *Bi-interval*. We also included the classical RPC and distributed shared memory (DSM-L) models, both lock-based, in our experiments, as baselines. RPC and DSM-L are based on a client-server architecture in which a server has to hold shared objects, and clients request the object from the server. In contrast, in distributed TM, a shared object is distributed for each node. The LAC and Relay protocols include a procedure to find a node that holds or will hold an object. However, for

Algorithm 2. Algorithm of Local and Remote TM

```
1  Local TM
2  if a requested object is in the local memory and it is validated then
3      return the object;
4  else
5      send a request message;
6  if the response is not an abort message then
7      wait for the requested object during a timer t
8  if t is expired then
9      resend a request message;
10 Remote TM
11 if a requested object is validated then
12     invalidate and send the object
13 else
14     a conflict is detected.
15     invoke a CM.
16     enqueue the aborted request;
17 if a transaction is committed then
18     invoke the scheduler algorithm;
```



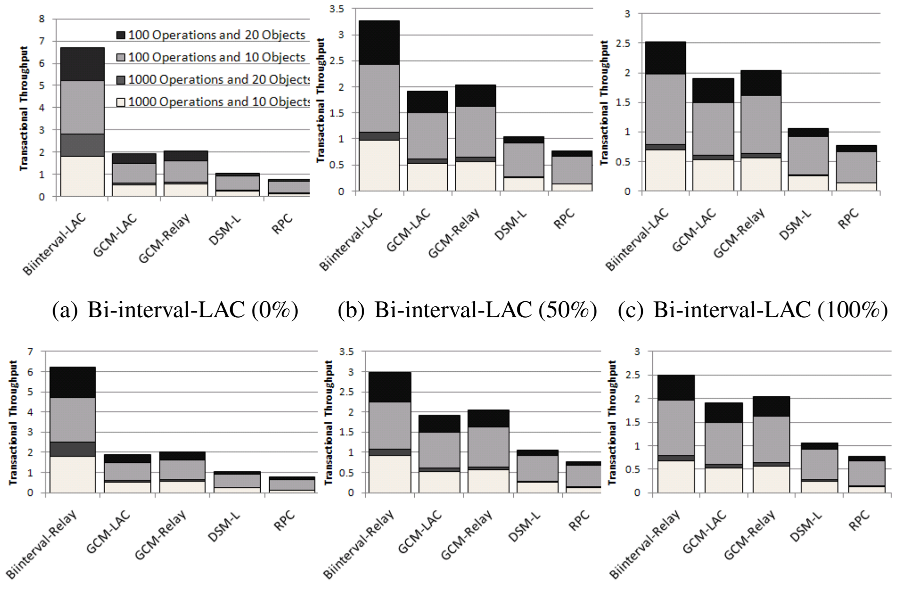
(a) Bi-interval-LAC (0%) (b) Bi-interval-LAC (50%) (c) Bi-interval-LAC (100%)

(d) Bi-interval-Relay (0%) (e) Bi-interval-Relay (50%) (f) Bi-interval-Relay (100%)

Fig. 4. Transactional Throughput Under Increasing Requesting Nodes/Single Object

fair comparison, we assume that all nodes know the location of the shared objects. Each node runs varying number of transactions, which write to and/or read from the objects. We used an 18-node testbed in the experimental study.

Two types of transactions were used in the experiments: insertion for a writing transaction and lookup for a reading transaction in a red-black tree. We measured the transactional throughput—i.e., the number of completed (committed) transactions per second under an increasing number of requesting nodes, for the different schemes. Since



(a) Bi-interval-LAC (0%) (b) Bi-interval-LAC (50%) (c) Bi-interval-LAC (100%)

(d) Bi-interval-Relay (0%) (e) Bi-interval-Relay (50%) (f) Bi-interval-Relay (100%)

Fig. 5. Transactional Throughput Under Multiple Objects and Increasing Operations

the throughput varies depending on the nodes' locations, we measured the total time to complete all transactions for each requesting node and computed the average number of committed transactions per second. We also varied the number of writing transactions. In the plots, 100% means that all requesting nodes are involved in writing transactions.

Figure 4 shows the transactional throughput under 6, 10, 14, and 18 nodes requesting an object. (In all the figures, we abbreviate the Greedy CM as GCM.) We observe that Bi-interval-LAC and GCM-LAC, and Bi-interval-Relay and GCM-Relay exhibit the same behavior under no conflicts. However, once a conflict occurs, an aborted transaction is not aborted again in Bi-interval-LAC (Relay). In GCM-LAC and GCM-Relay, the requesting nodes that have been aborted request an object again, so that increases the communication delay. Unless a request arrives at the object holding node right after a transaction is committed, the up-to-date copy of the object has to wait until the request arrives. However, the object holding node with *Bi-interval* immediately sends the copy to the requesting node right after it commits. The purpose of the Relay protocol is to reduce the number of aborts, so GCM-Relay has better performance than GCM-LAC. If no conflict occurs, Relay performs better than LAC. However, when a conflict occurs, Bi-interval-LAC performs better than Bi-interval-Relay due to the advantage of the parallelism of the reading transactions involved in the aborts. Bi-interval-LAC (0%) boosts the performance to the maximum possible transactional throughput.

It is interesting to observe the throughput difference between Bi-interval-LAC (100%) and Bi-interval-Relay (100%) shown in Figures 4(c) and 4(f), respectively. The cause of the throughput deterioration of Bi-interval-Relay (100%) is the high object moving delay. Even though transactions are aborted less in Relay, the copy of an object moves along a fixed spanning tree, which may not be the shortest path. Bi-interval-LAC, which has a relatively higher number of aborts, achieves the nearest node for aborted transactions.

In DSM-L and RPC, if an object is invalidated, they block new requests to protect the previous request. Specifically, Bi-interval-LAC and Relay (0%) simultaneously send all requesting nodes involved in aborted transactions to the object. Thus, they significantly outperform the other schemes.

We now turn our attention to throughput when a transaction uses multiple objects, and performs increasing number of object operations, causing longer transaction execution times. We measured throughput under 10 and 20 objects and 1000 and 100 operations per transaction. The objects are not related to each other, but the transaction has to use them together, so a transaction's execution time is longer. Figure 5 shows the results. GCM-LAC and GCM-Relay suffer from large number of aborts due to increasing number of objects and operations. They show greater throughput degradation from the number of aborts than that under shorter transactions. We observe a maximum improvement of 30% for Bi-interval under 100% updates.

6 Conclusions

Our work shows that the idea of grouping concurrent requests into read and write intervals to exploit concurrency of reading transactions — originally developed in BIMODAL for multiprocessor TM — can also be successfully exploited for distributed TM. Doing so poses a fundamental trade-off, however, one between object moving times and concurrency of reading transactions. *Bi-interval*'s design shows how this trade-off can be exploited towards optimizing transactional throughput.

Several directions for further work exist. First, we do not consider link or node failures. Second, *Bi-interval* does not support nested transactions. Additionally, we assume that transactional objects are unlinked. If objects are part of a linked structure (e.g., graph), scalable cache-coherence protocols and schedulers must be designed.

References

1. Ansari, M., et al.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)
2. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. In: PODC 2006, pp. 308–315 (2006)
3. Attiya, H., Milani, A.: Transactional scheduling for read-dominated workloads. In: OPODIS 2009, pp. 3–17. Springer, Heidelberg (2009)
4. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP 2008, pp. 247–258 (2008)
5. Dalessandro, L., Marathe, V.J., Spear, M.F., Scott, M.L.: Capabilities and limitations of library-based software transactional memory in c++. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing, Portland, OR (August 2007)

6. Demmer, M.J., Herlihy, M.: The arrow distributed directory protocol. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 119–133. Springer, Heidelberg (1998)
7. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In: PODC 2008, pp. 125–134 (2008)
8. Dragojević, A., Guerraoui, R., et al.: Preventing versus curing: avoiding conflicts in transactional memories. In: PODC 2009, pp. 7–16 (2009)
9. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC 2005, pp. 258–264 (2005)
10. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41(3), 315–324 (2007)
11. Hammond, L., Wong, V., et al.: Transactional memory coherence and consistency. SIGARCH Comput. Archit. News 32(2), 102 (2004)
12. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS 2003, p. 522 (2003)
13. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)
14. Kleinberg, J., Tardos, E.: Algorithm design (2005)
15. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: DiSTM: A software transactional memory framework for clusters. In: ICPP 2008, Washington, DC, USA, pp. 51–58. IEEE Computer Society, Los Alamitos (2008)
16. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid transactional memory. In: PPoPP 2006, pp. 209–220 (2006)
17. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: PPoPP 2006, pp. 198–208 (March 2006)
18. 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)
19. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC 2005, pp. 240–248 (2005)
20. Sonmez, N., Harris, T., Cristal, A., Unsal, O.S., Valero, M.: Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In: Parallel and Distributed Processing Symposium, International, pp. 1–10 (2009)
21. Yoo, R.M., Lee, H.-H.S.: Adaptive transaction scheduling for transactional memory systems. In: SPAA 2008, pp. 169–178 (2008)
22. Zhang, B., Ravindran, B.: Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In: OPODIS 2009, pp. 48–53 (2009)
23. Zhang, B., Ravindran, B.: Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In: SRDS 2009, pp. 268–277 (2009)

Recursion in Distributed Computing

Eli Gafni[1] and Sergio Rajsbaum[2,*]

[1] University of California, Los Angeles,
Computer Science Department,
Los Angeles, CA 90095
eli@ucla.edu

[2] Instituto de Matemáticas, Universidad Nacional Autónoma de México
Ciudad Universitaria, D.F. 04510
Mexico
rajsbaum@math.unam.mx

Abstract. The benefits of developing algorithms via recursion are well known. However, little use of recursion has been done in distributed algorithms, in spite of the fact that recursive structuring principles for distributed systems have been advocated since the beginning of the field. We present several distributed algorithms in a recursive form, which makes them easier to understand and analyze. Also, we expose several interesting issues arising in recursive distributed algorithms. Our goal is to promote the use and study of recursion in distributed algorithms.

1 Introduction

The benefits of designing and analyzing sequential algorithms using recursion are well known. Recursive algorithms are discussed in most textbooks, notably in Udi Manber's book [23]. However, little use of recursion has been done in distributed algorithms, in spite of the fact that recursive structuring principles for distributed systems have been advocated since the beginning of the field e.g. [13,24], and have been used before e.g. [12].

In this paper we describe simple and elegant recursive distributed algorithms for some important tasks, that illustrate the benefits of using recursion. We consider the following tasks: snapshots [1], immediate snapshots [6,27], renaming [4], and swap [2,29], and recursive distributed algorithms for each one. We work with a wait-free shared memory model where any number of processes can fail by crashing. We hope to convince the reader that thinking recursively is a methodology that facilitates the process of designing, analyzing and proving correctness of distributed algorithms.

We propose that studying recursion in a distributed setting is a worthwhile aim, although not without its drawbacks. There is no doubt that recursion should be covered starting with the first year introductory computer science courses, but it has been suggested recursive programming teaching be postponed until after iterative programs are well mastered, as recursion can lead to extremely

* Supported by UNAM-PAPIIT.

inefficient solutions e.g. [28]. In distributed algorithms, a well known example is the original Byzantine Agreement algorithm [21]. It exhibits the case that recursive distributed algorithms can be even more "dangerous" than in the centralized setting. The recursive algorithm looks simple and convincing, yet to really understand what it is doing, i.e. to unfold the recursion, took researchers a few years [5]. Even the seminal distributed spanning tree algorithm of [18], which is not recursive, can be viewed as a recursive algorithm that has been optimized [11]. Thus, one of the goals of this paper is to open the discussion of when and how recursion should be used in distributed algorithms.

There are several interesting issues that appear in recursive distributed algorithms, that do not appear in sequential recursion.

Naming recursive calls. Consider the classical binary search recursive algorithm. It searches an ordered array for a single element by cutting the array in half with each pass. It picks a midpoint near the center of the array, compares the data at that point with the data being searched. If the data is found, it terminates. Otherwise, there are two cases. (1) the data at the midpoint is greater than the data being searched for, and the algorithm is called recursively on the left part of the array, or (2) the data at the midpoint is less than the data being searched for, and the algorithm is called recursively on the right part of the array. Namely, only one recursive call is invoked, either (1) or (2), but not both. In contrast, in a distributed recursive algorithm, there are several processes running concurrently, and it is possible that both recursive calls are performed, by different processes. Thus, we need to name the recursive calls so that processes can identify them.

Concurrent branching. In sequential computing, recursive functions can be divided into linear and branched ones, depending on whether they make only one recursive call to itself (e.g. factorial), or more (e.g. fibonacci). In the first case, the recursion tree is a simple path, while in the second, it is a general tree. The distributed algorithms we present here are all of linear recursion, yet the recursion tree may not be a simple path, because, as explained above, different processes may invoke different recursive calls. See Figures 2, 4 and 9.

Iterated memory. As in sequential recursion, each distributed recursive call should be completely independent from the others. Even in the sequential binary search recursive algorithm, each recursive call (at least theoretically) operates on a new copy of the array, either the left side or the right side of the array. In a distributed algorithm, a process has local variables and shared variables. Both should be local to the recursive invocation. Thus, each recursive invocation has associated its own shared memory. There are no "side effects" in the sense that one recursive invocation cannot access the shared memory of another invocation. Thus, recursive distributed algorithms run on an *iterated model* of computation, where the shared memory is divided in sections. Processes run by accessing each section at most once, and in the same order. Iterated models have been exploited in the past e.g. [14,11,16,26,25,19].

Shared objects. In the simplest case, the shared memory that is accessed in each iteration is a single-writer/multi-reader shared array. Namely, a recursive distributed algorithm writes to the array, reads each of its elements, and after some local computation, either produces an output or invokes recursively the algorithm. Assuming at least one process decides, fewer processes participate in each recursive invocation (on a new shared array), until at the bottom of the recursion, only one process participates and decides. More generally, in each recursive invocation, processes communicate through a shared object that is more powerful than a single-writer/multi-reader shared array. Such an object is invoked at most once by each process. We specify the behavior of the object as a *task*, essentially, its input/output relation (see Section 2).

Inductive reasoning. As there are no side effects among recursive invocations, we can logically imagine all processes going in lockstep from a shared object to the next shared object, just varying the order in which they invoke each one. This ordering induces a structured set of executions that facilitates an inductive reasoning, and greatly simplifies the understanding of distributed algorithms. Also, the structure of the set of executions is recursive. This property has been useful to prove lower bounds and impossibility results as it facilitates a topological description of the executions e.g. [8,17,19]. We include Figure 5 to illustrate this point.

While recursion in sequential algorithms is well understood, in the distributed case we don't know much. For example, we do not know if in some cases side effects are unavoidable, and persistent shared objects are required to maintain information during recursive calls. We don't know if every recursive algorithm can be unfolded, and executed in an iterated model (different processes could invoke tasks in different order).

2 Model

We assume a shared memory distributed computing model, with processes $\Pi = \{p_1, \ldots, p_n\}$, in an asynchronous *wait-free* setting, where any number of processes can crash.

Processes communicate through shared objects that can be invoked at most once by each process. The input/output specification of an object is in terms of a *task,* defined by a set of possible inputs vectors, a set of possible output vectors, and an input/output relation Δ. A more formal definition of a task is given e.g. in [20]. Process p_i can invoke an instance of an object solving a task once with the instruction $\textsc{Task}_{tag}(x)$, and eventually gets back an output value, such that the vector of outputs satisfies the task's specification. The subindex *tag* identifies the particular instance of the object, and x is the input to the task.

The most basic task we consider is the *write/scan* task, that is a specification of a single-writer/multi-reader shared array. An instance *tag* of this object is accessed with $\text{WS}\textsc{can}_{tag}(x)$. Each instance *tag* has associated its own shared array $SM[1..n]$, where $SM[j]$ is initialized to $\perp$, for each j. When process p_i

invokes $\text{WSCAN}_{tag}(x)$, the value x is written to $SM[i]$, then p_i reads in arbitrary order $SM[j]$, for each j, and the vector of values read is what the invocation to $\text{WSCAN}_{tag}(x)$ returns.

Also, we are interested in designing distributed algorithms that solve a given task. The processes start with private input values, and must eventually decide on output values. For an execution where only a subset of processes participate an *input vector* I specifies in its i-th entry, $I[i]$, the input value of each participating process p_i, and $\perp$ in the entries of the other processes. Similarly, an *output vector* O contains a decision value $O[i]$ for a participating process p_i, and $\perp$ elsewhere. If an algorithm solves the task, then O is in $\Delta(I)$.

A famous task is *consensus*. Each process proposes a value and the correct processes have to decide the same value, which has to be a proposed value. This task is not wait-free solvable using read/write registers only [22]. We introduce other tasks as we discuss them.

3 Recursive Distributed Algorithms

We start with two basic forms of recursive distributed algorithms: linear branching in Section 3.1, and binary branching in Section 3.2. Here we only analyze the behavior of these algorithms, in later sections we show how they can be used to solve specific tasks. A multi-way branching version is postponed to Section 5.2, and a more general multi-way branching version is in Section 6.

3.1 Linear Recursion

Consider the algorithm, called IS, of Figure 1. In line 1 processes communicate through a single-writer/multi-reader shared array, invoked with the operation WSCAN (we omit its tag, because always the invocation is local to the recursive call). Process p_i writes its id i, and stores in *view* the set of ids read in the array (the entries different from $\perp$). In line 2 the process checks if *view* contains all n ids. The last process to write to the array sees n such values, i.e., $|view| = n$, it returns *view*, and terminates the algorithm. Namely, at least one process terminates the algorithm, but perhaps more than one (all process that see $|view| = n$), in each recursive call of the algorithm. In line 3 processes that have $|view| < n$ invoke the algorithm recursively (each time a different shared array is used). When $n = 1$, the single process invoking the algorithm returns a view that contains only itself.

```
Algorithm IS($n$);
 (1)  $view \leftarrow \text{WSCAN}(i)$;
 (2)  if $|view| = n$ then return $view$
 (3)  else return IS($n-1$)
```

Fig. 1. Linear recursion (code for p_i)

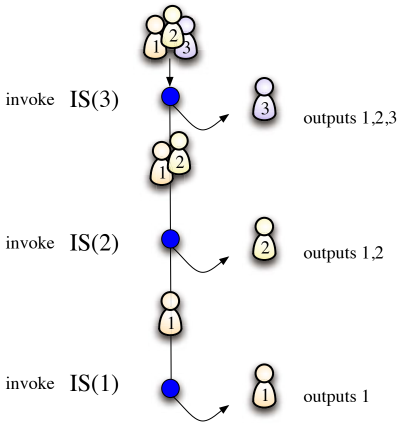


Fig. 2. Linear recursion, for 3 processes $1, 2, 3$

As we shall see in Section 4, this algorithm solves the immediate snapshot task. For now, we describe only its behavior, as linear recursion. Namely, the recursion tree is a simple path; one execution of the algorithm is illustrated in Figure 2 for 3 processes, $1, 2$ and 3.

In Figure 2 all three processes invoke IS(3), each one with its own id. In this example, only process 3 sees all three values after executing WScan, and exits the algorithm with view $1, 2, 3$. Processes $1, 2$ invoke IS(2), and a new read/write object instance through WScan. Namely, in an implementation of the write/scan task with a shared array, in the first recursive invocation IS(3) a shared array is used, in the second recursive invocation IS(2), a second shared array is used, and finally, when process 1 invokes IS(1) alone, it uses a third shared array, sees only itself, and terminates with view 1.

The total number of read and write steps by a process is $O(n^2)$. In more detail, a process p_i that returns a view with $|view| = k$ executes $\Theta(n(n-k+1))$ read/write steps. Process p_i returns *view* during the invocation of IS(k). Thus, it executed a total of $n-k+1$ task invocations, one for each recursive call, starting with IS(n). Each WScan invocation involves one write and n read steps.

3.2 Binary Branching

Let us now consider a small variation of algorithm IS, called algorithm BR in Figure 3. It uses $tag \in \{L, R, \emptyset\}$. The first time the algorithm is invoked by all n processes, with $tag = \emptyset$. Recursive invocations are invoked with smaller values of n each time, and with *tag* equal to L or to R. Until at the bottom of the recursion, the algorithm is invoked with $n = 1$ by only one process, which returns with a *view* that contains only its own id. In line 3, after seeing all n processes, process p_i checks if its own id is the largest it saw, in *view*. If so, it terminates the algorithm. If it is not the largest id, it invokes recursively an instance of BR identified by the $tag = R$, and size $n-1$ (at most $n-1$ processes invoke it). Line 5 is executed by processes that saw less than n ids in their views obtained in Line 2; they all invoke the same instance of BR, now identified by the $tag = L$, and size $n-1$ (at most $n-1$ invoke it).

```
Algorithm BR_{tag}(n);
(1)  view ← WScan(i);
(2)  if |view| = n then
(3)        if i = max view then return view ;
(4)        return BR_R(n − 1);
(5)  else return BR_L(n − 1)
```

Fig. 3. Branching recursion algorithm (code for p_i)

This time the recursive structure is a tree, not a path. An execution of the algorithm for 4 processes is illustrated in Figure 4. Each of the nodes in the tree has associated its own shared array. In the first recursive call, all processes write to the first shared array, and processes $3, 4$ see only themselves, hence invoke $\text{BR}_L(3)$, while processes $1, 2$ see all 4 processes, and as neither is the largest among them, they both invoke $\text{BR}_R(3)$. The rest of the figure is self-explanatory.

Here we are not interested in the task solved by Algorithm BR, only in that it has the same recursive structure as the renaming algorithm of Section 5, and hence the same complexity. Each process executes at most n recursive calls, and hence at most n invocations to a write/scan task. Thus, the total number of read and write steps by a process is $O(n^2)$.

4 Snaphots

In this section we describe an immediate snapshot [6,27] recursive algorithm. As the specification of the snapshot [1] task is a weakening of the immediate snapshot task, the algorithm solves the snapshot task as well.

Immediate snapshot task. An immediate snapshot task *IS* abstracts a shared array $SM[1..n]$ with one entry per process. The array is initialized to $[\bot, \ldots, \bot]$, where $\bot$ is a default value that cannot be written by a process. Intuitively, when a process p_i invokes the object, it is as if it instantaneously executes a write operation followed by a snapshot of the whole shared array. If several processes invoke the task simultaneously, then their corresponding write operations are executed concurrently, followed by a concurrent execution of their snapshot operations.

More precisely, in an *immediate snapshot* task, for each p_i, the result of its invocation satisfies the three following properties, where we assume i is the value written by p_i (without loss of generality) and sm_i is the set of values or *view* it gets back from the task. If $SM[k] = \bot$, the value k is not added to sm_i. We define $sm_i = \emptyset$, if the process p_i never invokes the task. These properties are:

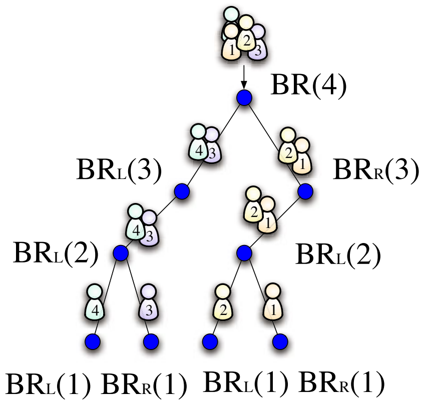


Fig. 4. Branching recursion, for 4 processes

- Self-inclusion. $\forall i : i \in sm_i$.
- Containment. $\forall i, j : sm_i \subseteq sm_j \;\vee\; sm_j \subseteq sm_i$.
- Immediacy. $\forall i, j : i \in sm_j \;\Rightarrow sm_i \subseteq sm_j$.

The immediacy property can be rewritten as $\forall i, j : \big(i \in sm_j \wedge j \in sm_i\big) \;\Rightarrow\; sm_i = sm_j$. Thus, concurrent invocations of the task obtain the same view. A *snapshot* task is required to satisfy only the first two properties.

The set of all possible views obtained by the processes after invoking an object implementing an immediate snapshot task can be represented by a *complex*, consisting of sets called *simplexes*, with the property that if a simplex is included in the complex, so are all its sub-simplexes. The set of all possible views, for 3 processes, is represented in Figure 5. Each vertex is labeled with a pair i, sm_i. The simplexes are the triangles, the edges, and the vertexes. The corners of each simplex are labeled with compatible views, satisfying the three previous properties. In the case of 4 processes, the complex would be 3-dimensional, including sets up to size 4, and so on for any number of processes. For more details about complexes and their use in distributed computing, see e.g. [20].

Recursive algorithm. A wait-free algorithm solving the immediate snapshot task was described in [6]. We include it in Appendix A for comparison. We encourage the reader to try to come up with a correctness proof, before reading the recursive version, where the proof will be trivial. Actually, algorithm IS of Figure 1 solves the immediate snapshot task.

Theorem 1. *Algorithm* IS *solves the immediate snapshot task for* n *processes in* $O(n^2)$ *steps. Moreover, a process obtains a snapshot of size* k *from the algorithm in* $\Theta(n(n-k+1))$ *steps.*

Proof. The complexity was analyzed in the previous section. Here we prove that IS solves the immediate snapshot task. Let S be the set of processes that terminate the algorithm in line 2, namely, with $|view| = n$. Each $p_i \in S$, terminates

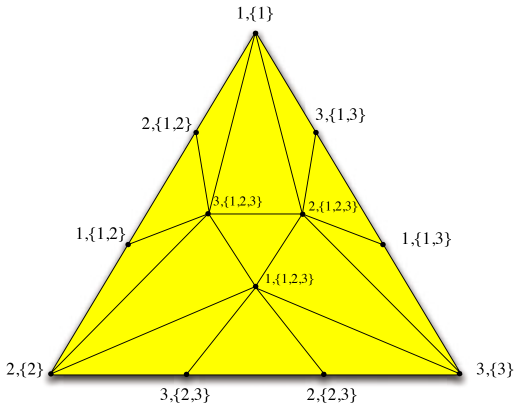


Fig. 5. All immediate snapshot subdivision views, for 3 processes

the algorithm with a view sm_i that contains all processes. Thus, for each such p_i, the **self-inclusion** property holds. Also, for any two p_i, p_j in S, we have $sm_i = sm_j$, and the properties of **containment** and **immediacy** hold.

By induction hypothesis, the three properties also hold for the other processes, that call recursively IS_{n-1}. It remains to show that the two last properties of the immediate snapshot task hold for a $p_i \in S$, and a $p_j \notin S$. First, property **containment** holds: clearly $sm_j \subset sm_i$, because p_i does not participate in the recursive call. Finally, property **immediacy** holds: j is in sm_i (p_i sees every process participating), and we have already seen that $sm_j \subseteq sm_i$. And it is impossible that $i \in sm_j$, because p_i does not participate in the recursive call.

5 Renaming

In the *renaming task* [4] each of n processes than can only compare their ids must choose one of $2n - 1$ new distinct names, called *slots*. It was proved in [20] that renaming is impossible with less than $2n - 1$ slots, except for some special values of n [10]. The algorithm of [4] solves the problem with $2n - 1$ slots, but is of exponential complexity [15]. Later on, [7] presented a recursive renaming algorithm based on immediate snapshots, of $O(n^3)$ complexity. We restate this algorithm in Section 5.2, and show that its complexity is actually $O(n^2)$. Also, we present a new renaming algorithm in Section 5.1 that is not based on immediate snapshots, also of $O(n^2)$ complexity; it is very simple, but requires knowledge of n.

To facilitate the description of a recursive algorithm, the slots are, given an integer $First$ and $Direction \in \{-1, +1\}$, the integers in the range $First + [0..2n - 2]$ if $Direction = +1$, or in the range $First + [-(2n - 2)..0]$ if $Direction = -1$. Combining the two, we get slots $First + Direction * [0..2n - 2]$. Thus, the number of slots is $2n - 1$ as required; i.e., $|Last - First| + 1 = 2n - 1$, defining $Last = First + Direction * (2n - 2)$.

5.1 Binary Branching Renaming Algorithm

The algorithm has exactly the same structure as the binary branching algorithm of Figure 3, using WScan. It partitions the set of processes into two subsets, and then solves renaming recursively on each subset. The algorithm Renaming$(n, \mathit{First}, \mathit{Direction})$ is presented in Figure 6. It uses tags of the form $\{\leftarrow, \rightarrow\}$, to represent the intuition of renaming "left to right" and "right to left" as indicated by the value of *Direction*. Given *First* and *Direction*, the algorithm is invoked by k processes, where $k \leq n$, and each process decides on a slot in the range $\mathit{First} + \mathit{Direction} * [0..2k-2]$. The algorithm ends in line 4, with a slot selected.

In the algorithm, the processes first write and scan a shared array, in line 1. According to the size of the view they get back, they are partitioned in 2 sets– the processes that get back a view of size n and the processes that get back a view of size less than n. If k processes, $k < n$, invoke it then, of course, nobody can get a view of size n. In this case they all go to line 6 and solve the problem recursively executing Renaming$_{\rightarrow}(n-1, \mathit{First}, \mathit{Direction})$. Thus, such recursive calls will be repeated until $k = n$. The variant of the algorithm described below evades these repeated calls, using immediate snapshots instead of write/scans.

Consider the case of $k = n$ invoking the algorithm. In this case, some processes will get a view of size n in line 1. If one of these, say p_i, sees that it has the largest id i in its view S_i (line 4), terminates the algorithm with slot *Last*. The other processes, Y, that get a view of size n, will proceed to solve the problem recursively, in line 5, renaming from slot $\mathit{Last} - 1$ down (reserving slot *Last* in case it was taken by p_i), by calling Renaming$_{\leftarrow}(n-1, \mathit{Last}-1, -\mathit{Direction})$. The processes X, that get a view of size less than n, solve the problem recursively in line 6, going up from position *First*, by calling Renaming$_{\rightarrow}(n-1, \mathit{First}, \mathit{Direction})$. Thus, we use the arrow in the superscript to distinguish the two distinct recursive invocations (to the same code). The correctness of the algorithm, in Theorem 1, is a simple counting argument, that consists of the observation that the two ranges, going down and up, do not overlap.

Theorem 2. *Algorithm* Renaming *solves the renaming task for n processes, in $O(n^2)$ steps.*

Proof. Clearly, the algorithm terminates, as it is called with smaller values of n in each recursive call, until $n = 1$, when it necessarily terminates. A process executes at most n recursive calls, and in each one it executes a write/scan. Each write/scan involves $O(n)$ read and write steps, for a total complexity of $O(n^2)$. If $n = 1$, then $|S_i| = 1$ in line (1) so the algorithm terminates with slot $\mathit{Last} = \mathit{First}$ in line (4). At the basis of the recursion, $n = 1$, the algorithm terminates correctly, renaming into 1 slot, as $\mathit{Last} = \mathit{First}$ when $n = 1$.

Assume the algorithm is correct for k less than n. The induction hypothesis is that when k' processes, $k' \leq k$, invoke Renaming$(k, \mathit{First}, \mathit{Direction})$, then they get new names in the range $\mathit{First} + \mathit{Direction} * [0..2k' - 2]$.

Now, assume the algorithm is invoked as Renaming$(n, \mathit{First}, \mathit{Direction})$, with $n > 1$, by $k \leq n$ processes. Let X be the set of processes that get a view

```
Algorithm $\textsc{Renaming}(n, First, Direction)$;
(1) $S_i \leftarrow \text{WS}\textsc{can}(i)$;
(2) $Last \leftarrow First + Direction * (2n-2)$;
(3) if $|S_i| = n$ then
(4)       if $i = \max S_i$ then return $Last$;
(5)       return $\textsc{Renaming}_{\leftarrow}(n-1, Last-1, -Direction)$;
(6) else return $\textsc{Renaming}_{\rightarrow}(n-1, First, Direction)$
```

Fig. 6. Write/scan binary branching renaming (code for p_i)

smaller than n in line (1), $|X| = x$. Notice that $0 \le x \le n-1$. If $k < n$ then all get a view smaller than n, and they all return $\textsc{Renaming}(n-1, First, Direction)$ in line (6), terminating correctly. So for the rest of the proof, assume $k = n$.

Let Y be the set of processes that get a view of size n in line (1), $|Y| = y$, excluding the process of largest *id*. Thus, $0 \le y \le n-1$.

The processes in X return from $\textsc{Renaming}(n-1, First, Direction)$ in line (6), with slots in the range $First + [0..2x-2]$. The processes in Y return from $\textsc{Renaming}(n-1, Last-1, (-1) * Direction)$ in line (5), with slots in the range $[Last - 1 - (2y-2)..Last - 1]$. To complete the proof we need to show that

$$First + 2x - 2 < Last - 1 - (2y - 2).$$

Recall $Last = First + Direction * (2n-2)$. Thus, we need to show that

$$2x - 2 < 2n - 2 - 1 - (2y - 2).$$

As $x + y \le n$, the previous inequality becomes $2(n) - 2 < 2n - 2 - 1 + 2$, and we are done.

5.2 A Multi-way Branching Renaming Algorithm

In the previous $\textsc{Renaming}_n$ algorithm the set of processes, X, that get back a view of size less than n, will waste recursive calls, calling the algorithm again and again (with the same values for *First* and *Direction*, but smaller values of n) until n goes down to n', with $n' = |X|$. In this recursive call, $\textsc{Renaming}_{n'}$, the processes that get back a view of size n', will do something interesting; that is, one might get a slot, and the others will invoke recursively $\textsc{Renaming}$, but in opposite direction. Therefore, using immediate snapshots, we can rewrite the $\textsc{Renaming}$ algorithm in the form of Figure 7. This is the renaming algorithm presented in [7].

Notice that in is$\textsc{Renaming}_{tag}$ the subindex *tag* is a sequence of integers: in line 4, the new tag $tag \cdot |S_i|$ is the old *tag*, appending at the end S_i. These tags are a way of naming the recursive calls, so that a set of processes that should participate in the same execution of is$\textsc{Renaming}$, can identify using the same

```
Algorithm isRenaming_{tag}(First, Direction);
(1)  $S_i \leftarrow$ Immediate_Snapshot($i$);
(2)  $Last \leftarrow First + Direction * (2|S_i| - 2)$;
(3)  if $i = \max S_i$ then return $Last$;
(4)  return isRenaming$_{tag \cdot |S_i|}(Last - 1, -Direction)$
```

Fig. 7. Immediate snapshot multi-way branching renaming (code for p_i)

tag. The first time isRenaming is called, *tag* should take some default value, say the empty set.

Although the analysis in [7] bounded the number of steps by $O(n^3)$, we observe a better bound can be given:

Theorem 3. *Algorithm is*Renaming *solves the renaming task for n processes, in $O(n^2)$ steps.*

Proof. As explained above, the algorithm is equivalent to Renaming, and hence correctness follows from Theorem 2. Now, to show that the complexity is $O(n^2)$ steps we do an amortized analysis, based on Theorem 1: a process obtains a snapshot of size s from the Immediate_Snapshot$_n$ algorithm in $\Theta(n(n-s+1))$ steps.

Assume a process runs isRenaming until it obtains a slot, invoking the algorithm recursively k times. In the i-th call, assume it gets a snapshot of size s_i (in line 1). For example, if $s_1 = n$, then $k = 1$, and using the result of Theorem 1, the number of steps executed is n. In general, the number of steps executed by a process is n times

$$[n - s_1] + [(n - s_1) - (n - s_2)] + [(n - s_2) - (n - s_3)] + \cdots + [(n - s_{k-1}) - (n - s_k)]$$

which gives a total of $O(n(n - s_k))$.

6 SWAP

Here we consider two tasks, Tournament$_\pi$ and Swap$_\pi$. A process can invoke these tasks with its input id, where π is an id of a processes that does not invoke the task, or 0, a special virtual id. Each process gets back another process id, or π. A process never gets back its own id. Exactly one process gets back π. We think of this process as the "winner" of the invocation. The *induced digraph* consists of all arcs $i \rightarrow j$, such that process i received process j as output from the task; it is guaranteed that the digraph is acyclic. We say that j is the *parent* of i. As every vertex has exactly one outgoing arc, except for the root, π, which has none, there is exactly one directed path from each vertex to the root. The Swap task always returns a directed path, while the Tournament can return any acyclic digraph.

Afek et al [2,29] noticed that these two tasks cannot be solved using read/write registers only, but can be solved if 2-consensus tasks are also available, namely, tasks that can be used to solve consensus for 2, but not for 3 processes.[1] They presented a wait-free algorithm that solves SWAP using read/write registers and TOURNAMENT tasks. The following is a recursive version of this algorithm, of the same complexity.

The SWAP algorithm is in Figure 8. Process p_i invokes the algorithm with $\text{SWAP}_{tag}(i)$, where $tag = 0$. In line 1 process i invokes $\text{TOURNAMENT}_{tag}(i)$ and in case it wins, i.e., gets back tag, it returns tag. By the specification of the tournament task, one, and only one process wins. All others invoke recursively a SWAP task: all processes with the same parent π invoke the same $\text{SWAP}_\pi(i)$ task.

```
Algorithm SWAP_{tag}(i);
 (1)  $\pi \leftarrow \text{TOURNAMENT}_{tag}(i)$;
 (2)  if $tag = \pi$ then return $\pi$;
 (3)  else return $\text{SWAP}_\pi(i)$
```

Fig. 8. The SWAP algorithm (code for p_i)

Assuming initially for each process p_i, $\pi_i = 0$, we get an invariant: the induced digraph (arcs from p_i to π_i) is a directed tree rooted in 0. Initially, all processes point to 0 directly. Each time TOURNAMENT_{tag} is executed, it is executed by a set of processes pointing to tag. The result of the execution is that exactly one process p keeps on pointing to tag, while the others induce a directed graph rooted at p.

An execution for 5 processes appears in Figure 9. Notice that although it is a tree, it has a unique topological order, as opposed to the tree of Figure 4. The result of executing TOURNAMENT_0 is that 1 wins, $2, 3, 4$ point to 1, while 5 point to 3. The result of executing TOURNAMENT_1 is that 2 wins, while $3, 4$ point to 2. The result of executing TOURNAMENT_2 is that 3 wins, while 4 point to 3. The result of executing TOURNAMENT_3 is that 4 wins, while 5 point to 4. Finally, 5 executes TOURNAMENT_4 by itself and wins.

For the following theorem, we count as a step either a read or write operation, or a 2-consensus operation, and assume a TOURNAMENT task can be implemented using $O(n)$ steps [2,29].

Theorem 4. *Algorithm* SWAP *solves the swap task for n processes, in $O(n^2)$ steps.*

Proof. The algorithm terminates, as in each invocation one process returns, the winner of the TOURNAMENT. Also, the basis is easy, as when only one process invokes the TOURNAMENT, it is necessarily the winner. Assume inductively that

[1] The task of Afek et al is not exactly the same as ours; for instance, they require a linearizability property, stating that the winner is first.

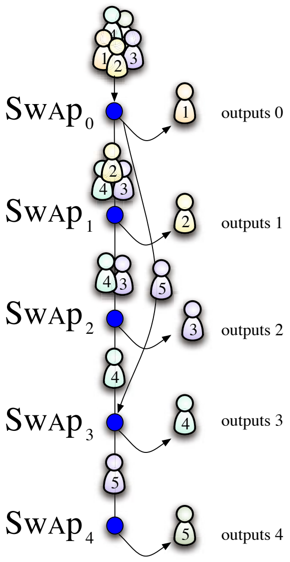


Fig. 9. Branching deterministic recursion, for 5 processes

the algorithm solves swap for less than n processes. Consider an invocation of SWAP, where the winner in line 1 is some process p. Consider the processes W that get back p in this line. Every other process will get back a descendant of these processes. Thus, the only processes that invoke SWAP$_{tag}$ with $tag = p$ are the processes in W. Moreover, it is impossible that two processes return the same tag, say $tag = x$, because a process that returns x does so during the execution of SWAP$_x$, after winning the TOURNAMENT invocation, and only one process can win this invocation.

Acknowledgments. We thank Michel Raynal and the anonymous referees for their comments on an earlier version of this paper.

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., Weisberger, E., Weisman, H.: A Completeness Theorem for a Class of Synchronization Objects (Extended Abstract). In: Proc. 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), Ithaca, New York, USA, August 15-18, pp. 159–170 (1993)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message-Passing Systems. J. ACM 42(1), 124–142 (1995)
4. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an Asynchronous Environment. J. ACM 37(3), 524–548 (1990)

5. Bar-Noy, A., Dolev, D., Dwork, C., Strong, H.R.: Shifting Gears. Changing Algorithms on the Fly to Expedite Byzantine Agreement. Inf. Comput. 97(2), 205–233 (1992)
6. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on the Theory of Computing (STOC), pp. 91–100. ACM Press, New York (1993)
7. Borowsky, E., Gafni, E.: Immediate Atomic Snapshots and Fast Renaming (Extended Abstract). In: 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), Ithaca, New York, USA, August 15-18, pp. 41–51 (1993)
8. Borowsky, E., Gafni, E.: A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). In: Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC 1997), Santa Barbara, California, USA, August 21-24, pp. 189–198 (1997)
9. Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: The BG Distributed Simulation Algorithm. Distributed Computing 14(3), 127–146 (2001)
10. Cañeda, A., Rajsbaum, S.: New combinatorial topology upper and lower bounds for renaming. In: Proceedings of the 27-th Annual ACM Symposium on Principles of Distributed Computing (PODC), Toronto, Canada, August 18-21, pp. 295–304 (2008)
11. Chou, C.-T., Gafni, E.: Understanding and Verifying Distributed Algorithms Using Stratified Decomposition. In: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC), Toronto, Ontario, Canada, August 15-17, pp. 44–65 (1988)
12. Coan, B.A., Welch, J.L.: Modular Construction of an Efficient 1-Bit Byzantine Agreement Protocol. Mathematical Systems Theory 26(1), 131–154 (1993)
13. Dobson, J., Randell, B.: Building Reliable Secure Computing Systems out of Unreliable Insecure Components. In: Proc. IEEE Conference on Security and Privacy, Oakland, USA, pp. 187–193 (1986)
14. Elrad, T., Francez, N.: Decomposition of Distributed Programs into Communication-Closed Layers. Sci. Comput. Program 2(3), 155–173 (1982)
15. Fouren, A.: Exponential examples for two renaming algorithms (August 1999), http://www.cs.technion.ac.il/~hagit/publications/expo.ps.gz
16. Gafni, E.: Round-by-Round Fault Detectors, Unifying Synchrony and Asynchrony (Extendeda Abstract). In: Proc. 17th Annual ACM Symposium on Principles of Distributed Computing (PODC), Puerto Vallarta, Mexico, June 28-July 2, pp. 143–152 (1998)
17. 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)
18. Gallager, R.G., Humblet, P.A., Spira, P.M.: A Distributed Algorithm for Minimum-Weight Spanning Trees. ACM Trans. Program. Lang. Syst. 5(1), 66–77 (1983)
19. Herlihy, M., Rajsbaum, S.: The Topology of Shared-Memory Adversaries. In: Annual ACM Symposium on Principles of Distributed Computing, Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Zurich, Switzerland, pp. 105–113 (2010), ISBN: 978-1-60558-888-9
20. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
21. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)

22. Loui, M.C., Abu-Amara, H.H.: Memory requirements for agreement among unreliable asynchronous processes. In: Preparata, F.P. (ed.) Advances in Computing Research, vol. 4, pp. 163–183. JAI Press, Greenwich (1987)
23. Manber, U.: Introduction to Algorithms: A Creative Approach. Addison-Wesley, Reading (1989)
24. Randell: Brian Recursively structured distributed computing systems. In: Proc. IEEE Symposium on Reliability in Distributed Software and Database Systems, pp. 3–11 (1983)
25. Rajsbaum, S., Raynal, M., Travers, C.: An impossibility about failure detectors in the iterated immediate snapshot model. Inf. Process. Lett. 108(3), 160–164 (2008)
26. Rajsbaum, S., Raynal, M., Travers, C.: The Iterated Restricted Immediate Snapshot Model. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 487–497. Springer, Heidelberg (2008)
27. 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)
28. Stojmenovic, I.: Recursive algorithms in computer science courses: Fibonacci numbers and binomial coefficients. IEEE Trans. on Education 43(3), 273–276 (2000)
29. Weisman, H.: Implementing shared memory overwriting objects. Master's thesis, Tel Aviv University (May 1994)

A Non-recursive Immediate Snapshots Algorithm

A wait-free algorithm solving the immediate snapshot task was described in [6]. We present it here for comparison with the recursive version presented in Section 4. The algorithm is in Figure 10.

```
Algorithm IMMEDIATE_SNAPSHOT($i$);
    repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
           for $j \in \{1,\dots,n\}$ do $level_i[j] \leftarrow LEVEL[j]$ end for;
           $view_i \leftarrow \{j : level_i[j] \le LEVEL[i]\}$;
    until $(|view_i| \ge LEVEL[i])$ end repeat;
    return($\{j$ such that $j \in view_i\}$)
```

Fig. 10. Non-recursive one-shot immediate snapshot algorithm (code for p_i)

It is argued in [6] that this algorithm solves the immediate snapshot task, with $O(n^3)$ complexity.

On Adaptive Renaming under Eventually Limited Contention

Damien Imbs and Michel Raynal

IRISA, Université de Rennes 1, 35042 Rennes, France
{damien.imbs,raynal}@irisa.fr

Abstract. The adaptive M-renaming problem consists in designing an algorithm that allows a set of $p \leq n$ participating asynchronous processes (where n is the total number of processes) not known in advance to acquire pair-wise different new names in a name space whose size M depends on p (and not on n). Adaptive $(2p-1)$-renaming algorithms for read/write shared memory systems have been designed. These algorithms, which are optimal with respect to the value of M, consider the wait-freedom progress condition, which means that any correct participant has to acquire a new name whatever the behavior of the other processes (that can be very slow or even crashed).

This paper addresses the design of an adaptive M-renaming algorithm when considering the k-obstruction-freedom progress condition. This condition, that is weaker than wait-freedom, requires that every correct participating process acquires a new name in all runs where during "long enough periods" at most k processes execute steps (p-obstruction-freedom and wait-freedom are actually equivalent). The paper presents an optimal adaptive $(p+k-1)$-renaming algorithm and, consequently, contributes to a better understanding of synchronization and concurrency by showing that weakening the liveness condition from wait-freedom to k-obstruction-freedom allows the new name space to be reduced from $2p-1$ to $\min(2p-1, p+k-1)$. Last but not least, the proposed algorithm is particularly simple, a first class property. This establishes an interesting tradeoff linking progress conditions with the size of the new name space.

Keywords: Asynchronous system, Fault-Tolerance, Liveness, Obstruction-freedom, Process crash, Progress condition, Shared memory system, Wait-freedom.

1 Introduction

1.1 Context of the Work

Renaming problem. Renaming is among the basic problems that lie at the core of computability in asynchronous distributed systems prone to process crashes [2]. It consists in the following. Each of the n processes that define the system has an initial name taken from a very large domain $[1..N]$ ($n << N$). Initially a process knows only its name, the value n and the fact that no two processes have the same initial name. The processes have to cooperate to choose new names from a name space $[1..M]$ such that $M << N$ and no two processes have the same new name. Given an integer M, the problem is called *M-renaming*.

Wait-freedom. This problem, first defined in the context of message-passing systems [2], has received a lot of attention in the context of asynchronous shared memory made up of atomic read/write registers. Numerous wait-free renaming algorithms have been designed (e.g., [5,7]). *Wait-free* means here that a process that does not crash has to obtain a new name in a finite number of its own computation steps whatever the behavior of the other processes (that can be very slow of even crashed) [14]. This means that, t denoting the maximum number of processes that may crash, wait-freedom implies $t = n - 1$.

A lower bound. An important theoretical result associated with the renaming problem in asynchronous read/write systems is the following [16]. Except for some specific values of n, $M = 2n-1$ is the lower bound on the size of the new name space. (For the specific values we have $M = 2n - 2$. These specific values, characterized in [8], involve sets of relatively prime integers[1].)

Adaptive renaming. A renaming algorithm is *adaptive* if the size of the new name space depends only on the number p of processes that ask for a new name (and not on the total number n of processes). Several adaptive algorithms have been designed such that the size of the new name space is $M = 2p - 1$ (e.g., [7]). This means that if "today" p' processes acquire new names, their new names belong to the interval $[1..2p' - 1]$. If "tomorrow" p'' additional processes acquire new names, these processes will have their new names in the interval $[1..2p - 1]$ where $p = p' + p''$.

Renaming in enriched systems. A way to circumvent the adaptive $M = 2p - 1$ lower bound consists in enriching the asynchronous crash-prone read/write shared memory system with base objects stronger than atomic registers (see [20] for a short survey). Two such objects have received a particular attention. The first is the k-Test&Set object. Among the processes that access it, such an object ensures that at least one and at most k obtain value 1 (they win), while the others obtain 0 (they lose). An adaptive M-renaming algorithm based on k-Test&Set objects and atomic registers for $M = 2p - \lceil \frac{p}{k} \rceil$ is described in [19][2].

The second object that has received attention is the k-set agreement object. This object allows each process to propose a value and decide a value such that a decided value is a proposed value and at most k different values are decided. It is shown in [9] that it exists an adaptive M-renaming algorithm based on k-set agreement objects and atomic registers for $M = p + k - 1$. (It is important to recall that [9] shows such an algorithm exists but does not exhibit it.) In the same vein, enriching the shared memory system with Compare&Swap objects allows solving adaptive M-renaming with $M = p$ [20]. More generally, relations between adaptive renaming, k-Test&Set, and k-set agreement have been investigated in [11,12]. A failure detector suited to the design of an adaptive $(p + k - 1)$-renaming algorithm is defined in [21].

[1] More precisely, there is a $(2n-2)$-renaming algorithm for the values of n such that the integers in the set $\{\binom{n}{i} \ : \ 1 \le i \le \lfloor \frac{n}{2} \rfloor\}$ are relatively prime [8].

[2] The adaptive algorithm presented in [19] is actually based on k-set agreement objects but, as noticed by Gafni, it can be easily observed that these objects can be replaced by k-Test&Set objects without affecting the behavior of the algorithm.

1.2 Content of the Paper

Obstruction-freedom and k-obstruction-freedom. Wait-freedom is the strongest possible liveness condition proposed so far. It is starvation-freedom in presence of failures. *Obstruction-freedom* is a weaker progress condition whose definition is concurrency-oriented. An obstruction-free implementation of an object guarantees that a process that invokes an operation -and does not crash- returns from that invocation if it runs "long enough" in isolation [15] ("long enough" is used to capture the arbitrary duration required by that process to execute the operation; intuitively, it means that the process cannot run in isolation for an infinite time). While both wait-freedom and obstruction-freedom are progress conditions whose definition is independent of the failure pattern, the second one guarantees progress only in "favorable" concurrency patterns. (The interested reader will find in [13] a failure detector-based approach that boosts obstruction-freedom to wait-freedom.)

k-Obstruction-freedom is a generalization of obstruction-freedom [23]. It guarantees that, for every set of processes P, $|P| \leq k$, every process in P -that does not crash- returns from its operation invocation if no process outside P takes steps for "long enough". It is easy to see that k-obstruction-freedom and wait-freedom are equivalent in an n-process system such that $n \leq k$. Differently, when $k < n$, k-obstruction-freedom depends on the concurrency pattern while wait-freedom does not.

If we want that all operation invocations issued by correct processes do terminate, while the liveness condition satisfied by an object implementation is k-obstruction-freedom, concurrency is the main adversary. Actually, crashes favor the coverage assumption on which k-obstruction relies to become effective.

It is important to notice that a k-obstruction-free implementation has to always guarantee the safety property of the object that is built. This means that, while at most k processes execute for a "long enough" period of time, the others can be stopped at any point in the execution of an object operation. As wait-freedom, (k)-obstruction-freedom is a liveness property. Moreover, it is easy to see that no wait-free or k-obstruction-free implementation can be lock-based.

Content of the paper. When considering the obstruction-freedom liveness condition, consensus is solvable in asynchronous shared memory read/write systems. Similarly, it is possible to design an "optimal" adaptive M-renaming algorithm (optimal means here $M = p$) when we consider the obstruction-freedom liveness condition. Hence, the natural question: "Which is the smallest value $M(k)$ that can be attained for adaptive $M(k)$-renaming when we consider k-obstruction-freedom?"

This paper answers this question by presenting an adaptive $M(k)$-renaming algorithm that satisfies this progress condition and is such that $M(k) = \min(2p-1, p+k-1)$. This means that, in all executions whose concurrency degree is eventually bounded by k (eventual k-bounded concurrency), all correct processes do terminate. It follows that, when $k \geq p$, the algorithm solves $(2p-1)$-renaming in a wait-free manner.

As shown in [10,19], one way to ensure k-bounded concurrency is to use a k-set agreement object. Such an object can be used to reduce parallelism up to k processes. It follows that by combining such an algorithm and the algorithm described in the paper, we obtain an adaptive $(p+k-1)$-renaming algorithm based on a k-set agreement object and, consequently, we explicitly provide an optimal algorithm whose existence was claimed in [9]. Hence, the paper contributes to a better understanding of the renaming problem by showing that weakening the liveness condition from wait-freedom to k-obstruction-freedom allows us to reduce the new name space from $2p - 1$ to $p + k - 1$.

1.3 Roadmap

The paper is made up of 5 sections. Section 2 presents the computation model and Section 3 defines the adaptive M-renaming problem. Then, Section 4 presents the adaptive k-obstruction-free M-renaming algorithm with $M = \min(2p-1, p+k-1)$, and proves its correctness and its optimality. Finally, Section 5 concludes the paper.

2 Underlying Shared Memory Model

Process model. The system consists of n processes that we denote $p_1, \ldots, p_n$. The integer i is the index of p_i. Each process p_i has an initial name id_i such that $id_i \in [1..N]$ (N is arbitrarily large). Moreover, a process does not know the initial names of the other processes; it only knows that no two processes have the same initial name. A process can crash. Given an execution, a process that crashes is said to be *faulty*. Otherwise, it is *correct* in that execution. Each process progresses at its own speed, which means that the system is asynchronous.

Notation. A process can have local registers. Such registers are denoted with lowercase letters with the process index appearing as a subscript (e.g., $prop_i$ is a local register of p_i). The notation $\perp$ is used to denote a default value. Uppercase letters are used to denote shared objects.

Communication model. Processes communicate by accessing snapshot objects and arrays of size n of atomic one-writer/multi-reader registers.

As far as an array $X[1..n]$ is concerned, this means that only p_i can write $X[i]$, while any process p_j can read the whole array. Such a read, denoted $aa \leftarrow X[1..n]$ where aa is a local variable of the reader, is not atomic. The reader reads asynchronously all entries of the array in any order.

Snapshot objects have been introduced in [1]. A snapshot object X is an array of size n (the number of processes) that provides each process p_i with two operations denoted $X.\mathsf{update}_i(v)$ and $X.\mathsf{snapshot}_i()$. The former assigns v to $X[i]$ (and is consequently also denoted $X[i] \leftarrow v$). The important point is that only p_i can write $X[i]$. The latter operation, $X.\mathsf{snapshot}_i()$, returns to p_i the current value of the array X. The important point is that all update and snapshot operations appear as if they have been executed atomically, which means that a snapshot object is linearizable [17]. (These operations can be wait-free built on top of atomic read/write registers.)

3 Adaptive M-Renaming

Definition. The renaming problem has been informally stated in the introduction [2]. Each process p_i has an initial name id_i such that no two processes have the same initial name. These initial names belong to a set $\{1, \ldots, N\}$ such that $n << N$. Let new_name() be the (only) operation provided by an adaptive M-renaming object, i.e., an object that allows processes to obtain new distinct names belonging to the interval $[1..M]$. The behavior of this object (that defines the adaptive M-renaming problem) is defined by the following properties where p denotes the number of processes that invoke new_name() during an execution (the set of $p$articipating processes).

- Termination. If a correct process invokes new_name() it obtains a new name.
- Agreement. No two processes obtain the same new name.
- Adaptivity. A new name belong to $[1..M]$ where M is a function of p.
- Index independence. The behavior of a process is independent of its index.

The last property states that, if, in a run, a process whose index is i obtains the new name v, that process would have obtained the very same new name if its index was j. This means that, from an operational point of view, the indexes define an underlying communication infrastructure, namely, an addressing mechanism that can be used only to access entries of shared arrays. Indexes cannot be used to compute new names.

k-Obstruction-free termination. When considering the k-obstruction-freedom progress condition, it is possible that no process ever terminates if no set P of at most k processes execute in isolation for a long enough period. The termination property has then to be weakened as follows.

- Termination. For any subset P of correct processes, with $|P| \leq k$, if the processes of P execute new_name() in isolation, each process of P eventually obtains a new name and terminates its invocation.

"In isolation" means that the processes of P are the only ones to execute the operation new_name(). Let us notice that the previous progress property does not prevent processes outside P to be active (as long as they do not execute new_name()).

4 A k-Obstruction-free Renaming Algorithm

4.1 The Algorithm

Underlying shared objects. The processes cooperate through two arrays of atomic one-writer/multi-reader registers denoted $OLDNAMES[1..n]$, and $LEVEL[1..n]$, and a snapshot object denoted $NAMES$.

- Register $OLDNAMES[i]$, that is initialized to $\perp$, is used by p_i to store its identity id_i. Hence $OLDNAMES[i] \neq \perp$ means that p_i participates in the renaming.
- Register $LEVEL[i]$ is initialized to 0. In order to obtain a new name, the processes progress asynchronously from a level (starting from 1) to the next one. $LEVEL[i]$ contains the current level attained by process p_i. As we will see, if during a long enough period at most k processes take steps, they will stabilize at the same level and obtain new names.
- $NAMES[1..n]$ is a snapshot object initialized to $[\perp, \ldots, \perp]$. $NAMES[i]$ contains the new name that p_i tries to acquire. When p_i returns from new_name(), $NAMES[i]$ contains its new name.

Process behavior. Algorithm 1 describes the behavior of a process p_i. When it invokes $\mathsf{new\_name}(id_i)$, p_i deposits its identity id_i in $OLDNAMES[i]$ and proceeds from level 0 to level 1. The local variable $prop_i$ contains p_i's current proposal for its new name. Its initial value is $\perp$. Then, p_i enters a loop (lines 03-21) that it will exit at line 06 with its new name.

Each time it starts a new execution of the loop body, p_i first posts its current name proposal in $NAMES[i]$ and reads (with a $NAMES.\mathsf{snapshot}()$ invocation) the values of all current proposals (line 04). If its current proposal $prop_i$ is not $\perp$ and no other process has proposed the same new name (line 05), p_i defines its new name as the value of $prop_i$ and exits the loop (line 06). Otherwise, there is a conflict: several processes are trying to acquire the same new name $prop_i$. In that case, p_i enters lines 08-19 to solve this conflict. These lines constitute the core of the algorithm.

In case of conflict, p_i first reads asynchronously all entries of $LEVEL[1..n]$ and computes the highest level attained (line 07) $highest\_level_i$. If its current level is smaller than $highest\_level_i$, p_i jumps to that level, indicates it by writing $LEVEL[i]$ (lines 08-09) and proceeds to the next loop iteration.

If its current level is equal to $highest\_level_i$, p_i computes the set of processes it is competing with in order to acquire a new name, namely the set $contending_i$ (lines 10-11). Those are the processes whose level is equal to $highest\_level_i$. Then, the behavior of p_i depends on the size of the set $contending_i$ (predicate at line 12).

– If $|contending_i| > k$, there are too many processes competing when we consider k-obstruction-freedom. Process p_i progresses then to the next level and proceeds to the next loop iteration (line 13).
– If $|contending_i| \leq k$, p_i selects a new name proposal before proceeding to the next iteration. This selection is similar to what is done in other renaming algorithms (e.g., [5]). As defined at lines 14-15, *free* denotes the list of names that are currently available. Accordingly, p_i defines its new name proposal as the rth value in the list *free* where r is its rank in the set of (at most k) competing processes (hence, $1 \leq r \leq k$).

4.2 Proof of the Algorithm

Lemma 1. *No two processes decide the same new name.*

Proof. Let us assume, by way of contradiction, that two processes p_i and p_j decide the same new name nm (they decide it at line 06). It follows from the write and snapshot operations at line 04 and the associated predicate at line 05 that $names_i[i] = nm$ and $names_i[j] \neq nm$ when p_i checks the predicate at line 05. Similarly, as p_j decides nm we have $names_j[i] = nm$ and $names_j[i] \neq nm$ when it executes line 05. Moreover, none of p_i and p_j modifies $NAMES$ after it decides.

On another side, as all snapshot invocations are linearizable, the last invocations issued by p_i and p_j are totally ordered. Let us assume without loss of generality that the snapshot invocation by p_i precedes the one by p_j. It follows that we have $NAMES[i] = nm$ when p_j invokes $NAMES.\mathsf{snapshot}()$. Hence, the array $names_j$ obtained by p_j at line 04 is such that $names_j[i] = nm$ and $names_j[j] = nm$. consequently, the predicate

```
operation new_name($id_i$):
(01) $prop_i \leftarrow \bot$; $my\_level_i \leftarrow 1$;
(02) $OLDNAMES[i] \leftarrow id_i$; $LEVEL[i] \leftarrow my\_level_i$;
(03) repeat forever
(04)   $NAMES[i] \leftarrow prop_i$; $names_i \leftarrow NAMES$.snapshot();
(05)   if $\big((prop_i \neq \bot) \wedge (\forall j \neq i : names_i[j] \neq prop_i)\big)$
(06)      then return($prop_i$)
(07)      else $levels_i \leftarrow LEVEL[1..n]$; $highest\_level_i \leftarrow \max(\{levels_i[j]\})$;
(08)           if ($my\_level_i < highest\_level_i$)
(09)              then $my\_level_i \leftarrow highest\_level_i$; $LEVEL[i] \leftarrow my\_level_i$
(10)              else $oldnames_i \leftarrow OLDNAMES[1..n]$;
(11)                   $contending_i \leftarrow \{oldnames_i[j] \mid levels_i[j] = highest\_level_i\}$;
(12)                   if ($|contending_i| > k$)
(13)                      then $my\_level_i \leftarrow highest\_level_i + 1$; $LEVEL[i] \leftarrow my\_level_i$
(14)                      else let $X$ $= \{names_i[j] \mid names_i[j] \neq \bot\}$;
(15)                           let $free$ = the increasing sequence $1, 2, 3, \ldots$ from which
                                             the integers in $X$ have been suppressed;
(16)                           let $r$ = rank of $id_i$ in $contending_i$;
(17)                           $prop_i$ $\leftarrow$ the $r$th integer in the increasing sequence $free$
(18)                   end if
(19)           end if
(20)   end if
(21) end repeat.
```

Algorithm 1. x-Obstruction-free adaptive M-renaming with $M = \min(2p-1, p+k-1)$

checked by p_j at line 05 is false which contradicts the initial assumption and concludes the proof of the lemma. $\Box_{Lemma\ 1}$

Lemma 2. *Let p be the number of processes that participate in renaming. The size of the new name space is $M = \min(2p-1, p+x-1)$.*

Proof. Let us consider a run in which at most p processes participate. Let p_i be a process that returns a new name (line 06). It follows from the text of the algorithm that this name has been obtained by p_i at line 17.

Due to the very definition of the value p, when p_i defined its last name proposal, at most $p-1$ other processes have previously defined a name proposal. Moreover, because it proposes a new name only if $|contending| \leq k$ (line 12), the rank r of p_i in the set $contending$ (line 16) is such that $r \leq \min(p,k)$. It follows that the last name proposal computed by p_i from the pair $(free, r)$ (lines 14-17) is upper bounded by $(p-1) + \min(p,k)$. Hence we have $M = \min(2p-1, p+k-1)$. $\Box_{Lemma\ 2}$

Lemma 3. *For any subset P of processes, with $|P| \leq k$, if the processes of P run in isolation long enough, then all correct processes of P execute line 06 (they decide a new name and terminate).*

Proof. Let us assume, by way of contradiction, that there is an infinite execution of the algorithm in which, after a finite time τ, all the processes of P take an infinite number of steps and no process outside P takes any step.

Because $|P| \leq k$ and all processes in P take an infinite number of steps, there is a finite time $\tau' \geq \tau$ from which every process p_j of P has set $LEVEL[j]$ to the value $\max(\{LEVEL[j]\}_{1 \leq j \leq n}\}$ (lines 07-09) and does not modify $LEVEL[j]$ anymore (test at line 08). This follows from the fact that the test at line 08 is then always false and, consequently, line 09 is no longer executed). Hence, $LEVEL$ does not change after τ' and all processes in P eventually obtain the same set $contending$ (line 11). It follows that each process of P eventually obtains a distinct rank in $contending$.

Let p_i be the process of P with the smallest identity id_i and r its rank in $contending$ (this rank is not necessarily 1 because there may be crashed processes outside P that are in $contending$). Let z be the r-th integer in the sequence *free* not taken by a process outside P (lines 15-18). Because, after τ', $LEVEL$ does not change and p_i has the smallest rank among the processes of P, once all the processes of P have executed lines 15-18 after τ', p_i will be the only process to propose $prop_i = z$ (all the other processes of P will propose greater names). Hence the predicate evaluated by p_i at line 05 is eventually satisfied and, consequently, p_i terminates at line 06, which contradicts the initial assumption and completes the poof. $\square_{Lemma}$ 3

Theorem 1. *Algorithm 1 is an adaptive k-obstruction-free M-renaming algorithm with* $M = \min(2p-1, p+k-1)$.

Proof. The agreement property follows from Lemma 1. The adaptability property follows from Lemma 2. The index independence property follows directly from the text of the algorithm (indexes are used only to address array entries).

The k-obstruction-freedom termination property requires that for any set P of correct processes such that (1) $|P| \leq k$ and (2) the processes of P run in isolation long enough, the processes of P decide a new name. This is exactly what is proved by Lemma 3.
$\square_{Theorem}$ 1

4.3 Impossibility and Optimality Results

As mentioned previously, Castañeda and Rajsbaum [8] have recently shown that, for wait-free *non-adaptive* M-renaming, there are *exceptional* numbers n of processes for which the lower bound on the new name size M of the new name space $M = 2n-2$ (while $M = 2n-1$ for the other values of n). These exceptional values of n correspond to the cases when the set $\{\binom{n}{i} \ : \ 1 \leq i \leq \lfloor \frac{n}{2} \rfloor\}$ is relatively prime.

Adaptive $(2n-2)$-renaming, on the other hand, allows solving $(n-1)$-set agreement when $t = n-1$ [11]. Solving wait-free $(n-1)$-set agreement in shared memory has been shown to be impossible for any number of processes [7,16,22]. Thus the lower bound of $2p-1$ on the new name space size for wait-free adaptive renaming of p processes still holds. This lower bound is used in the proof of the following theorem.

Theorem 2. *Let p be the number of processes that participate in the renaming. There is no k-obstruction-free adaptive M-renaming with* $M < \min(2p-1, p+k-1)$.

Proof. There are two cases: $p \leq k$ and $p > k$.

First case: $p \leq k$*.* When less than k processes participate, k-obstruction-freedom is equivalent to wait-freedom, so this case boils down to wait-free adaptive renaming. The minimum size of the name space is then $2p - 1 = \min(2p - 1, p + k - 1)$.

Second case: $p > k$*.* Consider an execution in which $p' = p - k$ correct processes invoke the renaming and decide new names in a non-concurrent way (one after the other). The k-obstruction-freedom progress condition requires that all these processes terminate their invocations. These p' processes will then occupy p' names that cannot be decided by processes that will invoke renaming later. After these p' processes have returned from their invocation, $k' \leq k$ of the k remaining processes invoke the renaming and start executing concurrently. Because only $k' \leq k$ processes are executing concurrently, k-obstruction-freedom requires that all the correct processes terminate. These k' processes will then need a new name space of $2k' - 1 \leq 2k - 1$ (not including the names of the first p' processes) in order to obtain new names. The resulting new name space will then have a size not less than $p' + 2k - 1 = p + k - 1 = \min(2p - 1, p + k - 1)$, which concludes the proof of the theorem. $\square_{Theorem\ 2}$

The following corollary is an immediate consequence of Algorithm 1 (sufficiency) and the previous theorem (necessity).

Corollary 1. *Algorithm 1 is optimal with respect to the size of the new name space.*

5 Concluding Remarks

Using a non-linearizable object to replace the snapshot object. A process p_i accesses the snapshot object $NAMES[1..n]$ only at line 04 where it first writes $NAMES[i]$ and then reads atomically the whole array. This object can actually be replaced by an *immediate snapshot* object as defined in [7]. Such an object provides processes with a single operation, denoted write_snapshot(), that simultaneously writes a value and returns the value of the array (including the written value). In addition to the properties of a snapshot() operation, the write_snapshot() operation allows concurrent invocations to return the very same array value. Differently from snapshot objects, immediate snapshot objects are not required to be linearizable, but only to be set-linearizable (different operations can be considered as happening at the same point of the time line). Because of this, their implementation is less expensive.

The other direction. Assuming the k-obstruction-freedom progress condition, the proposed algorithm solves the adaptive $(p + k - 1)$-renaming problem for any value of k.

Assuming a system in which at most t processes may crash, we have the following in the other direction. An algorithm is described in [11] that solves the t-set agreement problem from any solution to the adaptive $(p + t - 1)$-renaming problem. Another construction is informally described in [10] that, given a k-set agreement algorithm, ensures that no more than k processes execute concurrently (k-bounded concurrency). It follows that, given an adaptive $(p + t - 1)$-renaming algorithm, the stacking of these two constructions provides us with a t-concurrent execution, thus satisfying the t-obstruction-freedom progress condition.

Design simplicity and efficiency. The proposed algorithm is particularly simple, which is a first class property. It could be possible to use the (pretty involved) construction suggested in [10] to obtain a k-obstruction-free adaptive $(p+k-1)$-renaming algorithm. The resulting algorithm would be difficult to understand and inefficient. Differently, the proposed ad-hoc algorithm is particularly simple. Albeit it is partially subjective and cannot be easily "measured", simplicity is a first class property.

When we consider sequential computing, it is not because a polynomial problem can be reduced to an NP-complete problem that simple and efficient solutions have not to be looked for. The situation is similar for renaming: even if a given size M of the new name space can be obtained from an "heavy machinery", simple and efficient solutions have to be looked for. Moreover, a simple solution is usually very close to the very essence of the problem.

Acknowledgments

The authors would like to thank the referees for their constructive comments.

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. 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)
3. Attiya, H., Guerraoui, R., Ruppert, E.: Partial Snapshot Objects. In: Proc. 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA 2008), pp. 336–343. ACM Press, New York (2008)
4. Attiya, H., Rachman, O.: Atomic Snapshots in O($n \log n$) Operations. SIAM Journal on Computing 27(2), 319–340 (1998)
5. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn., p. 414. Wiley-Interscience, Hoboken (2004)
6. Borowsky, E., Gafni, E., Generalized, F.L.P.: Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on Theory of Computing (STOC 1993), pp. 91–100. ACM Press, New York (1993)
7. Borowsky, E., Gafni, E.: Immediate Atomic Snapshots and Fast Renaming. In: Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC 1993), pp. 41–51 (1993)
8. Castaneda, A., Rajsbaum, S.: New Combinatorial Topology Upper and Lower Bounds for Renaming. In: Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC 2008), pp. 295–304. ACM Press, New York (2008)
9. 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)
10. Gafni, E., Guerraoui, R.: Simulating Few by Many (2009) (unpublished manuscript), http://www.cs.ucla.edu/~eli/eli/kconc.pdf
11. Gafni, E., Mostéfaoui, A., Raynal, M., Travers, C.: From Adaptive Renaming to Set Agreement. Theoretical Computer Science 410, 1328–1335 (2009)
12. 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 Press, Los Alamitos (2007)

13. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The Weakest Failure Detectors to Boost Obstruction-Freedom. Distributed Computing 20(6), 415–433 (2008)
14. Herlihy, M.P.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
15. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free Synchronization: Double-ended Queues as an Example. In: Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529 (2003)
16. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
17. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
18. Imbs, D., Raynal, M.: Help when needed, but no more: Efficient Read/Write Partial Snapshot (33/117). In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 142–156. Springer, Heidelberg (2009)
19. Mostefaoui, M., Raynal, M., Travers, C.: Exploring Gafni's Reduction Land: from Ω^k to Wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set Agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 1–15. Springer, Heidelberg (2006)
20. Raynal, M.: Locks Considered Harmful: a Look at Non-traditional Synchronization. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 369–380. Springer, Heidelberg (2008)
21. Raynal, M., Travers, C.: In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-free Set Agreement. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 1–17. Springer, Heidelberg (2006)
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)
23. Taubenfeld, G.: Contention-Sensitive Data Structure and Algorithms. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 157–171. Springer, Heidelberg (2009)

RobuSTM: A Robust Software Transactional Memory

Jons-Tobias Wamhoff[1], Torvald Riegel[1], Christof Fetzer[1], and Pascal Felber[2]

[1] Dresden University of Technology, Germany
{first.last}@tu-dresden.de
[2] University of Neuchâtel, Switzerland
{first.last}@unine.ch

Abstract. For software transactional memory (STM) to be usable in large applications such as databases, it needs to be *robust*, i.e., live, efficient, tolerant of crashed and non-terminating transactions, and practical. In this paper, we study the question of whether one can implement a robust software transactional memory in an asynchronous system. To that end, we introduce a system model – the *multicore system model* (MSM) – which captures the properties provided by mainstream multicore systems. We show how to implement a *robust software transactional memory* (RobuSTM) in MSM. Our experimental evaluation indicates that RobuSTM compares well against existing blocking and nonblocking software transactional memories in terms of performance while providing a much higher degree of robustness.

1 Introduction

Software transactional memory (STM) is a promising approach to help programmers parallelize their applications: it has the potential to simplify the programming of concurrent applications, when compared to using fine-grained locks. Our general goal is to investigate the use of STM in large software systems like application servers, databases, or operating systems. Such systems are developed and maintained by hundreds of programmers, and all that code lives in the same address space of the system's process. Ensuring the robustness of such applications requires the use of techniques that guarantee the recovery from situations in which individual threads crash or behave improperly (e.g., loop infinitely) while executing critical sections. For example, commercial databases guarantee such robustness using custom mechanisms for lock-based critical sections [12].

A system that uses transactions to perform certain tasks typically relies on their completion. Thus, a robust STM must guarantee that all *well-behaved* transactions will terminate within a finite number of steps. A transaction is *well-behaved* if it is neither *crashed* nor *non-terminating*. Both crashed and non-terminating transactions can interfere with the internal synchronization mechanism of the underlying STM implementation, possibly preventing other transactions from making progress if not handled correctly. A *crashed* transaction will stop executing prematurely, i.e., it executes a finite number of steps and stops before committing (e.g., due to failure of the associated thread). A *non-terminating* transaction executes an infinite number of steps without attempting to commit.

Note that a robust STM provides guarantees that are very similar to a *wait-free* STM, which guarantees to commit all concurrent transactions in a bound number of steps. Yet, the definition of the *wait-free* property requires the use of an asynchronous model of computation, but it has been shown recently [10] that one *cannot* implement a wait-free STM in an such a system model. However, current multicore systems provide stronger guarantees than those postulated in the asynchronous system model. Therefore, we try to answer the question whether one can implement a robust STM in today's multicore computer architectures.

In this paper, we introduce a new *multicore system model* (MSM). It is asynchronous in the sense that it does not guarantee any bounds on the absolute or relative speed of threads but additionally reflects the properties of mainstream multicore systems. We show that one can implement a robust STM (RobuSTM) in MSM that guarantees progress for individual threads. Our RobuSTM implementation exhibits performance comparable to state-of-the-art lock-based STMs on various types of benchmarks. Therefore, we not only show that one can implement *robust* STMs but also that one can implement them *efficiently*.

The paper is organized as follows: We first introduce MSM in Section 2. Section 3 presents the algorithm of RobuSTM. We evaluate our approach in Section 4 and discuss related work in Section 5. We conclude in Section 6.

2 System Model

Our multicore system model (MSM) satisfies the following nine properties. (1) A process consists of a non-empty set of threads that share an address space. (2) All non-crashed threads execute their program code with a non-zero speed. Neither the absolute nor the relative speed of threads is bounded. (3) Threads can fail by crashing. A crash can be caused by a programming bug or by a hardware issue. In the case of a hardware issue, we assume that the process crashes. In case of a software bug, only a subset of the threads of a process might crash. (4) We assume that STM is correctly implemented, i.e., crashes of threads are caused by application bugs and not by the STM itself. The motivation is that a STM has typically a much smaller code size that is reused amongst multiple applications. (5) A process can detect the crash of one of its threads. (6) Threads can synchronize using CAS and *atomic-or* operations (see below). (7) The state of a process is finite. (8) A thread can clone the address space of the process. (9) Each thread has a performance counter that counts the number of instructions it executed.

Software Transactional Memory. In our model, we assume that transactions are executed concurrently by threads. Within transactions, all accesses to shared state must be redirected to the STM and neither non-transactional accesses to global data nor external IO operations are permitted. If a thread failed to commit the transaction, it is retried (see Section 3). We further assume that transactions are non-deterministic and allow transactions to execute different code paths or access different memory locations during the retry.

Detection mechanisms. Modern operating systems permit the detection of thread crash failures. A thread can crash for various reasons like an uncaught exception. To detect the crash of a thread that is mapped to an operating system process, one can read the list of all processes that currently exist and check their status or search for missing threads [1]. The MSM assumes the existence of a thread crash detector that detects within a finite number of steps when a thread has crashed (i.e., the thread stopped executing steps) and will not wrongly suspect a correct thread to have crashed. For simplicity, in our implementation we assume that a signal handler is executed whenever a thread crashes.

Progress mechanisms. Like many concurrent algorithms, the MSM assumes the existence of a compare-and-swap (CAS) operation: `bool CAS(addr, expect, new)`. CAS atomically replaces the content of address `addr` with value `new` if the current value is `expect`. It returns `true` iff it stored `new` at `addr`. A CAS is often used in loops in which a thread retries until its CAS succeeds (see Figure 1). Note that sometimes such a loop might contain a contention manager to resolve a conflict with another thread but in the meantime a third thread might have successfully changed `addr`. In other words, a contention manager might not be able to ensure progress of an individual thread since this thread might have continuous contention with two or more other threads.

```
repeat
    expect = *addr;            ▷ Read current value
    new = function(expect);    ▷ Get new value
until CAS(addr, expect, new)
```

Fig. 1. While CAS is wait-free, there is no guarantee that the CAS will ever succeed, i.e., that the loop ever terminates

```
repeat
    if has_priority() then     ▷ Privileged priority
        atomic-or(addr, F);            ▷ Set fail bit
        expect = *addr;        ▷ Expect bit in CAS
    else                          ▷ All other threads
        expect = *addr & ∼F;        ▷ No fail bit
    end if
until CAS(addr, expect, new)
```

Fig. 2. Using an *atomic-or*, we can make sure the CAS of the privileged priority thread always succeeds

The problem is that there is no guarantee that a thread will ever be successful in performing a CAS. To address this issue, the MSM assumes an *atomic-or* operation. Note that the x86 architecture supports such an operation: a programmer can just add a `LOCK` prefix to a logical `or` operation. It is guaranteed, that a processor will execute the *atomic-or* operation in a finite number of steps. Also note that such an operation does not exist on, for example, Sparc processors.

We use the *atomic-or* to ensure that each correct transaction will eventually commit. RobuSTM will select at most one thread with a privileged priority level in the sense that this thread should win all conflicts. To ensure that all CAS operations performed by a privileged thread succeed, it uses the *atomic-or* to make sure that all competing CASes fail. To do so, we reserve a bit (`F`) in each word that is used with a CAS (see Figure 2). If a privileged thread performs an *atomic-or* just before another thread tries to perform a CAS, the latter will fail because its expected value assumes the `F` bit to be cleared.

Our goal is not only to implement wait-free transactions in the face of crash failures, but also in the face of non-terminating transactions. We assume however that the STM code itself is well-behaved and only application code can crash or loop infinitely often. For tolerating non-terminating transactions, we assume two more mechanisms that can be found in current systems. First, the MSM assumes that we can *clone* a thread, i.e., the operating system copies the address space of a process (using copy-on-write) and the cloned thread executes in a new address space fully isolated from all threads of the original process. Second, the MSM assumes the existence of a performance counter that (1) counts the cycles executed by a thread, and (2) permits other threads to read this performance counter. The intuition of the performance counter is as follows. The privileged thread can keep its privilege for a certain number of cycles (measured by the performance counter), after which it is not permitted anymore to steal the locks of other threads. If we can prove that the thread is well-behaved and would have simply needed more time to terminate, we increase the time quantum given to the privileged thread. Since the state space of threads is finite (but potentially very large), there exists a finite threshold S such that each transaction will either try to commit in at most S steps, or it will never try to commit. The problem is how to determine an upper bound on this threshold for non-deterministic transactions (see Section 3). Our system ensures that non-terminating transactions are eventually isolated to ensure the other threads can make progress while ensuring that long running but correct transactions will eventually commit.

3 Design and Implementation

Our STM algorithm runs in different modes. In this section, we first present the basic algorithm optimized for the good case with well-behaved transactions (Mode 1). When conflicts are detected and fairness is at stake, we switch to Mode 2 by prioritizing transactions. If the system detects a lack of progress, we switch to Mode 3 for dealing with crashed and non-terminating transactions. The mode is set for each transaction individually.

3.1 Why a Lock-Based Design?

Our robust STM algorithm uses a lock-based design. The reason for basing our work on a blocking approach instead of an obstruction-free one is driven by performance considerations. Non-blocking implementations suffer from costly indirections necessary for meeting their obstruction-free progress guarantee [5,4,16]. Although many techniques known from blocking implementations were applied to avoid indirection under normal operation with little contention, indirection is still necessary when it comes to conflicts with transactions that are not well-behaved (see Section 5). Our own experiments (see Section 4) still show a superior performance of lock-based designs.

Several reasons can explain the good performance of blocking STMs. They have a simpler fast path and more streamlined implementations of the read/write operations with no extra indirection. In addition, the combination of invisible reads and time-based validation [17] provides significant performance benefits.

In this paper, we use a C++ version of the publicly-available TinySTM [6] as the basis for our robust STM algorithm. TinySTM is an efficient lock-based implementation of the lazy snapshot algorithm (LSA) [17].

3.2 Optimizing for the Good Case

For completeness, we briefly recall here the basic algorithm used by TinySTM. Like several other word-based STM designs, TinySTM relies upon a shared array of *locks* to protect memory from concurrent accesses (see Figure 3). Each lock covers a portion of the address space. In our implementation, it uses a per-stripe mapping where addresses are mapped to locks based on a hash function.

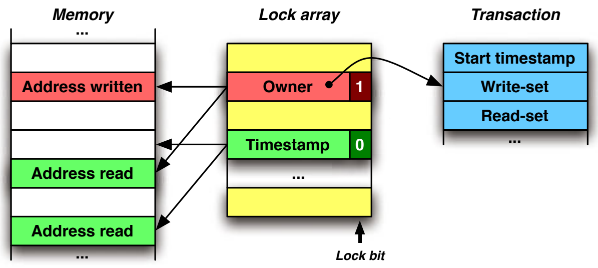


Fig. 3. Data structures for the lock-based design of TinySTM

Each lock is the size of an address on the target architecture. Its least significant bit is used to indicate whether the lock has been acquired by some transaction. If it is free, the STM stores in the remaining bits a version number that corresponds to the commit timestamp of the transaction that last wrote to one of the memory locations covered by the lock. If the lock is taken, the STM stores in the remaining bits a pointer to an entry in the write-set of the owner transaction. Note that addresses point to structures that are word-aligned and their least significant bits are always zero on 64-bit architectures; hence one of these bits can safely be used as lock bit.

When writing to a memory location, a transaction first identifies the lock entry that covers the memory address and atomically reads its value. If the lock bit is set, the transaction checks if it owns the lock using the address stored in the remaining bits of the entry. In that case, it simply writes the new value into the transaction-private write set and returns. Otherwise, there is a conflict and the default contention management policy is to immediately abort the transaction (we will show how one can change this behavior to provide fairness shortly).

If the lock bit is not set, the transaction tries to acquire the lock using a CAS operation. Failure indicates that another transaction has acquired the lock in the meantime and the whole procedure is restarted. If the CAS succeeds, the transaction becomes the owner of the lock. This basic design thus implements visible writes with objects being acquired when they are first encountered.

When reading a memory location, a transaction must verify that the lock is neither owned nor updated concurrently. To that end, the transaction reads the lock, then the memory location, and finally the lock again (obviously, appropriate memory barriers are used to ensure correct ordering of accesses). If the lock is not owned and its value (i.e., version number) did not change between both reads, then the value read is consistent. If the lock is owned by the transaction itself, the transaction returns the value from its write set. Once a value has been read, LSA checks if it can be used to construct a consistent snapshot. If that is not the case and the snapshot cannot be extended, the transaction aborts.

Upon commit, an update transaction that has a valid snapshot acquires a unique commit timestamp from the shared clock, writes its changes to memory, and releases the locks (by storing its commit timestamp as version number and clearing the lock bit). Upon abort, it simply releases any lock it has previously acquired. Refer to [17] for more details about the LSA algorithm.

3.3 Progress and Fairness

An important observation is that the basic TinySTM algorithm does not provide liveness guarantees even when considering only well-behaved transactions. In particular, a set of transactions can repeatedly abort each other, thus creating livelocks. Furthermore, there is no fairness between transactions: a long-running transaction might be taken over and aborted many times by shorter update transactions, in particular if the former performs numerous invisible reads

To address these problems, we introduce two mechanisms that make up Mode 2. The first one consists of introducing "visible reads" after a transaction has aborted a given number of times because of failed validation (i.e., due to invisible reads). To that end, in addition to the `WR` bit used for writers, we use an additional `RD` bit in the lock metadata to indicate that a transaction is reading the associated data (see Figure 4). Using a different bit for visible readers allows more concurrency because an invisible reader is still allowed to read data that is locked in read mode. Other conflicts with visible readers are handled as for writers, i.e., only one transaction is allowed to proceed. The use of visible reads makes all conflicts detectable at the time data is accessed: a well-behaved transaction that wins all conflicts is guaranteed not to abort.

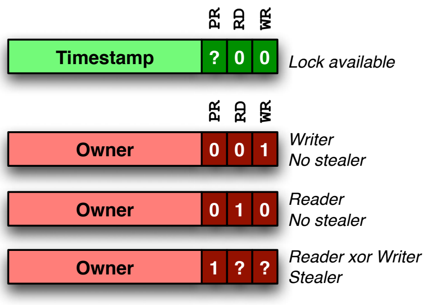


Fig. 4. Free and possibly reserved lock, owned lock, and owned lock with stealer

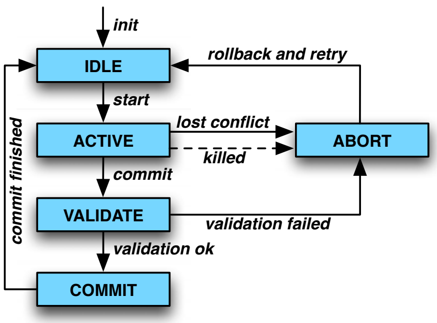


Fig. 5. States during the lifetime of a transaction

This mechanism alone is not sufficient to guarantee neither progress nor fairness. Depending on the contention management strategy, transactions can repeatedly abort each other, or a transaction might always lose to others and never commit. To address the fairness problem, we need to be able to prioritize transactions and choose which one to abort upon conflict. That way, we can ensure that the transaction with the highest priority level wins all its conflicts.

A transaction that cannot commit in Mode 1, first switches to visible reads. If it still fails to commit after a given number of retries with visible reads enabled, it tries to enter a *privileged priority* level that accepts only one thread at a time. Entry into this priority level is guarded using Lamport's bakery algorithm [13] that provides fairness by granting permission in the order in which transactions try to acquire the bakery lock. Because the number of steps that transactions are allowed to execute with priority is limited (see Section 3.6), each acquire attempt will finish in a finite number of steps. The privileged thread can steal a lock from its current owner by *atomic-or*ing the `PR` bit to 1 before acquiring it. The bit indicates that a transaction is about to steal the lock (see Figure 4). As explained in Section 2, this will ensure that any other thread attempting to CAS the lock metadata will fail (because it expects the `PR` bit to be cleared), while the privileged thread will succeed.

3.4 Safe Lock Stealing

Due to the lock-based nature of our base STM, being able to safely steal locks from transactions is necessary to build a robust STM. Our system model eases this because it requires that STM code is well-behaved and only application code can crash or loop infinitely often.

To understand how lock stealing works, consider Figure 5 that shows the different states a transactions can take. The normal path of a transaction is through states `IDLE`, `ACTIVE` (when transaction has started), `VALIDATE` (upon validation when entering commit phase), and `COMMIT` (after successful validation when releasing locks). A transaction can abort itself upon conflict (from `ACTIVE` state), or when validation fails (from `VALIDATE` state).

An active transaction can also be forcefully aborted (or killed) by another transaction in privileged priority (dashed arrow in the figure). This happens when the privileged transaction *tx* wants to acquire a lock that is already owned, i.e., with the `RD` or `WR` bit set. In that case, *tx* first reserves this lock for the privileged transaction by *atomic-or*ing the `PR` bit to 1. This wait-free operation also ensures that other non-privileged transactions will notice the presence of *tx* and will not be able to acquire the lock or clear the `PR` bit anymore (see Figure 2). In RobuSTM, all lock acquire and release operations must be performed using `CAS`, which will fail for non-privileged transactions if the `PR` bit is set.

After reserving the lock, *tx* can continue with actually stealing the lock. It loads the value of the lock again and determines whether there was an owner transaction. If so and if the owner is in the `IDLE` state, it can just acquire the lock. If the owner is in the `VALIDATE` or `COMMIT` states, *tx* waits for the owner to either abort (e.g., because validation failed) or finish committing. We do not abort validating transactions because they might be close to successfully committing. Because we assume that STM code is well-behaved and because read sets are finite, commit attempts execute in a finite number of steps. Note that a successfully committed transaction releases only the locks whose `PR` bit is not set. This process works as long as there is at most one transaction in the privileged priority level that can steal locks.

If the owner transaction is in `ACTIVE` state, *tx* attempts to abort the owner by using `CAS` to change the state to `ABORT`. After that or if the owner is already in state `ABORT`, *tx* acquires the lock using `CAS` but while doing so expects the value that the lock had after the *atomic-or*. The `PR` bit is only used during lock stealing and is not set after *tx* acquired the lock. Transactions check whether they have been aborted within each STM operation (e.g., loads). Note that a transaction's state is versioned to avoid ABA issues on lock owners, i.e., *tx* can distinguish if the transaction that previously owned the lock aborted and retried while performing the lock stealing.

3.5 Dealing with Crashed Transactions

Using a traditional lock-based STM could lead to infinite delays in case a thread that has acquired some locks crashes. Because RobuSTM supports lock stealing, the crash of a transaction that is not in privileged priority level and that is not in the `COMMIT` state does not prevent other transactions from safe lock stealing introduced in Section 3.4.

RobuSTM makes use of the crash detector included in the MSM to deal with crashed transactions. In practice, events that cause a thread to crash (e.g., a segmentation fault or an illegal instruction) are detected by the operating system and a thread can request to be notified about such events by registering a signal handler. If a signal is received by a thread that indicates a crash, the thread will abort itself if it is in the `ACTIVE` state to speed up future acquisitions by other threads. If the thread is in the `COMMIT` state and already started writing its modifications to memory, it will finish comitting. The intention there is to always keep the shared state consistent and to reduce the contention on locks. Transactions in privileged priority level that encountered a thread crash additional release their priority.

3.6 Dealing with Non-terminating Transactions

The main problem that we face when designing a robust STM is how to deal with non-terminating transactions as the locks they hold can prevent other transactions from making progress. Two different kinds of non-terminating transactions have to be distinguished: (1) transactions that are in `ACTIVE` state but stopped executing STM operations, and (2) transactions that still perform STM operations (e.g., in an infinite loop). Both correspond to non-crashed threads and never reach the `VALIDATE` state.

Let us first consider how RobuSTM handles threads that stopped executing STM operations (e.g., the thread is stuck in an endless loop). In the simplest case, the thread did not acquire any locks and thus does not prevent other threads from making progress and can be tolerated by the system. If the non-terminating transaction already acquired locks, it may run into a conflict with another thread. Eventually, the conflicting thread will reach the privileged priority level and again run into a conflict with the non-terminating transaction. It will then force the non-terminating transaction to abort and steals the lock. Since the status of a thread is only checked during STM operations, the non-terminating transaction will not discover the update and will remain in the `ABORT` state. Other transactions that encounter a conflict with a transaction in `ABORT` state can simply steal the lock.

A non-terminating transaction that still performs STM operations will discover the update of its state to `ABORT`. It will roll back and retry its execution. If, during the retry, it becomes again a non-terminating transaction that owns locks, it will be killed and retried again. It can therefore enter the privileged priority level and still behave as a non-terminating transaction, hence preventing all other transactions from making progress because it wins all conflicts. Since we assume that the state of a computer is finite, for each well-behaved and privileged transaction tx there exists a maximum number of steps, $maxSteps$, such that tx will execute at most $maxSteps$ before trying to commit. $maxSteps$ is not known a priori and hence, we cannot reasonably bound the number of steps that a privileged transaction is permitted to execute without risking to prevent some well-behaved transactions from committing.

MSM permits us to deal with non-terminating transactions running at the privileged priority level as follows. The privileged thread t_h receives a budget of at most a finite number of steps but at least $quantum$ steps, where $quantum$ is a dynamically updated value. Initially, $quantum$ is set to some arbitrary value that we assume to be smaller than $maxSteps$. The privileged thread t_h is forced to the `ABORT` state after $quantum$ steps (determined with the help of the performance counters) and is removed from the privileged priority level.

If the formerly privileged transaction t_h notices that it has been aborted and exceeded its quantum, it clones its thread. The clone consists of a separate address space that is copied on write from the parent and a single thread that runs in isolation. Transactional meta data of all threads in the parent is copied with the address space. The clone then continues to execute the transaction in a *checker run* using the meta data to resolve conflicts. There are two cases to consider. (1) If t_h is well-behaved, it will terminate after running for, say, $childSteps$. At this point, the child will return *success* and the parent thread will increase $quantum$ by setting it to a value of at least $childSteps$. Then, the parent thread will re-execute t_h at privileged priority with at least $quantum$ steps. If the new $quantum$ was not sufficient , e.g., because of non-determinism, it will be increased iteratively. (2) If t_h is not well-behaved, it will not terminate the checker run. In this case, the parent thread will wait forever for the child thread to terminate. Because the parent thread has aborted, it will not prevent any of the well-behaved threads from making progress.

4 Evaluation

In this section, we evaluate the performance of RobuSTM. We are specifically interested in showing that (1) it provides high throughput in good cases with little contention, (2) it provides fairness by guaranteeing progress of individual transactions, and (3) it tolerates crashed and non-terminating transactions.

We compare RobuSTM against four state-of-the-art STM implementations: TinySTM [6]; TinyETL, a C++ implementation of the encounter-time locking variant of TinySTM; TL2 [4], an STM implementation that uses commit-time locking; and NB STM [15], which combines efficient features of lock-based STM implementations with a non-blocking design, as our algorithm does. The NB STM implementation that we use is a port of the original SPARC implementation to the `x86` architecture.

For our evaluation, we use well-known micro-benchmarks and applications of the STAMP [2] benchmark suite. The *intset* micro-benchmarks perform queries and updates on integer sets implemented as *red-black tree* and *linked list*. We use the *bank* micro-benchmark to evaluate fairness: some threads perform money transfers (i.e., one withdrawal followed by a deposit) concurrently with long read-only transactions that compute the aggregated balance of all accounts. From the STAMP benchmark suite [2] we chose *Vacation*, *KMeans* and *Genome*. Vacation emulates a travel reservation system, reading and writing different tables that are implemented as red-black trees. KMeans clusters a set of points in parallel. Genome performs gene sequencing using hash sets and string search.

Our tests have been carried out on a dual-socket server with two Intel quad-cores (Intel XEON Clovertown, executing 64-bit Linux 2.6). We compiled all micro-benchmarks using the Dresden TM Compiler [3], which parses and transforms C/C++ transaction statements and redirects memory accesses to an STM.

4.1 Throughput for Well-Behaved Transactions

We first evaluate transaction throughput for lock-based and nonblocking STM implementations. There are no crashes or non-terminating transactions present.

Figure 6 shows the *bank* benchmark with low load under different STM runtimes. The left and middle plots show throughput for both transfer and aggregate-balance transactions. The lock-based STMs perform significantly faster

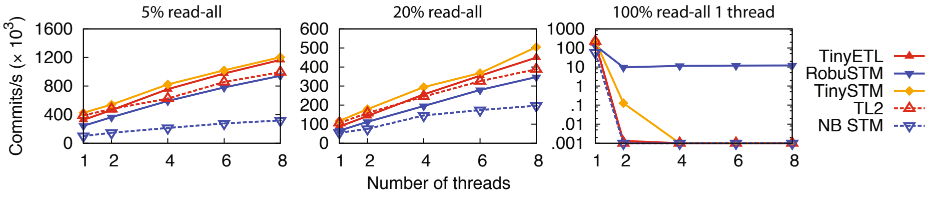


Fig. 6. Comparison lock-based vs. nonblocking STM (bank benchmark, 4096 accounts)

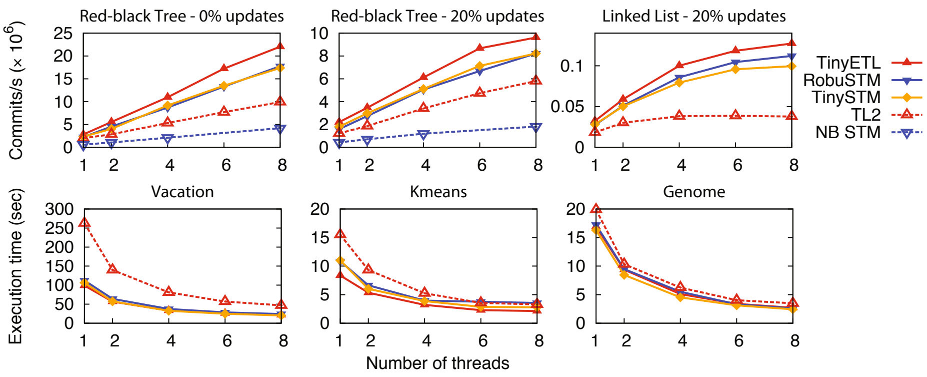


Fig. 7. Comparison of the performance of RobuSTM with micro-benchmarks (4096 initial elements) and STAMP applications (all with high contention)

than NB STM because the chosen nonblocking STM still requires an indirection step in case of contention. These results show why we would like RobuSTM to perform as well as blocking STMs. RobuSTM has more runtime overhead than TinyETL and TinySTM but is on par with TL2. Figure 7 shows performance results for additional micro-benchmarks and STAMP applications. Results for NB STM are only presented for the red-black tree because it requires manual instrumentation and it is not supported by the STAMP distribution. These results are in line with the *bank* benchmark results, showing that TinySTM and TinyETL perform best, followed by RobuSTM, then TL2 and finally NB STM.

The right plot of Figure 6 shows that the fairness that RobuSTM helps avoid starvation of the long aggregate-balance transactions with visible reads. In this plot, we only show the throughput of a single thread that is performing aggregate-balance (read-all) transactions. The guarantee for individual threads to make progress under MSM provides fairness for transactions that otherwise would not have a good chance to commit. Other STMs with invisible reads that simply abort upon conflict do not perform well because the read-only transaction will be continuously aborted.

4.2 Tolerating Crashes and Non-terminating Transactions

We now evaluate transaction throughput in the presence of crashes or non-terminating transactions. Ill-behaved transactions are simulated by injecting faults at the end of a transaction that performed write operations (i.e., it holds locks). We inject thread crashes by raising a signal and simulate non-terminating transactions by entering an infinite loop. The infinite loop either performs no operations on shared memory or continuously executes STM operations (e.g., transactional loads).

Orthogonal to the robustness that RobuSTM offers for synchronization, applications must be tolerant against faults of its threads. During the setup of our experiments we discovered two major problems with the thread-based benchmarks. (1) Barriers must be tolerant to faulty threads that never reach the barrier because of a crash or non-terminating code. (2) The workload cannot be pre-partitioned to the initial number of threads. Instead, it must be assigned dynamically, e.g., in each loop iteration. Therefore, we chose only a selection of STAMP applications that could be easily adapted. Using RobuSTM, an adapted application with initially N threads can tolerate up to $N-1$ faults because even ill-behaved transactions prevent the thread from processing its work. Increasing the number of tolerated faults would require a change in the programming model, e.g., based on a thread pool, and is not in the scope of this paper.

Figure 8 shows the performance of RobuSTM compared to TinyETL, the most efficient STM in the previous measurements. In each plot of the figure, we show the performance when some threads are faulty in the baseline 8-thread

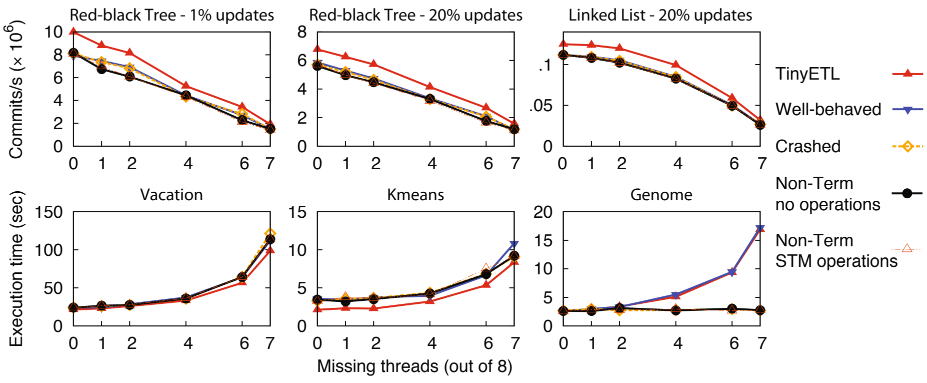


Fig. 8. Comparison of the performance for the benchmarks from Figure 7 with injected crashes and non-terminating transactions executing an infinite loop with or without STM operations

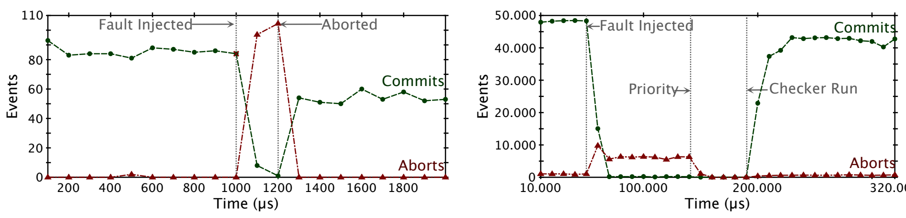


Fig. 9. Throughput for the red-black tree over time intervals under the presence of non-terminating transactions executing an infinite loop without (left) and with (right) STM operations. Vertical lines mark events of the fault-injected thread.

run. TinyETL is a run where faulty threads are simply not started in the runs, and thus shows the baseline. "Well-behaved" is similar (only well-behaved transactions), but uses RobuSTM. The other three lines show the performance in the presence of transactions that are not well-behaved. Faults were injected as early as possible, except for Genome, where they were injected in the last phase and can only be compared to the 8-thread runs. The results show that RobuSTM can ensure progress for an increasing number of injected faults. In fact, it can even compete with the throughput of the "well-behaved" case for the considered benchmarks.

To illustrate how RobuSTM behaves when non-terminating transactions are present, Figure 9 shows the number of commits and aborts over periods of time for the red-black tree benchmark. In the left graph, the benchmark is executed with two threads and one transaction enters an infinite loop that does not call STM operations. The remaining thread runs into a conflict and aborts repeatedly until it enters the privileged priority level. It then is allowed to kill the non-terminating transaction in order to steal its locks. Afterwards, the throughput picks up to the level of a single threaded execution. The right graph shows a scenario taken from Figure 8 with eight threads and one transaction that enters an infinite loop with STM operations. All remaining seven threads abort after running into a conflict and eventually reach privileged priority. Because the non-terminating transaction detects that it has been aborted during its STM operations, it retries. Thus, it must be aborted multiple times until it gains privileged priority. While the non-terminating transaction executes privileged, other threads wait and check the quantum of the non-terminating transaction. After the transaction detects that it was aborted because its quantum expired, it will clone its thread to enter the checker run. The period between gaining privileged priority and entering the checker run is much longer than the allowed quantum because it includes the costly clone of the process. After the initialization of the checker is finished, all locks are released in the parent process and the other threads can continue.

The results show that despite several crashed or non-terminating threads, RobuSTM is able to maintain a good level of commit throughput, effectively shielding other threads from failed transactions. We tested injecting faults in other STM implementations to justify our design decisions. Lock-based designs that acquire locks at commit-time (e.g., TL2) seem promising towards tolerating crashes and non-terminating transactions. Problems arise when fairness is at stake because memory accesses cannot be easily made visible. For implementations with encounter-time locking that simply abort on conflict (e.g., TinyETL), transactions that are not well-behaved and own locks lead to deadlocks. To overcome deadlocks, lock stealing and external abort of transactions must be supported. This will allow to tolerate crashes but not non-terminating transactions as they might continuously retry. We found that none of further existing approaches for contention management (see Section 5) met our robustness requirements.

5 Related Work

Non-blocking concurrent algorithms (e.g., lock-free or wait-free) ensure progress of some or all remaining threads even if one thread stops making progress. While many early STMs where non-blocking, most of the recent implementations use blocking algorithms because of their simpler design and better performance. Recent work on non-blocking STM [16,21] has shown that its performance can be substantially increased by applying techniques known from blocking STM implementations. This includes (1) timestamp-based conflict detection and (2) a reduced number of indirections while operating on transactional data by accessing memory in place in the common case. Depending on the algorithm, costly indirection is still required either during commit [16] or when stealing ownership records on conflict [21]. For the later, deflating the indirection is only possible after the original owner transaction moved to the abort state, but this might never happen for transactions that are not well-behaved.

Contention management was originally introduced to increase the throughput and avoiding possible livelocks (e.g., *Polite*, *Karma*, *Polka* [19,18]). An interesting observation is to back off the losing transaction after a conflict to avoid encountering the same conflict immediately upon retry. Contention managers that aim to provide fairness between short and long-running transactions usually rely on prioritization. The priority can be derived from the time when a transaction started or the amount of work it has done so far [18,9]. This helps long-running transactions reach their commit point but can delay short transactions extensively in case of high contention. Furthermore, crashed transactions will gain a high priority if it is based on the start time. An alternative is to derive the priority from the number of times the transaction has already been retried [19] and favor transactions with problems reaching their commit point.

In combination with priorities, simple mechanisms such as recency timestamps or liveness flags were introduced to determine the amount of time that contending transactions should back off. The goal is to increase the likelihood that a transaction that has already modified a memory location can commit (e.g., *Timestamp*, *Published Timestamp* [19,18]). These mechanisms are also used by transactions to show that they are not crashed. However, this approach does not work for non-terminating transactions because they may well update the timestamp or flag forever. The length of potential contention intervals can be reduced if locks are not acquired before commit time [20]. This would allow us to tolerate non-terminating transactions because they never try to commit [10], but by detecting conflicts lazily one cannot ensure that a transaction will eventually manage to commit (it can be repeatedly forced to abort by concurrent transactions that commit updates to shared memory).

Contention managers can also try to ensure progress of individual transactions. In the initial proposal of the *Greedy* contention manager [9], which guarantees that every well-behaved transaction commits within a bounded amount of time, thread failures could prevent global progress (i.e., the property that at least some thread makes progress). This issue was solved by giving each transaction a bounded period of time during which it could not be aborted by other transactions [8]. If a correct transaction exceeds this time limit and is aborted, it can retry with a longer delay. This approach works for crash failures but not for non-terminating transactions because the delay can grow arbitrarily large if the transaction is retried infinitely often.

Fich et al. [7] proposed an algorithm that converts any obstruction-free algorithm [11] into a practically wait-free one. The idea is that, in a semi-synchronous system, it is impossible to determine if a thread has crashed by observing its executed steps, as a step can take a bounded but unknown amount of time to complete. Thus, it is not possible to know a priori how long to wait for a possibly crashed transaction. Instead, one has to wait for increasingly longer periods. To decide if a thread had indeed crashed after expiration of the waiting period, they observe the *instruction counter* of the thread used to track progress. This approach cannot be applied straightforwardly to STMs because transactions can contain loops or perform operations with variable durations, e.g., allocate memory, so we cannot automatically and efficiently determine the abstract linear instruction counter of a running transaction.

Guerraoui and Kapalka were the first to take non-terminating transactions explicitly into account [10]. Their result is that the strongest progress guarantee that can be ensured in asynchronous systems is global progress, which is analogous to lock freedom. Since thread crashes and non-terminating transactions are not detected but tolerated, one cannot give to a single transaction an exclusive execution right because the thread might gain the right and never release it. We show in this paper that relying on a different but yet practical system model (see Section 2) allows us to build robust STMs that avoid these limitations and work on current multicore systems.

6 Conclusion

Robustness of transactional memory has often been ignored in previous research as the main focus was on providing performance. Yet, robustness to software bugs and application failures is an important property if one wants to use transactional memory in large mission-critical or safety-critical systems.

In this paper, we have introduced the multicore system model (MSM) that is practical in the sense that it reflects the properties of today's multicore computers. We have shown that (1) it is possible to build a robust STM with performance comparable to that of non-robust state-of-the-art STMs, and (2) we can implement such an STM under MSM.

Our experimental evaluation indicates that robustness only has a small additional overhead in the good case (i.e., no or few ill-behaved transactions), and performance remains good even when there are crashed and non-terminating threads. We expect to further improve efficiency by tuning the configuration parameters at runtime. For RobuSTM, these are especially the number of retries (1) after which transactions switch to using visible reads and (2) after which they attempt to run as a privileged transaction. Previous work has shown this to be very beneficial in the case of other STM configuration parameters [6]. We also expect that pairing this work with operating-system scheduling [14] could enable interesting optimizations.

Acknowledgements. This work is supported in part by the European Commission FP7 VELOX project (ICT-216852).

References

1. Aguilera, M., Walfish, M.: No time for asynchrony. In: HotOS 2009: Proceedings of the 12th Workshop on Hot Topics in Operating Systems. USENIX (2009)
2. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008: Proceedings of The IEEE International Symposium on Workload Characterization (2008)
3. Christie, D., Chung, J.-W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., Riviere, E.: Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In: EuroSys 2010 (2010)
4. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
5. Ennals, R.: Software transactional memory should not be obstruction-free. Technical report, Intel Research (2006)
6. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, New York (2008)
7. Fich, F., Luchangco, V., Moir, M., Shavit, N.: Obstruction-Free Algorithms can be Practically Wait-Free. LNCS. Springer, Heidelberg (2005)
8. Guerraoui, R., Herlihy, M., Kapalka, M., Pochon, B.: Robust contention management in software transactional memory. In: Workshop on Synchronization and Concurrency in Object-Oriented Languages (2005)
9. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC 2005: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing. ACM, New York (2005)
10. Guerraoui, R., Kapalka, M.: How Live Can a Transactional Memory Be? Technical report, EPFL (2009)
11. Herlihy, M., Luchangco, V., Moir, M., William, I., Scherer, N.: Software-transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd annual symposium on Principles of distributed computing. ACM, New York (2003)
12. Lahiri, T., Ganesh, A., Weiss, R., Joshi, A.: Fast-start: quick fault recovery in oracle. SIGMOD Rec. (2001)
13. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (1974)
14. Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J., Muller, G.: Scheduling support for transactional memory contention management. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010) (January 2010)
15. Marathe, V.J., Moir, M.: Efficient nonblocking software transactional memory. Technical report, Department of Computer Science, University of Rochester (2008)

16. Marathe, V.J., Moir, M.: Toward high performance nonblocking software transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, New York (2008)
17. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, Springer, Heidelberg (2006)
18. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC 2005: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing. ACM, New York (2005)
19. Scherer III, W., Scott, M.: Contention Management in Dynamic Software Transactional Memory. In: PODC Workshop on Concurrency and Synchronization in Java programs (2004)
20. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, New York (2008)
21. Tabba, F., Moir, M., Goodman, J.R., Hay, A.W., Wang, C.: NZTM: Nonblocking zero-indirection transactional memory. In: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. ACM, New York (2009)

A Provably Starvation-Free Distributed Directory Protocol*

Hagit Attiya[1,2], Vincent Gramoli[2,3], and Alessia Milani[4]

[1] Technion, Israel
[2] EPFL, Switzerland
[3] University of Neuchâtel, Switzerland
[4] LIP6, Université Pierre et Marie Curie, France

Abstract. This paper presents COMBINE, a distributed directory protocol for shared objects, designed for large-scale distributed systems. Directory protocols support move requests, allowing to write the object locally, as well as lookup requests, providing a read-only copy of the object. They have been used in distributed shared memory implementations and in data-flow implementations of distributed software transactional memory in large-scale systems.

The protocol runs on an overlay tree, whose leaves are the nodes of the system; it ensures that the cost of serving a request is proportional to the cost of the shortest path between the requesting node and the serving node, in the overlay tree. The correctness of the protocol, including starvation freedom, is proved, despite asynchrony and concurrent requests. The protocol avoids race conditions by *combining* requests that overtake each other as they pass through the same node. Using an overlay tree with a good stretch factor yields an efficient protocol, even when requests are concurrent.

1 Introduction

Distributed applications in large-scale systems aim for good *scalability*, offering proportionally better performance as the number of processing nodes increases, by exploiting communication to access nearby data [19].

An important example is provided by a *directory-based consistency* protocol [7]; in such a protocol, a *directory* helps to maintain the coherence of objects among entities sharing them. To access an object, a processor uses the directory to obtain a copy; when the object changes, the directory either updates or invalidates the other copies. In a large-scale system, the directory manages copies of an object, through a communication mechanism supporting the following operations: A writable copy of the object is obtained with a move request, and a read-only copy of the object is obtained with a lookup request.

A directory-based consistency protocol is better tailored for large-scale distributed systems, in which remote accesses require expensive communication,

* Hagit Attiya is partially supported by the *Israel Science Foundation* (grant number 953/06). Alessia Milani is financed by ANR grant R-DISCOVER.

several orders of magnitude slower than local ones. Reducing the *cost of communication* with the objects and the number of remote operations is crucial for achieving good performance in distributed shared memory and transactional memory implementations. Several directory protocols for maintaining consistency have been presented in the literature, e.g., [1, 6, 7, 9]. (They are discussed in Section 6.)

In large-scale systems, where the communication cost dominates the latency, directory protocols must order the potentially large number of requests that are contending for the same object. Rather than channeling all requests to the current location of the object, some directory protocols, e.g., [9], implement a *distributed queue*, where a request from p gets enqueued until the object gets acquired and released by some *predecessor* q, a node that p detects as having requested the object before.

This paper presents COMBINE, a new directory protocol based on a distributed queue, which efficiently accommodates concurrent requests for the same object, and non-fifo message delivery. COMBINE is particularly suited for systems in which the cost of communication is not uniform, that is, some nodes are "closer" than others. Scalability in COMBINE is achieved by communicating on an *overlay tree* and ensuring that the cost of performing a lookup or a move is therefore proportional to the cost of the shortest path between the requesting node and the serving node (its predecessor), in the overlay tree. The simplicity of the overlay tree, and in particular, the fact that the object is held only by leaf nodes, facilitates the proof that a node finds a previous node holding the object.

To state the communication cost more precisely, we assume that every pair of nodes can communicate, and that the cost of communication between pairs of nodes forms a *metric*, that is, there is a symmetric positive *distance* between nodes, denoted by $\delta(.,.)$, which satisfies the triangle inequality. The *stretch* of a tree is the worst case ratio between the cost of direct communication between two nodes p and q in the network, that is, $\delta(p,q)$, and the cost of communicating along the shortest tree path between p and q.

The communication cost of COMBINE is proportional to the cost of the shortest path between the requesting node and the serving node, *times the stretch of the overlay tree*. Thus, the communication cost improves as the stretch of the overlay tree decreases. Specifically, the cost of a lookup request by node q that is served by node p is proportional to the cost of the shortest tree path between p and q, that is, to $\delta(p,q)$ times the stretch of the tree. The cost of a move request by node p is the same, with q being the node that will pass the object to p.

When communication is asynchronous and requests are concurrent, however, bounding the communication cost does not ensure that a request is eventually served (*starvation-freedom*). It remains to show that while p is waiting for the object from a node q, a finite, acyclic waiting chain is being built between q and the node owning the object (the head of the queue). Possibly, while the request of p is waiting at q, many other requests are passing over it and being placed ahead of it in the queue, so p's request is never served. We prove that this does not happen, providing the first complete proof for a distributed queue protocol, accommodating asynchrony and concurrent requests.

A pleasing aspect of Combine is in not requiring fifo communication links. Prior directory protocols [9, 12, 23] assume that links preserve the order of messages; however, ensuring this property through a link-layer protocol can significantly increase message delay. Instead, as its name suggests, Combine handles requests that overtake each other by *combining* multiple requests that pass through the same node. Originally used to reduce contention in multistage interconnection networks [16, 20], combining means piggybacking information of distinct requests in the same message.

Organization: We start with preliminary definitions (Section 2), and then present Combine in Section 3. The termination proof and analysis of our protocol are given in Section 4. Section 5 discusses how to construct an overlay tree. Finally, related work is presented in Section 6 and we conclude in Section 7.

2 Preliminaries

We consider a set of nodes V, each with a unique identifier, communicating over a complete communication network. If two nodes p and q do not have a direct physical link between them, then an underlying routing protocol directs the message from p to q through the physical communication edges.

The cost of communication between nodes is non-uniform, and each edge (p, q) has a weight which represents the cost for sending a message from p to q, denoted $\delta(p, q)$. We assume that the weight is symmetric, that is, $\delta(p, q) = \delta(q, p)$, and it satisfies the triangle inequality, that is, $\delta(p, q) \leq \delta(p, u) + \delta(u, q)$.

The *diameter* of the network, denoted Δ, is the maximum of $\delta(p, q)$ over all pairs of nodes.

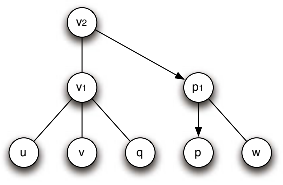
We assume reliable message delivery, that is, every message sent is eventually received. A node is able to receive a message, perform a local computation, and send a message in a single atomic step.

We assume the existence of a rooted *overlay tree* T, in which all physical nodes are leaves and inner nodes are mapped to physical nodes.

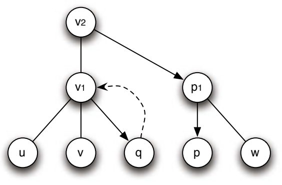
Let $d_T(p, q)$ be the number of hops needed to go from a leaf node p up to the lowest common ancestor of p and leaf node q in T, and then down to q (or vice versa); $\delta_T(p, q)$ is the sum of the costs of traversing this path, that is, the sum of $\delta(.,.)$ for the edges along this path.

The *stretch* of an overlay tree T is the ratio between the communication cost over T and the direct communication cost. That is, $stretch_T = \max_{p,q} \frac{\delta_T(p,q)}{\delta(p,q)}$. Section 5 presents ways to construct overlay trees with small stretch.

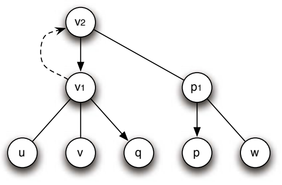
The *depth* of T, denoted D_T, is the number of hops on the longest path from the root of T to a leaf; the *diameter* of T, denoted Δ_T, is the maximum of $\delta_T(p, q)$, over all pairs of nodes.



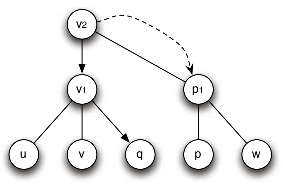
(a) Initially, p owns the object



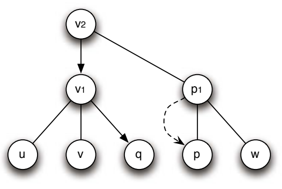
(b) Node q issues a move request



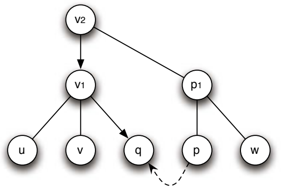
(c) Pointers from the root lead to q



(d) Previous pointers are discarded



(e) The request reaches the predecessor p



(f) Object is moved from p to q

Fig. 1. A move request initialized by node q when the object is located at p

3 The Combine Protocol

The protocol works on an overlay tree. When the algorithm starts, each node knows its parent in the overlay tree. Some nodes, in particular, the root of the overlay tree, also have a *downward* pointer towards one neighbor (other than its parent).

The downward pointers create a path in the overlay tree, from the root to the leaf node initially holding the object; in Figure 1(a), the arrows indicate downward pointers towards p.

A node requesting the object x tries to find a *predecessor*: a nearby node waiting for x or the node currently holding x. Initially, p, the node holding the object is this predecessor.

We *combine* multiple requests, by piggybacking information of distinct requests in the same message, to deal with concurrent requests.

- A node q obtains the current value of the object by executing a lookup request. This request goes up in the overlay tree until it discovers a pointer towards the downward path to a predecessor; the lookup records its identifier at each visited node. When the request arrives at this predecessor, it sends a read-only copy directly to q. Each node stores the information associated to at most one request for any other node.
- A node q acquires an object by executing a move request. This request goes up in the overlay tree until it discovers a pointer towards the downward path to a predecessor. This is represented by the successive steps of a move as indicated in Figure 1. The move sets downward pointers towards q while going

up the tree, and resets the downward pointers it follows while descending towards a predecessor. If the move discovers a stored lookup it embeds it rather than passing over it. When the move and (possibly) its embedded lookup reach a predecessor p, they wait until p receives the object. After p receives the object and releases it, p sends the object to q and a read-only copy of the object to nodes who issued the embedded lookup requests.

Since the downward path to the object may be changing while a lookup (or a move) is trying to locate the object, the lookup may remain blocked at some intermediate node u on the path towards the object. Without combining, a move request could overtake a lookup request and remove the path of pointers, thus, preventing it from terminating. However, the identifier stored in all the nodes a lookup visits on its path to the predecessor allows an overtaking move to embed the lookup. This guarantees termination of concurrent requests, even when messages are reordered. Information stored at the nodes ensures that a lookup is not processed multiple times.

We now present the algorithm in more detail. The state of a node appears in Algorithm 1. Each node knows its *parent* in the overlay tree, except the root, whose $parent = \perp$. A node might have a *pointer* towards one of its children (otherwise it is $\perp$); the *pointer* at the root is not $\perp$. Each node also maintains a variable *lookups*, holding information useful for combining.

Algorithm 1. State and Message

```
1:  State of a node $u$:
2:     $parent \in \mathbb{N} \cup \{\perp\}$, representing the parent node in the tree
3:     $pointer \in \mathbb{N} \cup \{\perp\}$, the direction towards the known predecessor, initially $\perp$
4:     $lookups$ a record (initially empty) of lookup entries with fields:
5:        $id \in \mathbb{N}$, the identifier of the node initiating the request
6:        $ts \in \mathbb{N}$, the sequence number of the request
7:        $status \in \{$not-served, served, passed$\}$, the request status
8:  Message type:
9:     $message$ a record with fields:
10:       $phase \in \{$up, down$\}$
11:       $type \in \{$move, lookup$\}$
12:       $id \in \mathbb{N}$, the identifier of the request
13:       $ts \in \mathbb{N}$, the sequence number of the request
14:       $lookups$, a record of embedded lookup entries
```

The lookup() *operation:* A lookup request r issued by a node q carries a unique identifier including its sequence number, say $ts = \tau$, and its initiator, $id = q$. Its pseudocode appears in Algorithm 2. A lookup can be in three distinct states: it is either running and no move overtook it (not-served), it is running and a move request overtook and embedded it (passed), or it is over (served).

The lookup request proceeds in two subsequent phases. First, its initiator node sends a message that traverses its ancestors up to the first ancestor whose *pointer*

Algorithm 2. Lookup of object x at node u

```
1:  Receiving ⟨up, lookup, $q, \tau, *$⟩ from $v$:                                ▷ Lookup up phase
2:    if $\nexists\langle q, \tau_1, *\rangle \in u.lookups : \tau_1 \geq \tau$ then                 ▷ Not a stale message?
3:      if $\exists r_q = \langle q, \tau_2, *\rangle \in u.lookups : \tau_2 < \tau$ then         ▷ First time we hear about?
4:        $u.lookups \leftarrow u.lookups \setminus \{r_q\} \cup \{\langle q, \tau, \text{not-served}\rangle\}$   ▷ Overwrite stored lookup
5:      else $u.lookups \leftarrow u.lookups \cup \{\langle q, \tau, \text{not-served}\rangle\}$              ▷ Store lookup
6:      if $u.pointer = \perp$ then
7:        send ⟨up, lookup, $q, \tau, \perp$⟩ to $u.parent$                          ▷ Resume up phase
8:      else send ⟨down, lookup, $q, \tau, \perp$⟩ to $u.pointer$                    ▷ Start down phase

9:  Receiving ⟨down, lookup, $q, \tau, *$⟩ from $v$:                              ▷ Lookup down phase
10:   if $\nexists\langle q, \tau_1, *\rangle \in u.lookups : \tau_1 \geq \tau$ then
11:     if $\exists r_q = \langle q, \tau_2, *\rangle \in u.lookups : \tau_2 < \tau$ then
12:       $u.lookups \leftarrow u.lookups \setminus \{r_q\} \cup \{\langle q, \tau, \text{not-served}\rangle\}$
13:     else $u.lookups \leftarrow u.lookups \cup \{\langle q, \tau, \text{not-served}\rangle\}$
14:     if $u$ is a leaf then
15:       send $x_{read\text{-}only}$ to $q$        ▷ Blocking send (i.e., executes as soon as u releases x)
16:     else send ⟨down, lookup, $q, \tau, \perp$⟩ to $u.pointer$                    ▷ Resume down phase
```

indicates the direction towards a predecessor—this is the *up phase* (Lines 1–8). Second, the lookup message follows successively all the downward pointers down to a predecessor—this is the *down phase* (Lines 9–16). The protocol guarantees that there is a downward path of pointers from the root to a predecessor, hence, the lookup finds it (see Lemma 2).

A node keeps track of the lookups that visited it by recording their identifier in the field lookups, containing some lookup identifiers (i.e., their initiator identifier *id* and their sequence number *ts*) and their *status*. The information stored by the lookup at each visited node ensures that a lookup is embedded at most once by a move. When a new lookup is received by a node u, u records the request identifier of this freshly discovered lookup. If u had already stored a previous lookup from the same initiator, then it overwrites it by the more recent lookup, thus keeping the amount of stored information bounded (Lines 3–4).

Due to combining, the lookup may reach its predecessor either by itself or embedded in a move request. If the lookup request r arrives at its predecessor by itself, then the lookup sends a read-only copy of the object directly to the requesting node q (Line 15 of Algorithm 2).

The move() *operation:* The move request, described in Algorithm 3, proceeds in two phases to find its predecessor, as for the lookup. In the up phase (Lines 1–11), the message goes up in the tree to the first node whose downward pointer is set. In the down phase (Lines 12–26), it follows the pointers down to its predecessor. The difference in the up phase of a move request is that an intermediate node u receiving the move message from its child v sets its $u.pointer$ down to v (Line 8). The difference in the down phase of a move request is that each intermediary node u receiving the message from its parent v resets its $u.pointer$ to $\perp$ (Line 16).

Algorithm 3. Move of object x at node u

```
1:  Receiving $m = \langle \mathsf{up}, \mathsf{move}, q, \tau, lookups \rangle$ from $v$:        ▷ Move up phase
2:    $\mathsf{clean}(m)$
3:    for all $r_a = \langle a, \tau, \mathsf{not\text{-}served} \rangle \in u.lookups$ do
4:      if $\nexists \langle a, \tau', * \rangle \in m.lookups : \tau' \geq \tau$ then
5:        $m.lookups \leftarrow m.lookups \cup \{r_a\}$        ▷ Embed non-served lookups
6:      $u.lookups \leftarrow u.lookups \setminus \{r_a\} \cup \{\langle a, \tau, \mathsf{served} \rangle\}$        ▷ Mark lookups as served
7:    $oldpointer \leftarrow u.pointer$
8:    $u.pointer \leftarrow v$        ▷ Set downward pointer
9:    if $oldpointer = \perp$ then
10:     send $\langle \mathsf{up}, \mathsf{move}, q, \tau, m.lookups \rangle$ to $u.parent$        ▷ Resume up phase
11:   else send $\langle \mathsf{down}, \mathsf{move}, q, \tau, m.lookups \rangle$ to $oldpointer$        ▷ Start down phase

12: Receiving $m = \langle \mathsf{down}, \mathsf{move}, q, \tau, lookups \rangle$ from $v$:        ▷ Move down phase
13:   $\mathsf{clean}(m)$
14:   if $u$ is not a leaf then        ▷ Is predecessor not reached yet?
15:     $oldpointer \leftarrow u.pointer$
16:     $u.pointer \leftarrow \perp$        ▷ Unset downward pointer
17:     for all $r_a = \langle a, \tau, \mathsf{not\text{-}served} \rangle \in u.lookups$ do
18:       if $\nexists \langle a, \tau', * \rangle \in m.lookups : \tau' \geq \tau$ then
19:         $m.lookups \leftarrow m.lookups \cup \{r_a\}$
20:       $u.lookups \leftarrow u.lookups \setminus \{r_a\} \cup \{\langle a, \tau, \mathsf{served} \rangle\}$
21:     send $m$ to $oldpointer$        ▷ Resume down phase
22:   else        ▷ Predecessor is reached
23:     for all $\langle a, \tau, status \rangle \in m.lookups : \nexists \langle a, \tau', * \rangle \in u.lookups$ with $\tau' \geq \tau$ do
24:       send $x_{read\text{-}only}$ to $a$        ▷ Blocking send of read-only copy
25:     send $x$ to $q$        ▷ Blocking send of object
26:     $\mathsf{delete}(x)$        ▷ Remove object local copy

27: $\mathsf{clean}(m)$:        ▷ Clean-up the unused information
28:   for all $\langle a, \tau, \mathsf{not\text{-}served} \rangle \in m.lookups$ do
29:     if $\exists \langle a, \tau', status \rangle \in u.lookups : (status = \mathsf{served} \wedge \tau' = \tau) \vee (\tau' > \tau)$ then
30:       $m.lookups \leftarrow m.lookups \setminus \{\langle a, \tau, \mathsf{not\text{-}served} \rangle\}$
31:     if $\langle a, \tau, * \rangle \notin u.lookups$ then
32:       $m.lookups \leftarrow m.lookups \setminus \{\langle a, \tau, \mathsf{not\text{-}served} \rangle\} \cup \{\langle a, \tau, \mathsf{passed} \rangle\}$
33:       $u.lookups \leftarrow u.lookups \cup \{\langle a, \tau, \mathsf{passed} \rangle\}$
```

For each visited node u, the move request embeds all the lookup requests stored at u that need to be served and marks them as served in u (Lines 3–6, 17–20 of Algorithm 3).

Along its path, the move may discover that either some lookup r it embeds has been already served or that it overtakes some embedded lookup r' (Line 29 or Line 31, respectively, of Algorithm 3). In the first case, the move just erases r from the lookups it embeds, while in the second case the move marks, both in the tuple it carries and locally at the node, that the lookup r' has been passed (Line 30 or Lines 32–33, respectively, of Algorithm 3).

Once obtaining the object at its predecessor, the move request first serves all the lookup requests that it embeds (Lines 23, 24), then sends the object to the node that issued the move (Line 25) and finally deletes the object at the current node (Line 26).

If the object is not at its predecessor when the request arrives, the request is enqueued and its initiator node will receive the object as soon as the predecessor releases the object (after having obtained it).

Concurrent request considerations: Note that a lookup may not arrive at its predecessor because a concurrent move request overtook it and embeds it, that is, the lookup r found at a node u that $u.pointer$ equals v, later, a move m follows the same downward pointer to v, but arrives at v before r. The lookup detects the overtaking by m and stops at node v (Line 13 of Algorithm 3, and Lines 2 and 10 of Algorithm 2). Finally, the move m embeds the lookup r and serves it once it reaches its predecessor (Lines 23, 24 of Algorithm 3 and Lines 31, 32 of Algorithm 3).

Additionally, note that no multiple move requests can arrive at the same predecessor node, as a move follows a path of pointers that it immediately removes. Similarly, no lookup arrives at a node where a move already arrived, unless embedded in it. Finally, observe that no move is issued from a node that is waiting for the object or that stores the object.

4 Analysis of Combine

A request initiated by node p is served when p receives a copy of the object, which is read-only in case of a lookup request. This section shows that every request is eventually served, and analyzes the communication cost. We start by considering only move requests, and then extend the analysis to lookup requests.

Inspecting the pseudocode of the up phase shows that, for every $\ell > 1$, a move request m sets a downward pointer from a node u at level ℓ to a node u' at level $\ell - 1$ only if it has previously set a downward pointer from u' to a node at level $\ell - 2$. Thus, *assuming no other move request modifies these links*, there is a downward path from u to a leaf node. The proof of the next lemma shows that this path from u to a leaf exists even if another move request redirects a pointer set by m at some level $\ell' \leq \ell$.

Lemma 1. *If there is a downward pointer at a node u, then there is a downward path from u to a leaf node.*

Proof. We prove that if there is a downward pointer at node u at level ℓ, then there is a path from u to a leaf node. We prove that this path exists even when move requests may redirect links on this path from u to a leaf node.

The proof is by induction on the highest level ℓ' such that no pointer is redirected between levels ℓ' and ℓ. The base case, $\ell' = 1$, is obvious.

For the inductive step, $\ell' > 1$, assume that there is always a path from u to a leaf node, even if move requests change any of the pointers set by m at some level below $\ell' - 1$. Let m' be a move request that redirects the downward pointer at level ℓ', that is, m' redirects the link at node u' to a node v at level $\ell' - 1$. (Note that the link redirection is done atomically by assumption, so that two nodes can not redirect the same link in two different directions.) However, by the inductive hypothesis (applied to m'), this means that there is a downward path from v to a leaf node. Hence, there is a downward path from u' to a leaf node, and since no pointer is redirected at the levels between ℓ' and ℓ, the inductive claim follows. □

Lemma 2. *At any configuration, there is a path of downward pointers from the root to a leaf node.*

Proof. Initially, the invariant is true by assumption. The claim follows from Lemma 1, since there is always a downward pointer at the root. □

Observation 1. *A* move *request is not passed by another* move *request, since setting of the link and sending the same request in the next step of the path (upon receiving a* move *request) happen in an atomic step.*

We next argue that a request never backtracks its path towards the object.

Lemma 3. *A* move *request m does not visit the same node twice.*

Proof. Assume, by way of contradiction, that m visits some node twice. Clearly, a move request does not backtrack during its down phase. Then, let u be the first node that m visits twice during its up phase.

Since m does not visits the same node twice during the down phase, m does not find a downward link at u, when visiting u for the first time. Thus, m continues to u', the parent of u.

If m finds a downward link to u at u', then we obtain a contradiction, by Observation 1 and since the only way for this downward link to exist is that m has been passed by another move request. Otherwise, m does not find a downward link at u', and due to the tree structure, this contradicts the fact that u is the first node that m visits twice. □

A node p is the *predecessor* of node q if the move message sent by node p has reached node q and p is waiting for q to send the object.

Lemma 4. *A* move *request m by node p reaches its predecessor q within $d_T(p, q)$ hops and $\delta_T(p, q)$ total cost.*

Proof. Since there is always a downward pointer at the root, Lemma 1 and Observation 1 imply that the request m eventually reaches its predecessor q. Moreover, by Lemma 3 and the tree structure, the up-phase eventually completes by reaching the lowest common ancestor of p and q. Then m follows the path towards q using only downward pointers. Thus, the total number of hops traversed by m during both phases is $d_T(p, q)$ and the total cost is $\delta_T(p, q)$. □

This means that D_T is an upper bound on the number of hops for a request to find its predecessor.

Lemma 4 already allows to bound the *communication cost* of a request issued by node q, that is, the cost of reaching the node p from which a copy of the object is sent to q. Observe that once the request reaches p, the object is sent directly from p to q without traversing the overlay tree.

Theorem 1. *The communication cost of a request issued by node q and served by node p is $O(\delta_T(p, q))$.*

Clearly, embedding lookup requests in move requests does not increase the message complexity, but it might increase the bit complexity. In [4], we measure the total number of bits sent on behalf of a request, and show that combining does not increase the number of bits transmitted due to a lookup request. (The argument is straightforward for move requests, which are never embedded.)

Note that finding a predecessor does not immediately imply that the request of p does not starve, and must be eventually served. It is possible that although the request reaches the predecessor q, q's request itself is still on the path to its own predecessor. During this time, other requests may constantly take over p's request and be inserted into the queue ahead of it. We next limit the effect of this to ensure that a request is eventually served.

For a given configuration, we define a *chain of requests* starting from the initiator of a request m. Let u_0 be the node that initiated m; the node before u_0 is its predecessor, u_1. The node before u_1 is u_1's predecessor, *if u_1 has reached it.* The chain ends at a node that does not have a predecessor (yet), or at the node that holds the object.

The *length* of the chain is the number of nodes in it. So, the chain ends at a node whose request is still on its way to its predecessor, or when the node holds the object. In the last case, where the end of the chain holds the object, we say that the chain is *complete.*

Observe that at a configuration, a node appears at most once in a chain, since a node cannot have two outstanding requests at the same time.

For the rest of the proof, we assume that each message takes *at most* one time unit, that is, d hops take at most d time units.

Lemma 5. *A chain is complete within at most $n \cdot D_T$ time units after m is issued.*

Proof. We show, by induction on k, that after $k \cdot D_T$ time units, the length of the chain is either at least k, *or the chain is complete.* Base case, $k = 0$, is obvious. For the induction step, consider the head of the chain. If it holds the object, then the chain is complete and we are done. Otherwise, as it has already issued a request, by Lemma 4, within at most D_T hops, and hence, time units, it finds its predecessor, implying that the length of the chain grows by one.

Since a node appears at most once in a chain, its length, k, can be at most n, and hence, the chain is complete within $n \cdot D_T$ time units. $\square$

Once a chain is complete, the position of r in the queue is fixed, and the requests start waiting.

Assume that the time to execute a request at a node is negligible, and that the object is sent from one node in the chain to its successor within one hop. Thus, within n hops the object arrives at i_0, implying the next theorem.

Theorem 2 (No starvation). *A request is served within* $n \cdot D_T + n$ *time units.*

We now discuss how to modify the proof to accommodate a lookup request r. Lemma 1 (and hence, Lemma 2) does not change since only move requests change the downward paths. For Lemma 3, the path can be changed only if r is passed by m, so r stops once at u'. The move request m that embeds r will not visit u twice, as argued above. Hence, the claim follows. For Lemma 4, if a lookup request is passed at a node u, then a move request m embeds the lookup at a node u', that is, the parent or the child of u, respectively, if m passed the lookup in the up or down phase. Thus, the move request will reach its predecessor, which is also the predecessor of the lookup. A read-only copy of the object will be sent to the lookup before the object is sent to the node that issued the move request that embeds the lookup.

The lookup and move requests can be used to support read and write operations, respectively, and provide a linearizable read/write object [14], in a manner similar to Arrow [9].

5 Constructing an Overlay Tree

Constructing an overlay tree with good stretch is the key to obtaining good performance in Combine. One approach is a direct construction of an overlay tree, in a manner similar to [12]. Another approach is to derive an overlay tree T from any spanning tree ST, without deteriorating the stretch. The rest of this section describes this approach.

Pick a *center* u of ST as the root of the overlay tree. (I.e., a node minimizing the longest hop distance to a leaf node.)

Let k be the level of u in the resulting rooted tree.

By backwards induction, we augment ST with virtual nodes and virtual links to obtain an overlay tree T where all nodes of ST are leaf nodes, without increasing the stretch of the tree. (See Figure 2.) The depth of T, D_T, is k. At level $k-1$ we add to T a duplicate of the root u and create a virtual link between this duplicate and u itself. Then, for every level $\ell < k-1$, we augment level ℓ of the spanning tree with a virtual node for each (virtual or physical) node at level $\ell+1$ and create a virtual link between a node at level ℓ and its duplicate at level $\ell+1$.

To see why the stretch of the overlay tree T is equal to the stretch of the underlying spanning tree ST, note that we do not change the structure of the spanning tree, but augment it with virtual paths consisting of the same node, so that each becomes a leaf. Since the cost of sending a message from a node u to itself is negligible compared with the cost of sending a message from u to any

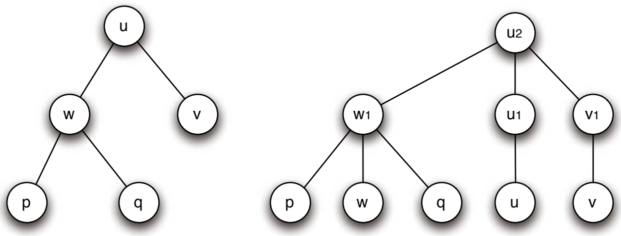


Fig. 2. Deriving an overlay tree from a spanning tree

other node in the system, the cost of these virtual paths (which have at most k hops) is also negligible.

There are constructions of a spanning tree with low stretch, e.g., [10], which can be used to derive an overlay tree with the same stretch.

6 Related Work

We start by discussing directory protocols that are implemented in software, and then turn to hardware protocols.

Arrow [9] is a distributed directory protocol, maintaining a distributed queue, using path reversal. The protocol operates on *spanning tree*, where all nodes (including inner ones) may request the object. Every node holds a pointer to one of its neighbors in the tree, indicating the direction towards the node owning the object; the path formed by all the pointers indicates the location of a *sink* node either holding the object or that is going to own the object. A move request redirects the pointers as it follows this path to find the sink, so the initiator of the request becomes the new sink. In this respect, Arrow and Combine are quite similar; however, the fact that in Combine only leaf nodes request the object, allows to provide a complete and relatively simple proof of the protocol's behavior, including lack of starvation.

The original paper on Arrow analyzes the protocol under the assumption that requests are sequential. Herlihy, Tirthapura and Wattenhofer [13] analyze Arrow assuming concurrent requests in a *one-shot* situation, where all requests arrive together; starvation-freedom is obvious under this assumption. Kuhn and Wattenhofer [17] allow requests at arbitrary times, but assume that the system is synchronous. They provide a competitive analysis of the distance to the predecessor found by a request (relative to an optimal algorithm aware of all requests, including future ones); it seems that this analysis also applies to Combine. The communication cost of Arrow is proportional to the stretch of the spanning tree used. Herlihy, Kuhn, Tirthapura and Wattenhofer [11] merge these works in a complex competitive analysis of Arrow for the asynchronous case. The difference with our analysis is twofold. First, they do not prove starvation freedom yet they analyze latency by assuming that requests are assigned a fixed location in the queue. In contrast, we consider worst-case scenarios where a request finds its predecessor before its predecessor finds its own predecessor, in which the requests order is undefined. Second, they restrict their analysis to exclusive accesses, and it does not handle the shared read-only requests discussed in [9]—the reason is that requests get reordered due to message asynchrony, delaying arbitrarily read-only requests. COMBINE provides shared and exclusive requests and we provide a simple proof that they do not starve despite asynchrony.

More recently, Zhang and Ravindran [23] presented RELAY, a directory protocol that also runs on a spanning tree. In RELAY, pointers lead to the node *currently* holding the object (rather than to a node that is already waiting for the object), and they are changed only after the object moves from one node to another. (This is similar to the tree-based mutual exclusion algorithm of Raymond [21].) When requests are concurrent, they are not queued one after the other, and a request may travel to a distant node currently holding the object, while it ends up obtaining the object from a nearby node (which is going to receive the object first). (See [4].)

BALLISTIC [12] assumes a hierarchical overlay structure, whose leaves are the physical nodes; this structure is similar to the overlay tree used in COMBINE, but it is enriched with *shortcuts*, so that a node has several *parents* in the level above. Requests travel up and down this overlay structure in a manner similar to COMBINE; since requests might be going over parallel links, however, performance may deteriorate due to concurrent requests. It is possible to construct a situation where a request travels to the root of the structure, but ends up obtaining the object from a sibling node (see [4, 22]). Sun's thesis [22] discusses this problem and suggests a variant that integrates a mutual exclusion protocol in each level. The cost of this variant depends on the mutual exclusion algorithm chosen, but it can be quite high since many nodes may participate in a level. Sun's thesis also includes a proof that requests do not starve, but it relies on a strong synchrony assumption, that *all messages on a link incur a fixed delay.*

The next table compares the communication cost of a request by node p, served by node q. In the table, $\delta_{ST}(p,q)$ denotes the (weighted) distance between p and q on a *spanning tree*; while Δ_{ST} is the (weighter) diameter of the spanning tree. The construction of Section 5 implies that they are not better (asymptotically) than the distance and diameter of the overlay tree ($\delta_T(p,q)$ and Δ_T).

Protocol	Cost	Assumes FIFO
COMBINE	$O(\delta_T(p,q))$	No
ARROW	$O(\delta_{ST}(p,q))$	Yes
RELAY	$O(\Delta_{ST})$	Yes
BALLISTIC	$O(\Delta_T)$	Yes

In hardware cache coherent systems, a directory is used to store the memory addresses of all data that are present in the cache of each node. It maintains access coherence by allowing cache hits or by (re)reading a block of data from the memory. In [6], blocks have three states indicating whether they are shared, exclusive or invalid. A block is either invalidated when some specific action may violate coherence or upon receiving an invalidation broadcast message [3]. In addition, the directory can maintain information to restrict the broadcast to affected nodes, as suggested in [2]. Finally, the directory size can be reduced by linking the nodes that maintain a copy of the block [15]. Upon invalidation of a block, a message invalidates successively the caches of all linked nodes.

The design principle of this later approach is similar to the distributed queue that is maintained by our protocol except that we use it for passing exclusive accesses and not for invalidating read-only copies.

7 Discussion

We have proposed Combine to efficiently solve a key challenge of a directory protocol in highly-contended situations: ensuring that all requests eventually get served. Some prior protocols prove that requests do not starve under the assumption that requests execute one by one and / or that communication is synchronous. Others have high communication complexity as they require that conflicting requests run additional mutual exclusion algorithms or because concurrent requests may result in an unbounded number of message exchanges. Our combining technique does not incur any communication overhead to handle concurrency. It can be easily adapted to tolerate unreliable communication, by a traditional retransmission technique based on timeouts and acknowledgements.

Consistency protocols play an important role in *data-flow* distributed implementations of software transactional memory (DTM) in large-scale distributed memory systems [12]. The consistency protocols of existing DTMs [5, 8, 18] seem less suited for large-scale systems than directory-based consistency protocols. They follow a lazy conflict detection strategy either by acquiring a global lock [18] or by broadcasting [5, 8], at commit-time—two techniques that do not scale well when the number of nodes grow.

We leave to future work the interesting question of integrating the directory protocol into a full-fledged distributed transactional memory.

References

[1] Agarwal, A., Chaiken, D., Kranz, D., Kubiatowicz, J., Kurihara, K., Maa, G., Nussbaum, D., Parkin, M., Yeung, D.: The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In: Proceedings of Workshop on Scalable Shared Memory Multiprocessors (1991)

[2] Agarwal, A., Simoni, R., Hennessy, J.L., Horowitz, M.: An evaluation of directory schemes for cache coherence. In: ISCA, pp. 280–289 (1988)

[3] Archibald, J.K., Baer, J.-L.: An economical solution to the cache coherence problem. In: ISCA, pp. 355–362 (1984)

[4] Attiya, H., Gramoli, V., Milani, A.: Combine: An improved directory-based consistency protocol. Technical Report LPD-2010-002, EPFL (2010)

[5] Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP, pp. 247–258 (2008)

[6] Censier, L.M., Feautrier, P.: A new solution to coherence problems in multicache systems. IEEE Trans. on Comp. C-27(12), 1112–1118 (1978)
[7] Chaiken, D., Fields, C., Kurihara, K., Agarwal, A.: Directory-based cache coherence in large-scale multiprocessors. Computer 23(6), 49–58 (1990)
[8] Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: Dependable distributed software transactional memory. In: PRDC, pp. 307–313 (2009)
[9] Demmer, M., Herlihy, M.: The Arrow directory protocol. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 119–133. Springer, Heidelberg (1998)
[10] Emek, Y., Peleg, D.: Approximating minimum max-stretch spanning trees on unweighted graphs. SIAM J. Comput. 38(5), 1761–1781 (2008)
[11] Herlihy, M., Kuhn, F., Tirthapura, S., Wattenhofer, R.: Dynamic analysis of the arrow distributed protocol. Theory of Computing Systems 39(6) (2006)
[12] Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)
[13] Herlihy, M., Tirthapura, S., Wattenhofer, R.: Competitive concurrent distributed queuing. In: PODC, pp. 127–133 (2001)
[14] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Prog. Lang. Syst. 12(3), 463–492 (1990)
[15] James, D.V., Laundrie, A.T., Gjessing, S., Sohi, G.: Scalable coherent interface. Computer 23(6), 74–77 (1990)
[16] Kruskal, C.P., Rudolph, L., Snir, M.: Efficient synchronization of multiprocessors with shared memory. In: PODC, pp. 218–228 (1986)
[17] Kuhn, F., Wattenhofer, R.: Dynamic analysis of the arrow distributed protocol. In: SPAA, pp. 294–301 (2004)
[18] Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: PPoPP, pp. 198–208 (2006)
[19] Nussbaum, D., Agarwal, A.: Scalability of parallel machines. Commun. ACM (March 1991)
[20] Pfister, G.F., Norton, V.A.: "hot spot" contention and combining in multistage interconnection networks. IEEE Trans. on Comp. 34(10), 943–948 (1985)
[21] Raymond, K.: A tree-based algorithm for distributed mutual exclusion. TOCS 7(1), 61–77 (1989)
[22] Sun, Y.: The Ballistic Protocol: Location-aware Distributed Cache Coherence in Metric-Space Networks. PhD thesis, Brown University (May 2006)
[23] Zhang, B., Ravindran, B.: Relay: A cache-coherence protocol for distributed transactional memory. In: OPODIS, pp. 48–53 (2009)

Lightweight Live Migration for High Availability Cluster Service*

Bo Jiang[1], Binoy Ravindran[1], and Changsoo Kim[2]

[1] ECE Dept., Virginia Tech
{bjiang,binoy}@vt.edu
[2] ETRI, Daejeon, South Korea
cskim7@etri.re.kr

Abstract. High availability is a critical feature for service clusters and cloud computing, and is often considered more valuable than performance. One commonly used technique to enhance the availability is live migration, which replicates services based on virtualization technology. However, continuous live migration with checkpointing will introduce significant overhead. In this paper, we present a lightweight live migration (LLM) mechanism to integrate whole-system migration and input replay efforts, which aims at reducing the overhead while providing comparable availability. LLM migrates service requests from network clients at high frequency during the interval of checkpointing system updates. Once a failure happens to the primary machine, the backup machine will continue the service based on the virtual machine image and network inputs at their respective last migration rounds. We implemented LLM based on Xen and compared it with Remus—a state-of-the-art effort that enhances the availability by checkpointing system status updates. Our experimental evaluations show that LLM clearly outperforms Remus in terms of network delay and overhead. For certain types of applications, LLM may also be a better alternative in terms of downtime than Remus. In addition, LLM achieves transaction level consistency like Remus.

1 Introduction

High availability (HA) is a critical feature of modern enterprise-scale data and service clusters. Any downtime that a server cluster experiences may result in severe loss on both revenue and customer loyalty. Therefore, high availability is often considered more valuable than performance [1]. Especially along with the development of cloud computing—one of the most remarkable development opportunities for the Internet—computation and storage are gradually moving from clients to cluster servers in a cloud [2]. Thus the availability of the resources in a cloud is essential to the success of cloud computing. Nowadays, high availability is still a very challenging problem [3], because there are many failure categories to handle so as to guarantee the continuous operation. Among the failure models, hardware fail-stop failure is one of the most commonly studied [4].

* This work was supported by the IT R&D program of MKE/KEIT, South Korea [2007S01602, Development of Cost Effective and Large Scale Global Internet Service Solution].

Naturally, replication is an important approach to increase the availability by providing redundancy—once a failure occurs to a replica, services that run upon it can be taken over by other replicas [5]. Replication may be realized as several redundancy types: spatial redundancy, temporal redundancy, and structural (or contextual) redundancy [6]. For example, service migration used in server clusters [7] provides spatial redundancy, since it requires extra hardware as running space of services.

For any redundancy type, the consistency among multiple replicas needs to be guaranteed, in a certain consistency level. Based on Brewer's CAP theorem [8], the consistency is a competing factor to the availability, i.e., there is a trade-off between them.

By running multiple virtual machines (VM) on a single physical machine, virtualization technology can facilitate the management of services, such as replication via migration. Virtualization technology separates service applications from physical machines with a virtual machine monitor (VMM), thus provides increased flexibility and improved performance [9]. With these advantages, virtualization technology makes it easy to migrate services across physical machines. Usually we call the machine which provides regular services as the *primary machine*, and the one which takes over the services at a failure as the *backup machine*.

To achieve high availability, live migration is typically used to minimize the downtime. Here live migration means executing the migration without suspending the primary machine. Instead, the primary machine keeps running until the migration is completed. Live migration was first studied in [7], where the migration is executed only once and triggered on demand of users. Such a one-time live migration is suitable for data processing or management purposes. However, it does not work for disaster recovery because the migration cannot be triggered by a failure event.

In [10], the authors introduced the idea of checkpointing to live migration by presenting Remus—a periodical live migration process for disaster recovery. Using checkpointing, the primary machine keeps migrating a whole system, including CPU/memory status updates as well as writes to the file system to the backup machine at configured frequency. Once a failure happens so that the migration data stream is broken, the backup machine will take over the service immediately starting from the latest stop point of checkpointing. However, checkpointing at high frequency will introduce significant overhead, as plenty of resources such as CPU and memory are consumed by the migration. In this case clients that request services may experience significantly long delays. If on the contrary the migration runs at low frequency trying to reduce the overhead, there maybe many service requests that are duplicately served. Actually this will produce the same effect of increasing the downtime from the perspective of those new requests that come after the duplicately served requests.

In fact, there is another approach for service replication—input replay [11]. With input replay, the data to replicate will be much less than whole-system replication. Although input replay cannot replicate the system status exactly, such a Point-in-Time consistency is actually very challenging equally for all the replication approaches in real implementations [12].

Based on input replay, the objective of this paper is to reduce the overhead of whole-system checkpointing when achieving comparable downtime and the same level consistency as those of Remus. In this way, we will be able to leverage the advantage of input replay without suffering from its flaw on consistency.

Based on the checkpointing approach of Remus, we developed an integrated live migration mechanism, called Lightweight Live Migration (LLM), which consists of both whole-system checkpointing and input replay. The basic idea is as follows:

1) The primary machine keeps migrating to the backup machine: a) the guest VM image (including CPU/memory status updates and new writes to the file system) at low frequency; and b) service requests from network clients at high frequency; and

2) Once a failure happens to the primary machine, the backup machine will continue the service based on the guest VM image and network inputs at their respective last migration rounds.

Especially when the network service involves a lot of computation or database updates, CPU/memory status updates and writes to the file system will be a big bulk of data. Compared with past efforts such as Remus, migrating the guest VM image at low frequency with input replay as an auxiliary may significantly reduce the migration overhead.

We compared LLM with Remus in terms of the following metrics: 1) downtime, which demonstrates the availability; 2) network delay, which reflects the client experience; and 3) overhead, which is measured with kernel compilation time. The experimental evaluations show that LLM sharply reduces the overhead of whole-system checkpointing and network delay on the client side compared with Remus. In addition, LLM demonstrates a downtime that is comparable, or even better for certain type of applications, to that of Remus. We also analyzed that LLM achieves transaction level consistency, which is the same as Remus.

The paper makes the following contributions:

1) We integrate the idea of input replay with whole-system checkpointing mechanism. Such an integrated effort outperforms the existing work with a single effort of checkpointing, especially for applications with intensive network workload;

2) LLM migrates the service requests from the primary machine independently, instead of depending on a special load balancer hardware. This means we can apply LLM more generally in practical use; and

3) We developed a fully functional prototype for LLM, which can be used as a basis for further research and practical application.

The rest of the paper is organized as follows. Related work is discussed in Section 2. In Section 3, we describe the system model and assumptions. We then introduce the design and implementation of LLM in Section 4. In Section 5, we report our experiment environment, benchmarks and the evaluation results. In Section 6, we finally conclude and discuss the future work.

2 Related Work

In [13], high availability is defined as a system design protocol and associated implementation that ensures a certain degree of operational continuity during a given measurement period. Based on this definition, the availability may be estimated with a percentage of continuously operation time in a year, also known as “X nines” (for example “five nines”, i.e., 99.999%) [6]. In fact, this percentage is determined by two factors—the number of failure events and the downtime of each failure event. We mainly consider the downtime at a failure in this paper.

State migration was studied in many literatures such as [14,15]. For Xen [16], live migration was added in [7] to reduce the downtime thereby increasing the availability. However, this one-time migration effort is not suitable for disaster recovery, which requires frequent checkpointing protection. The wide-area live migration was also studied in [17]. But it is out of the scope of this paper. We only study the live migration locally in a cluster.

Checkpointing is a commonly used approach for fault tolerance. The idea of checkpointing was introduced to live migration by Cully *et al.* in Remus [10]. Remus is a remarkable effort on live migration, which aims to handle hardware fail-stop failure on a single host with whole-system migration. By checkpointing the status updates of a whole system, Remus can achieve generality, transparency, and seamless failure recovery. It is designed to use pipelined checkpoints, which means the active VM is bounded by short pauses, in each of which the state change is quickly migrated to the backup machine. Moreover, both memory and CPU state backup and network/disk buffering were carefully designed based on live migration [7] and Xen’s intrinsic services. In general, Remus is a practical effort based on Xen, and most of its functions have already been merged into Xen.

In terms of shortcomings, though the downtime of Remus can be controlled within one second, it experiences about 50% performance penalty such as on network delay and CPU execution time. This penalty comes from data migration at high frequency—it supports up to 40 times of migration per second. If we decrease the frequency, the backup machine may serve a lot of service requests that have already been served by the primary machine.

Input replay is also a commonly studied approach for high availability. In [11], Bressoud *et al.* provided fault tolerance by forwarding the input events and deterministically replay them. Another example is ReVirt [18], in which VM logging and replay were discussed yet for intrusion analysis instead of high availability. However like Remus, these efforts also involve a single approach only. On the contrary, we integrate the efforts of both checkpointing and input replay to provide high availability with reduced overhead. Manetho is also an integrated effort including both rollback-recovery and process replication [19]. However, Manetho requires the clients of replicated services to participate in the recovery protocol, which will result in the “visibility” of a failure to the clients. LLM, unlike Manetho, can fully leverage the fault tolerance of Internet so as to completely hide the migration as well as the recovery process from clients.

Xen [16] is an open source virtual machine monitor under the GPL2 license, which makes it very flexible for the purpose of research. We evaluated LLM on the platform of Xen.

3 System Model

In this paper, we discuss a whole-system replication. Therefore, for each primary machine, we assume there is a backup machine as the replica, and there is a high-speed network connection between the two machines. We also assume that the primary machine and the backup machine share a single storage, so that we do not have to migrate the whole file system.

We only consider hardware fail-stop failure model. Fail-stop failure makes the following assumptions: 1) any correct server can detect whether any other server has failed; and 2) every server employs a stable storage which reflects the last correct service state of the crashed server. This stable storage can be read by other servers, even if the owner of the storage has crashed.

We implemented LLM based on the existing codes of Remus, which was developed on Xen [16]. With Xen, network services run in guest virtual machines (called domain U or domU in Xen terminology). There is a unique VM for management purpose, called domain 0 or dom0, which has direct access to all physical hardware. Therefore service requests go through the back-end driver (called netback) in dom0 first, and then are distributed to the front-end driver (called netfront) in a specific domU. This will facilitate our network buffer management and migration.

We do not make any assumptions about the load balancer in a cluster, since a load balancer is a special hardware which is out of scope of this paper. This means LLM migrates the service requests from the primary machine independently. Moreover, since we do not consider the application-level migration, we do not distinguish the services running on a single guest virtual machine. All the migrated service requests are managed in the same manner.

Finally, we assume that there is an algorithm to map each egress response (i.e., a response from the server to clients) packet to a specific ingress request (i.e., a request from clients to a server) packet. A straight-forward approach is to append a sequence number for each ingress request packet and egress response packet, and keep this sequence number during the service. Hence, it is easy to pare up the two types of packets and map them accordingly.

4 Design and Implementation

We design the implementation architecture of LLM as shown in Figure 1. Beyond Remus, we also migrate the change in network driver buffers. The entire process works as follows:

1) First, on the primary machine, we setup the mapping between the ingress buffer and the egress buffer, signifying which packets are generated corresponding to which service request(s), and which requests are yet to be served. Moreover, LLM hooks a copy for each ingress service request.

2) Second, at each migration pause, LLM migrates the hooked copy as well as the boundary information to the backup machine asynchronously, using the same migration socket as the one used by Remus for CPU/memory status updates and writes to the file system.

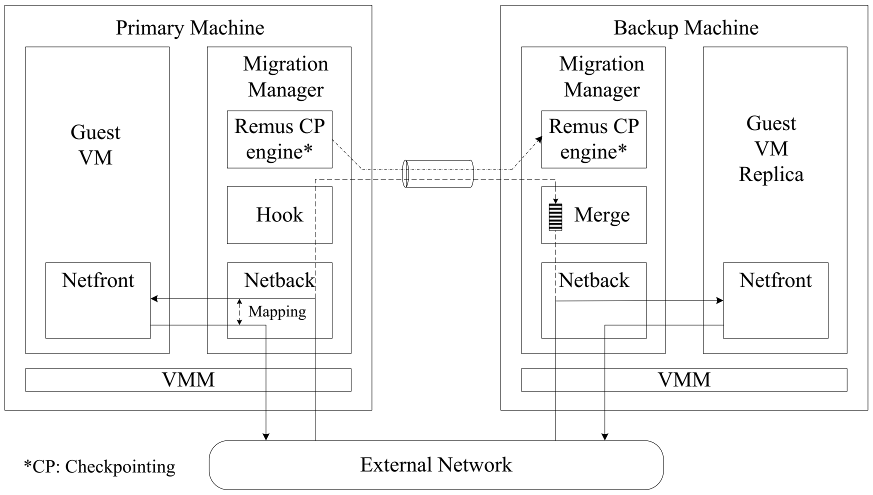


Fig. 1. LLM Architecture

3) Third, all the migrated service requests are buffered in a queue in the "merge" module. Those buffered requests that have been served will be removed based on the migrated boundary information. Once a failure occurs on the primary machine that breaks the migration data stream, the backup machine recovers the migrated memory image and merges the service requests into the corresponding driver buffers.

In Figure 1, solid lines represent the regular input/output of network packets, dash-dotted lines show the migration of system status updates, and dashed lines mean the migration of network buffers.

Next, we will first introduce LLM's release of egress responses, analyze the consistency, then discuss the function modules in separate subsections: 1) egress response release and consistency analysis in Section 4.1; 2) mapping and hooking of services in Section 4.2; 3) asynchronous migration, especially its time sequence, in Section 4.3; and 4) buffering and merging of requests in Section 4.4.

4.1 Egress Response Release and Consistency Analysis

Unlike the block/commit case used by Remus, LLM releases the response packets immediately after they are generated on VMs. In the block/commit case, all the outputs are blocked using IMQ [20] until the migration in a checkpointing epoch is acknowledged. This can avoid losing any externally visible state, which helps to maintain the consistency. However, at low checkpointing frequency, network clients may experience very long delays with the block/commit mechanism of Remus.

In fact, the immediate release case and the block/commit case can achieve the same operation correctness on the client side, and the same consistency level on the server side. First on the client side: 1) for the block/commit case, there won't be any duplicated response packets. However, network clients may re-transmit requests after the timer expires; 2) for the immediate release case, on the contrary, there maybe duplicated response packets. Nevertheless, the re-transmission of service requests won't increment because of the increased delay. Given the fault tolerance of Internet, either the duplication or the disorder of both service requests and responses, which are introduced by suspended service or multiprocessing, will be handled correctly. Although the performance that a client experiences in two cases may be different, the correctness is guaranteed and transparent to the clients.

Then on the server side, LLM achieves the same level of consistency, i.e., transactional consistency, as that of Remus. In Remus, all the service to request packets in the speculative execution phases will be re-transmitted by the client and re-served by the backup machine. In fact, this is equivalent to input replay, and the backup machine will have to recover the status at the failure point by replaying these re-transmitted requests. Though LLM migrates the requests directly, it recovers the status at the failure point in the same fashion. The only difference in consistency of LLM from Remus is the workload of input replay: as LLM usually runs at low frequency, it produces more requests to replay.

Therefore with the immediate release of egress response packets, LLM achieves the same operation correctness on the client side, and the same consistency level on the server side.

4.2 Hooking and Mapping of Service Requests

Without a special load balancer hardware, LLM has to migrate the service requests itself. Obviously, the first step of this migration is to make a copy.

Linux, which Xen is built on, provides a netfilter system consisting of a series of hooks in various points in a protocol stack. Figure 2 shows the netfilter system in the IPv4 protocol stack. This system makes it easy to copy or filter network packets to and from a specific guest VM [21].

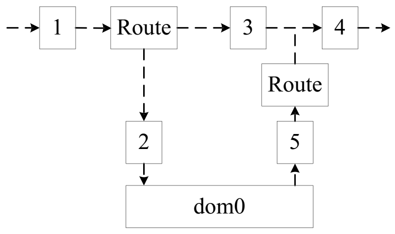


Fig. 2. Netfilter System

There is a netback driver in domain 0 of Xen, which is responsible for routing the packets to and from the guest VM. Domain "U"s are considered as external network devices. Thus, all the packets to and from guest VMs are routed through the path from 1 to 3 to 4 in the figure. Along this path, we choose point 3—NF_IP_FORWARD—to hook both ingress requests and egress responses.

We implemented this hook function module in two parts: 1) a hook module in the kernel that copies *sk_buff* and sends it up to the user space, and 2) a separate thread in the user space that receives copies and analyzes (for egress responses) or write them into the migration buffer (for ingress requests).

In Linux kernel, network packets are managed using *sk_buff*. The information in the *sk_buff* header is specific to the local host. Therefore we only copy the contents between the head pointer and the tail pointer. To recognize the packet header offsets when the *sk_buff* header is absent, we append a metadata in front of the packet content, which includes the header offsets of each layer as well as the content length. This metadata will help the backup machine to create a new *sk_buff* header.

LLM manages a mapping table in the user space on the primary machine. For each hooked ingress request, we append an entry in the mapping table including 1) a sequence number, 2) a completion flag, and 3) a pointer to memory in the migration buffer. For each entry, the sequence number helps to distinguish requests from each other, and setting the completion flag as "True" means the service for this request has been completed. Then for each hooked egress response packet, we decide which request packet should be matched with using the algorithm that we assumed. Then we will set the completion flag of this request packet in the mapping table as appropriate.

4.3 Asynchronous Network Buffer Migration

Checkpointing was used to migrate the ever-changing updates of CPU/memory/disk to the backup machine by Remus. Only at the beginning of each checkpointing cycle, the migration occurs in a burst mode after the guest virtual machine resumes. Most of the time, there is no traffic flowing through the network connection between the primary machine and the backup machine. During this interval, we can migrate the service requests at higher frequency than that of checkpointing.

Like the migration of CPU/memory/disk updates, the migration of service requests is also in an asynchronous manner, i.e., the primary machine can resume its service without waiting for the acknowledgement from the backup machine.

Figure 3 shows the time sequence of migrating the checkpointed resources and the incoming service requests at different frequencies on a single network socket. The entire sequence within an epoch is described as follows:

1) The dashed blocks represent the suspension period when the guest virtual machine is paused. During this suspension period, all the status updates of CPU/memory/disk are collected and stored in a migration buffer.

2) Once the guest VM is resumed, the content stored in the migration buffer is migrated first (shown as a block shaded area that is adjacent to the dashed area in the figure).

3) Then, the network buffer migration starts at high frequency until the guest VM is suspended again. At the end of each network buffer migration cycle (the thin, shaded strips in the figure), LLM transmits two boundary sequence numbers for the moment: one is for the first service request in the current checkpointing period, and the other is for the first service request that has a "False" completion flag. All the services after the first boundary need to be replayed on the backup machine for consistency, but only

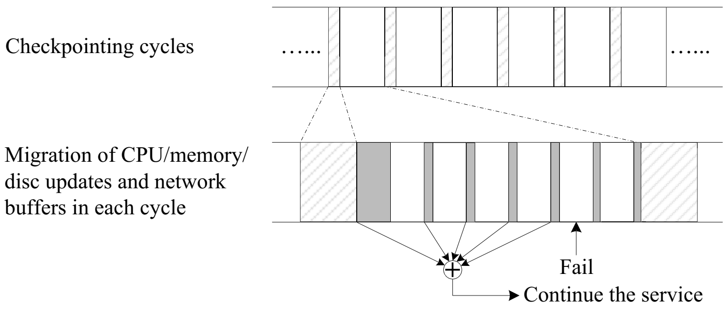

Checkpointing cycles
Migration of CPU/memory/ disc updates and network buffers in each cycle
Fail
Continue the service

Fig. 3. Checkpointing Sequence

those after the second boundary need to be responded to the clients. If there is no new requests, LLM transmits the boundary sequence numbers only.

Anytime a failure happens to a primary machine, the backup machine will 1) continue the execution of VM from the latest checkpointed status; 2) replay the requests after the first boundary to achieve consistency; and 3) respond to those un-responded requests after the second boundary. This recovery process is shown with the "+" combination signal in the figure.

4.4 Buffering and Merging of Requests

The migrated service requests are first stored in a queue (implemented with a double linked list as shown in Figure 1) in the user space on the backup machine. With this double linked list, the storage can be allocated dynamically and the complexity of insertion at tail and removal from head will be constant.

Everytime when a network buffer migration burst arrives, the backup machine will first enqueue the incoming new requests at the tail, then dequeue and free requests from the head until the one with the first boundary sequence number. In this way, the queue only stores those needed to recover the system state. These requests will not be released to the kernel space until the migration data stream is broken.

As on the primary machine, there is also a module running in the kernel space on the backup machine. This module is responsible for creating a *sk_buff* header based on the metadata and inserting the requests into the queue in the kernel space. As shown in Figure 2, the requests will also be inserted to point 3, i.e., NF_IP_FORWARD. In this way, the protocol stack in the kernel will be able to recognize these migrated requests, just like local ingress request packets.

5 Evaluation

We evaluated LLM and compared it with Remus in terms of its correctness, downtime, delay for clients, and overhead under various checkpointing periods. The downtime is the primary factor to estimate the availability of a cluster, whereas network delay mainly represents clients' experience. Finally, the overhead must be considered in the picture so that the effectiveness of the service will not be overly compromised by checkpointing.

5.1 Experiment Environment

The hardware experiment environment included two machines (one as primary and the other as backup), each with an IA32 architecture processor and a 3 GB RAM. We set up a 100 Mbps network connection between the two machines specifically used for migration. In addition, we used a third PC as a network client to transmit service requests and examine the results based on responses.

As for the software environment, we built Xen from source which was downloaded from its unstable tree [22], and let all the protected virtual machines run PV (i.e., paravirtualization) guests with Linux 2.6.18. We also downloaded Remus version 0.9 codes from [23]. Then we allocated 256 MB RAM for each guest virtual machine, the file system of which is an image file of 3 GB shared by two machines using NFS (i.e., Network File System).

5.2 Benchmarks and Measurements

We utilized three network application examples to evaluate the downtime, network delay and overhead of LLM and Remus:

1) Example 1 (HighNet)—The first example is flood ping [24] with the interval of 0.01 second, and there is no significant computation task running on domain U. In this case, the network load will be extremely high, but the system updates are not significant. We named it "HighNet" to signify the intensity of network load.

2) Example 2 (HighSys)—In the second example, we designed a simple application to taint 200 pages (4 KB per page on our platform) per second, and there are no service requests from external clients. Therefore, this example involves a lot of computation workload on domain U. The name "HighSys" reflects its intensity on system updates.

3) Example 3 (Kernel Compilation)—We used kernel compilation as the third example which involves all the components in a system, including CPU/memory/disk updates. As part of Xen, we used Linux kernel 2.6.18 directly. Given the limited resource on domain U, we cut the configuration to a small subset in order to reduce the time required to run each experiment.

Here example 1 and 2 represent opposite types of network application, whereas example 3 is a typical application type entailing almost all aspects of system workload.

We measured the downtime and network delay under example 1 and 2, and the overhead under example 3. The details of each measurement are described below.

The downtime and network delay were measured using ping program on the client side, and the key index we selected here is the round-trip time of ping packets. We believe it makes more sense to measure on the client side since the client experience during a downtime is what actually matters. The flood ping used by example 1 is in itself a way of measurement. Yet for example 2, since it does not involve network activities, we have to use ping as an additional measure. To avoid the extra migration load, we increased the ping interval to 0.1 second and disabled the hooking function of

LLM for ping packets. For each test case, we stopped the ping program after breaking the migration data stream. Then, in each ping program log file, we record the last peak value of the round-trip time as the downtime, since it represents the delay of the first ping packet at the beginning of the disruption, therefore reflects the wait time of network clients. Lastly, we calculated the average value of round-trip times in a checkpoint period as the network delay.

Though there is no response for ping requests before the VM that fails is recovered completely, we are still able to guarantee that no ping packets are lost during the downtime. This is based on the configurable timer that ping program provides. As long as this timer does not expire, ping program will wait for the response (the transmission of following requests will not be influenced) without acknowledging a ping failure. In the experiments, we configured this timer long enough regarding the downtime that we may experience, so that each ping request will be responded, sooner or later. In this way, the response with the longest round-trip time could be used to estimate the downtime.

The overhead was measured using the incremental time (as a percentage) of kernel compilation. Specifically, the baseline, i.e., a 0% overhead, is the kernel compilation time without checkpointing, whereas a 100% overhead, for example, stands for a doubling of kernel compilation time when checkpointing exists. We measured kernel compilation time using a stop watch, so that it includes both the execution time and suspension time of domain U.

Finally, we measured the performance and the overhead under various checkpointing periods. For Remus, the checkpointing period is the time interval of system updates migration, whereas for LLM, the checkpointing period represents the interval of network buffer migration. By configuring the same value for the checkpointing frequency of Remus and the network buffer frequency of LLM, we are able to guarantee the fairness of the comparison to the greatest extent. Furthermore, we executed our experiments starting from one second of checkpointing period for two reasons. One is that the network connection specifically between the primary and backup machines has limited bandwidth in our experiment environment, thus will increase the migration time in each period. The other is that the timer in the existing migration implementation used by Remus and Xen is still under experiment. Therefore, at high checkpointing frequency, the actual checkpointing period is highly likely to exceed what we configured, thus it does not make sense to set checkpointing period of less than one second in the experiments.

5.3 Evaluation Results

First, we verified the correctness of LLM using two approaches:

1) We verified that the flood ping can be served continuously by the VM which is taken over by the backup machine after a failure occurs. Moreover, we carefully examined the sequence numbers, and observed that there was no disruption in the ping flood; and

2) We verified that the kernel image, which was compiled in domU on the primary machine during the migration process, can be successfully installed and booted in domU on the backup machine after being migrated.

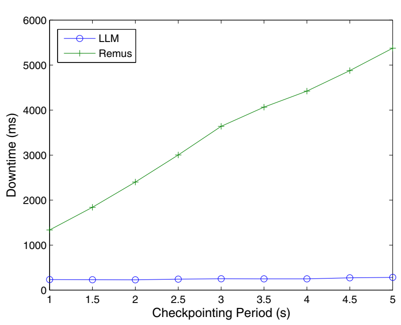


Fig. 4. Downtime under HighNet

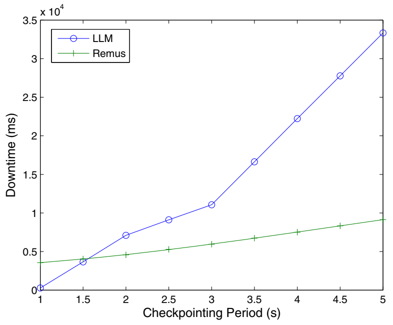


Fig. 5. Downtime under HighSys

These two approaches can fully prove that LLM functions correctly after the migration.

Secondly, we measured the downtime under HighNet and HighSys, the results of which are shown in Figures 4 and 5.

We observe that under HighSys, LLM demonstrates a downtime that is longer than, yet comparable to, that of Remus. The reason is that LLM runs at low frequency, hence the migration traffic in each period will be higher than that of Remus. Under HighNet, the downtime of LLM and Remus show a reverse relationship where LLM outperforms Remus. This is because, from the client side, there are too many duplicated packets to be served again by the backup machine in Remus. In LLM, on the contrary, the primary machine migrates the request packets as well as boundaries to the backup machine, i.e., only those packets yet to be served will be served by the backup. Thus the client does not need to re-transmit the requests, therefore will experience a much shorter downtime.

Thirdly, we evaluated the network delay under HighNet and HighSys as shown in Figures 6 and 7. In both cases, we observe that LLM significantly reduces the network delay by removing the egress queue management and releasing responses immediately.

In Figures 6 and 7, we only recorded the average network delay in a migration period. Next, we show the details of the network delay in a specific migration period in Figure 8, in which the interval between two adjacent peak values represents one migration period.

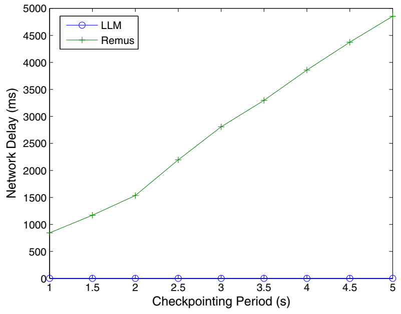


Fig. 6. Network Delay under HighNet

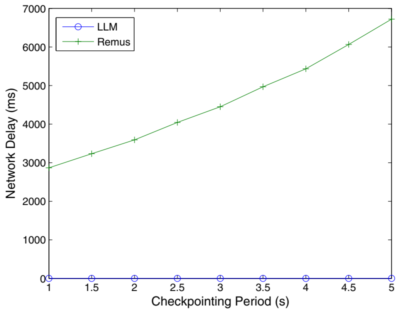


Fig. 7. Network Delay under HighSys

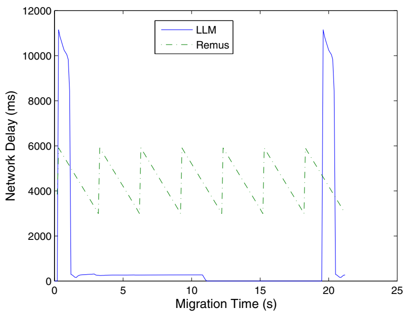


Fig. 8. Detailed Network Delay

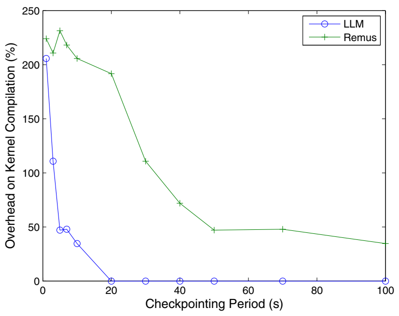


Fig. 9. Overhead under Kernel Compilation

We observe that the network delay of Remus decreases linearly within a period but remains at a plateau. In LLM, on the contrary, the network delay is very high at the beginning of a period, then quickly decrease to nearly zero after a system update is over. Therefore, most of the time, LLM demonstrates a much shorter network delay than Remus.

Finally, Figure 9 shows the overhead under kernel compilation. Actually, the overhead significantly changes only in the checkpointing period interval of $[1,60]$ seconds, as shown in the figure. For checkpointing with shorter periods, the migration of system updates may last longer than a configured checkpointing period, therefore the kernel compilation time for these cases are almost the same with minor fluctuation. For checkpointing with longer periods, especially when it is longer than the baseline (i.e., kernel compilation without any checkpointing), a VM suspension may or may not occur during one compilation process. Therefore, the kernel compilation time will be very close to the baseline, meaning a zero percent overhead. Right in this interval, LLM's overhead due to the suspension of domain U is significantly lower than that of Remus, as it runs at much lower frequency than Remus.

In the experiments above (except for the specific sample shown in Figure 8), each data point was averaged from five sample values. The reason that we did not provide standard deviations is that the sample values remain very stable for given application examples. Generally the standard deviation is less than 5% of the mean value.

In summary, LLM clearly outperforms Remus in terms of network delay and overhead. For certain types of applications, LLM may also be a better alternative in terms of downtime than Remus.

6 Conclusions

In this paper, we designed an integrated live migration mechanism consisting of both whole-system checkpointing and input replay. LLM achieves transactional consistency, which is in the same level as Remus. From the experimental evaluations, we observed that LLM can successfully reduce the overhead of whole-system checkpointing and network delay on the client side while providing comparable downtime. Especially for

HighNet-like application types with intensive network workload, LLM demonstrates an even shorter downtime than the state-of-the-art efforts. As most services that require high availability usually involve a lot of network activities, LLM will be more efficient than other high availability approaches.

Finally, we want to provide some thoughts and clarifications for further discussion in this topic, namely, load balancer and multiple backup:

1) Load balancer—We did not depend on a special load balancer hardware to migrate the requests. If we do, a load balancer may duplicate the ingress request packets at the gateway, and distribute them to the primary and backup machines at the same time. In this way, what we need to migrate besides the system updates will simply be the boundary information. However, since the network buffer migration happens in the interval between system updates migration, the savings from the migration traffic may be negligible compared to the required investment on a load balancer.

2) Multiple backup—There are many scenarios for multiple backup, which involves the internal architecture of clusters. This is out of scope of this paper, and that is why we did not evaluate it in the experiment. If there is need to support multiple backup, we expect the changes to the prototype will not be significant.

The directions of future work include: 1) evaluate the impact of LLM on the consistency; and 2) compare LLM's performance and overhead in an environment with a load balancer.

References

1. Kopper, K.: The Linux Enterprise Cluster: build a highly available cluster with commodity hardware and free software. No Starch Press (2004)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Technical report (2009)
3. Blake, V.: Five nines: A telecom myth. Communications Technology (2009)
4. Poledna, S.: Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism. Kluwer Academic Publishers, Dordrecht (1996)
5. Mullender, S.: Distributed Systems. Addison Wesley Publishing Company, Reading (1993)
6. Carwardine, J.: Providing open architecture high availability solutions. HA forum (2005)
7. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: NSDI 2005: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation, pp. 273–286. USENIX Association, Berkeley (2005)
8. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
9. Mergen, M.F., Uhlig, V., Krieger, O., Xenidis, J.: Virtualization for high-performance computing. SIGOPS Oper. Syst. Rev. 40(2), 8–11 (2006)
10. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: high availability via asynchronous virtual machine replication. In: NSDI 2008: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, pp. 161–174. USENIX Association (2008)
11. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. In: SOSP 1995: Proceedings of the fifteenth ACM symposium on Operating systems principles, pp. 1–11. ACM, New York (1995)

12. Aguilera, M.K., Spence, S., Veitch, A.: Olive: distributed point-in-time branching storage for real systems. In: NSDI 2006: Proceedings of the 3rd conference on Networked Systems Design & Implementation, Berkeley, CA, USA, pp. 27–27 (2006)
13. Hawkins, M., Piedad, F.: High Availability: Design, Techniques and Processes. Prentice Hall PTR, Upper Saddle River (2000)
14. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD 1996: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, pp. 173–182. ACM, New York (1996)
15. Milojíčić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Comput. Surv. 32(3), 241–299 (2000)
16. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP 2003: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 164–177. ACM, New York (2003)
17. Bradford, R., Kotsovinos, E., Feldmann, A., Schiöberg, H.: Live wide-area migration of virtual machines including local persistent state. In: VEE 2007: Proceedings of the 3rd international conference on Virtual execution environments, pp. 169–179. ACM, New York (2007)
18. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: Revirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. 36(SI), 211–224 (2002)
19. Elnozahy, E.N.: Manetho: fault tolerance in distributed systems using rollback-recovery and process replication. PhD thesis, Houston, TX, USA, Chairman-Zwaenepoel, Willy (1994)
20. Mchardy, P.: Linux imq, `http://www.linuximq.net/`
21. Russell, R., Welte, H.: Linux netfilter hacking howto, `http://www.iptables.org/documentation/HOWTO/netfilter-hacking-HOWTO.html`
22. Xen Community: Xen unstable source, `http://xenbits.xensource.com/xen-unstable.hg`
23. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus source code, `http://dsg.cs.ubc.ca/remus/`
24. Stevens, W.R.: TCP/IP illustrated. The protocols, vol. 1. Addison-Wesley Longman Publishing Co., Inc., Boston (1993)

Approximation of δ-Timeliness[*]

Carole Delporte-Gallet[1], Stéphane Devismes[2], and Hugues Fauconnier[1]

[1] Université Paris Diderot, LIAFA
{Carole.Delporte,Hugues.Fauconnier}@liafa.jussieu.fr
[2] Université Joseph Fourier, Grenoble I, VERIMAG UMR 5104
Stephane.Devismes@imag.fr

Abstract. In asynchronous message-passing distributed systems prone to process crashes, a communication link is said δ-*timely* if the communication delay on this link is bounded by some constant δ. We study here in which way processes may approximate and find structural properties based on δ-timeliness (*e.g.*, find δ-timely paths between processes or build a ring between correct processes using only δ-timely links).

To that end, we define a notion of approximation of predicates. Then, with help of such approximations, we give a general algorithm that enables to choose and eventually agree on one of these predicates. Finally, applying this approach to δ-timeliness, we give conditions and algorithms to approximate δ-timeliness and dynamically find structural properties using δ-timeliness.

1 Introduction

Assume an asynchronous message-passing system prone to process crash failures and consider the following problem: we know that after some unknown time there is at least one path from process p to process q such that every message sent along this path arrives to process q by δ time units (such a path is said δ-timely). Now, how can we determine one of these paths or at least one path that is Δ-timely for a Δ close to δ? By "determine" we mean that eventually all processes agree on the chosen path.

To that end, the processes must be at least able to test the δ-timeliness of paths and one of the contribution of this paper is to give some necessary and sufficient conditions to do this. In particular, we prove that without synchronized clocks, the system has to ensure strong synchrony properties in the sense that there must not only exist δ-timely paths from p to q but also from q to p.

The δ-timeliness of a path is a property that is only eventually ensured. Moreover, δ-timeliness of a link can only be approximated by processes. We would like that an approximation algorithm outputs boolean values in such a way that if the path is truly δ-timely, then eventually only true is output, and if the path is not δ-timely, then false is output infinitely often. However as we will see, such approximations of δ-timeliness are generally too strong because there is some incertitude on the time it takes to go from p to q on the path. Therefore the approximation algorithm only ensures that if the path is δ-timely, then true is eventually output forever and if the path is not Δ-timely (with

[*] This work has been supported in part by the ANR project *SHAMAN* and the INRIA project *GANG*.

$\Delta > \delta$), then false is output infinitely often. Hence, between δ and Δ the approximation algorithm may output true or false just as well.

The existence of such approximations enables to answer to our initial problem (but with the little restriction on δ and Δ): if at least one δ-timely path exists from p to q, it is then possible to ensure that all processes eventually agree on the same Δ-timely path from p to q. An algorithm which ensures that all correct processes eventually agree on the same structure verifying some properties is called an *extraction* algorithm [7]. Many other problems can be solved in the same way: for example extraction of a tree containing all correct processes whose all paths from the root are Δ-timely, or a ring containing all correct processes and whose links are Δ-timely, etc.

Actually, all these aforementioned algorithms use methods very similar to the ones used, for example, in the eventual leader election problem in [1–4, 9, 11]. In fact, we prove here that this approach can be expressed in a very general framework. Approximation of δ-timeliness and extraction of some structures based on the δ-timeliness relation are special cases of the more general problem of approximating properties on runs and extracting structures based on these properties. Instead of δ-timeliness, one can consider more general properties on runs and consider when they are eventually true forever. Then, assuming approximation algorithms for these predicates, it is possible to extract (*i. e.* choose and agree) one of these predicates that is true in the run. More precisely, as for δ-timeliness, the approximation is defined by pairs of predicates (P, Q), where P specifies when the approximation has to eventually output true forever and $\neg Q$ when the approximation has not to eventually output true forever (in other words has to output infinitely often false). Then, given a set of pairs of predicates (P_i, Q_i) indexed by some set I, the extraction algorithm, assuming that at least one P_i is true, will converge to some i_0 for all correct processes such that Q_{i_0} is true. This generalization enables to have the same algorithms for many different problems.

In this way, the extraction of structures based on the δ-timeliness relation is only a special case of this general case. For example, assuming that processes have a trusting mechanism (like a failure detector) giving to each processes lists of process supposed to be alive, and consider the predicate "p is eventually in all lists", then the extraction algorithm gives one of such processes. Assuming that these lists are given by failure detector $\Diamond\mathcal{S}$ ([5]) then the extraction algorithm gives an implementation of failure detector Ω and the algorithm is rather close to algorithm of [6].

Contributions. In this paper, we first define a general framework in which processes may approximate predicates on runs and give a generic extraction algorithm that enables processes to converge on one of the predicates satisfied in the run. Then, we apply these concepts by proposing algorithms and impossibility results about the approximation of δ-timeliness. In particular we give sufficient conditions to approximate δ-timeliness on links. More precisely, we prove that we need either to have perfectly synchronized clocks or to assume very strong timeliness requirements. Finally, we give examples of general extractions based on approximation of link δ-timeliness. These illustrations emphasizes the two mains points of our contribution. Firstly, this general approach allows to drastically simplify the design of algorithm. Indeed, from a simple algorithm that approximates a local predicate, like "a link is δ-timely", we can easily derive an algorithm to extract a more complex structure such as a δ-timely path or a

δ-timely ring. Secondly, our algorithms have practical applications, as they can be used to find efficient routes in a network (that is, δ-timely paths), or efficient overlay such as δ-timely rings.

Related works. Many works [1–4, 9, 11] on eventual leader election or Ω implementations uses the same techniques as here. This paper may be also seen as a formalization and abstraction of these techniques. To the best of our knowledge [7] was the first one to study timeliness for itself and not as a mean to get information about process crashes. Note that as link timeliness only means that the communication delay of the link is bounded, its interest is mainly theoretical.

Roadmap. In Section 2 we present the model. Section 3 gives the basic definitions for approximations of predicates and the general extraction algorithm. Section 4 gives conditions for the existence algorithms for approximation of δ-timeliness. In Section 5 we give some examples of applications of extraction algorithms for δ-timeliness.

2 Model

Processes. We consider distributed systems composed of n processes that communicate by message-passing through directed links. We denote the set of processes by $\Pi = \{p_1, ..., p_n\}$. We assume that the communication graph is complete, *i.e.*, for each pair of distinct processes p, q, there is a directed link from p to q, denoted by (p, q). A process may fail by crashing, in which case it definitely stops its local algorithm. A process that never crashes is said to be *correct*, *faulty* otherwise.

We assume the existence of a discrete global clock to which processes cannot access. The range $\mathcal{T}$ of the clock's ticks is the set of natural integers. All correct processes p are *synchronous*, *i.e.*, they are able to take step at each clock tick. So, they can accurately measure time intervals.

As processes can accurately measure time intervals, our algorithms can use *local timers*. A process p starts a timer by setting `settimer`(F) to a positive value. F is a flag that identifies the timer. The timer F is then decremented until it expires. When the timer expires, `timerexpire`(F) becomes $true$. Note that a timer can be restarted (by setting `settimer`(F) to some positive value) before it expires. Finally, `unsettimer`(F) allows to disable timer F.

Communication pattern. A *communication pattern* CP is a function from $\mathcal{T} \times \Pi \times \Pi$ to $\mathbb{N}$: $CP(\tau, p, q) = k$ means that if process p sends a message to process q at time τ then the message is received by process q at time $\tau + k$.

Communication between processes is assumed to be *reliable* and *FIFO*. *Reliable* means that (1) every message sent through a link (p, q) is eventually received by q if q is correct, also (2) if a message m from p is received by q, then m is received by q at most once and only if p previously sent m to q. *FIFO* means that messages from p to q are received in the order they are sent. The links being reliable, an implementation of *reliable broadcast* [8] is possible and in the following reliable broadcast is defined by two primitives: `rbroadcast`$\langle\rangle$ and `rdeliver`$\langle\rangle$.

Runs. An algorithm $\mathcal{A}$ consists of n deterministic (infinite) automata, one per process. The execution of an algorithm $\mathcal{A}$ proceeds as a sequence of process *steps*. Each process performs its steps atomically. During a step, a process may send and/or receive some messages and changes its state.

A run R of algorithm $\mathcal{A}$ is a tuple $R = \langle CP, I, E, S, F\rangle$ where CP is a communication pattern, I is the initial state of the processes in Π, E is an infinite sequence of steps of $\mathcal{A}$, S is a list of increasing time values indicating when each step in E occurred, and F is a failure pattern, *i.e.* a non decreasing function from $\mathcal{T}$ to 2^{Π} such that $F(\tau)$ is the set of processes that are crashed at time τ. $Faulty(R) = \bigcup_{\tau\in\mathcal{T}} F(\tau)$ is the set of faulty processes and $Correct(R) = \Pi - Faulty(R)$ is the set of correct processes.

Process p may take a step at time τ only if it is not crashed at this time. A process p takes an infinite number of steps if and only if $p \in Correct(R)$. Moreover a run satisfies properties concerning sending and receiving messages according to CP: if p sends message m to q at time τ, and q is correct then m is received by q at time $\tau + CP(\tau, p, q)$.

δ-timeliness. Given some δ and two correct processes p and q, we say that *link* (p, q) *is δ-timely* if and only if there is a time τ such that for all time $\tau' \geq \tau$ we have $CP(\tau', p, q) \leq \delta$. By convention, if q is faulty then link (p, q) is δ-timely, and if p is faulty and q is correct, link (p, q) is not δ-timely.

3 Approximation and Extraction

Predicates. Given a run R, the local state of process p in R at time τ is denoted $S_R(p, \tau)$. Predicates considered here are defined from functions ϕ from $\Pi \times \mathcal{T}$ to $\{true, false\}$ that define the truth value in local states $S_R(p, \tau)$. We always assume that $\phi(p, \tau) = true$ if p is crashed at time τ. By extension, $\phi(\tau)$ denotes $\bigwedge_{p\in\Pi} \phi(p, \tau)$. By definition, the predicate associated to ϕ, denoted P_ϕ, is true for run R if and only if there is a time τ_0 such that for all time $\tau \geq \tau_0$, $\phi(\tau)$ is true. In this case, we also say that ϕ is eventually forever true. In the following, a predicate P on run R is always associated in this way to some ϕ. When the context is clear we do not give ϕ explicitly.

For example let Q be the predicate "the boolean variable v of process p is eventually forever true". Q is a predicate on the run for which $\phi(q, \tau)$ is true if and only if v is true in $S_R(p, \tau)$. Remark that $\neg Q$ is true if and only if we have "the boolean variable v of process p is infinitely often false". Generally, $\neg P_\phi$ is equivalent to for all time τ there is some $\tau' \geq \tau$ and some process p such that $\phi(p, \tau')$ is false, that is, there is a process p such that $\phi(p, \tau)$ is false infinitely often.

Predicate P_ϕ is said to be *local to process* p if and only if for all time τ we have $\phi(\tau) = \phi(p, \tau)$. Let $\psi_{p,q}$ be the function defined by $\psi_{(p,q)}(r, \tau) = true$ if and only if $CP(\tau, p, q) \leq \delta$. $P_{\psi_{p,q}}$ is the predicate corresponding to the δ-timeliness of the link (p, q). In the following, $P_{\psi_{(p,q)}}$ will be abbreviated as $T_{pq}(\delta)$. With help of the assumption made on predicates for faulty processes, this predicate is local to process q. In the same way, predicate "p is eventually crashed", denoted $P_{p\ is\ crashed}$, is local to p.

Approximation algorithms. In the following, we are interested in algorithms that approximate some predicates in the sense that to approximate P we want to get some variable v such that: P is true if and only if v is eventually forever true. Actually, such approximations are too strong, so we consider here weaker approximations: we define the approximation by a pair of predicates (P, Q) such that (1) $P \Rightarrow Q$, (2) if P is true then v must be eventually forever true, and (3) if Q is false then v is not eventually forever true.

More precisely, consider pair (P, Q) of predicates such that $P \Rightarrow Q$, an *approximation algorithm* $\mathcal{A}$ for process p of predicates (P, Q) is an algorithm with a special boolean variable local to p $Out^p_{\mathcal{A}}$ (actually the output of $\mathcal{A}$ for p) written only by algorithm $\mathcal{A}$ such that:

- if P is true then $Out^p_{\mathcal{A}}$ is eventually forever true
- if Q is false then $Out^p_{\mathcal{A}}$ is not eventually forever true (*i.e.* is infinitely often false)

By convention, we assume that if p is correct then $Out^p_{\mathcal{A}}$ is written infinitely often (as processes are synchronous it is always possible). Not that if P is false but Q is true then $Out^p_{\mathcal{A}}$ may be eventually true forever or infinitely often false. In this way, for $Q \wedge \neg P$ there is no requirement on the output of $\mathcal{A}$.

By extension, predicates (P, Q) are local to process p if both P and Q are local to p. In the case of approximation of predicates (P, Q) local to p, an approximation algorithm can be implemented for every correct process:

Proposition 1. *If $\mathcal{A}$ is an approximation algorithm for p of predicates local to p (P, Q), then for every correct process q there exists an approximation algorithm for q of predicate (P, Q).*

Sketch of Proof. Let $\mathcal{A}$ be an approximation algorithm for p of predicates local to p (P, Q).

Algorithm given in Figure 1 implements an approximation algorithm $\mathcal{B}$ of (P, Q) for every correct process. In this algorithm, each time algorithm $\mathcal{A}$ modifies $Out^p_{\mathcal{A}}$, (1) $Out^p_{\mathcal{B}}$ is written and (2) a message is (reliably) broadcast to inform every process. When a process q delivers this message, it writes its output value $Out^q_{\mathcal{B}}$ with the new value. If this new value is false then $true$ is written again into $Out^q_{\mathcal{B}}$.

Assume that p is faulty. Then, by assumption about faulty processes and local predicates, P is $true$. Also, by definition of the algorithm, every correct process q only finitely delivers messages from p. As a consequence, $Out^q_{\mathcal{B}}$ is eventually forever true.

Assume that p is correct. First, by (1) $Out^p_{\mathcal{B}}$ is eventually forever $true$ if and only if $Out^p_{\mathcal{A}}$ is eventually forever $true$. Then, by (2), for every correct process $q \neq p$, $Out^q_{\mathcal{B}}$ is eventually forever $true$ if and only if $Out^p_{\mathcal{A}}$ is eventually forever $true$.

Hence, if $\mathcal{A}$ is an approximation algorithm for p of predicates local to p (P, Q), then $\mathcal{B}$ is an approximation algorithm for every correct process of predicate (P, Q). $\square$

The next proposition can be verified using an algorithm very similar to the one given in Figure 1.

```
In Code for $\mathcal{A}$ of $p$:
1:    whenever $\mathcal{A}$ writes $Out^p_{\mathcal{A}}$ with $b$    /* $b$ is a boolean value */
2:        $Out^p_{\mathcal{B}} := b$;rbroadcast$\langle(P, b)\rangle$
In Code for process $q \neq p$:
1:    whenever rdeliver$\langle$(P,b)$\rangle$
2:        $Out^q_{\mathcal{B}} := b$
3:        if $\neg b$ then $Out^q_{\mathcal{B}} := true$
```

Fig. 1. $\mathcal{B}$, approximation algorithm of (P, Q) for every correct process

Proposition 2. *If $\mathcal{A}_{P,Q}$ and $\mathcal{A}_{P',Q'}$ are approximation algorithms for correct process p of predicates (P, Q) and correct process q of predicates (P', Q'), respectively, then for all correct processes r, there are approximation algorithms for r of predicates $(P \wedge P', Q \wedge Q')$.*

In the following we are only interested in approximation algorithms for *all* correct processes. Then, by default, an approximation algorithm of (P, Q) means an approximation algorithm for all correct processes.

Given a finite set of indexes I and a set of predicates indexed by I say $(P_i, Q_i)_{i\in I}$, the set $(\mathcal{A}_i)_{i\in I}$ denotes the *set of approximation algorithms of* $(P_i, Q_i)_{i\in I}$, that is, for each $i \in I$, $\mathcal{A}_i$ is an approximation algorithm for (P_i, Q_i).

Extraction algorithms. Consider a set of predicates indexed by some finite set and assume that at least one of these predicates is true. We would like that all correct processes choose and converge to the same predicate satisfied in the run. To evaluate these predicates, processes use approximations algorithms as defined just before. Hence, we do not have sets of single predicates but sets of pair of predicates (P_i, Q_i) indexed by some set in which P_i specifies when the approximation outputs true forever and $\neg Q_i$ when the approximation does not output true forever. Hence, if at least one of the P_i is true, the extraction algorithm has to converge to one index j such that Q_j is true. In this way, from a property guaranteed in the run for at least one P_i, we find an index i_0 such that Q_{i_0} is true. Of course, if $Q_i = P_i$ then the chosen index verifies P_{i_0}.

More precisely, let $(P_i, Q_i)_{i\in I}$ be a set of predicates indexed by I, an *extraction algorithm* for $(P_i, Q_i)_{i\in I}$ is an algorithm such that in each run R where at least one P_i is true, all correct processes eventually choose the same index $i_0 \in I$ such that Q_{i_0} is true. In other words, each process p has a variable d_p and in each run R where $\bigvee_{i\in I} P_i = true$, there is an index $i_0 \in I$ satisfying the following two properties:

- **Eventual Agreement:** there is a time τ after which for all correct processes p $d_p = i_0$,
- **Validity:** Q_{i_0} is true.

Let I be a finite set of indexes, and $(P_i, Q_i)_{i\in I}$ a set of predicates indexed by I, if $(\mathcal{A}_i)_{i\in I}$ is a set of approximations of $(P_i, Q_i)_{i\in I}$, then the algorithm in Figure 2 is an extraction algorithm for $(P_i, Q_i)_{i\in I}$.

In this algorithm, each process p associates a (local) counter variable $Acc[i]$ to each variable $Out^p_{\mathcal{A}_i}$. Each time $Out^p_{\mathcal{A}_i}$ becomes $false$ at p, p increments $Acc[i]$. Moreover,

```
Code for each process $p$
1: Procedure $updateExtracted()$
2:     $d \leftarrow i$ such that $(Acc[i], i) = \min_{\prec_{lex}}\{(Acc[i'], i')$ such that $i' \in I\}$

3: On initialization:
4:     for all $i \in I$ do $Acc[i] \leftarrow 0$
5:     $updateExtracted()$
6:     start tasks 1, 2 and 3

7: task 1:
8:     loop forever
9:         each time $Out^p_{\mathcal{A}_i}$ becomes $false$
10:            $Acc[i] \leftarrow Acc[i] + 1$
11:            $updateExtracted()$
12: task 2:
13:     loop forever
14:         send$\langle(ACC, Acc)\rangle$ to every process except $p$ every $\eta$        /* $\eta$ is a constant */
15: task 3:
16:     upon receive$\langle(ACC, a)\rangle$ do
17:         for all $i \in I$ do
18:             $Acc[i] \leftarrow max(Acc[i], a[i])$
19:         $updateExtracted()$
```

Fig. 2. Extraction algorithm for $(P_i, Q_i)_{i\in I}$ assuming $(\mathcal{A}_i)_{i\in I}$ is a set of approximations of $(P_i, Q_i)_{i\in I}$

each p regularly sends Acc to all other processes. Upon receiving a message containing the array acc, a process locally updates $Acc[i]$ for all i with the maximum value between $Acc[i]$ and $acc[i]$. This way:

– If there is a time after which $Out^p_{\mathcal{A}_i}$ is true forever for all processes, then the value of $Acc[i]$ is eventually fixed to the same value for all correct processes.
– If $Out^q_{\mathcal{A}_i}$ is false infinitely often at some process q, then $Acc[i]$ is incremented infinitely often by q. Consequently, $Acc[i]$ is unbounded for all correct processes.

If for some j, P_j is true in the run then, as $\mathcal{A}_j$ approximates (P_j, Q_j), at least $Acc[j]$ is eventually fixed to the same value for all correct processes. To agree on some i, the processes call $updateExtracted()$ each time they modify their array: this function sets the variable d to the index i_0 such that $Acc[i_0]$ is minimum (we use the order on the indices to break tie). Hence, if for some j, P_j is true in the run then, eventually the d-variable of every correct process p is set to the same index i_0 such that $Out^p_{\mathcal{A}_{i_0}}$ is eventually forever true. As $\mathcal{A}_{i_0}$ approximates (P_{i_0}, Q_{i_0}), Q_{i_0} is true in the run and we can conclude:

Proposition 3. *Let I be a finite set of indexes, and $(P_i, Q_i)_{i\in I}$ a set of predicates indexed by I, if $(\mathcal{A}_i)_{i\in I}$ is a set of approximation algorithms of $(P_i, Q_i)_{i\in I}$, then there exists an extraction algorithm for $(P_i, Q_i)_{i\in I}$.*

Note that this extraction algorithm has two additional properties: it is self-stabilizing and may tolerates fair lossy links.

Unfortunately in this extraction algorithm all correct processes send infinitely many messages and consult infinitely many times all approximation algorithms $(\mathcal{A}_i)$. Now, it is possible to achieve a more efficient extraction concerning communication.

Let $(\mathcal{A}_i)_{i\in I}$ be a set of approximation algorithms of $(P_i, Q_i)_{i\in I}$. The extraction algorithm $\mathcal{A}$ obtained with $(\mathcal{A}_i)_{i\in I}$ is *communication-efficient* [10] if: (1) $\mathcal{A}$ is an extraction algorithm for $(P_i, Q_i)_{i\in I}$, and (2) for each run R if at least one P_i of $(P_i, Q_i)_{i\in I}$ is true then there is a time τ after which (a) there exists some j in I such that every correct process p reads only $Out^p_{\mathcal{A}_j}$, and (b) no message is sent by $\mathcal{A}$ after τ.

If $(\mathcal{A}_i)_{i\in I}$ is a set of approximations of $(P_i, Q_i)_{i\in I}$, then the algorithm in Figure 3 is an efficient extraction algorithm for $(P_i, Q_i)_{i\in I}$.

Again in this algorithm, a counter variable $Acc[i]$ is associated to each variable $Out^p_{\mathcal{A}_i}$. Again, $updateExtracted()$ returns the index i_0 such that $Acc[i_0]$ is minimum and the variable d is set to this index. However, to get the efficiency, each process p now only tests the value of $Out^p_{\mathcal{A}_d}$. Each time $Out^p_{\mathcal{A}_d}$ becomes $false$, process p blames d by reliably broadcasting the message (ACC, d) to every process. Upon delivering (ACC, x), a process increments $Acc[x]$ and calls $updateExtracted()$ to refresh the value of d.

If for some i, P_i is true in the run then, as $\mathcal{A}_i$ approximates (P_i, Q_i), $Out^q_{\mathcal{A}_i}$ is eventually forever true at all correct processes p and the message (ACC, i) can only be finitely broadcasted. Moreover, by the property of the reliable broadcast, all correct processes delivers the same number of (ACC, i) messages. Hence, eventually every correct process agrees on a fixed value of $Acc[i]$. As the value in Acc are monotically increasing, by definitions of $updateExtracted()$ and the reliable broadcast, the d-variables of all correct processes eventually converge to the same index i_0.

Assume now that the d-variables of all correct processes eventually converge to the same index i_0 but $Out^q_{\mathcal{A}_{i_0}}$ is $false$ infinitely often for some correct process q. In this case, q continuously tests $Out^q_{\mathcal{A}_{i_0}}$ and, consequently, (reliably) broadcasts infinitely many (ACC, i_0) messages. So, the value of $Acc[i_0]$ grows infinitely often at every correct process and eventually the d-variables of all correct processes are set to some other index.

Hence, if for some j, P_j is true in the run then, eventually the d-variable of every correct process p is set to the same index i_0 such that $Out^p_{\mathcal{A}_{i_0}}$ is eventually forever true. As $\mathcal{A}_{i_0}$ approximates (P_{i_0}, Q_{i_0}), Q_{i_0} is true and we can conclude:

Proposition 4. *If $(\mathcal{A}_i)_{i\in I}$ is a set of approximation algorithms of $(P_i, Q_i)_{i\in I}$, then there exists an communication-efficient extraction algorithm for $(P_i, Q_i)_{i\in I}$.*

4 Approximation Algorithms for δ-Timeliness

We now consider the predicates $T_{qp}(\delta)$ on δ-timeliness of links (q, p). We notice that even in the good case where processes are equipped with perfectly synchronized clocks, process q has to send a message every tick of time to approximate $(T_{qp}(\delta), T_{qp}(\delta))$. This does not seem reasonable, so we consider the predicate $(T_{qp}(\delta), T_{qp}(\Delta))$ with $\Delta > \delta$. Obviously, we have a good approximation when Δ is close to δ. That is, if there exist two reasonable constants m and a such that $\Delta = m\delta + a$.

We first show that without additional assumptions there is no approximation algorithm for $(T_{qp}(\delta), T_{qp}(\Delta))$. Then we consider two assumptions that make the problem solvable: (1) each process is equipped with a perfectly synchronized clock, and (2)

```
Code for each process p
1: Procedure $updateExtracted()$
2:     $d \leftarrow i$ such that $(Acc[i], i) = \min_{\prec_{lex}}\{(Acc[i'], i')$ such that $i' \in I\}$

3: On initialization:
4:     for all $i \in I$ do $Acc[i] \leftarrow 0$
5:     $updateExtracted()$
6:     start tasks 1 and 2

7: task 1:
8:     loop forever
9:         each time $Out^p_{\mathcal{A}_d}$ becomes $false$
10:            rbroadcast$\langle(ACC, d)\rangle$

11: task 2:
12:     upon rdeliver$\langle(ACC,x)\rangle$ do
13:         $Acc[x] \leftarrow Acc[x] + 1$
14:         $updateExtracted()$
```

Fig. 3. Communication-efficient extraction algorithm for $(P_i, Q_i)_{i\in I}$ assuming $(\mathcal{A}_i)_{i\in I}$ is a set of approximations of $(P_i, Q_i)_{i\in I}$

there is a Γ-timely path from q to p. We show that in both cases, there exist two constants m and a such that if $\Delta \geq m\delta + a$, then there is an algorithm to approximate $(T_{qp}(\delta), T_{qp}(\Delta))$.

4.1 Impossibility Results

If the system does not have additional assumptions, like perfectly synchronized clocks in processes p and q or a Γ-timely path from q to p, then there is no algorithm to approximate $(T_{qp}(\delta), T_{qp}(\Delta))$. This result holds even if we consider a system without crash failures.

Proposition 5. *There is no approximation algorithm for* $(T_{qp}(\delta), T_{qp}(\Delta))$.

Sketch of Proof. In a run, it is possible that p starts executing its code at some time τ_0 while q starts at some time τ'_0. (If we know that p and q start executing their local code at the same time, then as their local clock can accurately measure the time, they have a perfect synchronized clock.)

We proceed by contradiction. Assume there is an approximation algorithm $\mathcal{A}$ for $(T_{qp}(\delta), T_{qp}(\Delta))$. Let R be a run of $\mathcal{A}$ in which (i) the link q to p is δ-timely, (ii) all messages from any process different from q take a time $K > \Delta + 2$, and (iii) all messages to any process different from p take a time $K > \Delta + 2$. R defines the real time at which processes takes steps, but by hypothesis processes do not have access to this time. It is then possible to construct a run R_K of $\mathcal{A}$ in which (1) process q takes the same steps at the same time than in R, and (2) for each other process r, r takes at time $\tau + K - 1$ the step it takes at time τ in R. For every process, R and R_K are indistinguishable. Now, the properties of the approximation algorithm gives that there is a time after which $Out_{\mathcal{A}}$ is forever true in R, and consequently, in R_K. However, in R_K, the messages sent by q are received by p with a delay of $K - 1 > \Delta$. That is, the communication from q to p is not Δ-timely in R_K, contradicting the properties of the approximation algorithm. □

Note that when it is possible to design, an approximation algorithm for $(T_{qp}(\delta), T_{qp}(\delta))$ is really expensive. To see this, assume that q sends a message at time $2\tau - 1$ and at time $2\tau + 1$. These messages are received by process p at $2\tau - 1 + \delta$ and $2\tau + 1 + \delta$. As q has omitted to send a message at time 2τ, we cannot evaluate if $CT(2\tau, q, p) = \delta$ or $\delta + 1$. In this latter case, the link is not δ-timely but p cannot observe that. Hence, follows:

Proposition 6. *When an approximation algorithm for* $(T_{qp}(\delta), T_{qp}(\delta))$ *can be designed, then in the algorithm, if* q *is correct, it must eventually send messages at every clock tick.*

4.2 Approximation Algorithms

Algorithm assuming perfectly synchronized clocks. We first assume that processes are equipped with perfectly synchronized clocks denoted $clock()$, *i.e.*, for every time τ, every process p and q, we have $clock_p() = clock_q()$ at time τ. Considering the link (q, p), a constant $K > 1$, algorithm given in Figure 4 allows process p to *approximate* $(T_{qp}(\delta), T_{qp}(\delta + K))$. In the algorithm, process p has a boolean variable $Out_{\mathcal{A}_{(q,p)}}$ that is initialized to $true$ and reset to $true$ every η time. This variable remains $true$ until p learns that a message from q to p may take more than $\delta + K$ time units. In this case $Out_{\mathcal{A}_{(q,p)}}$ is set to $false$. If in the extraction algorithm, p waits that $Out_{\mathcal{A}_{(q,p)}}$ is false, then it is notified.

To test the timeliness of the link, we proceed as follows: every K time, q sends to p a `TIMELY?` message where it stores the current value of its local clock. As the clocks are perfectly synchronized, upon receiving a `TIMELY?` message tagged with the clock value c, p knows the exact time the message spends to traverse the link. If the message spends more that δ time units, p has an evidence that was not timely and, consequently, sets $Out_{\mathcal{A}_{(q,p)}}$ to $false$. Moreover, we use a *timer* of period $K + \delta$. If p does not receive any message from q during a period of $K + \delta$ time units, p suspects q and consequently sets $Out_{\mathcal{A}_{(q,p)}}$ to $false$.

If the link (q, p) is δ-timely, then q is correct. Hence, q sends `TIMELY?` messages to p infinitely often and eventually all these messages are received by p on time. So, eventually p stops setting $Out_{\mathcal{A}_{(q,p)}}$ to $false$. Hence, $Out_{\mathcal{A}_{(q,p)}}$ is eventually forever true.

If the link (q, p) is not $(\delta + K)$-timely, there is two cases to consider:

- If q eventually crashes, the timer guarantees that $Out_{\mathcal{A}_{(q,p)}}$ is $false$ infinitely often.
- Assume now that q is correct. If the link is not $(\delta + K)$-timely then for infinitely many time τ, $CT(\tau, q, p) > \delta + K$. Let τ be such a time. There exists τ' such that process q sends (`TIMELY?`) messages at τ' and at $\tau' + K$ with $\tau' < \tau \leq \tau' + K$. By the FIFO property on the links, the message sent at $\tau' + K$ is received after $\tau + \delta + K + 1 > (\tau' + K) + \delta + 1$. Hence, $Out_{\mathcal{A}_{(q,p)}}$ is set to $false$.

In both cases $Out_{\mathcal{A}_{(q,p)}}$ is $false$ infinitely often.

Lemma 1. *If processes are equipped with perfectly synchronized clocks, algorithm given in Figure 4 is an* approximation *algorithm of* $(T_{qp}(\delta), T_{qp}(\delta + K))$ *for process* p.

```
Code for process p
1: Initialization
2:     $Out_{\mathcal{A}_{(q,p)}} \leftarrow true$
3:     settimer($qp$) $\leftarrow K + \delta$
4:     start Task 1, Task 2 and Task 3

5: Task 1
6:     upon receive⟨TIMELY?, $c$⟩ from $q$ do
7:         settimer($qp$) $\leftarrow K + \delta$
8:         if clock() $- c > \delta$ then
9:             $Out_{\mathcal{A}_{(q,p)}} \leftarrow false$     /* the extraction algorithm at $p$ will notice that $Out_{\mathcal{A}_{(q,p)}}$ $isfalse$ */

10: Task 2
11:     upon timerexpire($qp$) do
12:         settimer($qp$) $\leftarrow K + \delta$
13:         $Out_{\mathcal{A}_{(q,p)}} \leftarrow false$     /* the extraction algorithm at $p$ will notice that $Out_{\mathcal{A}_{(q,p)}}$ $isfalse$ */
14: Task 3
15:     do every $\eta$ time     /* $\eta$ is a constant */
16:         $Out_{\mathcal{A}_{(q,p)}} \leftarrow true$

Code for process q
1: Initialization
2:     start Task 4

3: Task 4
4:     do every $K$ time
5:         send⟨TIMELY?, clock()⟩ to $p$
```

Fig. 4. Algorithm for p to approximate $(T_{qp}(\delta), T_{qp}(\delta + K))$, assuming perfectly synchronized clocks

With Proposition 1, we obtain:

Theorem 1. *If processes are equipped with perfectly synchronized clocks, there is an algorithm for all processes to approximate* $(T_{qp}(\delta), T_{qp}(\delta + K))$.

Algorithm assuming a Γ-timely path in the reverse side. We now assume that local clocks may not be synchronized. Instead, we assume that if the link (q, p) is δ-timely, then there exists a Γ-timely path from p to q, that is, a path such that if p and q are correct and p sends a message to q at time τ, then there exists some correct processes r_1,...,r_k and some time τ_1,...,τ_k such that (1) $r_1 = p$, (2) $r_k = q$, (3) $\tau_1 = \tau$, (4) $\tau_k \leq \tau + \Gamma$, and (5) for all $1 \leq i < k$, if r_i sends a message at time τ_i, it is received by r_{i+1} at time τ_{i+1}.

Considering the link from q to p, the algorithm given in Figure 5 allows process p to *approximate* the predicate $(T_{qp}(\delta), T_{qp}(2\delta + \Gamma + K))$.

In the algorithm, process p has a boolean variable $Out_{\mathcal{A}_{(q,p)}}$ that behaves as in the previous algorithm.

To test the timeliness of (q, p), we proceed by phases. Every K times, p broadcasts to every other process a `TIMELY?` message where it stores the current phase number and a counter value initialized to 0. Then, using the counter, the message is relayed at most $n - 1$ times in all directions except p until reaching q. These relays guarantee that if there exists a Γ-timely path from p to q, then at least one `TIMELY?` message tagged with the current phase number arrives to q in less than Γ time units. Upon receiving

```
Code for process $p$
 1: **Initialization**
 2:     $phase\_qp \leftarrow 0$
 3:     $Out_{\mathcal{A}_{(q,p)}} \leftarrow true$
 4:     start Task 1, Task 2, Task 3 and Task4

 5: **Task 1**
 6:     **do every** $K$ **time**
 7:         $phase\_qp \leftarrow phase\_qp + 1$
 8:         send⟨TIMELY?, $phase\_qp$, 0⟩ **to** every process except $p$
 9:         settimer($phase\_pq$) $\leftarrow \delta + \Gamma$

10: **Task 2**
11:     **upon** receive⟨TIMELY?, $\ell$, $-$⟩ **from** $q$
12:         unsettimer($\ell$)

13: **Task 3**
14:     **upon** timerexpire($\ell$) **do**
15:         $Out_{\mathcal{A}_{(q,p)}} \leftarrow false$      /* the extraction algorithm at $p$ will notice that $Out_{\mathcal{A}_{(q,p)}} isfalse$ */

16: **Task 4**
17:     **do every** $\eta$ **time**      /* $\eta$ is a constant */
18:         $Out_{\mathcal{A}_{(q,p)}} \leftarrow true$

Code for process $q$
 1: **Initialization**
 2:     start Task 5

 3: **Task 5**
 4:     **upon** receive⟨TIMELY?, $t$, $k$⟩ **from** any process $r$ **do**
 5:         send⟨TIMELY?, $t$, $k+1$⟩ **to** $p$

Code for every process except $p$ and $q$
 1: **Initialization**
 2:     start Task 6

 3: **Task 6**
 4:     **upon** receive⟨TIMELY?, $t$, $k$⟩ **from** any process $r$ **do**
 5:         **if** $k \leq n-1$ **then**
 6:             send⟨TIMELY?, $t$, $k+1$⟩ **to** every process except $p$
```

Fig. 5. Algorithm for p to *approximate* $(T_{qp}(\delta), T_{qp}(2\delta + \Gamma + K))$ assuming a Γ-timely path in the reverse side

such a message, q relays the message a last time to p only. Hence, if (q, p) is δ-timely, p receives from q at least one TIMELY? message tagged with the current phase number in less than $\delta + \Gamma$ time units. If p does not receive this message by time $\delta + \Gamma$, it sets $Out_{\mathcal{A}_{(q,p)}}$ to $false$. To this end, it uses one timer per phase that is activated when it sends a TIMELY? message and this timer is disabled if the corresponding TIMELY? message is received from q on time.

If the link (q, p) is δ-timely, then q is correct. So, there is a time after which, during every phase, p receives at least one TIMELY? message from q in less than $\delta + \Gamma$ time. Hence, $Out_{\mathcal{A}_{(q,p)}}$ is eventually forever true.

If the link (q, p) is not $(2\delta + \Gamma + K)$-timely, there is two cases to consider:

– If q eventually crashes, eventually p stops receiving TIMELY? messages and, consequently, sets $Out_{\mathcal{A}_{(q,p)}}$ to $false$ infinitely often.
– Assume now that q is correct. If the link is not $(2\delta + \Gamma + K)$-timely then for infinitely many time τ, $CT(\tau, q, p) > 2\delta + \Gamma + K$. Let τ be such a time.

By definition, if q sends a message to p at time τ, then p receives the message at a time τ' such that $\tau' > \tau + 2\delta + \Gamma + K$. Moreover, the link being FIFO, (1) every message sent by q to p after time τ is received after time $\tau' > \tau + 2\delta + \Gamma + K$.

Every K time, p increments ph, sends a message (TIMELY?,ph,0), and starts a timer identified by ph that will expire $\delta + \Gamma$ times later. In particular, p sends such a message, say (TIMELY?,ph_i,0), at a time τ_i such that $\tau \leq \tau_i \leq \tau + K$ and starts a timer ph_i that will expire at the latest at time $\tau + \delta + \Gamma + K$. Moreover, (2) before time τ_i no (TIMELY?,ph_i,$-$) message exists in the system. So by (1) and (2), p cannot receive any (TIMELY?,ph_i,$-$) message before time $\tau' > \tau + 2\delta + \Gamma + K$. Hence, p cannot receive any (TIMELY?,ph_i,$-$) message before timer ph_i expires. Consequently, p will set $Out_{\mathcal{A}_{(q,p)}}$ to $false$. Hence, if (q, p) is not $(2\delta + \Gamma + K)$-timely, p sets $Out_{\mathcal{A}_{(q,p)}}$ to $false$ infinitely often.

In both cases $Out_{\mathcal{A}_{(q,p)}}$ is $false$ infinitely often.

Lemma 2. *If there is a Γ-timely path from p to q, algorithm given in Figure 4 is an* approximation *algorithm $(T_{qp}(\delta), T_{qp}(2\delta + \Gamma + K))$ for process p.*

With Proposition 1, we obtain:

Theorem 2. *If there is a Γ-timely path from p to q, then there is an algorithm for all processes to approximate $(T_{qp}(\delta), T_{qp}(2\delta + \Gamma + K))$*

δ-timely paths. In the same way it possible to approximate a δ-timely path P from p to q, (every message sent along the path arrives to process q within δ), we denote this predicate $T_P(\delta)$. The algorithms that approximate $(T_P(\delta), T_P(\Delta))$ are similar to those Figure 4 and Figure 5. Let P be a path from p to q, we have:

Theorem 3. *If processes are equipped with perfectly synchronized clocks, there is an algorithm for all processes to approximate $(T_P(\delta), T_P(\delta + K))$.*

Theorem 4. *If there is a Γ-timely path from q to p, then there is an algorithm for all processes to approximate $(T_P(\delta), T_P(2\delta + \Gamma + K))$*

5 Extraction of δ-Timeliness Graphs

In this section, we apply our previous results to extract graphs. To that end, we give few examples of sets of predicates on graphs. These predicates concern the δ-timeliness of the edges of the graph. We show that we can extract an element of such sets under some assumptions. We also exhibit some desirable properties of the extracted graphs.

We begin with some definitions and notations about graphs.

5.1 Graphs

For a directed graph $G = \langle N, E \rangle$, $Nodes(G)$ and $Edges(G)$ denote N and E, respectively. The tuple (X, Y) is a *directed cut* (*dicut* for short) of G if and only if X and Y define a partition of $Nodes(G)$ and there is no directed edge $(y, x) \in Edges(G)$ such that $x \in X$ and $y \in Y$.

Lemma 3. *For every path $p_0, \dots p_m$ constituted of Δ-timely links, there exists $k \in [0 \dots m+1]$ such that:*

- *for all j such that $0 \leq j < k$, p_j is a correct process, and*
- *for all j such that $k \leq j \leq m$, p_j is faulty.*

Proof: For all $i \in [0 \dots m-1]$, (p_i, p_{i+1}) is Δ-timely and, by definition, if p_{i+1} is correct then p_i is also correct, which proves the lemma. □

We deduce the following corollary from Lemma 3:

Corollary 1. *Let P be a path from p to q constituted of Δ-timely links.*

- *If p and q are correct, then all processes in P are correct.*
- *If P is a cycle, then either all processes are correct or all processes are faulty.*

5.2 Extracting an Elementary δ-Timely Path from p to q

Let $(Path_i)_{i \in I}$ be the set of all possible elementary paths from p to q. If $(\mathcal{A}_i)_{i \in I}$ is a set of approximation algorithms of $(T_{Path_i}(\delta)), T_{Path_i}(\Delta))_{i \in I}$, then from Proposition 3, there is an extraction algorithm for $(T_{Path_i}(\delta), T_{Path_i}(\Delta))_{i \in I}$, and from Proposition 4, there is a communication-efficient extraction algorithm for $(T_{Path_i}(\delta), T_{Path_i}(\Delta))_{i \in I}$. By Proposition 2 and Theorem 3, assuming perfectly synchronized clocks, there is an approximation algorithm for $(T_{Path_i}(\delta), T_{Path_i}(\Delta))_{i \in I}$ for every $\Delta > \delta$. Hence, we can conclude:

Theorem 5. *Assuming perfectly synchronized clocks and $\Delta > \delta$, there exists a (communication-efficient) extraction algorithm for $(T_{Path_i}(\delta), T_{Path_i}(\Delta))_{i \in I}$.*

Following a similar reasoning, using Theorem 2, we have:

Theorem 6. *Assuming a Γ-timely path in the reverse side, if $\Delta > 2\delta + \Gamma$, there is a (communication-efficient) algorithm for $(T_{Path_i}(\delta), T_{Path_i}(\Delta))_{i \in I}$.*

By Corollary 1, if p and q are correct, then every Δ-timely path from p to q only contains correct processes. Hence, the algorithm we obtained allows to efficiently route message in the network. Moreover, our approach being modular, one can design a routing algorithm for only a restricted subset of processes. For example, if we consider a clustered network, one may want to design an efficient routing algorithm only for the set of clusterheads (the communication inside a cluster being usually local or managed using an overlay like a tree).

5.3 Extracting δ-Timely Graphs

We want now to extract δ-timely graphs containing all correct processes. Below, we give some definitions to formally explain our approach.

Consider the set of all graphs $(G_i)_{i \in I}$ that can be constructed with $Nodes(G_i) \subseteq \Pi$ and $Edges(G_i) \subseteq Nodes(G_i) \times Nodes(G_i)$.

```
Code for process $p$
1: **Initialization**
2:     start Task 1

3: **Task 1**
4:     **do every** $\eta$ **time**    /* $\eta$ is a constant */
5:         send⟨$(ALIVE)$⟩ **to** every process except $p$
6:         $Out_{\mathcal{A}_p} \leftarrow false$    /* the extraction algorithm at $p$ will notice that $Out_{\mathcal{A}_p}$ is $false$ */

Code for process $q \neq p$
1: **Initialization**
2:     start Task 2 and Task 3

3: **Task 2**
4:     **upon** receive⟨ALIVE⟩ **from** $p$ **do**
5:         $Out_{\mathcal{A}_p} \leftarrow false$
6: **Task 3**
7:     **do every** $\eta$ **time**    /* $\eta$ is a constant */
8:         $Out_{\mathcal{A}_p} \leftarrow true$
```

Fig. 6. Algorithm to approximate $(P_{p\ is\ crashed}, P_{p\ is\ crashed})$

$TG_{G_i}(\delta)$ is true in run R if and only if eventually (1) all nodes of $\Pi - G_i$ are crashed, and (2) all edges (p,q) of G_i are δ-timely.

By definition, if $TG_{G_i}(\delta)$ is true in a run R, G_i contains all correct processes. If $(\mathcal{A}_i)_{i\in I}$ is a set of approximations of $(TG_{G_i}(\delta), TG_{G_i}(\Delta))_{i\in I}$, then by Propositions 2, 3, and 4, there is an (efficient) extraction algorithm for $(TG_{G_i}(\delta), TG_{G_i}(\Delta))_{i\in I}$.

To define $TG_{G_i}(\delta)$ we need a local predicate $P_{p\ is\ crashed}$ that states if a process p is crashed. A local algorithm that approximates this predicate is given below.

Approximate crashed processes. We can easily design an approximation algorithm $\mathcal{A}_p$ for $(P_{p\ is\ crashed}, P_{p\ is\ crashed})$ at every process $q \neq p$: process p regularly sends messages. Each time a process receives such a message it sets $Out_{\mathcal{A}_p}$ to false. Moreover, q regularly resets $Out_{\mathcal{A}_p}$ to $true$. If p is faulty, $Out_{\mathcal{A}_p}$ is eventually forever $true$. Otherwise (p is correct), $Out_{\mathcal{A}_p}$ is infinitely often $false$. Hence, follows:

Proposition 7. *Algorithm given in Figure 6 allows every process q to* approximate $(P_{p\ is\ crashed}, P_{p\ is\ crashed})$.

Using $P_{p\ is\ crashed}$ and $T_e(\delta)$, we can now define $TG_{G_i}(\delta)$ as follows: $TG_{G_i}(\delta) \equiv \bigwedge_{e\in Edges(G_i)} T_e(\delta) \wedge \bigwedge_{v\notin Nodes(G_i)} Crash(v)$.

Assuming perfectly synchronized clocks, by Propositions 7, 2, and Theorem 1, there is an approximation algorithm $\mathcal{A}_{G_i}$ for $(TG_{G_i}(\delta), TG_{G_i}(\Delta))_{i\in I}$ if $\Delta > \delta$. By Propositions 3 and 4, we can conclude:

Theorem 7. *Assuming perfectly synchronized clocks and $\Delta > \delta$, there exists a (communicationn-efficient) algorithm that extracts $(TG_{G_i}(\delta), TG_{G_i}(\Delta))_{i\in I}$.*

Following a similar reasoning, by Proposition 2, 3, 4, 7, and Theorem 2, we have:

Theorem 8. *Assuming a Γ-timely path in the reverse side, if $\Delta > 2\delta + \Gamma$, there is a (communication-efficient) algorithm that extracts $TG_{G_i}(\delta), TG_{G_i}(\Delta))_{i\in I}$.*

From the previous theorem and Corollary 1, we can deduce the next corollary, which states that the extracted graph is a dicut between correct and faulty processes. Note that this property is very useful. For example, one can design an approximation algorithm to extract a tree. Then, if not all the processes are faulty, the extracted tree will be rooted at a correct process, the tree will contain all correct processes, and in the tree there will exist a Δ-timely path from the root to every other correct process. Hence, the algorithm will allow to communication-efficiently route message from a correct to all others.

Corollary 2. *If G_{i_0} is the extracted graph, $G_{i_0}[Correct(R)]$, $G_{i_0}[Faulty(R)]$ is a directed cut of G_{i_0}*

From Corollaries 1 and 2, we have the next corollary, which gives a sufficient condition to evaluate $\Diamond\mathcal{P}$, the eventually perfect failure detector [5] that eventually outputs exactly the set of correct processes.

Corollary 3. *If there is at least one correct process and if $(G_i)_{i \in I}$ contains only strongly connected graphs, the extracted graph G_{i_0} contains only correct processes.*

Extracting a ring containing all correct processes. We now want to extract a δ-timely ring among all correct processes. Consider the set of all graphs $(Ring_i)_{i \in I}$, that are all possible rings among any non-empty subset of Π.

$TG_{Ring_i}(\delta)$ is true in run R if and only if eventually (1) all nodes of $\Pi - G$ are crashed, and (2) all edges (p, q) of G are δ-timely.

By Corollary 1, if $Ring_G(\delta)$ is true in run R then G contains exactly the set of correct processes of run R.

Assuming perfectly synchronized clocks, by Propositions 2, 7, and Theorem 1, there is an approximation algorithms for $(TG_{Ring_i}(\delta), TG_{Ring_i}(\Delta))_{i \in I}$. By Propositions 3 and 4, we can conclude:

Theorem 9. *Assuming perfectly synchronized clocks, if $\Delta > \delta$, there exists a (communication-efficient) extraction algorithm for $(TG_{Ring_i}(\delta), TG_{Ring_i}(\Delta))_{i \in I}$.*

In a ring composed of δ-timely links, each link (p, q) of the ring is δ-timely and there is a path from q to p that is $(n-1)\delta$-timely. Then, by Propositions 2, 3, 4, and Theorem 2 , we have:

Theorem 10. *If $\Delta > (n + 1)\delta$, there is a (communication-efficient) extraction algorithm for $(TG_{Ring_i}(\delta), TG_{Ring_i}(\Delta))_{i \in I}$.*

As noted previously the extracted ring contains exactly the correct processes.

6 Concluding Remarks

In this paper, we studied in which way processes may approximate and agree on structural properties based on δ-timeliness (*e.g.*, find δ-timely paths between processes or build a ring between correct processes using only δ-timely links).

We focused on δ-timeliness of the links, however other properties are also interesting to approximate. For example, with general timeliness defined as the existence of some unknown bound on communication delays, we can get the same results as in [7]. Approximation and extraction algorithms may be considered as a first step to dynamically evaluate predicates expressed in some kind of temporal logic.

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 108–122. Springer, Heidelberg (2001)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. In: PODC, pp. 306–314 (2003)
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: PODC, pp. 328–337. ACM, New York (2004)
4. Aguilera, M.K., Deporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega in systems with weak reliability and synchrony assumptions. Distributed Computing 21(4), 285–314 (2008)
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
6. Chu, F.: Reducing Ω to $\Diamond W$. Information Processing Letters 67(6), 298–293 (Sep 1998)
7. Delporte-Gallet, C., Devismes, S., Fauconnier, H., Larrea, M.: Algorithms for extracting timeliness graphs. In: SIROCC0 (2010)
8. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Tech. Rep. TR 94-1425, Department of Computer Science, Cornell University (1994)
9. Hutle, M., Malkhi, D., Schmid, U., Zhou, L.: Chasing the weakest system model for implementing omega and consensus. IEEE Trans. Dependable Sec. Comput. 6(4), 269–281 (2009)
10. Larrea, M., Arévalo, S., Fernández, A.: Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 34–48. Springer, Heidelberg (1999)
11. Mostéfaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: DSN, pp. 351–360. IEEE Computer Society, Los Alamitos (2003)

A Framework for Adaptive Optimization of Remote Synchronous CSCW in the Cloud Computing Era

Ji Lu, Yaoxue Zhang, and Yuezhi Zhou

Key Laboratory of Pervasive Computing, Ministry of Education
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, Beijing, China
`luji07@mails.tsinghua.edu.cn`

Abstract. The systems for remote synchronous computer supported cooperative work (CSCW) are significant to facilitate people's communication and promote productivity. However, in the Internet, such systems often suffer from the problems of relatively large latency, low bandwidth and relatively high cost of wide-area networking. Previous works tried to improve various mechanisms of communication, but till now we still cannot get rid of these problems due to the nature of the Internet data transmission mechanism. Rather than making optimizations based on the traditional CSCW computing style as previous work did, this paper proposes an idea of moving appropriate collaborative instances to the proper computing nodes which are just born in the emerging Cloud computing environments. Moreover, the paper presents a formal framework AORS to optimally organize the collaborative computing upon the emerging computational resources from the perspectives of both performance and cost. The formulization of the framework is proposed, and an analytic theory is developed. Directly solving the modeled problem has to refer to the exhaustive search, which is of exponential computational complexity; so we develop two heuristics. The experimental evaluations demonstrate the high efficiency and effectiveness of the heuristics. Furthermore, we conduct extensive simulation experiments on the current collaborative computing style and AORS. They illustrate that AORS brings the CSCW applications better communication quality and lower cost.

1 Introduction

The remote synchronous CSCW systems [1], which denote the systems for geographically distributed individuals collaboratively computing in the same period of time, have provided us with a lot of benefit. The important instances include electronic meeting systems, multiplayer online gaming systems, various real-time groupware, etc. However, in the Internet environments, the networked collaborations often suffer from the problems of relatively low quality and relatively high cost of wide-area networking. A lot of research has been done to address these problems, which can be divided to three categories. The first category tries to improve the network infrastructure [10][16], few of which however are adopted in the actual deployment. The works in the second category concentrate on optimizing the mechanisms or protocols of the communication among collaborative instances: [14] speeds-up the slow uplink direction by means of a compression scheme based on a dictionary transmitted through the downlink channel; in [11] IPComp is proposed as an IP payload compression protocol to reduce the data size in general IP packet transmission; Fast TCP [4] and UDT [9] enhance on the specific transport layer protocols, attempting to make better use of the existing resources. The third category emphasizes on exploring the CSCW-specific approaches: [3] introduces software design patterns for building optimistic constructs into networked games through optimistic programming, expecting to reduce the effects of the network latency; [13] designs a latency prediction system Htrae that allows online game players to cluster themselves so that those in the same session might have relatively low latency.

Although these works have been proposed, the original problems are still extremely difficult to solve. The relatively low quality, such as the relatively large delay, stems from the Internet nature of the multi-hop store-and-forward transmission. And in the aspect of the cost, Jim Gray examined technological trends in 2003, and concluded that the cost of wide-area networking had fallen more slowly and still remained relatively higher than other IT hardware costs [7]. Furthermore, from 2003 to 2008, the cost/performance improvement of wide-area networking was only 2.7, the least one among the comparisons with computing and storage, which were 16 and 10, respectively [2]. Such problems are especially more serious in the emerging mobile device based CSCW systems. The connection quality and the computing capability of the mobile devices are generally lower than the common PCs or workstations. Moreover, except the WiFi-enabled occasions, the price of GPRS or 3G connection is relatively high.

We observe that there is a basic fact: almost all these previous works are based on the traditional computing style in which all the collaborators (software) are arranged to run on their pre-defined locations. For example, in a networked collaborative drawing scenario, each collaborator, that is, an instance of collaborative drawing software as well as the related data, performs on one user's computer and communicates with others in the drawing process. So in such current style, we *have to* bear the relatively large latency and the relatively high cost of communication because this distributed computing *has to* suffer such best-effort long-distance Internet data transmission.

Currently, we begin to reconsider these problems due to a new characteristic, emerging in the Internet. It is observed that in the Internet some nodes for the public to compute with enough freedom and capability finally come out. Since the naissance of computer networks, we could employ some capabilities of other machines. However, so far such employment has been quite limited actually; for instance, although common users can publish an essay in a blog server, they could hardly find a place in the Internet to run a program at their will. The formerly developed Grids mainly focused on the research and scientific computing, and could hardly be employed freely for the public. Currently, with the industry zealously promoting Cloud computing [2][17], some nodes with powerful computing capability, e.g., the datacenter nodes, begin to come out. The public actually have computing capabilities geographically distributed in the Internet to carry out tasks with enough computational freedom. In the meantime, the portability of computational components across different underlying computing nodes could be well handled by the extensive virtual computing mechanisms.

Instead of computing in the current style, we envision what about moving appropriate collaborators to the capable and proper computing nodes in the Internet, and more importantly, organizing the CSCW in a better way? Specifically, take a concrete CSCW application for example: there are five collaborators {*P*1, *P*2, *P*3, *P*4, *P*5}, each of which consists of the necessary program and associated data; every collaborator belongs to the corresponding user {*U*1, *U*2, *U*3, *U*4, *U*5}. In the traditional/current computing style, each collaborator resides and computes in one user's own computer, as the concentric-circles illustrate. However, if some computing nodes with enough computational freedom and capability exist in the Internet, the collaborator can compute on such places, namely *E*, *F*, *G*, *H*, *I*, *J* demonstrated in Fig. 1(a). If the volume of the communication data among the collaborators is so large, all of the five collaborators may migrate to one datacenter, e.g., the node *H*, to let the communication cost tend to zero. However, it should be first verified that the migration cost of the collaborators would not counteract the decrease of the communication cost. Furthermore, it should be guaranteed that the networking quality between each user node and the node where the corresponding collaborator computes is within the acceptable range. Moreover, the time consumed for moving the collaborator from its original node to the new computing node should be beneath the maximal user-bearable threshold. So comprehensively considering all these goals and constraints, it is likely that the final optimal organization scheme is just as Fig. 1(b) illustrates. This paper is to formally explore how we can organize the CSCW computing in an optimal way.

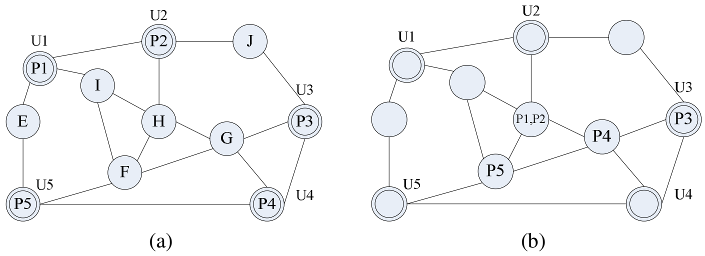


Fig. 1. Illustration of the organization of CSCW computing

2 Overview of AORS Framework

In a networked CSCW task, as mentioned above, we call each distributed collaborative instance as a *collaborator*. Initially, each collaborator resides in its *default computing node*. For instance, in a collaborative drawing application, a collaborative instance, which includes the drawing program and associated data, is a collaborator; the machine in which the collaborator resides is the collaborator's default computing node. As a matter of fact, each collaborator computing at its default computing node is just the current collaborative computing style. In the network, the emerging computing nodes capable of hosting the collaborator, such as the datacenters in Cloud computing, are called *candidate computing nodes*. Both these two classes, namely, the default computing node and candidate computing nodes are also named *qualified computing nodes*.

In one CSCW task, there could be several collaborators, which can be either homogenous or heterogeneous in terms of computational requirements and behavior. In the networks, for any collaborator, there could be multiple qualified computing nodes that are capable of hosting it. Facing these various distributed computational entities, the goal of the optimal organization of the CSCW computing refers to choosing an optimal computing node in the networks for each collaborator. Such optimization is considered from two sides: both the performance and the cost.

Performance View: Through the survey on numerous kinds of collaborative applications in the Internet, we discover that two aspects are most significant to user experience. The first is the time for the desired application to be prepared well for use. The main part of such preparation time here is the collaborator transferring time. So we take this into account. The second aspect is the connection quality between pairs of collaborators, and between the users and the collaborators. To characterize such connection quality, we adopt the metric goodput. Goodput can reflect the real transmission rate between two nodes in the network, and it is easy to be obtained.

Cost Perspective: The other side is considered in the view of the cost on the utilization of common computing resources. In contrast, the organization of computing in traditional related fields, e.g., parallel computing, always involves the adjustment on workload, such as load balance. However, the load is not considered in this context, because most computing nodes emerging currently in reality are born just as super data centers, aiming at providing any scale-up and scale-down elasticity. Here, the cost does not directly mean the monetary expenditure. It aims at reflecting the system effort to complete a given task. More specifically, as analyzed in the first section, in general, the networking becomes the most precious resource in a current computation. So we choose the cost of data transfer in the Internet as a factor in the optimization.

The framework AORS consists of four main parts: Application Feature Capture Engine, Network Characteristic Detection Engine, Optimization Unit and Computing Support Platform. A high level illustration of the framework is given in Fig. 2. Application Feature Capture Engine is responsible for obtaining the characteristics and demands of a collaborative computing application, and Network Characteristic Detection Engine uses a series of mechanisms to acquire the essential network parameters and status. All the knowledge above flows to the Optimization Unit to compute the optimal organization scheme of the distributed entities for the collaborative task. Then according

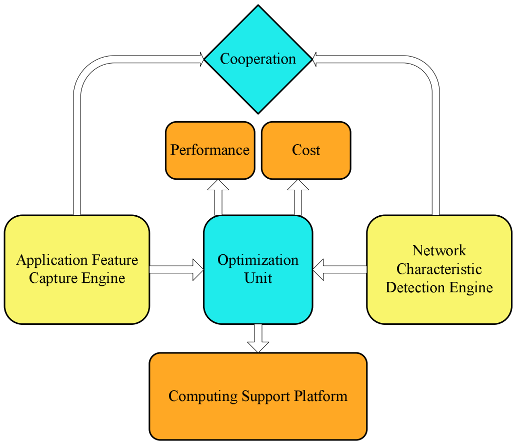

Cooperation
Performance
Cost
Application Feature Capture Engine
Optimization Unit
Network Characteristic Detection Engine
Computing Support Platform

Fig. 2. The high level view of the AORS framework

to the optimal scheme, Computing Support Platform will transfer the migratable collaborators from their default computing nodes to the optimal candidate computing nodes, and may also transfer some data back when the task is completed, if needed. No matter where the collaborator is, (e.g., at the default computing node or at the candidate computing node), the user, if existing, generally associates with the default computing node. So if a collaborator is running in a remote candidate computing node, the default computing node will provide necessary remote output and input function for the user, acting like a SSH or VNC client.

3 Formulization of AORS

3.1 Application Feature Capture Engine

Application Feature Capture Engine (AFCE) collects the essential features of a collaborative computing application that is to be performed in the networks. This process is completed by AFCE interacting with the application, and cooperating with Network Characteristic Detection Engine.

The features are obtained through two ways. The first is when a collaborative application is going to start, it actively reports the computational parameters to AFCE. The second is AFCE uses history to predict the value of the parameter which is needed. The captured features include the collaborator information and the application-specific parameters, which can be formalized as follows.

In a collaborative computing application, there are m collaborators, which form the collaborator set $P = \{p_k \mid 1 \le k \le m\}$. Initially, these collaborators reside in a set of the default computing nodes $G = \{g_k \mid 1 \le k \le m\}$. Specifically, assume p_k is in the node g_k. Besides the initially resident place g_k, in the network, there might exist a set of candidate computing nodes. The default computing node and all the candidate computing nodes of p_k form the set of qualified computing nodes which is of size n_k, denoted via $D^k = \{d_j^k \mid 1 \le j \le n_k\}$. So $g_k \in D^k$; moreover, $g_k = d_1^k$. In particular, in the case that collaborator p_k is not migratable as mentioned previously, $n_k = 1$. If moving the collaborator p_k from g_k to any other qualified computing node d_j^k, the data of size h_k should be transferred. At the end of the application, the collaborator which was moved from the default computing node to one of the candidate computing nodes, might migrate back, and let the size of data transferred in this process be v_k. The above features of the application are generally easy to be obtained in the optimization phase; however, the size $e_{k'k''}(t)$ of the data that collaborator $p_{k'}$ will communicate to $p_{k''}$ during the collaborative task is relatively difficult to determine, which will be addressed later.

3.2 Network Characteristic Detection Engine

Network Characteristic Detection Engine (NCDE) is responsible for detecting the necessary network information for the collaborative application. NCDE uses three ways to detect the needed network parameters: the engine actively requests the detected object for the needed information; it leverages the history of the corresponding network status and calculates the current value on their own; in the case of large data centers promulgating their parameters, through passively listening to the reports, NCDE updates the corresponding data.

The characteristics that NCDE detects comprise the particular network connection parameters, which can be formalized as follows.

$w^{k}_{j'j''}$: the cost per unit load transferring from $d^{k}_{j'}$ to $d^{k}_{j''}$;

$\chi^{k'k''}_{j'j''}$: the cost per unit load transferring from $d^{k'}_{j'}$ to $d^{k''}_{j''}$.

Although the cost per unit load transferring is relatively static, the goodput is likely to change due to dynamics of networking. Let $r^{k}_{j'j''}(t)$ be the goodput from $d^{k}_{j'}$ to $d^{k}_{j''}$, and $r^{k'k''}_{j'j''}(t)$ be the goodput from $d^{k'}_{j'}$ to $d^{k''}_{j''}$.

3.3 Optimization Unit

The optimal scheme can be either computed once before an application starts or recomputed periodically during a long collaborative computing process in order to take into account the dynamics of the computing environments. Uniformly, we assume an optimal scheme is generated for the organization of the collaborative computing in the next short period of time *s*.

In both AFCE and NCDE, some needed parameters are obtained through prediction. So to well handle this, we use Genetic Programming to estimate the tendency of such parameters, specifically $e_{k'k''}(t)$, $r^{k}_{j'j''}(t)$, $r^{k'k''}_{j'j''}(t)$, in the next short period of time *s*.

Define the function

$$\xi:\quad \mathrm{Z}^{+} \rightarrow \mathrm{Z}^{+}. \tag{1}$$

More specifically, for $y = \xi(x)$, $x \in [1, m]$, $y \in [1, n_x]$. This function establishes a mapping from the index of collaborator set to the index of computing node set. If given a collaborator p_x , $x \in [1, m]$, the computing node chosen for hosting the computation of p_x can be denoted by $d^{x}_{\xi(x)}$.

If we decide to move a collaborator p_k from its default computing node to a candidate computing node in the networks, the data of size h_k will be transferred. In practice, the time for such transfer is expected to be within a user-acceptable limit. According to the goodput measured by Network Characteristic Detection Engine, the time *t* for transferring the collaborator p_k can be computed as follows. First, we have

$$\int_0^t r^{k}_{1\xi(k)}(t)\, d(t) = h_k \tag{2}$$

Let $R^{k}_{1\xi(k)}(t)$ be an antiderivative of $r^{k}_{1\xi(k)}(t)$ on [o, *t*]. From the fundamental theorem of calculus, we have

$$\int_0^t r^{k}_{1\xi(k)}(t)\, d(t) = R^{k}_{1\xi(k)}(t) - R^{k}_{1\xi(k)}(0) = h_k \tag{3}$$

$$R^{k}_{1\xi(k)}(t) = h_k + R^{k}_{1\xi(k)}(0) \tag{4}$$

$$t = {R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0)) \tag{5}$$

where ${R^{k}_{1\xi(k)}}^{-1}$ is the inverse function of $R^{k}_{1\xi(k)}$.

Assuming that all the collaborators which need to be moved begin to migrate at the same time, we can compute the overall time spent on the migration procedure as $\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0))\right)$. For a given collaborative application, if τ denotes the maximal user-acceptable collaborator migration time, the following inequality should be guaranteed.

$$\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0))\right) \le \tau \tag{6}$$

As mentioned above, the considered networking quality is the user-to-collaborator goodput, collaborator-to-user goodput and collaborator-to-collaborator goodput. Users are generally with the default computing nodes, no matter where the collaborators are moved to. So the average user-to-collaborator goodput of one user to the corresponding collaborator p_k should be computed as $\dfrac{\int_{\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k}_{1\xi(k)}(t)\,d(t)}{s-\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0))\right)}$. And the collaborator-to-collaborator goodput of collaborator $p_{k'}$ to $p_{k''}$ can be obtained through $\dfrac{\int_{\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k\bar{k}}_{\xi(k)\xi(\bar{k})}(t)\,d(t)}{s-\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0))\right)}$.

For a given collaborative application, let δ^{+} be the minimal allowable average goodput from the user to the corresponding collaborator, δ^{-} be the minimal allowable average goodput from the collaborator to its user, and let Ψ be the minimal bearable average collaborator-to-collaborator goodput. We should make sure the following inequality hold.

$$\forall\, k \in [1, m], \frac{\int_{\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k}_{1\xi(k)}(t)\,d(t)}{s-\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0))\right)} \ge \delta^{+} \tag{7}$$

$$\forall\, k \in [1, m], \frac{\int_{\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k}_{\xi(k)1}(t)\,d(t)}{s-\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0))\right)} \ge \delta^{-} \tag{8}$$

$$\frac{\sum_{k=1}^{m}\sum_{\bar{k}=1}^{m}\left(\dfrac{\int_{\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k\bar{k}}_{\xi(k)\xi(\bar{k})}(t)\,d(t)}{s-\max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k + R^{k}_{1\xi(k)}(0))\right)}\right)}{2m} \ge \Psi \tag{9}$$

Let $c(w,a,b)$ be the cost of transferring the data of size w from node a to node b. We define $c(w,a,b) = w\cdot\sigma_{ab}$, where σ_{ab} is the cost per unit load transferring from node a to node b, which can be presented via different kinds of metrics, such as hops, or the money paid to the ISP or to the Cloud service provider.

So it can be obtained that for a given collaborative application, the cost of transferring all necessary collaborators from their default computing nodes to the chosen candidate computing nodes is

$$\psi(\xi) = \sum_{k=1}^{m} h_k w_{1\xi(k)}^{k} \tag{10}$$

And during the collaborative computing, $e_{k'k''}(t)$, the size of the data transferred from the collaborator $p_{k'}$ to $p_{k''}$ at the moment *t* can be zero, if no communication is needed. According to $c(w,a,b) = w \cdot \sigma_{ab}$, we can compute the total cost due to the communication among all the collaborators

$$\varpi(\xi) = \sum_{k'=1}^{m}\sum_{k''=1}^{m}\int_{\max\limits_{1\le k\le m}\left(R_{1\xi(k)}^{k}{}^{-1}(h_k+R_{1\xi(k)}^{k}(0))\right)}^{s} e_{k'k''}(t)\chi_{\xi(k')\xi(k'')}^{k'k''} d(t) \tag{11}$$

At the end of the application, those collaborators that have left their default computing nodes may need to transfer some parts back or migrate back. The overall cost of such kind of data movement across the network can be obtained as follows

$$\varphi(\xi) = \sum_{k=1}^{m} v_k w_{\xi(k)1}^{k} \tag{12}$$

Thus, we can compute the total cost of transferring all essential data in the networks.

$$\begin{aligned} \eta(\xi) &= \omega(\xi) + \varpi(\xi) + \varphi(\xi) \\ &= \sum_{k=1}^{m} h_k w_{1\xi(k)}^{k} + \sum_{k=1}^{m} v_k w_{\xi(k)1}^{k} + \sum_{k'=1}^{m}\sum_{k''=1}^{m}\int_{\max\limits_{1\le k\le m}\left(R_{1\xi(k)}^{k}{}^{-1}(h_k+R_{1\xi(k)}^{k}(0))\right)}^{s} e_{k'k''}(t)\chi_{\xi(k')\xi(k'')}^{k'k''} d(t) \end{aligned} \tag{13}$$

According to the surveys on various practical systems and applications, we model the whole problem into two sub-problems which reflect the main application demands. Problem A constrains the uplink, downlink and collaborator-to-collaborator data transfer quality, as well as the collaborator migration time, and expects to minimize the total communication cost that depicts the expenditure on the employment of the current most precious resource as mentioned above. Problem B aims at maximizing the collaborator-to-collaborator communication quality, which is regarded as the inter-collaborator communication quality, under the constraints of the maximal acceptable cost, lowest tolerated quality between the user and the collaborator, as well as the maximal bearable collaborator migration time.

Problem A.

$$\min \quad \eta(\xi) = \sum_{k=1}^{m} h_k w_{1\xi(k)}^{k} + \sum_{k=1}^{m} v_k w_{\xi(k)1}^{k} + \sum_{k'=1}^{m}\sum_{k''=1}^{m}\int_{\max\limits_{1\le k\le m}\left(R_{1\xi(k)}^{k}{}^{-1}(h_k+R_{1\xi(k)}^{k}(0))\right)}^{s} e_{k'k''}(t)\chi_{\xi(k')\xi(k'')}^{k'k''} d(t)$$

$$s.t. \quad \frac{\sum\limits_{k=1}^{m}\sum\limits_{\bar{k}=1}^{m}\left(\dfrac{\int_{\max\limits_{1\le k\le m}\left(R_{1\xi(k)}^{k}{}^{-1}(h_k+R_{1\xi(k)}^{k}(0))\right)}^{s} r_{\xi(k)\xi(\bar{k})}^{k\bar{k}}(t)\, d(t)}{s-\max\limits_{1\le k\le m}\left(R_{1\xi(k)}^{k}{}^{-1}(h_k+R_{1\xi(k)}^{k}(0))\right)}\right)}{2m} \ge \Psi$$

$$\forall\, k \in [1,m], \quad \frac{\int_{\max\limits_{1\le k\le m}\left(R_{1\xi(k)}^{k}{}^{-1}(h_k+R_{1\xi(k)}^{k}(0))\right)}^{s} r_{1\xi(k)}^{k}(t)\, d(t)}{s-\max\limits_{1\le k\le m}\left(R_{1\xi(k)}^{k}{}^{-1}(h_k+R_{1\xi(k)}^{k}(0))\right)} \ge \delta^{+}$$

$$\forall\, k \in [1,m], \frac{\int_{\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k}_{\xi(k)1}(t)\, d(t)}{s-\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)} \ \ge\ \delta^{-}$$

$$\forall\, k \in [1,m], \max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right) \ \le\ \tau$$

$$\forall\, k \in [1,m], 1 \le \xi(k) \le n$$

$$1 \le k, k', k'' \le m$$

$$h_k,\, e_{k'k''},\, v_k,\, \chi^{k'k''}_{j'j''} \ge 0$$

$$\forall\, k,\quad \begin{cases} w^{k}_{j'j''} > 0 & j' \ne j'' \\ w^{k}_{j'j''} = 0 & j' = j'' \end{cases}.$$

Problem B.

$$\max \quad \phi(\xi) = \frac{\sum\limits_{k=1}^{m}\sum\limits_{\bar{k}=1}^{m}\left(\frac{\int_{\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k\bar{k}}_{\xi(k)\xi(\bar{k})}(t)\, d(t)}{s-\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}\right)}{2m}$$

$$s.t. \quad \sum_{k=1}^{m} h_k w^{k}_{1\xi(k)} + \sum_{k=1}^{m} v_k w^{k}_{\xi(k)1} + \sum_{k'=1}^{m}\sum_{k''=1}^{m}\int_{\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} e_{k'k''}(t)\chi^{k'k''}_{\xi(k')\xi(k'')} d(t) \le \Omega$$

$$\forall\, k \in [1,m], \frac{\int_{\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k}_{1\xi(k)}(t)\, d(t)}{s-\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)} \ \ge\ \delta^{+}$$

$$\forall\, k \in [1,m], \frac{\int_{\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)}^{s} r^{k}_{\xi(k)1}(t)\, d(t)}{s-\max\limits_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right)} \ \ge\ \delta^{-}$$

$$\forall\, k \in [1,m], \max_{1\le k\le m}\left({R^{k}_{1\xi(k)}}^{-1}(h_k+R^{k}_{1\xi(k)}(0))\right) \ \le\ \tau$$

$$\forall\, k \in [1,m], 1 \le \xi(k) \le n$$

$$1 \le k, k', k'' \le m$$

$$h_k,\, e_{k'k''},\, v_k,\, \chi^{k'k''}_{j'j''} \ge 0$$

$$\forall\, k,\quad \begin{cases} w^{k}_{j'j''} > 0 & j' \ne j'' \\ w^{k}_{j'j''} = 0 & j' = j'' \end{cases}.$$

4 Solving the Optimization Problems

The formulization in Section 3 proposes and formalizes the problem of how to optimally organize the collaborative computing and communication in the Internet; however, handling either the goal or the constraints of the optimization problem is of great computational complexity. Specifically, due to the randomness nature of the parameters in practice, as the increase of *m*, the scale of solution space for the problem expands exponentially.

So, we have to seek efficient heuristics to solve the problems.

Define two new operators ∇_x, ∇_x^c on ξ:

$$\nabla_{x^*}\xi = \begin{cases} \xi(x) & x \in [1, m] \wedge x \neq x^* \\ y & x = x^* \end{cases} \tag{14}$$

$$\nabla_{x^*}^c\xi = \begin{cases} \xi(x) & x \in [1, m] \wedge x \neq x^* \\ c & x = x^* \end{cases} \tag{15}$$

where $y \in [1, n_k]$, and c is a given integer between 1 and n_k.

Lemma 1: Assume integer $x \in [1, m]$ $\xi(x) \in [1, n_k]$, $\forall\ \xi$, if there exists a positive integer sequence λ : λ_1 , λ_2 , λ_3 … , in which $\lambda_i \in [1, m]$, such that $\forall\, i \geq 1$

$$\eta(\nabla_{\lambda_i}\nabla_{\lambda_{i-1}}\nabla_{\lambda_{i-2}}...\nabla_{\lambda_i}\xi) < \eta(\nabla_{\lambda_{i-1}}\nabla_{\lambda_{i-2}}...\nabla_{\lambda_i}\xi), \tag{16}$$

then the λ sequence is finite.

Proof: Compared to $\eta(\xi)$, $\eta(\nabla_{\lambda_i}\xi)$ will make every part of η changed: some may become smaller, while some may go larger. However, because for each operation of ∇ on ξ, $\eta(\nabla_{\lambda_i}\nabla_{\lambda_{i-1}}\nabla_{\lambda_{i-2}}...\nabla_{\lambda_1}\xi) < \eta(\nabla_{\lambda_{i-1}}\nabla_{\lambda_{i-2}}...\nabla_{\lambda_1}\xi)$, it can be guaranteed that there will be no loop in the ∇ operations.

Because there exists a minimum of η, the ∇_{λ_i} operations will end up with η getting a minimum (either the global minimum or local minimum). According to the process of generating the λ sequence, if ∇_{λ_i} operations will finally end, then we can obtain that λ sequence is finite.

If following the similar steps, we can obtain the lemma for Problem B.

Lemma 2: Assume integer $x \in [1, m]$, $\xi(x) \in [1, n_k]$, $\forall\ \xi$, if there exists a positive integer sequence λ : λ_1 , λ_2 , λ_3 … , in which $\lambda_i \in [1, m]$, such that $\forall\, i \geq 1$

$$\phi(\nabla_{\lambda_i}\nabla_{\lambda_{i-1}}\nabla_{\lambda_{i-2}}...\nabla_{\lambda_1}\xi) < \phi(\nabla_{\lambda_{i-1}}\nabla_{\lambda_{i-2}}...\nabla_{\lambda_1}\xi), \tag{17}$$

then the λ sequence is finite.

Theorem 1: Assume integer $x \in [1, m]$, $\xi(x) \in [1, n_k]$, $\forall\ \xi$, there exist two positive integer sequence u, v, which are both empty initially. Moreover, there exists another positive integer sequence λ, in which

$$\lambda_i = \begin{cases} i \bmod m & i \bmod m \neq 0 \\ m & i \bmod m = 0 \end{cases}. \tag{18}$$

For $i \geq 1$, if

$$\eta(\nabla_{\lambda_i}\nabla_{\mu_j}\nabla_{\mu_{j-1}}...\nabla_{\mu_1}\xi) < \eta(\nabla_{\mu_j}\nabla_{\mu_{j-1}}...\nabla_{\mu_1}\xi) \tag{19}$$

then add λ_i into sequence μ as a new element μ_{j+1}, and add i into sequence υ as a new element υ_{j+1}.

Then, $\forall\ \upsilon_j$, the following inequality holds

$$\max(\upsilon_j - \upsilon_{j-1}) \leq m - 1. \tag{20}$$

Proof: Suppose there exists an integer r, $r \geq 1$, such that

$$\max(\upsilon_r - \upsilon_{r-1}) > m-1.$$

So the following inequality will hold.

$$\eta(\nabla_{\mu_r}\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi) < \eta(\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi)$$

Because υ_r and υ_{r-1} are the adjacent elements in the sequence, according to the approach of constructing the sequences μ and υ, we have:

For $\forall$ integer $s \in (\upsilon_{r-1}, \upsilon_r)$,

$$\eta(\nabla_{\mu_{\lambda_s}}\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi) \geq \eta(\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi).$$

In particular, for $t = \upsilon_r - m$, because $\upsilon_{r-1} \leq t \leq \upsilon_r$, it can be obtained that

$$\eta(\nabla_{\mu_{\lambda_t}}\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi) \geq \eta(\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi).$$

However, because $\lambda_i = \begin{cases} i \bmod m & i \bmod m \neq 0 \\ m & i \bmod m = 0 \end{cases}$, $\lambda_{\upsilon_r} = \mu_r = \lambda_t$. This is contrary to the hypothesis made at the beginning, that is

$$\eta(\nabla_{\mu_r}\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi) < \eta(\nabla_{\mu_{r-1}}\nabla_{\mu_{r-2}}...\nabla_{\mu_1}\xi)$$

Therefore, $\forall\, \upsilon_j$ the inequality holds

$$\max(\upsilon_j - \upsilon_{j-1}) \leq m-1.$$

Theorem 2: Assume integer $x \in [1, m]$, $\xi(x) \in [1, n_k]$, $\forall\ \xi$, there exist two positive integer sequence u, v, which are both empty initially. Moreover, there exists another positive integer sequence λ, in which

$$\lambda_i = \begin{cases} i \bmod m & i \bmod m \neq 0 \\ m & i \bmod m = 0 \end{cases}. \tag{21}$$

For $i \geq 1$, if

$$\phi(\nabla_{\lambda_i}\nabla_{\mu_j}\nabla_{\mu_{j-1}}...\nabla_{\mu_1} f) < \phi(\nabla_{\mu_j}\nabla_{\mu_{j-1}}...\nabla_{\mu_1}\xi) \tag{22}$$

then add λ_i into sequence μ as a new element μ_{j+1}, and add i into sequence υ as a new element υ_{j+1}.

Then, $\forall\, \upsilon_j$, the following inequality holds

$$\max(\upsilon_j - \upsilon_{j-1}) \leq m-1. \tag{23}$$

The proof to the theorem is similar to that of Theorem 1.

As a matter of fact, Theorem 1 and Theorem 2, as well as Lemma 1 and Lemma 2, already clearly tell us the effective approach which can quickly approximate the optimal solution.

Simply put, we can first consider moving one collaborator while temporarily fixing the others; if there exist the qualified computing nodes which are more appropriate for the collaborator to migrate in the perspective of the optimization goal and the constraints, choose the most suitable one, and put the collaborator on it. Then we need to reestablish the relationship among the nodes and recompute the affected parameters. Following the similar way, we process each of the collaborators in a round-by-round manner. In this procedure, it is guaranteed by the theoretical proofs that the value of the goal function will continually decrease.

Formally, first we increase i, from (18) and (21) we can obtain new λ_i. Then through the method in Theorem 1 or Theorem 2, we can find the new elements μ_j and υ_j, as long as $\neg(\exists i \quad i-\upsilon_j \geq m)$. It is guaranteed by Theorem 1 and 2 that in this process, the value of the goal function can continually decrease. Thus, we can insistently drive such process. We call *the approach has run a round* if the value of λ_i has changed m times. More deeply, we can learn from the Theorem 1 and 2 that the value of the goal function will definitely decrease at lease once in one round, until $i-\upsilon_j \geq m$. And Lemma 1 and Lemma 2 have proven that this ceasing condition $i-\upsilon_j \geq m$ will finally hold. When ceased, the minimal (or near-minimal) value of the goal function can be obtained.

5 Evaluation and Results

5.1 Evaluation on Performance of the Heuristics

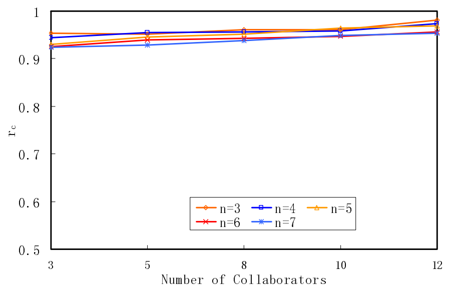
Experiment Methodology. We conduct a series of simulation experiments to evaluate the performance of the proposed heuristics in AORS (named AH) against the exhaustive search approach (named EA) which can generate the optimal organization scheme. The number of the candidate computing nodes for a collaborator follows the uniform distribution on the interval [1, *n*]. In the experiment, the time consumed to obtain the optimal solution via the exhaustive approach increases sharply due to the exponential expansion of the problem scale when the number of collaborators increases. For example, when *m*= 8, *n*=9, the time spent is 9035 seconds on a machine with a 2G HZ dual-core processor and 1GB RAM. So the intensive comparative experiments are taken in the circumstances of $m \leq 7$. For obtaining the data to draw one point in the figures, 1000 different networks are generated with the parameters illustrated in the graph; at last, the average of the data is computed and adopted.

In the evaluation, we focus mainly on two aspects. First, we want to check the overall status of the results generated by AH compared to the optimal results. Second, we hope to discover the impact of the number of collaborators and candidate computing nodes on the performance of AH. To quantify these purposes, two indicators are introduced. For Problem A, the indicator is the extent to which the cost minimized via AH approximates the optimal one. For Problem B, the factor is the extent to which the average goodput maximized by AH approximates the optimal result. Specifically, these two indicators are formulized as the ratio

$$r_c = 1-\frac{\eta(\xi^h)-\eta(\xi^*)}{\eta(\xi^*)} \qquad r_g = 1-\frac{\phi(\xi^*)-\phi(\xi^h)}{\phi(\xi^*)} \tag{24}$$

where ξ^h is the scheme obtained by AH, and ξ^* is the optimal scheme generated by EA.

Experiment Results. Fig. 3 illustrates the relationship among the number of collaborators, the number of candidate computing nodes and the ratio r_c. It presents four cases of different number of candidate computing nodes in the network, that is, *n* is equal to 3, 5, 8, 10 and 12, respectively. We can observe that the values of r_c of all the points are beyond 0.93. Although for a fixed number of collaborators the values of r_c are slightly different, the tendency of all the curves representing the different cases of distinct numbers of candidate computing nodes go upper right when more collaborators join in the CSCW computing.



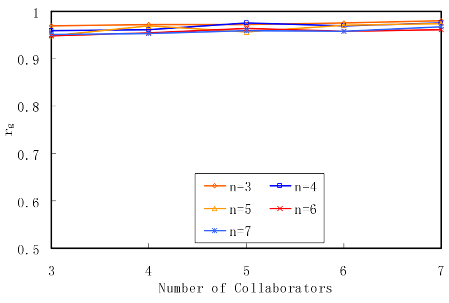


Fig. 3. Relationship among the number of collaborators, candidate computing nodes and r_c

Fig. 4. Relationship among the number of collaborators, candidate computing nodes and r_g

In Fig. 4, the studies for the AH performance on the optimization of average goodput are depicted. The number of candidate computing nodes, n, chooses the value of 3, 4, 5, 6 and 7. It is observed that the lowest value of the ratio r_g is 0.95 when n is equal to 3. While the number of candidate computing nodes increases, the ratio r_g changes slightly. However, similar to the situation in Fig. 3, if we study any circumstances with the given number of candidate computing nodes, in the application the more collaborators there are, the larger the ratio r_g is. Thus, from both Fig. 3 and Fig. 4, it is presented that even if the whole solution space expands exponentially as the number of collaborators increases, the scalability of the heuristics is nice.

5.2 Comparative Evaluation between AORS and Current Computing Style

Experiment Methodology. Instead of evaluating the AORS in some individual applications, we conduct extensive simulation experiments to compare the performance of AORS and the current CSCW computing style. In the experiment, various scenarios with distinct collaborator number 3, 6, 15, 25, 45 are thoroughly studied. In each scenario, we further divide and investigate the sub-scenarios which contain 3, 5, 8, 12, 16, 24, 30 candidate computing nodes, respectively. In each sub-scenario, 200 distinct networks are generated and the average results are computed and analyzed. In any topology, the default computing nodes are randomly distributed in leaf nodes, while the candidate computing nodes are distributed in the interior nodes. This is reasonable in that the default computing nodes, which are associated with users of the collaborative applications, are generally in the periphery of cyber-space; and as a public computing provider, a candidate computing node is usually expected to locate at a place with relatively abundant connections.

The cost of data transfer is concretely defined as the product of the size of transferred data and the hops between the source and destination. The sizes of data which needs to be moved from the default computing nodes to the candidate computing nodes follow the uniform distribution on the interval [1, 2000]MB. The sizes of data which need to be moved back from the candidate computing nodes to the default computing nodes follow the uniform distribution on the interval [0, 600] MB. The sizes of communication data among pairs of the collaborators follow the uniform distribution between 0 and 6000MB, in that there can be no communication between some pairs of collaborators in practice.

The goodput between the default computing nodes and candidate computing nodes, and the goodput between each pair of candidate computing nodes follow the uniform distribution between 4096kb/s and 80000kb/s. The hop between a candidate computing node and a default computing node follows the uniform distribution on the interval [5, 20]; the hops between pairs of candidate computing nodes follow the normal distribution with the mean of 10. Such settings are based on the observation of the current typical Internet communications.

Using the approaches in this paper above, before each session Optimal Unit of the AORS simulator computes the optimal organization scheme. During the session, Network Characteristic Detection Engine monitors the actual sizes of the data transferred; from the monitored information, on the one hand, the actual communication cost and goodput can be obtained, which reflect the actual performance of the current collaborative computing style; on the other hand, Optimal Unit puts the monitored information mentioned in Section 3, e.g., the goodput, onto the optimal scheme generated at the beginning of the session. So the metrics in the case of adopting the optimal scheme generated via AORS can be computed. Thus we can compare the communication quality and the cost achieved in the current collaborative computing style with those achieved in AORS.

Experiment Results. Fig. 5 depicts the relationship among the number of candidate computing nodes, the number of collaborators and the costs in the two environments. The vertical axis represents the ratio of the total network cost of the collaborative applications under AORS to the cost of exactly the same applications running within the same network environments in the current CSCW computing style. It is shown that all the points are beneath 1, indicating AORS always provides smaller cost than the current style. Although more collaborators in the collaborative environments tend to make the proportion of communication cost reduction smaller, it can be observed that with the number of candidate computing nodes increasing, the cost ratios always become smaller, which states that AORS reduces more communication cost.

Fig. 6 discloses the relationship among the number of candidate computing nodes, the group size and the average communication goodput among the collaborators in two environments. The vertical-axis indicates the ratio of the average goodput achieved in AORS to the goodput that the same application achieves in the current computing style.

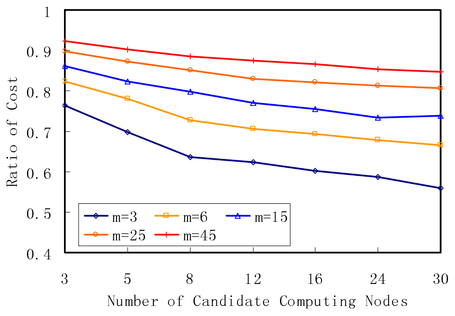


Fig. 5. Comparisons of the costs in AORS environments and current environments

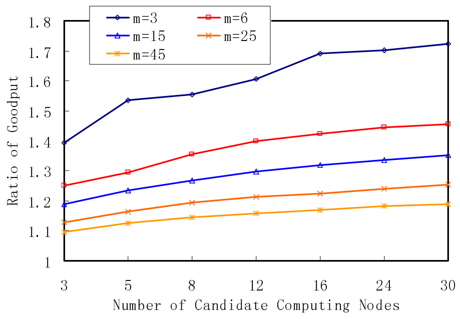


Fig. 6. Comparisons of the goodput in AORS environments and current enviroments

The observation that all the vertical-axis values are above 1 illustrates that AORS always can provide better goodput to CSCW applications. Although the ratio tends to decrease as the number of collaborators becomes large, the absolute value of the increased goodput from what the current computing style provides to what AORS brings constantly goes larger. On the other hand, in any circumstances with a given number of collaborators, it can be observed that larger amount of candidate computing nodes can also bring larger goodput for the collaborative applications.

6 Related Work

The most fruitful former studies related to organizing the computing resource in the large-scale networks are mainly in the domain of Grid computing. The need for the Grid system's ability to recognize the state of the resources is stressed in [15]. [12] realizes the significance of data location in job dispatching in Grid. It studies the jobs that use a single input file and assumes homogeneous sites with a simplified FIFO strategy. Like previous scheduling algorithms, the Close-to-Files algorithm [6] schedules a job to the processor close to a site where data is present. Assuming that a job only needs a single-file as input data, it uses an exhaustive algorithm to search across all the combinations of computing sites and data sites to find one with the minimal cost. The scheduling in the case of divisible loads is also analyzed [5]. To avoid performance deterioration due to the change of the Grid computing environments over time, adaptive scheduling techniques are proposed [15] [8].

However, these mechanisms can hardly be employed to address the problems in the current collaborative computing in the Internet. For example, there are few mechanisms to deal with organizing multiple distributed collaborative instances upon distributed computing facilities. Furthermore, the previously proposed approaches seldom consider the user-system interaction quality. The reasons can be ascribed to the orientation of the grid research. In contrast, this paper starts from analyzing the inherent characteristics of the current Internet communication, surveying the real problems in the Internet collaborative applications, and then models and solves the problem from both the performance perspective and the cost view.

7 Conclusion

Remote synchronous CSCW systems provide us with great opportunities in both our life and work. However, due to some intrinsic characteristics of the Internet, e.g., the relatively large latency and relatively low bandwidth, the collaborative applications are often affected or even unusable in present wide-area networks. Rather than making some improvements on the traditional CSCW computing style as former works did, this paper proposes an idea of changing the computing style, specifically, moving the proper collaborators of the application to appropriate computing nodes provided by the emerging Cloud computing environment. More importantly, this paper builds up a formal framework to optimally organize the collaborative computing and communication. A formulization of the framework is proposed, and an analytic theory of optimizing the communication quality and cost is developed. However, the solution space of the optimization problem scales up exponentially. To solve the problem, this paper develops two heuristics. The experiment results illustrate that the heuristics are effective and efficient. We also make extensive simulation experiments on comparing AORS with the current CSCW computing style. The results present that AORS improves communication quality, and in the meantime saves a lot of cost.

Acknowledgments. This work has been supported in part by the National High Technology Research and Development Program of China (Grant No.: 2009AA01Z151) and National Core-High-Base Major Project of China (Grant No.: 2009ZX01039-001-001). We would like to thank the anonymous reviewers for their helpful comments on this paper.

References

1. Baecker, R.M., Grudin, J., Buxton, W., Greenberg, S.: Readings in Human-Computer Interaction: Toward the Year 2000. Morgan Kaufmann Publishers, San Francisco (1995)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing (2009)
3. Shelley, G., Katchabaw, M.: Patterns of optimism for reducing the effects of latency in networked multiplayer games. In: FuturePlay (2005)
4. Wei, D.X., Jin, C., Low, S.H., Hegde, S.: FAST TCP: motivation, architecture, algorithms, performance. IEEE/ACM Trans. on Networking (2007)
5. Cardinale, Y., Casanova, H.: An evaluation of Job Scheduling Strategies for Divisible Loads on Grid Platforms. In: High Performance Computing and Simulation (2006)
6. Mohamed, H.H., Epema, D.H.J.: An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In: IEEE Conference on Cluster Computing (2004)
7. Gray, J.: Distributed Computing Economics. In: ACM Queue (2008)
8. Lee, L., Liang, C., Chang, H.: An adaptive task scheduling system for Grid Computing. In: 6th IEEE international Conference on Computer and information Technology (2006)
9. Gu, Y., Grossman, R.L.: UDT: UDP-based data transfer for high-speed wide area networks. International Journal of Computer and Telecommunications Networking (2007)
10. Braden, R., Zhang, L., Berson, S., Herzog, S., Jamin, S.: Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification. RFC 2208
11. Shacham, A., Monsour, B., Pereira, R., Thomas, M.: IP Payload Compression Protocol, RFC 3173
12. Foster, I., Ranganathan, K.: Decoupling computation and data scheduling in distributed data intensive applications. In: 11th Symposium on High Performance Distributed Computing (2002)
13. Agarwal, S., Lorch, J.R.: Matchmaking for Online Games and Other Latency-Sensitive P2P Systems. In: ACM SIGCOMM (2009)
14. Mazzini, G.: Asymmetric channel cooperative compression. IEEE Communications Letters (2008)
15. Othman, A., Dew, P., Djemame, K., Gourlay, K.: Adaptive grid resource brokering. In: IEEE Internetional Conference on Cluster Computing (2003)
16. Yang, L., Gani, A., Zakaria, O., Anuar, N.B.: Implementing lightweight reservation protocol for mobile network using crossover router & pointer forwarding scheme. In: WSEAS Conference on Electronics, Hardware, Wireless and Optical Communication (2009)
17. Amazon Elastic Compute Cloud Site, `http://aws.amazon.com/ec2/`

Chameleon-MAC: Adaptive and Self-$\star$ Algorithms for Media Access Control in Mobile Ad Hoc Networks[$\star$]

Pierre Leone[1], Marina Papatriantafilou[2], Elad M. Schiller[2], and Gongxi Zhu[2]

[1] University of Geneva, (Switzerland)
pierre.leone@unige.ch

[2] Chalmers University of Technology, Sweden
{ptrianta,elad,gongxi}@chalmers.se

Abstract. In mobile ad hoc networks (MANETs) mobile nodes do not have access to a fixed network infrastructure and they set up a communication network by themselves. MANETs require implementation of a wireless Medium Access Control (MAC) layer. Existing MAC algorithms that consider no mobility, solve the problem of eventually guaranteeing every node with a share of the communications bandwidth. In the context of MANETs, we ask: Is there an efficient MAC algorithm when mobility is considered?

MANETs are subject to transient faults, from which self-stabilizing systems can recover. The self-stabilization design criteria, and related concepts of self-$\star$, liberate the application designer from dealing with low-level complications, and provide an important level of abstraction. Whereas stabilization criteria are important for the development of autonomous systems, adaptation is imperative for coping with a variable environment. Adapting to a variable environment requires dealing with a wide range of practical issues, such as relocation of mobile nodes and changes to the motion patterns.

This work proposes the design and proof of concept implementation of an adapted MAC algorithm named Chameleon-MAC, which is based on a self-stabilizing algorithm by Leone et al., and uses self-$\star$ methods in order to further adapt its behavior according to the mobility characteristics of the environment. Moreover, we give an extensive treatment of the aspects and parameters that can bring the algorithm into the practical realm and we demonstrate documented behavior on real network studies (MICAz 2.4 GHz motes) as well as using simulation (TOSSIM [32]), showing improved overhead and fault-recovery periods than existing algorithms.

We expect that these advantages, besides the contribution in the algorithmic front of research, can enable quicker adoption by practitioners and faster deployment.

[$\star$] An extended version appears in [31] as a technical report. This work is partially supported by the ICT Programme of the European Union under contract number ICT-2008-215270 (FRONT'S).

1 Introduction

Mobile ad hoc networks (MANETs) are autonomous and self-organizing systems where mobile computing devices require networking applications when a fixed network infrastructure is not available or not preferred to be used. In these cases, mobile computing devices could set up a possibly short-lived network for the communication needs of the moment; in other words, an ad hoc network. MANETs are based on wireless communications that require implementation of a *Medium Access Control* (MAC) layer [40]. MAC protocols need to be robust and have high bandwidth utilization and low communication delay [38]. The analysis of radio transmissions in ad hoc networks [19] and the relocation analysis of mobile nodes [30] show that MAC algorithms that employ a scheduled access strategy, such as in TDMA, might have lower throughput than algorithms that follow a randomized strategy [such as slotted ALOHA [1], [34]. However, the scheduled approach offers greater predictability, which can facilitate fairness [21] and energy conservation. This work proposes the design and proof of concept implementation of an adapted MAC algorithm named Chameleon-MAC, which is based on a self-stabilizing algorithm [30] and uses new methods and techniques in order to further adapt its behavior according to the mobility characteristics of the environment. Through extensive treatment of the aspects and parameters of the new algorithm, we show that the algorithm in [30] can fit into a practical realm and we demonstrate documented behavior on real network studies (MICAz 2.4 GHz motes) as well as simulation (TOSSIM [32]), showing improved overhead and fault-recovery periods when compared with existing algorithms.

1.1 A Case for Adaptive Self-⋆ in MANETs

The dynamic and difficult-to-predict nature of mobile networks gives rise to many fault-tolerance issues and requires efficient solutions. MANETs, for example, are subject to transient faults due to hardware/software temporal malfunctions or short-lived violations of the assumed settings for modeling the location of the mobile nodes. Fault-tolerant systems that are *self-stabilizing* [10, 11] can recover after the occurrence of transient faults, which can cause an arbitrary corruption of the system state (so long as the program's code is still intact). The self-stabilization design criteria, and related concepts of self-⋆ [4], liberate the application designer from dealing with low-level complications, and provide an important level of abstraction. Consequently, the application design can easily focus on its task – and knowledge-driven aspects.

Whereas stabilization criteria (and the related self-⋆ concepts [4]) are important for the development of autonomous systems, adaptation is imperative for coping with a variable environment. In the context of MANETs, adapting to a variable environment requires dealing with a wide range of practical issues, such as relocation of mobile nodes, and changes to the rate or pattern by which they move [17]. Adaptation has additional aspects in the context of fault-tolerance, such as resource consumption [11, 16]. For example, high rates of churn and concurrent node relocations can significantly increase the overhead.

MANETs require MAC protocols that can provide robustness, high throughput, and low latency [38]. Broadly speaking, existing implementations of MAC protocols are based on carrier sense multiple access with collision avoidance (CSMA/CA) and cannot predictably guarantee these requirements [6, 7]. The design of the Chameleon-MAC algorithm takes a fresh look at time division multiple access (TDMA) algorithms. The Chameleon-MAC algorithm automatically adjusts the choice of timeslots according to the characteristics of a variable environment (just as the Chamaeleonidae lizards adapt their chromatophore cells according to their surrounding colors). The adaptive design of the Chameleon-MAC algorithm offers robustness, a greater degree of schedule predictability than existing CSMA/CA algorithms, and better resource adaptiveness than existing TDMA algorithms. Such advantages allow quicker technological developments and faster industrial deployment of MANETs.

1.2 Relocation Model

While MAC algorithms have been extensively studied in the context of MANETs, both numerically and empirically, analytical study has been neglected thus far. Until now, it has been speculated that existing MAC algorithms (that do not consider mobility of node) would perform well in MANETs. Alternatively, algebraic equations are used for modeling the kinetics of mobile nodes [5]. Kinetic models are difficult to analyze; it is hard to consider arbitrary behavior of mobile nodes and transient faults.

We consider an abstraction of model, named *relocations analysis*, which simplifies the proofs of impossibility results, lower bounds, and algorithms [30]. Models for relocation analysis focus on the location of mobile nodes rather than emphasizing the "movements of the mobile users" [as in 8]. The location of mobile nodes changes in discrete steps that are determined by a small set of parameters, rather than in continuous motion and by complex rules [as in kinetics model of 5]. The analytical results for this abstract model hold for a large set of concrete mobility models that can implement it. For example, a relocation analysis model can be used to estimate the throughput of MAC algorithms in realistic highway scenarios (cf. Section 5.4). Thus the studied model can improve the understanding of MANETs by facilitating an analytical study of its algorithms in system settings that can represent realistic scenarios.

1.3 Our Contribution

Algorithmic properties and systematic studies. We propose the Chameleon-MAC algorithm; an adaptive, self-$\star$ MAC algorithm that takes the scheduled approach and adapts to the variable environment of MANETs. We study the algorithm in an abstract and universal relocation model, and implement it in a real network of wireless MICAz 2.4 GHz motes. The relocation model allows us to compare the Chameleon-MAC algorithm to self-stabilizing MAC algorithms, such as the ones by Leone et al. [30], Herman-Tixeuil [21], slotted ALOHA [1] and p-persistent CSMA/CA (with and without back-off) [40]. This study and proof of concept includes an extensive treatment of the aspects and parameters that can bring the algorithm into the practical realm and demonstrate documented behavior on real network studies (MICAz 2.4 GHz motes) as well as using simulation (TOSSIM [32]). The study also shows that the Chameleon-MAC algorithm maintains higher throughput than [1, 21, 30, 40], while it also reveals other properties of interest.

Abstract-study-model properties and possibilities. Thus far, the designers of fault-tolerant algorithms for MANETs have considered a plethora of mobility models [8] for modeling the location of mobile nodes. Therefore, it is difficult to compare different MAC algorithms for MANETs. Our study model is an abstract relocation model used to analytically compare MAC algorithms [as in 30] or to conduct experimental study and evaluation as we do in this work. In [30], the properties of the model are briefly explored. The present work demonstrates further that the relocation model is sufficiently abstract to describe a variety of situations, and detailed enough to allow comparative studies of different algorithms.

Concrete-study-model scenarios and conclusions for practice. Based on the abstract relocation model, we show how to describe concrete mobility models for Vehicular Ad-Hoc Networks (VANETs) that, due to the nodes' mobility, have regular and transient radio interferences. We also systematically study the role of the local estimation of global parameters, because mobile nodes do not have direct knowledge about the model's global parameters and their impact on the throughput. Furthermore, we use the aforementioned models to study the algorithms with varying mobility parameters and demonstrate that the Chameleon-MAC algorithm adapts in variable environments that have radio interferences. Namely, the study shows that the Chameleon-MAC algorithm quickly recovers from radio interferences that occur due to the nodes' mobility in VANETs. Moreover, even in scenarios with no mobility, the Chameleon-MAC algorithm's throughput is higher than the one of existing implementations [1, 21, 30, 40]. We present measurements, on a real network of MICAz 2.4 GHz motes, that validate this observation, which we first obtain through TOSSIM [32] implementations.

2 Preliminaries

The system consists of a set of communicating entities, which we call *(mobile) nodes* (or *(mobile) motes*). Denote the set of nodes by P (processors) and every node by $p_i \in P$.

Synchronization. We assume that the MAC protocol is invoked periodically by synchronized *common pulses*. The term *(broadcasting) timeslot* refers to the period between two consecutive common pulses, t_x and t_{x+1}, such that $t_{x+1} = (t_x \bmod T) + 1$, where T is a predefined constant named the *frame size*, i.e., the number of timeslots in a TDMA frame (or broadcasting round).

Communication model. We consider a standard radio interference unit (such as `CC2420` [36]) that allows sensing the carrier and reading the energy level of the communication channel. Sometimes, we simplify the description of our algorithms and relocation models by considering concepts from graph theory. Nevertheless, the simulations consider a standard physical layer model [27].

At any instance of time, the ability of any pair of nodes to directly communicate is defined by the set, $N_i \subseteq P$, of *neighbors* that node $p_i \in P$ can communicate with directly. Wireless transmissions are subject to collisions and we consider the potential of nodes to interfere with each others' communications. We say that nodes $A \subseteq P$ *broadcast simultaneously* if the nodes in A broadcast within the same timeslot. We denote by $\mathcal{N}_i = \{p_k \in N_j : p_j \in N_i \cup \{p_i\}\} \setminus \{p_i\}$ the set of nodes that may interfere with p_i's communications when any nonempty subset of them, $A \subseteq \mathcal{N}_i : A \neq \emptyset$, transmit simultaneously with p_i. We call $\mathcal{N}_i$ the *interference neighborhood* of node $p_i \in P$, and $|\mathcal{N}_i|$ the *interference degree* of node $p_i \in P$.

3 Models for Relocation Analysis

We enhance the abstract relocation model of [30] using geometric properties that can limit the node velocity, unlike the earlier model [30]. In order to exemplify concrete models that can implement the proposed abstract one, we consider two mobility models that are inspired by vehicular ad hoc networks (VANETs).

3.1 Abstract Model Definitions

In [30], the authors use relocation steps, r_t, that relocate a random subset of nodes, P_{r_t}, and require that the number of nodes that relocate at time t is at most $\alpha|P|$, where $\alpha \in [0,1]$ (*relocation rate*) and time is assumed to be discrete. The relocation steps of [30] are random permutations of the locations of the nodes in P_{r_t}. Namely, within one relocation step, a mobile node can relocate to any location. Thus, the model in [30] does not limit the node velocity.

This work looks into different scenarios in which each mobile node randomly moves in the Euclidian plane $[0,1]^2$. Initially, n vertices are placed in $[0,1]^2$, independently and uniformly at random. The relocation steps in which a bounded number of nodes, $p_i \in P_r$, change their location, $p_i(t) = \langle x_i(t), y_i(t)\rangle$, to a random one, $p_i(t+1)$, that is at a distance of at most $\beta \geq \text{distance}(p_i(t), p_i(t+1))$, where t is a discrete time instant, $\beta \in [0,1]$ is a known constant named *(maximal) relocation distance* and $\text{distance}(p_i, p_j) = \sqrt{(y_j - y_i)^2 + (x_j - x_i)^2}$ is the geometric distance. We note that β limits the node speed in concrete mobile models that implement the proposed abstract model.

3.2 Concrete Model Definitions

Relocation analysis considers an abstract model that can represent several concrete mobility models. We consider two concrete mobility models that are inspired by vehicular ad hoc networks (VANETs). The models depict parallel and unison motion of vehicles that move at a constant speed and in opposite lanes. The first model assumes that the mobile nodes are placed on a grid and thus the radio interferences follow regular patterns. The second model considers vehicle clusters that pass by each other and thus their radio interferences are transient.

Regular radio interference. This model is inspired by traffic scenarios in which the vehicles are moving in parallel columns. We consider m rows, $r_0, r_1, \ldots, r_{m-1}$, of m nodes each ($m = 20$ in our experiments). Namely, $r_j = p_{j,0}, p_{j,1}, \ldots p_{j,m-1}$ is the j-th row. In this concrete model, the axes of the Euclidian plane $[0,1]^2$ are associated with scales that consider $\frac{1}{200}$ as its *distance units*. We assume that, at any time, the location, (x_i, y_i), of a mobile node, p_i, is aligned to the axes' scale. Namely, $200x_i$ and $200y_i$ are integers. Moreover, at any time, the distance between $p_{j,k}$ and $p_{j,k+1}$ (where $k, j \in [0, m-2]$) and the distance between $p_{j,k}$ and $p_{j+1,k}$ (where $j \in [0, m-2]$) is ψ distance units ($\psi = 10$ in our experiments). In the initial configuration, the nodes are placed on a $m \times m$ matrix (of parallel and symmetrical lines). At each relocation step, the nodes in the even rows move a constant distance, $speed > 0$, to the right. Moreover, when the nodes move too far to the right, they reappear on the left. Namely, the rightmost node in the rows, $p_{j,m-1}$, becomes the leftmost node in the row when the location of the $p_{j,m-2}$ is not to the right of the vertical line, $\ell_{up,down}$, that can be stretched between the location of that nodes $p_{up,m-1}$ and $p_{down,m-1}$, where $up = j - 1 \bmod m$ and $down = j + 1 \bmod m$. In Section 5, we show that the radio interferences of this model follow a regular pattern.

Transient radio interference. This model is inspired by two vehicle clusters that pass each other while moving in opposite lanes. The clusters are formed in a process that assures a minimal distance of ψ units between any two neighboring nodes ($\psi = 10$ in our experiments). Namely, we start by placing the nodes on a $m \times m$ matrix (of parallel and symmetrical lines), and let the nodes move toward their neighbors in a greedy manner that minimizes the distances among neighbors ($m = 20$ in our experiments). Once the clusters are formed, the cluster on the right moves towards the one on the left in a synchronized manner. At each relocation step, they reposition themselves to a location that is a constant number of units distance, $speed > 0$, from their current location. In Section 5, we show that the radio interferences of this model are transient.

4 The Chameleon-MAC Algorithm

An adaptive and self-$\star$ MAC algorithm for MANETs is presented. The algorithm is based on a non-adaptive, yet self-stabilizing, MAC algorithm for MANETs [30]. Leone et al. [30] explain how mobile nodes are able to learn some information about the success of the neighbors' broadcasts and base their algorithm on vertex-coloring; nodes avoid broadcasting in the timeslots in which their neighbors successfully broadcast. Namely, the algorithm assigns each node a color (timeslot) that is unique to its interference neighborhood.

The algorithm by Leone et al. [30] is self-stabilizing; however it is not adaptive. For example, the algorithm allows each node to broadcast at most once in every broadcasting round and assumes that the number of timeslots in the broadcasting rounds, T, is at least as large as the maximal size of the interference neighborhoods. Before presenting the Chameleon-MAC algorithm, we explain some details from [30] that are needed for understanding the new Chameleon-MAC algorithm.

4.1 Self-stabilizing MAC Algorithm for MANETs

Keeping track of broadcast history is complicated in MANETs, because of node relocations and transmission collisions. The algorithm by Leone et al. [30] presents a randomized solution that respects the recent history of the neighbors' broadcasts based on information that can be inaccurate. However, when the relocation rate or maximal relocation distance are not too great, the timeslots can be effectively allocated by the Leone et al. algorithm [30].

This is achieved using a randomized construction that lets every node inform its interference neighborhood on its broadcasting timeslot and allows the neighbors to record this timeslot as an occupied/unused one. The construction is based on a randomized competition among neighboring nodes that attempt to broadcast within the same timeslot. When there is a single competing node, that node is guaranteed to win. Namely, the node succeeds in informing its interference neighborhood on its broadcasting timeslot and letting the interference neighborhood mark its broadcasting timeslot as an occupied one. In the case where there are $x > 1$ competing nodes, there might be more than one "winner" (hence causing collisions). However, the expected number of winners will decrease after each subsequent round, because of the randomized construction. Why this procedure is guaranteed to converge is shown in [30]. For self-containment, we include the pseudo-code description of [30] in Fig. 1.

4.2 The Chameleon-MAC Algorithm

The Chameleon-MAC algorithm adapts to a variable environment in which relocation parameters, as well as the size of the interference neighborhoods, can change. Moreover, the algorithm adapts its behavior according to the state of the allocated resources, i.e., nodes adjust their timeslot allocation strategy according to the distribution of assigned timeslots. In order to do that, the algorithm employs new methods and techniques that achieve self-$\star$ properties [4].

Definition 1 is required for the presentation of the Chameleon-MAC algorithm.

Definition 1 (Timeslot properties). *Consider the interference neighborhood, $\mathcal{N}_i : p_i \in P$, and its timeslot assignment in $\mathcal{N}_i$, which we denote by $[\nu_{i,s}]$, where $\nu_{i,s} = \{p_j \in \mathcal{N}_i : s_j = s\}$, where s_j is the timeslot used by processor $p_j \in \mathcal{N}_i$. We say that timeslot $s \in [0, T-1]$ is:*

– *empty; if no neighbor in $\mathcal{N}_i$ transmits in timeslot s, i.e., $|\nu_{i,s}| = 0$,*

```
Variables and external functions
  $MaxRnd$ = number of rounds in the competition
  $s$:[0, $T$-1] ∪ {⊥} = next timeslot to broadcast or null (⊥)
  $competing$: boolean = competing for the channel
  $unique$: boolean = indicates a unique broadcasting timeslot
  $unused$[0,$T$-1]: boolean = marking unused timeslots
  MAC_fetch()/MAC_deliver(): layer interface
  broadcast/receive/carrier_sense/collision(): media primitives

Upon timeslot($t$)
  if $t = 0$ then (∗ On the first timeslot perform a test ∗)
    (∗ Was the previous broadcast unsuccessful? ∗)
    if $\neg$ $unique \vee s = \perp$ then
      (∗ Choose again the broadcasting timeslot ∗)
      $s \leftarrow$ get_random_unused()
    $unique \leftarrow$ false (∗ reset the state of unique ∗)
  $unused[t] \leftarrow$ true (∗ remove stale information ∗)
  (∗ Get a new message and sent it if everything is ok ∗)
  if $s \neq \perp \wedge t = s$ then send(MAC_fetch())

Upon receive($m$) do MAC_deliver($m$)

Function get_random_unused() (∗ selects unused timeslots ∗)
  return select_random({ $k \in [0,T\text{-}1]$ |$unused[k]$ = true })

Function send($m$)
  ($competing$, $unique$, $k$) ← (true, true, 1) (∗ start competing ∗)
  while $k \leq MaxRnd \wedge competing$ = true (∗ stop competing? ∗)
    with probability $2^{(-MaxRnd+k)}$ do
      broadcast($m$) (∗ try acquiring the channel ∗)
      $competing \leftarrow$ false (∗ quit the competition ∗)
    with probability $1 - 2^{(-MaxRnd+k)}$ do
      wait until the end of the competition time unit
    $k \leftarrow k + 1$

(∗ Stop competing when a neighboring node starts transmitting ∗)
Upon carrier_sense($t$) (∗ a neighbor is using timeslot $t$ ∗)
  if $competing$ = true then $unique \leftarrow$ false
  ($competing$, $unused[t]$) ← (false, false)

(∗ Collisions indicate unused timeslots ∗)
Upon reception_error($t$) do $unused[t] \leftarrow$ true
```

Fig. 1. The self-stabilizing MAC algorithm for MANETs by Leone et al. [30], code of processor p_i

- *congested; if more than one neighbor transmits in timeslot s, i.e., $|\nu_{i,s}| > 1$,*
- *well-used or unique; if exactly one neighbor transmits, i.e., $|\nu_{i,s}| = 1$, and*
- *unused; if no neighbor or more than one neighbor transmits, i.e., $|\nu_{i,s}| \neq 1$ and thus timeslot s is not properly assigned.*

Variable relocation rate and distance. MAC algorithms that follow the scheduled approach spend some of the communication bandwidth on allocating timeslots. We consider such algorithms and relocation models with constant parameters, in order to explain the existence of a trade-off between the throughput and the convergence time. Then, we explain how to balance this trade-off by adapting the algorithm's behavior according to the relocation parameters.

Obviously, there is a trade-off between the throughput, τ, and the communication overhead, h, because $\tau + h$ is bounded by the communication bandwidth. The throughput of MAC algorithms that follow the scheduled approach is guaranteed to converge within a bounded number of broadcasting rounds, ϱ. The rate by which they converge depends not only on the model's parameters, but also on the communication overhead, h. For example, the more bandwidth that the Leone et al. algorithm [30] is spending on competing for timeslots, the greater recovery each broadcasting round provides, and consequently, the shorter the convergence period is. This is a trade-off between the throughput, τ, and the convergence time, ϱ, which is settled by the communication overhead, h. Complementing the Leone et al. algorithm [30], the Chameleon-MAC algorithm balances this trade-off.

The Chameleon-MAC algorithm lets nodes adjust the communication overhead that they spend on competing for their timeslot according to an estimated number of unique timeslots (cf. Definition 1). The intuition behind the balancing technique is that the energy level becomes low as the nodes successfully allocate unique timeslots, because there are fewer collisions. Thus, after a convergence period, the nodes can reduce the communication overhead, h, by using less of the communication bandwidth on competing for their timeslots. Namely, they adjust the value of $MaxRnd$ of Figure 1. The eventual value of h depends on the parameters of the relocation model, because the nodes should not stop dealing with relocations. Namely, in order to cope with relocations, the nodes should always deal with events in which their broadcasting timeslots stop being unique due to the relocation of mobile nodes. Such relocations cause message collisions and higher energy level in the radio channels. The Chameleon-MAC algorithm copes with such changes by letting each node gradually adjust the amount of communication bandwidth it spends competing for timeslots.

Thus, when the relocation parameters are constants, the communication overhead, h, is set to a value that balances the trade-off between the throughput, τ, and convergence period, ϱ.

Variable size of interference neighborhoods. The Chameleon-MAC algorithm follows the scheduled approach because of its greater predictability compared to the random access approach. Namely, the nodes are allocated unique timeslots from the TDMA frame (cf. Definition 1). The Chameleon-MAC algorithm considers frames of a fixed size and adapts the timeslot allocation according to the size of interference neighborhoods complementing the Leone et al. algorithm [30].

The presentation of the techniques in use is simplified by considering the two possible cases: (1) the size of p_i's interference neighborhood, D_i, is not greater than the TDMA frame size, T, i.e., $D_i \leq T$ and (2) $D_i > T$ (where $p_i \in P$ is a node in the system). Notice that the procedures are concurrently executed and do not require explicit knowledge about D_i.

○ *$D_i \leq T$.* In this case, the Chameleon-MAC algorithm allocates to each node at least one timeslot. Node p_i publishes (in its data-packets) the number of timeslots, $T_i \in [0, T]$, that it uses in order to facilitate fairness requirements (i.e., attach this information to message m send in line 30 of Figure 1). The fairness criterion is that $|T_i - M_i| \leq 1$, where M_i is the median of timeslots allocated to the neighbors of p_i. The nodes apply this fairness criterion when deciding on the number of timeslots to use.

○ *$D_i > T$.* The algorithm employs two methods for dealing with cases in which there are more nodes in the neighborhood than TDMA timeslots; one for facilitating *fairness* and another for *contention control*.

Fairness can be facilitated by letting the nodes transmit at most ℓ consecutive data-packets before allowing other nodes to compete for their (already allocated) timeslots, where ℓ is a predefined constant.

The contention control technique is inspired by the p-persistent CSMA algorithm [40]. It assures that, at any time, an expected fraction of $p \in (0, 1)$ mobile nodes that do not have unique timeslots would compete for an unused one and $(1-p)$ of them would defer until a later broadcasting round (cf. Definition 1). i.e., each node that needs to change its broadcasting timeslot decides, with probability

p, to attempt to use the new timeslot in the broadcasting round that immediately follows [as in the Leone et al. algorithm [30]], and with probability $(1-p)$ it skips that broadcasting round. Namely, the function get_random_unused() in lines 23 to 24 of Figure 1, returns $\perp$ with probability $(1-p)$.

This process can be repeated for several timeslots until the node decides, with probability p, to attempt to use one of the unused timeslots.

The combination of the methods for facilitating fairness and contention control aims at allowing nodes to eventually acquire a unique timeslot. The successful reservation of unique timeslots for the duration of ℓ broadcasting rounds depends, of course, on the nodes' relocation.

Adaptive timeslot allocation strategy. The Chameleon-MAC algorithm employs two methods for adjusting the timeslot allocation according to the allocation's distribution.

∘ *Luby-algorithm-inspired method.* Luby [33] presents a round-based vertex-coloring algorithm. In each round, every uncolored vertex selects, uniformly at random, an unused color (cf. Definition 1). In Luby's settings, each vertex can accurately tell whether its color is well-used. This is difficult to achieve deterministically in wireless radio communications. However, the nodes can discover with probability $\Pr_{test} \in [\frac{1}{4}, \frac{1}{2}]$ whether their broadcasting timeslots are well-used [see [30], Section 4]. Therefore, a *non-uniform* selection of unused timeslots is the result of letting the nodes whose broadcasting timeslot is congested select, uniformly at random, an unused timeslot (as in line 24 of Figure 1). This is because a ratio of $1 - \Pr_{test}$ nodes cannot detect that their broadcasting timeslot is congested, and hence do not change their broadcasting timeslot.

The Luby-algorithm-inspired method prefers empty timeslots to congested ones when selecting a new broadcasting timeslot from the set of unused ones.

∘ *Žerovnik-algorithm-inspired method.* Žerovnik [41] accelerates the stabilization of vertex-coloring algorithms by favoring colors that are less represented in the neighborhood. Inspired by Žerovnik's technique, the Chameleon-MAC algorithm employs a method that aims at favoring timeslots that are less frequently used in the neighborhood when nodes are required to select a new timeslot. The heuristic favors the selection of congested timeslots with low energy level of the radio channel over congested timeslots with high level (cf. Definition 1).

In more detail, when node $p_i \in P$ considers a new timeslot, s_i, and s_i happens to be congested, then with probability $\Pr(s_i)$, p_i indeed uses timeslot s for broadcasting and with probability of $1 - \Pr(s_i)$ it does not change its broadcasting timeslot, where $\Pr(s_i) = 1 - \mathcal{O}(\exp(timeslot\_energy\_level[s_i]))$, the array $timeslot\_energy\_level_i[]$ stores the channel energy level that node p_i recorded during the latest broadcasting round. We note that the value of $timeslot\_energy\_level_i[t]$ is negative in our settings, because of the low energy used for transmission.

5 Experimental Evaluations

Throughput is a basic measure of communication efficiency. It is defined as the average fraction of time that the channel is employed for useful data propagation [40]. We study the relationship between the eventual throughput, τ, of the studied algorithms and the model parameters. The study considers: (1) simulated throughput, $\tau_{\mathsf{simulated}}$, obtained by simulation (TOSSIM [32]), (2) estimated throughput, $\tau_{\mathsf{estimated}}$, which is a MATLAB interpolation of the simulation results and (3) measured throughput, τ_{measured}, obtained by executing the Chameleon-MAC algorithm in a real network of wireless MICAz 2.4 GHz motes. The results of the experiment suggest that the throughput, τ, is a function that depends on the model parameters.

5.1 System Settings

The simulations were conducted in TOSSIM/TinyOS `2.1v` [32] and considered 400 motes. We have considered the default values for TOSSIM's radio model, which is based on the `CC2420` radio, used in the MICAz 2.4 GHz and telos [36] motes. In our simulations, we use timeslots of 2.5 ms.

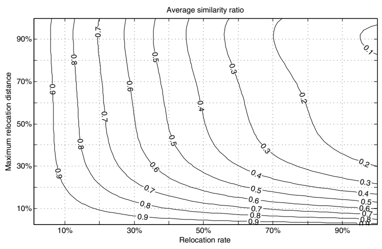


Fig. 2. Average similarity ratio is depicted as the function $ASR(\alpha, \beta)$ of the relocation rate, α (x-axis), and the maximal relocation distance β (y-axis). 5th polynomial interpolation was used.

Radio interference model. TOSSIM/TinyOS `2.1v` simulates the ambient noise and RF interference a mote receives, both from other radio devices as well as outside sources. In particular, it uses an SNR-based (Signal-to-Noise Ratio) packet error model with SINR-based (Signal to Interference-plus-Noise Ratio) interference [32]. Thus, common issues, such as the hidden terminal problem and the exposed terminal problem, are considered.

Simulation details. The simulation experiments considered 400 mobile nodes. Before the system execution, each node had selected its broadcasting timeslot uniformly at random from the set $[0, T-1]$ of timeslots. We then let the simulation run for a sufficiently long period (100 broadcasting rounds) during which the throughput, τ, eventually converges.

Empirical test-bed. This work demonstrates a proof of concept of the Chameleon-MAC algorithm using a real network of 35 MICAz 2.4 GHz motes, which are composed of the `ATmega128L` microcontroller and the `CC2420` radio chip. The motes were placed on a 5×7 matrix, the distance between two neighboring motes was 22 cm and the radio admission level was -90dBm [9, Section 6.3]. Throughout this particular experiment, the motes were not moved.

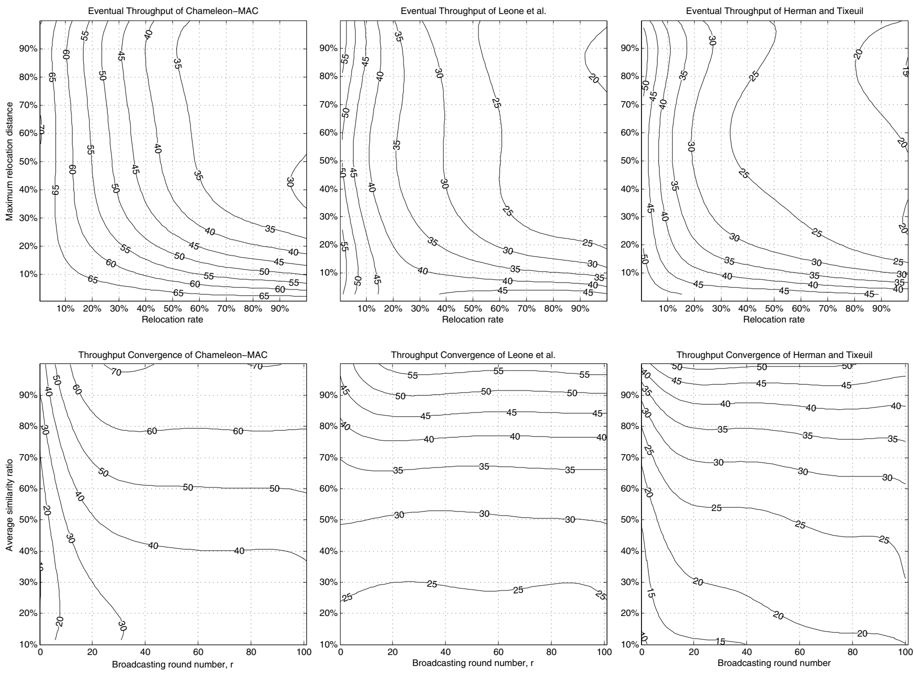


Fig. 3. Throughput of the Chameleon-MAC algorithm, τ, is compared to those of Leone et al. [30] and Herman-Tixeuil [21]. On the top row, the throughput, $\tau(\alpha, \beta)$, is depicted as a function of the relocation rate, α (x-axis), and the maximal relocation distance β (y-axis). On the bottom row, the throughput, $\tau(r, asr)$, is depicted as a function of the broadcasting round, r (x-axis), and ASR (y-axis). E.g., suppose that the ASR is 50%, then the throughput of the Chameleon-MAC algorithm will be about 20%, 30% and 40% within about 5, 10 and 20 broadcasting rounds, whereas the throughput of the Herman-Tixeuil [21] will be about 20% and 25% within about 10 and 60 broadcasting rounds. 5th polynomial interpolation was used.

The Chameleon-MAC algorithm assumes that each node has access to an accurate clock that can facilitate a common pulse. In our test-bed, the common pulse was emulated by starting every timeslot with a beacon message sent by the central mote, $p_{3,5}$.

5.2 Presentation

The relocation rate and distance are global parameters that define the relocation model. It is not clear how they can be estimated locally by the mobile nodes. Therefore, we consider the average similarity ratio (ASR) parameter, which can be calculated locally by the nodes [31].

The definition of the *average similarity ratio* (ASR) considers the unit disk graph (UDG); given the disk radius, $\chi \in [0, 1]$, we define $G_t = (P, E_t)$ as the UDG that the mobile nodes induce in time t, where $E_t = \{(p_i, p_j) | \text{distance}(p_i, p_j) \leq \chi\}$ is the set of edges at time t and $\text{distance}(p_i, p_j) = \sqrt{(y_j - y_i)^2 + (x_j - x_i)^2}$ is the geometric distance. The ASR is defined by a (non-aggregated) similarity ratio, $ASR_i = \frac{|N_i(G_t) \cap N_i(G_{t+1})|}{|N_i(G_t)|}$, that considers the neighbors that a mobile node maintains when relocating to a new neighborhood, where $N_i(G)$ is the set of p_i's neighbors in graph G. The nodes that are placed near the plane's borders have a lower (non-aggregated) similarity ratio upon relocation. Hence, $ASR = \text{average}(\{ASR_i | p_i \in P(t)\})$, considers the set of nodes, $P(t) = \{p_i \in P | \langle x_i(t), y_i(t) \rangle \in [\frac{1}{5}, \frac{4}{5}]^2\}$.

$ASR(\alpha, \beta)$ depicts ASR as a function of the relocation rate and distance; see Fig. 2. *Contour charts* present two parameter functions, e.g., $ASR(\alpha, \beta)$. They are often used in geographic maps to join points of the same height above sea level. Contour lines in Fig. 2 connect values of $ASR(\alpha, \beta)$ that are the same (see the text tags along the line).

The term *percentage of potential throughput* (PPT) is used in the presentation of the throughput: $\tau_{\mathsf{simulated}}$, $\tau_{\mathsf{estimated}}$ or τ_{measured}. It considers a TDMA frame in which only data-packets are sent and they are all received correctly. We define $bit_{potential}$ as the sum of bits in all the payloads of such a frame. Given a broadcasting round, r, and a node, $p_i \in P$, we define $bit_{actual}(r, i)$ as the number of bits in the payloads that were sent correctly by p_i or received correctly by p_i in r. Given a broadcasting round, r, we define PPT as $\frac{1}{|P(t_r)|} \sum_{p_i \in P(t_r)} \frac{bit_{actual}(r,i)}{bit_{potential}}$, where t_r is the time in which the broadcasting round begins and $P(t) = \{p_i \in P | \langle x_i(t), y_i(t) \rangle \in [\frac{1}{5}, \frac{4}{5}]^2\}$. We note that in our settings, the *maximum potential throughput* MPT is 76%, because of the time required for multiplexing and the transmission of the packet header/footer.

5.3 Throughput

The results of the experiments allow us to compare the CHAMELEON-MAC algorithm to Leone et al. [30], Herman-Tixeuil [21], slotted ALOHA [1] and p-persistent CSMA/CA (with and without back-off) [40] (Fig. 3, Fig. 4 and Fig. 5).

The simulated throughput values are compared as a function, $\tau(\alpha, \beta)$, of relocation rate and distance in Fig. 3-top. The charts show that the CHAMELEON-MAC algorithm has greater throughput than Leone et al. [30] and Herman-Tixeuil [21].

The simulated throughput values are depicted as a function, $\tau(r, asr)$, of the broadcasting rounds and the ASR

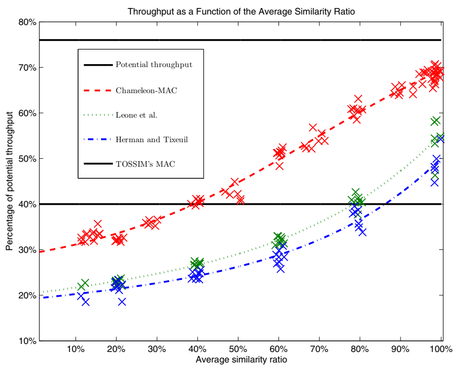


Fig. 4. Throughput of the CHAMELEON-MAC algorithm, τ, is compared to those of Leone et al. [30], Herman-Tixeuil [21], and CSMA/CA with back-off

in Fig. 3-bottom. The charts show that the Chameleon-MAC algorithm converges within 30 broadcasting rounds, whereas the Herman-Tixeuil [21] converge period may take more than 100 rounds.

The simulated throughput values are depicted as a function, $\tau(asr)$, of the ASR in Fig. 4. The chart shows that around $asr = 40\%$, there is a critical threshold, above which the throughput of the Chameleon-MAC algorithm is higher than TOSSIM's native MAC (CSMA/CA with back-off). We let TOSSIM's native MAC (CSMA/CA with back-off) represent the MAC algorithms that follow the randomized approach, such as slotted ALOHA [1] and p-persistent CSMA/CA without back-off [40], because TOSSIM's native MAC has greater throughput than slotted ALOHA [1] and p-persistent CSMA/CA without back-off [40]. The results show that the maximal eventual throughput of the Chameleon-MAC algorithm is $\tau = 70.4\%$ when the relocation model considers no mobility, which is 92.6% of the maximum potential throughput (MPT). In general, the interpolated function $\tau_{estimated}(asr) = 0.41(asr)^{1.353} + 0.29$ can be used for estimating the eventual throughput of the Chameleon-MAC algorithm when the ASR is constant.

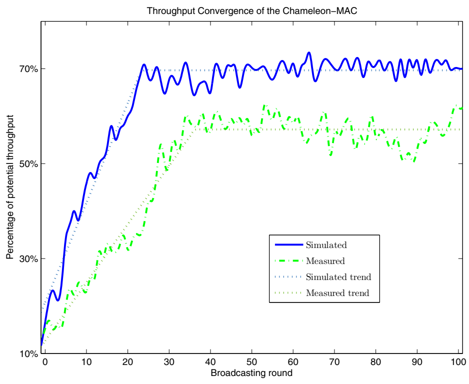


Fig. 5. Simulated and measured throughput (upper solid line and lower dash-dot line, respectively) together with their trends (dotted lines). The x-axes consider the broadcasting round number, r. The y-axes consider the percentage of potential throughput (PPT). The simulated throughput's trend is $\tau_{simulated\_trend}(r) = 0.02r + 0.19$ when $r < 25$, and 0.57 when $r \geq 25$. The measured throughput's trend is $\tau_{measured\_trend}(r) = 0.01r + 0.11$ when $r < 37$ and 0.70 when $r \geq 37$.

The simulated and measured throughput values are compared in Fig. 5. We note that the eventual measured throughput is about 82% of the simulated one with standard deviations of 2.93% and 1.33%, respectively, which are similar to the throughput deviation of TOSSIM's native MAC (CSMA/CA with back-off). Moreover, the measured convergence is 59%, slower than the simulated one. We attribute these differences in throughput, stability and convergence to the lack of detail in TOSSIM's radio interference model [32].

5.4 Validation of the Abstract Relocation Analysis in Concrete Mobility Models

We consider the two concrete mobility models for vehicular ad hoc networks (VANETs) that were presented in Section 2. They validate the abstract relocation model by showing that, in scenarios that exclude unrealistic situations, the

Chameleon-MAC algorithm can maintain throughput that is greater than the studied algorithms [1, 21, 30, 40] in the presence of regular and transient radio interference that occurs due to the nodes' mobility. Moreover, we show that the throughput of the Chameleon-MAC algorithm in VANETs is correlated to the function, $\tau_{estimated}(asr)$, that interpolates eventual throughput (cf. Section 5.3).

Regular radio interference. This model is inspired by traffic scenarios in which vehicles are moving in parallel columns. The experimental results are presented in Fig. 6. The experiment considers $speed \leq 20$ distance units per broadcasting round, because we exclude unrealistic situations.

The figure shows that within fewer than 30 broadcasting rounds, the throughput of the Chameleon-MAC algorithm is greater than TOSSIM's native MAC (CSMA/CA with back-off) and within about 50 broadcasting rounds, the throughput of the Chameleon-MAC algorithm converges to a value that is less than 5% from the eventual value and 50% greater than TOSSIM's native MAC.

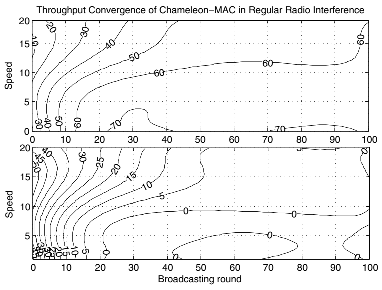


Fig. 6. Simulated and estimated throughput in VANETs with regular interference. The top chart presents the throughput, $\tau_{\mathsf{simulated}}(r, speed)$, as a function of the broadcasting rounds, r (x-axis), and the *speed* (y-axis) in unit distance per broadcasting round. The bottom chart presents the residual, $\tau_{\mathsf{simulated}}(r, speed) - \tau_{\mathsf{estimated}}(ASR(r))$, as a function of the broadcasting rounds, r (x-axis), and the *speed* (y-axis) in unit distance per broadcasting round.

Transient radio interference. This model is inspired by two vehicle clusters that pass each other while moving in opposite lanes. Fig. 7 considers such transient radio interference. The results show that, in scenarios that exclude unrealistic situations, the Chameleon-MAC algorithm maintains greater throughput than the studied MAC algorithms [1, 21, 30, 40] in the presence of transient radio interference and it can quickly recover from such faults.

Fig. 7 considers the simulated throughput, $\tau_{simulated}(r)$, estimated throughput, $\tau_{estimated}(r) = \tau_{estimated}(ASR(r))$ (cf. the interpolated eventual throughput for a constant ASR in Section 5.3), average similarity ratio, $ASR(r)$, and the residual, $residual(r) = \tau_{simulated}(r) - \tau_{estimated}(r) + 1$ (the addition of the constant 1 allows the reader to visually compare between $residual(r)$ and $ASR(r)$). The figure depicts an interference period during the broadcasting rounds $r \in [20, 40]$.

6 Discussion

Though some fundamental ad hoc networking problems remain unsolved or need optimized solutions, it is believed that ad hoc networks are not very far from

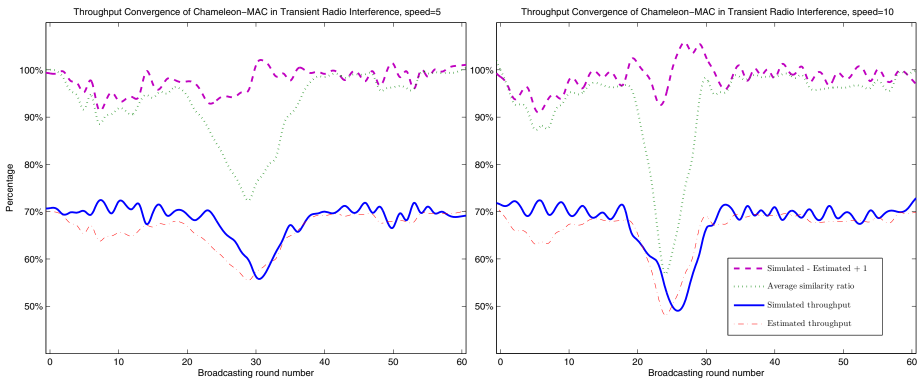


Fig. 7. Simulated throughput, $\tau_{simulated}(r)$, estimated throughput, $\tau_{estimated}(r)$, average similarity ratio, $ASR(r)$, and the residual, $residual(r) = \tau_{simulated}(r) - \tau_{estimated}(r) + 1$, are depicted by the solid line, dash-dot line, dotted line, and dashed line, respectively in a VANETs with transient interference. The left and right charts consider $speed = 5$ and $speed = 10$, respectively, of distance units per broadcasting round. The x-axes consider the broadcasting round number, r. The y-axes consider the percentage of the aforementioned functions.

being deployed on a large-scale commercial basis. For example, the TerraNet system allows voice and data communications via a peer-to-peer mobile mesh network comprised of modified handsets that were originally designed for cellular networks [39]. Vehicular ad hoc networks (VANETs) are a type of mobile network used for communication among mobile vehicles [22]. Nowadays, major automakers are developing future applications for vehicular communications on roads and highways. In sensor MANETs, the motes have joint monitoring tasks and the nodes' mobility can help achieve such tasks. Moreover, mobility can facilitate logistical services, say, by providing network connectivity between the MANETs' nodes and other communication endpoints, such as command-and-control units. Such services and applications can be constructed using the Virtual Node (VN) abstraction for MANETs [14, 15] without the use of fixed or stationary infrastructure.

6.1 Related Work

The `IEEE 802.11` standard is widely used for wireless communications. Nevertheless, the research field of MAC protocols is very active and has hundreds of publications per year. In fact, the `IEEE 802.11` amendment, `IEEE 802.11p`, for wireless access in vehicular environments (WAVE), is scheduled for November 2010. It was shown that the standard's existing implementations cannot guarantee channel access before a finite deadline [6, 7]. Therefore, VANETs' real-time applications cannot predictably meet their deadlines. The design of the Chameleon-MAC algorithm facilitates MAC protocols that address important issues, such as predictably [6, 7], fairness [21] and energy conservation.

The *algorithmic* study of MAC protocols is a research area that considers the analysis of distributed MAC algorithms using theoretical models that represent the communication environment. The scope of this work includes distributed MAC algorithms for wireless ad hoc networks and adaptive self-stabilization.

MAC **algorithms.** ALOHAnet and its synchronized version Slotted ALOHA [1] are pioneering wireless systems that employ a strategy of "random access". Time division multiple access (TDMA) is another early approach where nodes transmit one after the other, each using its own timeslot, say, according to a defined schedule. The analysis of radio transmissions in ad hoc networks [19] and the relocation analysis of mobile nodes [30] show that there are scenarios in which MAC algorithms that employ a scheduled access strategy have lower throughput than algorithms that follow the random access strategy. However, the scheduled approach offers greater predictability, which can facilitate fairness [21] and energy conservation.

∘ *Non-self-stabilizing MAC algorithms for wireless ad hoc networks.* Wattenhofer's fruitful research line of *local algorithms* considers both theoretical [18, 20, and references therein] and practical aspects of MAC algorithms [43, and references therein] and the related problem of clock synchronization [28, and references therein]. For example, the first partly-asynchronous self-organizing local algorithm for vertex-coloring in wireless ad hoc networks is presented in [37]. However, this line currently does not consider MANETs. An example of a self-organizing MAC protocol that considers a single hop and no mobility is [35].

∘ *Non-self-stabilizing MAC algorithms for MANETs.* An abstract MAC layer was specified for MANETs in [24]. The authors mention algorithms that can satisfy their specifications (when the mobile nodes very slowly change their locations). MAC algorithms that use complete information about the nodes' trajectory are presented in [42, and references therein] (without considering the practical issues that are related to maintaining the information about the nodes' trajectories). A self-organizing TDMA algorithm that maintains the topological structure of its communication neighborhood is considered in [6, 7]. The authors use computer simulations to study the protocol, assuming that nodes have access to a global positioning system, and that transmission collisions can be detected.

∘ *Self-stabilizing MAC algorithms for wireless ad hoc networks.* Two examples of self-stabilizing TDMA algorithms are presented in [21, 23]. The algorithms are based on vertex-coloring and consider ad hoc networks (in which the mobile nodes may move very slowly). Recomputation and floating output techniques [11, Section 2.8] are used for converting deterministic local algorithms to self-stabilizing ones in [29]. However, deterministic MAC algorithms are known to be inefficient in the context of MANETs, as shown in [30]. There are several other proposals for self-stabilizing MAC algorithms for sensor networks [such as 2, 3, 25, 26]; however, none of them considers MANETs.

Adaptive self-stabilization. In the context of self-stabilization, adaptive resource consumption was considered for communication [16] and memory [cf. Update algorithm in [11]. Namely, after the convergence period, the algorithm's resource consumption is asymptotically optimal [as in [11] or poly-logarithmic times the optimal [as in [16]. In the context of MANETs, the idea of self-stabilizing Virtual Node (VN) [13–15] and tokens that perform random walks in order to adapt to the topological changes [17] was widely adopted by the theory and practice of MANETs. These concepts can implement a wide range of applications, such as group communication services [17], and traffic coordination and safety [44].

6.2 Conclusions

The large number of MAC algorithms and protocols for ad hoc networks manifests both the interest in the problem, as well as the lack of some commonly accepted and well understood methods that enable balancing the large number of trade-offs. The proposed adaptive and self-⋆ Chameleon-MAC algorithm and the systematic study presented here shed light on and bridge this situation: The extensive study, using the TOSSIM [32] simulation and the actual MICAz 2.4 GHz mote platform, of the aspects and parameters that are of interest in the practical realm, offer a gnomon to facilitate adoption and deployment in practice.

The study includes a wide, multi-dimensional range of mobility situations and demonstrates that the Chameleon-MAC algorithm is an alternative with improved overhead and fault-recovery periods compared to existing algorithms in the literature and in practice [1, 21, 30, 40]. To highlight a small example, in scenarios that exclude unrealistic situations, the Chameleon-MAC algorithm maintains greater throughput than the studied ones [1, 21, 30, 40], including TOSSIM's native MAC (CSMA/CA with back-off).

As a side-result, this work demonstrates that the relocation model is sufficiently abstract to describe a variety of situations (from limited mobility scenarios to concrete mobility models for VANETs) and detailed enough to allow comparative studies of different algorithms.

Another contribution in the paper, which was an intermediate step in our study and which can be of independent interest, is the analysis of the role of the local estimation (by the nodes) of the global model parameters and their impact on the algorithms. Interestingly, we discovered the average similarity ratio (ASR) that can estimate well the throughput of the studied algorithms.

References

[1] Abramson, N.: Development of the ALOHANET. IEEE Transactions on Information Theory 31(2), 119–123 (1985)

[2] Arumugam, M., Kulkarni, S.: Self-stabilizing deterministic time division multiple access for sensor networks. AIAA Journal of Aerospace Computing, Information, and Communication (JACIC) 3, 403–419 (2006)

[3] Arumugam, M., Kulkarni, S.S.: Self-stabilizing deterministic TDMA for sensor networks. In: Chakraborty, G. (ed.) ICDCIT 2005. LNCS, vol. 3816, pp. 69–81. Springer, Heidelberg (2005)
[4] Berns, A., Ghosh, S.: Dissecting self-* properties. In: SASO, pp. 10–19. IEEE Computer Society, Los Alamitos (2009)
[5] Bettstetter, C.: Smooth is better than sharp: a random mobility model for simulation of wireless networks. In: Meo, M., Dahlberg, T.A., Donatiello, L. (eds.) MSWiM, pp. 19–27. ACM, New York (2001)
[6] Bilstrup, K., Uhlemann, E., Ström, E.G., Bilstrup, U.: Evaluation of the IEEE 802.11p MAC method for vehicle-to-vehicle communication. In: VTC Fall, pp. 1–5. IEEE, Los Alamitos (2008)
[7] Bilstrup, K., Uhlemann, E., Ström, E.G., Bilstrup, U.: On the ability of the 802.11p MAC method and STDMA to support real-time vehicle-to-vehicle communication. EURASIP Journal on Wireless Communications and Networking 2009, 1–13 (2009)
[8] Camp, T., Boleng, J., Davies, V.: A survey of mobility models for ad hoc network research. Wireless Communications and Mobile Computing 2(5), 483–502 (2002)
[9] Chipcon Products from Texas Instruments, Texas Instruments, Post Office Box 655303, Dallas, Texas 75265. 2.4GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver (2008)
[10] Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. ACM Commun. 17(11), 643–644 (1974)
[11] Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
[12] Dolev, S. (ed.): ALGOSENSORS 2009. LNCS, vol. 5804. Springer, Heidelberg (2009)
[13] Dolev, S., Gilbert, S., Lynch, N.A., Schiller, E., Shvartsman, A.A., Welch, J.L.: Brief announcement: virtual mobile nodes for mobile ad hoc networks. In: Chaudhuri, S., Kutten, S. (eds.) PODC, p. 385. ACM, New York (2004)
[14] Dolev, S., Gilbert, S., Lynch, N.A., Schiller, E., Shvartsman, A.A., Welch, J.L.: Virtual mobile nodes for mobile ad hoc networks. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 230–244. Springer, Heidelberg (2004)
[15] Dolev, S., Gilbert, S., Schiller, E., Shvartsman, A.A., Welch, J.L.: Autonomous virtual mobile nodes. In: DIALM-POMC, pp. 62–69 (2005)
[16] Dolev, S., Schiller, E.: Communication adaptive self-stabilizing group membership service. IEEE Trans. Parallel Distrib. Syst. 14(7), 709–720 (2003)
[17] Dolev, S., Schiller, E., Welch, J.L.: Random walk for self-stabilizing group communication in ad hoc networks. IEEE Trans. Mob. Comput. 5(7), 893–905 (2006)
[18] Goussevskaia, O., Wattenhofer, R., Halldórsson, M.M., Welzl, E.: Capacity of arbitrary wireless networks. In: INFOCOM, pp. 1872–1880. IEEE, Los Alamitos (2009)
[19] Haenggi, M.: Outage, local throughput, and capacity of random wireless networks. Trans. Wireless. Comm. 8(8), 4350–4359 (2009)
[20] Halldórsson, M.M., Wattenhofer, R.: Wireless communication is in APX. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555, pp. 525–536. Springer, Heidelberg (2009)
[21] Herman, T., Tixeuil, S.: A distributed TDMA slot assignment algorithm for wireless sensor networks. In: Nikoletseas, S.E., Rolim, J.D.P. (eds.) ALGOSENSORS 2004. LNCS, vol. 3121, pp. 45–58. Springer, Heidelberg (2004)
[22] Holfelder, W., Johnson, D.B., Hartenstein, H., Bahl, V. (eds.): Proceedings of the Third International Workshop on Vehicular Ad Hoc Networks, VANET 2006. ACM, New York (2006)

[23] Jhumka, A., Kulkarni, S.S.: On the design of mobility-tolerant TDMA-based media access control (MAC) protocol for mobile sensor networks. In: Janowski, T., Mohanty, H. (eds.) ICDCIT 2007. LNCS, vol. 4882, pp. 42–53. Springer, Heidelberg (2007)

[24] Kuhn, F., Lynch, N.A., Newport, C.C.: The abstract MAC layer. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 48–62. Springer, Heidelberg (2009)

[25] Kulkarni, S.S., Arumugam, M.U.: Sensor Network Operations, chapter SS-TDMA: A self-stabilizing MAC for sensor networks. IEEE Press, Los Alamitos (2006)

[26] Lagemann, A., Nolte, J., Weyer, C., Turau, V.: Mission statement: Applying self-stabilization to wireless sensor networks. In: Proceedings of the 8th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze" (FGSN 2009), pp. 47–49 (2009)

[27] Lee, H., Cerpa, A., Levis, P.: Improving wireless simulation through noise modeling. In: Abdelzaher, T.F., Guibas, L.J., Welsh, M. (eds.) IPSN, pp. 21–30. ACM, New York (2007)

[28] Lenzen, C., Locher, T., Sommer, P., Wattenhofer, R.: Clock Synchronization: Open Problems in Theory and Practice. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM 2010. LNCS, vol. 5901, pp. 61–70. Springer, Heidelberg (2010)

[29] 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)

[30] Leone, P., Papatriantafilou, M., Schiller, E.M.: Relocation analysis of stabilizing MAC algorithms for large-scale mobile ad hoc networks. In: Dolev [12], pp. 203–217

[31] Leone, P., Papatriantafilou, M., Schiller, E.M., Zhu, G.: CHAMELEON-MAC: Adaptive and stabilizing algorithms for media access control in mobile ad-hoc networks. Technical Report 2010:02, Chalmers University of Technology (2010), ISSN 1652-926X

[32] Levis, P., Lee, N., Welsh, M., Culler, D.E.: TOSSIM: accurate and scalable simulation of entire tinyos applications. In: Akyildiz, I.F., Estrin, D., Culler, D.E., Srivastava, M.B. (eds.) SenSys, pp. 126–137. ACM, New York (2003)

[33] Luby, M.: Removing randomness in parallel computation without a processor penalty. J. Comput. Syst. Sci. 47(2), 250–286 (1993)

[34] Metzner, J.J.: On Improving Utilization in ALOHA Networks. IEEE Transactions on Communications 24(4), 447–448 (1976)

[35] Patel, A., Degesys, J., Nagpal, R.: Desynchronization: The theory of self-organizing algorithms for round-robin scheduling. In: SASO, pp. 87–96. IEEE Computer Society, Los Alamitos (2007)

[36] Polastre, J., Szewczyk, R., Culler, D.E.: Telos: enabling ultra-low power wireless research. In: IPSN, pp. 364–369. IEEE, Los Alamitos (2005)

[37] Schneider, J., Wattenhofer, R.: Coloring unstructured wireless multi-hop networks. In: Tirthapura, S., Alvisi, L. (eds.) PODC, pp. 210–219. ACM, New York (2009)

[38] Specifications ASTM, E2213-03. Standard Specification for Telecommunications and Information Exchange between Roadside and Vehicle Systems - 5 GHz Band Dedicated Short Range Communications Medium Access Control and Physical Layer (September 2003)

[39] Stuedi, P., Alonso, G.: Wireless ad hoc VoIP. In: Workshop on Middleware for Next-generation Converged Networks and Applications, Newport Beach, California, USA (November 2007)

[40] Takagi, H., Kleinrock, L.: Throughput analysis for persistent CSMA systems. IEEE Transactions on Communications 33(7), 627–638 (1985)

[41] Žerovnik, J.: On the convergence of a randomized algorithm frequency assignment problem. Central European Journal for Operations Research and Economics (CEJORE) 6(1-2), 135–151 (1998)

[42] Viqar, S., Welch, J.L.: Deterministic collision free communication despite continuous motion. In: Dolev [12], pp. 218–229

[43] Wattenhofer, R.: Theory Meets Practice, It's about Time. In: 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Czech Republic (2010)

[44] Wegener, A., Schiller, E.M., Hellbrück, H., Fekete, S.P., Fischer, S.: Hovering data clouds: A decentralized and self-organizing information system. In: de Meer, H., Sterbenz, J.P.G. (eds.) IWSOS 2006. LNCS, vol. 4124, pp. 243–247. Springer, Heidelberg (2006)

A Comparative Study of Rateless Codes for P2P Persistent Storage

Heverson B. Ribeiro[1] and Emmanuelle Anceaume[2]

[1] IRISA / INRIA, Rennes Bretagne-Atlantique, France
heverson.ribeiro@inria.fr
[2] IRISA / CNRS UMR 6074, France
emmanuelle.anceaume@irisa.fr

Abstract. This paper evaluates the performance of two seminal rateless erasure codes, LT Codes and Online Codes. Their properties make them appropriate for coping with communication channels having an unbounded loss rate. They are therefore very well suited to peer-to-peer systems. This evaluation targets two goals. First, it compares the performance of both codes in different adversarial environments and in different application contexts. Second, it helps understanding how the parameters driving the behavior of the coding impact its complexity. To the best of our knowledge, this is the first comprehensive study facilitating application designers in setting the optimal values for the coding parameters to best fit their P2P context.

Keywords: peer-to-peer, p2p, storage, rateless codes, fountain codes, LT codes, online codes, data persistency.

1 Introduction

Typical applications such as file sharing or large scale distribution of data all have in common the central need for durable access to data. Implementing these applications over a peer-to-peer system allows to potentially benefit from the very large storage space globally provided by the many unused or idle machines connected to the network. In this case, however, great care must be taken since peers can dynamically and freely join or leave the system. This has led for the last few years to the deployment of a rich number of distributed storage solutions. These architectures mainly differ according to their replication scheme and their failure model. For instance, architectures proposed in [10,13] rely on full replication, the simplest form of redundancy. Here, full copies of the original data are stored at several peers, which guarantees optimal download latency. However, the storage overhead and the bandwidth for storing new replicas when peers leave may be unacceptable, and thus tend to overwhelm the ease of implementation and the low download latency of this replication schema. This has motivated the use of fixed-rate erasure coding as in [2,5]. Fixed-rate erasure codes such as Reed-Solomon and more recently Tornado codes mainly consist in adding a specific amount of redundancy to the fragmented original data and storing these redundant fragments at multiple peers. The amount of redundancy, computed according to the assumed failure model, guarantees the same level of availability as full replication, but with an overhead of only a fraction of the original data, and coding and decoding operations in linear time (only guaranteed by Tornado codes). Unfortunately, this replication scheme is intrinsically not adapted to unbounded-loss channels in contrast to Rateless codes (also called Fountain codes). As a class of erasure codes, they provide natural resilience to losses, and therefore are fully adapted to dynamic systems. By being rateless, they give rise to the generation of random, and potentially unlimited number of uniquely coded symbols, which make them fully adapted to dynamic systems such as peer-to-peer systems. Two classes of rateless erasure codes exist. Representative of the first class is the LT coding proposed by Luby [6], and representatives of the second one are Online codes proposed by Maymounkov [7] and Raptor codes by Shokrollahi [12]. The latter class differs from the former one by the presence of a pre-coding phase. A data storage architecture based on a compound of Online coding and full replication has been recently proposed in [8]. Previous analyses have shown the benefit of hybrid replication over full replication in terms of data availability, storage overhead, and bandwidth usage [4,9]. However, for the best of our knowledge, no experimental study comparing the two classes of fountain codes has ever been performed.

The objective of the present work is to provide such a comparison. Specifically, we propose to compare the experimental performance of both LT and Online codes. Note that Raptor codes [12] could have been analysed as a representative of the second class of fountain codes; however because part of their coding process relies on LT codes, we have opted for Online codes. This evaluation seeks not only to compare the performance of both codes in different adversarial environments, and in different application contexts (which is modeled through different size of data), but also to understand the impact of each coding parameter regarding the space and time complexity of the coding process. As such, this work should be considered, for the best of our knowledge, as the first comprehensive guideline that should help application designers to configure these codes with optimal parameters values.

The remainder of this paper is structured as follows: In Section 2 we present the main features of erasure codes. Section 3 presents the experiments performed on LT and Online codes to evaluate their recovery guarantees, their decoding complexity in terms of XOR operations, and their adequacy to varying size of input data. This section is introduced by a brief description of the architecture in which these experiments have been run. Section 4 concludes.

2 Backgroung on Rateless Erasure Codes

As previously said in the introduction, the main advantages of rateless erasure codes over traditional erasure codes are first that the ratio between input and encoded symbols is not fixed which eliminates the need for estimating the loss-rate beforehand, and second that the encoded symbols can be independently generated on-the-fly. The following section emphasizes the main features of both LT and Online codes. An in-depth description can be found in the respective original papers [6,7].

2.1 Principles of LT Rateless Codes

Coding Process. The LT coding process [6] consists in partitioning the data item or message to be coded into $k = n/l$ equal size symbols (also called input blocks), where n is the size of the data-item, and l a parameter that is typically chosen based on the packet payload to be transmitted. Each encoded symbol c_i (also called check block) is generated by *(i)* choosing a *degree* d_i from a particular degree distribution (see below), *(ii)* randomly choosing d_i distinct input symbols (called neighbors of c_i) among the k input symbols, and *(iii)* combining the d_i neighbors into a check block c_i by performing a bitwise XOR operation. Note that the degree and the set of neighbors information d_i associated with each check block c_i needs to be known during the decoding process. There are different ways to communicate this information during the coding process. For instance, a deterministic function may be used to define the degree of each check block and then both coder and decoder can apply the same function in order to recreate the information [6]. In the following, any check block cb_i is represented as a pair $\langle c_i, x_i \rangle$, where c_i is the check block generated by combining d_i neighbors and x_i is the set of the d_i combined neighbors. Figure 1(a) shows the LT coding process represented by a Tanner graph [14]. Specifically, the bipartite graph is such that the first set of vertices represents input symbols k_1, k_2 and k_3 and the second set represents the generated check blocks cb_A, cb_B, cb_C, cb_D and cb_E. Using the generation procedure described here above, a potentially infinite number of check blocks can be generated. Later on, we provide the lower bound CB_0 on the number of check blocks that need to be generated in order to guarantee the success of LT coding with high probability.

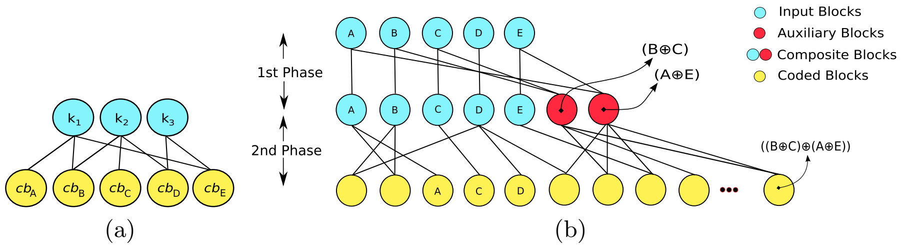


Fig. 1. (a) Check blocks $cb_A = k_1$, $cb_B = k_1 \oplus k_2$, $cb_C = k_2$, $cb_D = k_2 \oplus k_3$, and $cb_E = k_1 \oplus k_3$ coded from input symbols k_1, k_2, and k_3. (b) Online two-phase coding process

Decoding Process. The key idea of the decoding process is to rebuild the Tanner graph based on the set of received check blocks. Upon receipt of check blocks, the decoder performs the following iterative procedure: *(i)* Find any check block cb_i with degree equal to one (i.e. each *degree-one* check block cb_i has only one input symbol k_i as neighbor), *(ii)* copy the data c_i of $cb_i\langle c_i, x_i\rangle$ to k_i. (i.e. neighbor k_i of check block cb_i is an exact copy of data c_i), *(iii)* remove the edge between cb_i and k_i, and *(iv)* execute a bitwise XOR operation between k_i and any remaining check block cb_r that has k_i as neighbor ($cb_r = cb_r \oplus k_i$), and remove the edge between cb_r and k_i. These four steps are repeated until all k input symbols are successfully recovered.

Soliton Degree Distributions. The key point of LT codes is the design of a proper degree distribution. The distribution must guarantee the success of the LT process by using first, as few as possible check blocks to ensure minimum redundancy among them and second, an average degree as low as possible to reduce the average number of symbol operations to recover the original data. The first approach proposed by Luby was to rely on the *Ideal Soliton Distribution* inspired by Soliton Waves [11]. The idea behind the Ideal Soliton distribution is that, at each iteration of the decoding algorithm, the expected number of degree-one check blocks is equal to one. Specifically, if we denote by $\rho(d)$ the probability of an encoded symbol to be of degree d ($1 \le d \le k$), we have

$$\rho(d) = \begin{cases} \frac{1}{k} & \text{if } d = 1 \\ \frac{1}{d(d-1)} & \text{if } d = 2, \ldots, k \end{cases}$$

Unfortunately, the Ideal Soliton distribution $\rho(.)$ performs poorly in practice since for any small variation on the expected number of degree-one check blocks the recovering is bound to fail. This has led to the *Robust Soliton Distribution*, referred to as $\mu(d)$ in the following. By generating a larger expected number of degree-one check blocks the Robust Soliton distribution guarantees the success of LT decoding with high probability. Specifically, the Robust Soliton distribution is based on three parameters namely k, δ and C, where k is the number of input symbols to be coded, δ is the failure probability of the LT process and C is a positive constant that affects the probability of generating degree-one check blocks. The Robust Soliton distribution $\mu(d)$ is the normalized value of the sum $\rho(d) + \tau(d)$ where $\tau(d)$ is defined as

$$\tau(d) = \begin{cases} \frac{S}{k} \cdot \frac{1}{d} & \text{if } d = 1, \ldots, (\frac{k}{S}) - 1 \\ \frac{S \cdot \ln(\frac{S}{\delta})}{k} & \text{if } d > \frac{k}{S} \\ 0 & \text{if } d = \frac{k}{S} \end{cases}$$

where S, the expected number of degree-one check blocks in the decoding process, is given by $S = C \ln(k/\delta)\sqrt{k}$. Luby [6] proved that by setting the estimated minimum number CB_0 of check blocks to

$$CB_0 = k \cdot \sum_{d=1}^{k} \rho(d) + \tau(d) \tag{1}$$

the input symbols are recovered with probability $1 - \delta$, with δ arbitrarily small.

2.2 Principles of Online Rateless Codes

Coding Process. The general idea of Online codes [7] is similar to LT codes. The original data is partitioned into k input fragments and coded to redundant check blocks. Online codes are characterized by two main parameters ε and q. Parameter ε, typically satisfying $0.01 \leq \varepsilon \leq 0.1$, infers how many blocks are needed to recover the original message, while q affects the success probability of the decoding process (interesting values of q range from 1 to 5 as shown in the sequel). The main differences between Online and LT codes are the coding algorithm and the degree distribution. Briefly, in contrast to LT codes, Online codes are generated through two encoding phases. In the first phase, $\alpha\varepsilon qn$ auxiliary blocks are generated (typically, α is equal to .55). Each auxiliary block is built by XOR-ing q randomly chosen input blocks. The auxiliary blocks are then appended to the original n blocks message to form the so called composite message of size $n' = n(1 + \alpha\varepsilon q)$, which is suitable for coding. In the second phase, composite blocks are used to create check blocks. Similarly to LT codes, each check block is generated by selecting its degree (i.e. neighbor composite blocks) from a specific degree distribution. Selected neighbors are XOR-ed to form check blocks. Figure 1(b) illustrates these two phases.

Decoding Process. The decoding process of Online codes performs similarly to LT codes. The composite blocks are decoded and from these decoded blocks, the input blocks are recovered.

Online Degree Distribution. The Online distribution only depends on ε. This parameter is used to calculate an upper bound F on the degree of check blocks with $F = \lceil \ln(\frac{\varepsilon^2}{4}) / \ln(1 - \frac{\varepsilon}{2}) \rceil$. If we denote by $\rho(d)$ the probability of a check block to be of degree d, then the probability distribution $\rho(d)$ is defined as

$$\rho(d) = \begin{cases} 1 - \frac{1+\frac{1}{F}}{1+\varepsilon} & \text{if } d = 1 \\ \frac{[1-\rho(1)]F}{(F-1)d(d-1)} & \text{if } d = 2, \ldots, F \end{cases}$$

F is called the degree sample space of distribution $\rho(d)$. Note that in contrast to LT distribution, the Online distribution $\rho(d)$ does not depend on the number of input blocks. It has been theoretically shown in [7] that generating

$$CB_0 = n(1 + \varepsilon)(1 + \alpha\varepsilon q) \quad (2)$$

check blocks is sufficient to decode a fraction $(1 - \frac{\varepsilon}{2})$ of composite blocks, and to successfully recover the original data with probability $1 - (\frac{\varepsilon}{2})^{q+1}$.

3 Experimental Results

This section is devoted to an in-depth practical comparison of LT and Online codes. This comparison has been achieved by implementing both codes in DataCube, a persistent storage architecture [8]. Prior to comparing both codes performance, we briefly describe the main features of this architecture, and present the different policies that have been implemented to select and collect check blocks at the different peers of the system.

3.1 Experimental Platform

DataCube is a data persistent storage architecture robust against highly dynamic environments and adversarial behaviors. DataCube relies on the properties offered by cluster-based DHTs overlays (e.g. [1,3]), and by a compound of full replication and rateless erasure coding schemes. Briefly, cluster-based structured overlay networks are such that clusters of peers substitute peers at the vertices of the graph. Size of each cluster is both lower and upper bounded. The lower bound ℓ usually satisfies some constraint based on the assumed failure model. For instance $\ell \geq 4$ allows Byzantine tolerant agreement protocols to be run among these ℓ peers despite the presence of one Byzantine peer. The upper bound L is typically in $\mathcal{O}(logN)$ where N is the current number of peers in the system, to meet scalability requirements. Once a cluster size exceeds L, this cluster `splits` into two smallest clusters, each one populating with the peers that are closer to each other according to some distance D. Similarly, once a cluster undershoots its minimal size ℓ, this cluster `merges` with the closest cluster in its neighborhood. At cluster level, peers are organized as core and spare members. Each data-item is replicated at core members. This replication schema guarantees that in presence of a bounded number of malicious peers, data integrity is guaranteed through Byzantine agreement protocols, and efficient data retrieval is preserved (retrieval is achieved in $\mathcal{O}(logN)$ hops and requires $\mathcal{O}(logN)$ messages, with N the current number of peers in the system). In addition to this replication schema, each data D is fragmented, coded and spread outside its original cluster. The identifier cb_i of each check block i of D is unique and is derived by applying a hash-chain mechanism on the key, *key(D)*, of D such that $cb_i = \mathcal{H}^{(i)}(key(D))$. The adjacencies x_i of cb_i are derived by using a pseudo-random generator function $\mathcal{G}(.)$. The rationale of using $\mathcal{G}(.)$ is that any core member can generate exactly the same check blocks independently from the other core members. Each check block i is then placed at $\gamma \geq 2$ spare members of the cluster that matches cb_i. This coding schema guarantees that in presence of targeted attacks (i.e., the adversary manages to adaptively mount collusion against specific clusters of peers), recovery of the data those clusters were in charge of is self-triggered. For space reasons we do not give any more detail regarding the implementation of hybrid replication in DataCube. None of these details are necessary for the understanding of our present work, as DataCube will essentially be used as an evaluation platform. Anyway, the interested reader is invited to read its description in [8].

To evaluate the coding performance of both Online and LT codes in Datacube, we simulate targeted attacks on a set of clusters. Such attacks prevent any access to the data these clusters cache. By doing this, we force the retrieval of these data to be recovered through the decoding process. Let $\mathcal{C}$ be one these clusters, and D be the data some peer q is looking for. Let $\mathcal{C}'$ be the closest cluster to cluster $\mathcal{C}$ so that, by construction of DataCube, cluster $\mathcal{C}'$ is in charge of recovering cluster

$\mathcal{C}$ data, and in particular data D. Then for any peer p_i in the core set of $\mathcal{C}'$, p_i will request and collect sufficiently many check blocks to successfully recover D. Specifically, from the hash-chain mechanism applied to *key(D)*, p_j derives a set M of m check blocks keys, namely $cb_1 \dots cb_m$, with $cb_i = \mathcal{H}^{(i)}(key(D))$, and $m = m_0 \cdot CB_0$, where $m_0 = 1 \dots 5$. Then from $\mathcal{G}(.)$ p_j generates $x_1 \dots x_m$, the respective adjacencies of $cb_1 \dots cb_m$. From this set M, p_j has the opportunity to apply different policies to select the check blocks it will collect in DataCube, these policies differing according to the priority given to the adjacencies degree. The rational of these policies is to show whether in practice it makes sense to "help" the decoder by first collecting as many degree-one check blocks as possible (this is the purpose of both Policies 2 and 3), or on the contrary, whether it is more efficient to collect random check blocks as theoretically predicted (Role of Policy 1). Policy 4 implements the optimal decoders behaviors. Specifically,

- Policy 1 (*random policy*): no priority is given to the adjacencies degrees. That is, CB_0 check blocks identifiers cb_i are randomly chosen from set M, and the corresponding check blocks are collected in DataCube (through `lookup(`cb_i`)` operations).
- Policy 2 (*degree-one first, random afterwards policy*): the priority is given to degree-one check blocks. That is, all degree-one check blocks identifiers belonging to M are selected, and if less than CB_0 check blocks have been selected, the remaining ones are randomly selected from M. As for above, the corresponding check blocks are collected in DataCube through `lookup(.)` operations.
- Policy 3 (*degree-one-only policy*): similar to Policy 2 except that instead of randomly selected the non degree-one check blocks, degree-two check blocks that will be reduced thanks to the degree-one check blocks are selected. Note that less than CB_0 check blocks can be selected.
- Policy 4 (*optimal policy*): the bipartite graph is applied on the elements of set M, and only necessary check blocks are selected. Comparing to Policy 3, no redundant degree-one check blocks are selected. This makes Policy 4 optimal w.r.t. set M.

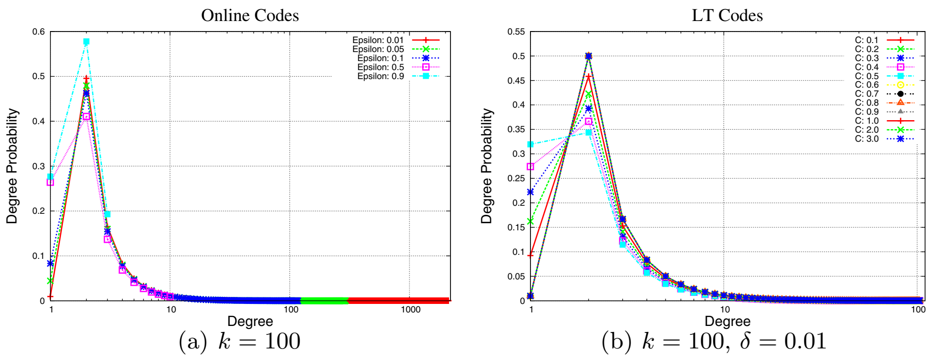
If this first phase is not sufficient for the decoding process to recover the input message M, then p_j regenerates a new set M and proceeds as above. It does this until D is fully recovered.

3.2 Setup

Our experiments are conducted over a Linux-based cluster of 60 dedicated Dell PowerEdge 1855 and 1955 computers, with 8 gigabytes of memory each, and 400 Bi-pro Intel Xeon processors. We developed a java-based prototype to simulate DataCube, and to implement both LT and Online coding schemes. Each coding schema is evaluated with a large range of input parameters. The number k of input fragments varies from 100 to $10,000$. For Online coding, ε varies from 0.01 to 0.9, and q is set to integer values from 1 to 5. For LT coding, δ varies from 0.01 to 0.9 and C from 0.1 to 5. All the plotted curves are the average of 50 experiments.

3.3 Degree Distributions

We start by highlighting some differences on the expected behavior of both coding processes. The design of each degree distribution is the most critical point for both coding processes to efficiently recover the original data. A good degree distribution must guarantee the input block coverage, that is, that all the input blocks have at least one edge to guarantee a successful recovery. In Online coding, composite blocks are used to ensure this coverage. Low degrees check blocks, and degree-one in particular, are crucial for the decoding process to start and keep running. On the other hand, too many low degrees may lead to an over redundancy of check blocks, and thus are useless. Figure 2(a) shows the degree distribution of Online codes. Two interesting behaviors can be observed. First, for $\varepsilon \leq 0.1$ the degree-one probability is very small (i.e., ≤ 0.09) while for $\varepsilon > 0.1$, the expected number of degree-one check blocks is greater than 26%. The second observation is that the range of degrees a check block can be built from (i.e., F values) drastically augments with decreasing values of ε (i.e., $F = 3$ for $\varepsilon = 0.9$, and $F = 2114$ for $\varepsilon = 0.01$). As will be shown later on, both features have a great impact on Online coding performance. Figure 2(b) shows the degree distribution of the LT coding process. The impact of C on check blocks degrees is clearly shown (i.e., increasing C values augments the probability of degree-one check blocks). An interesting point to observe is that degree-one probability increases up to $C \leq 0.5$, and abruptly drops for $C > 0.5$. Interpretation of this behavior is that combination of these specific values of C (i.e., $C > 0.5$) and those of k and δ lead distribution $\tau(d)$ to tend to zero, which makes the Robust Soliton distribution behaving exactly as the Ideal distribution. The value of C for which this phenomena occurs will be referred to in the following as the *cut-off* value of C (for $k = 100$, and $\delta = 0.01$, the cut-off value equals 0.6). We show in the following how the cut-off value impacts LT performance. Figures 3(a) and 3(b) show the degree distribution of LT codes for varying values of δ. We can see that the influence of δ increases with increasing values of C.



(a) $k = 100$ (b) $k = 100$, $\delta = 0.01$

Fig. 2. Distributions $\mu(.)$ and $\rho(.)$ as a function of the degree

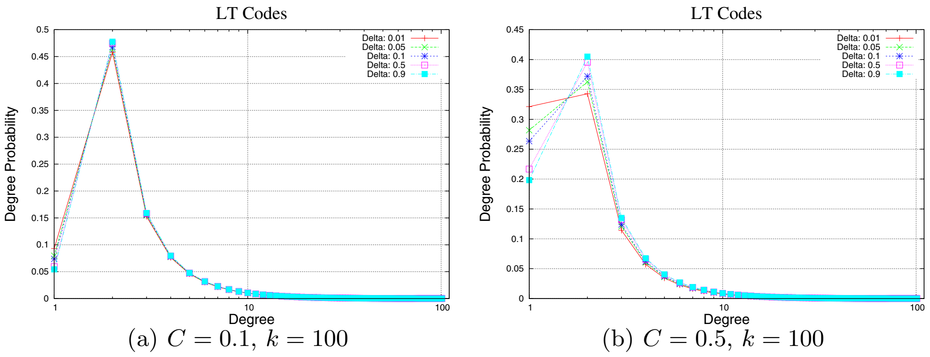
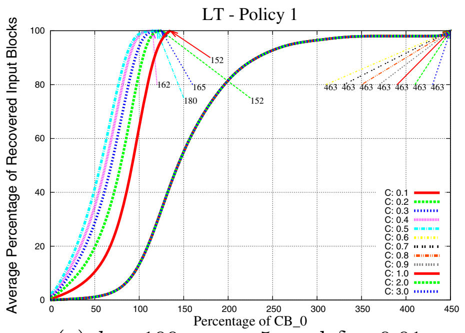


Fig. 3. Distributions $\mu(.)$ as a function of the degree for different values of δ

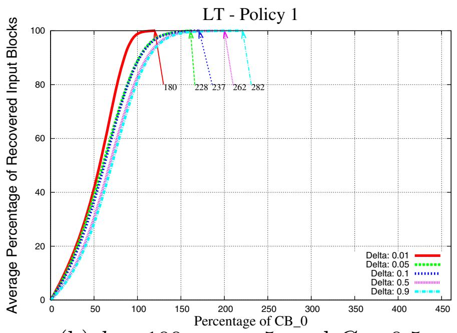
3.4 Recovery Performance of Coding Processes

This section evaluates the number of check blocks that need to be recovered in practice to successfully recover the original data D for both coding schemes, and for the four above described policies. In all the figures shown in this section, curves are depicted as a function of the fraction of the predicted minimal value CB_0. That is, an abscissa equal to 150 means 1.5 times CB_0 check blocks. Arrows point to the number of check blocks that are needed to recover exactly k input blocks, that is 100% of D.

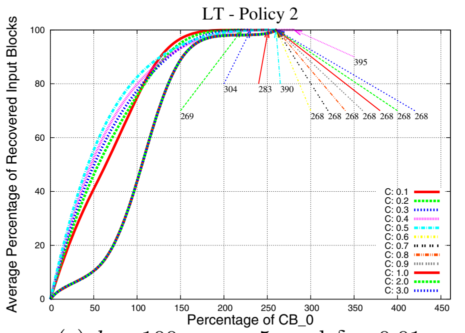
We first analyze the results obtained for LT codes using the four above described policies. The impacts of both C and δ on LT recovery performance when Policy 1 is run are illustrated in Figures 4(a) and 4(b). General observations drawn from Figure 4(a) are first that by randomly collecting check blocks, LT differs from the theoretical prediction in no more than 30% (for instance, to recover $k = 100$ input blocks with $C = 0.1$, CB_0 is equal to 113 (see Relation 1) while in average 152 check blocks are needed. Moreover, for increasing values of C, LT behavior progressively degrades with a sharp breakdown when C reaches its cut-off value (i.e., $C = 0.6$). Indeed, from $C = 0.6$ onwards, the Robust Soliton distribution behaves as the Ideal one, which also explain why LT behavior is independent from C values (i.e., 463 check blocks are necessary to successfully recover $k = 100$ input blocks for any $C \geq 0.6$). Figure 4(b) shows LT recovery performance for different values of δ. We can see that the number of check blocks augments with increasing values of δ. This feature is due to LT distribution $\mu(.)$ since increasing values of δ leads to a diminution of degree-one probability. We can also observe that with 100% of CB_0, the percentage of recovered input blocks increases with decreasing values of δ. A similar behavior observed for different values of C tends to confirm that δ highly impacts the probability of successful recovery. The impact of Policies 2 and 3 on LT is significant as shown in Figures 4(c) and 4(d). The quasi-exclusive collect of degree-one check blocks combined with the impact of C on the generation of degree-one check blocks overwhelms the decoder with too many redundant degree-one check blocks (and



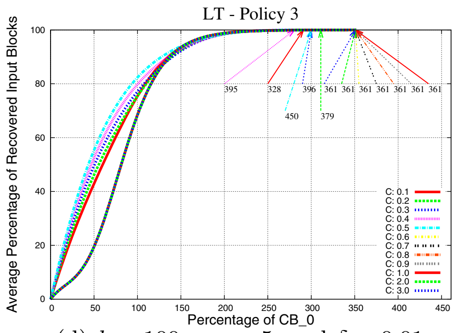
(a) $k = 100$, $m_0 = 5$, and $\delta = 0.01$



(b) $k = 100$, $m_0 = 5$, and $C = 0.5$

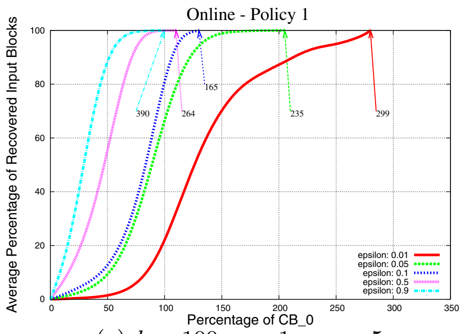


(c) $k = 100$, $m_0 = 5$, and $\delta = 0.01$

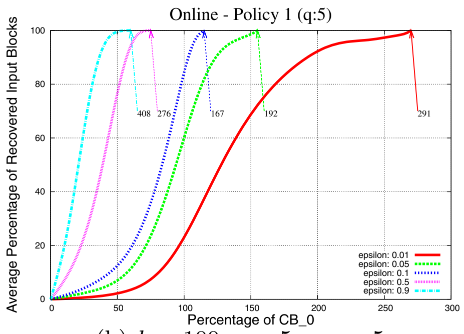


(d) $k = 100$, $m_0 = 5$, and $\delta = 0.01$

Fig. 4. Percentage of recovered input blocks as a function of the number of collected check blocks expressed as a fraction of CB_0 for different values of C



(a) $k = 100$, $q = 1$, $m_0 = 5$

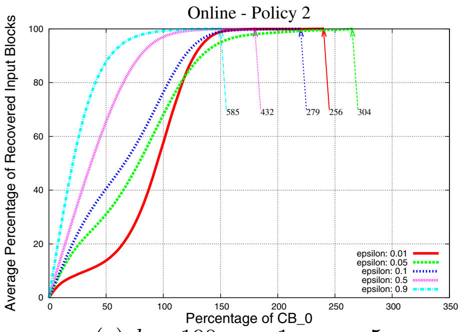


(b) $k = 100$, $q = 5$, $m_0 = 5$

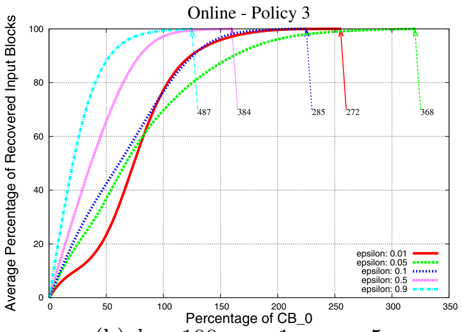
Fig. 5. Percentage of recovered input blocks as a function of the number of collected check blocks expressed as a fraction of CB_0

degree-two check blocks for Policy 3) which requires many check blocks (from 269 to 395) to successfully recover $k = 100$ original input blocks. On the other hand, when C exceeds its cut-off value, the large amount of degree-one check blocks collected by both policies is compensated by the very low number of check blocks generated by the Ideal distribution, leading to better recovery performance than when $C < 0.6$. Note that in all these experiments, m_0 equals 5 which gives a large choice for the policies to select check blocks that fit their properties. Finally, and as expected, the optimal policy behaves perfectly well. This policy guarantees the full recovery of input blocks in a linear number of check blocks. Indeed, each check block is the outcome of the bipartite graph decoding process, and thus each single selected check block is useful for the recovery (for instance, 103 check blocks in average allow to recover $k = 100$ input blocks. For space reasons we have not illustrated performance of Policy 4 in this paper).

We now analyze the results obtained for Online coding schema with the four policies. Policy 1 is illustrated for different values of ε and q in respectively Figures 5(a) and 5(b). A preliminary observation drawn from Figure 5(a) is that varying ε leads to a greater range of CB_0s values than what is obtained when one varies parameter C in LT (i.e., with Online, CB_0 varies from 106 to 390 while with LT, CB_0 varies from 103 to 150). Thus as a first approximation, we may expect that in average the number of check blocks that need to be collected for successfully recovering the input blocks is larger with Online than with LT. Another preliminary comment is that the gap between theoretical predictions and practical results increases with diminishing values of ε. Surprisingly enough, the number of check blocks that need to be collected for a successful recovery does not vary proportionally to ε, that is for extrema values of ε, this number is respectively equal to 290 and 390, while it decreases to reach its minimum 165 for $\varepsilon = 0.1$. Actually, for $\varepsilon = 0.9$, the number of auxiliary blocks is large (see Section 2.2) but the space degree F is very small (i.e., equal to 3, see Section 3.3). Thus a large number of redundant check blocks are *a priori* collected, which demands for more check blocks to successfully recover k input blocks. Now for $\varepsilon = 0.01$, the number of auxiliary blocks is small but they form a complete bipartite graph with their associated input blocks, which clearly make them useless. Moreover the probability of having degree-one check blocks is very small (i.e., 0.01), and the probability of having degree-two is large (i.e., 50%). However as the space degree F is very large too, the likelihood of having some very high degree check blocks is not null, and thus a large number of check blocks need also to be collected. Finally, for $\varepsilon = 0.1$, the distribution of check blocks degrees is relatively well balanced, in the sense that the proportion of degree-one is relatively high (i.e., 9%) but not too high to prevent redundancy. The proportion of degree-two is high (i.e., 47%) which combined with degree-one allows to cover many input blocks. Finally, degree-three and degree-four check blocks are also useful (i.e., respectively equal to 15% and 4%). This clearly make this value of ε optimal, which is confirmed by the fact that 165 check blocks successfully recover 100 input blocks. The very same argument applies for larger values of q as shown in Figure 5(b). The impact of Policies 2 and 3 on Online coding



(a) $k = 100$, $q = 1$, $m_0 = 5$



(b) $k = 100$, $q = 1$, $m_0 = 5$

Fig. 6. Percentage of recovered input blocks as a function of the number of collected check blocks expressed as a fraction of CB_0

shown in Figures 6(a), and 6(b) is similar to the one obtained on LT, in the sense that recovering essentially degree-one check blocks cumulates with the high proportion of degree-one check blocks generated with the degree distributions (see Section 3.3) and thus, leads to the collect of a large number of redundant check blocks. Note that sensibility of this phenomena augments with increasing values of ε.

3.5 Computational Cost in Terms of xor Operations

In this section, we discuss the computational costs of both LT and Online decoding process. The computational cost is quantified by the number of XOR operations that need to be run to successfully recover the input data when Policy 1 is applied. Note that the unit of XOR used by the encoder/decoder matches the length of an input block.

We first analyze the number of XOR operations in both LT and Online coding schemes as a function of their respective parameters (C, δ), and (ε, q). Results are depicted in Figures 7(a) and 7(b). The main observation drawn from Figure 7(a) is that the number of XOR operations slowly drops down with increasing values of C (provided that C cut-off values are not reached (i.e., $C = 0.6$ for $\delta = 0.01$ and $C = 1$ for $\delta = 0.5$). Specifically, for small values of C, the probability of degree-one check blocks is less than 10% and weakly depends on δ values. On the other hand, for larger values of C, the probability of degree-one check blocks is equal to respectively 22% and 33% for $\delta = 0.5$ and $\delta = 0.01$ which explain the negative slopes of both curves, and the increasing gap between both curves for increasing values of ε.

Now, regarding Online computational cost as a function of ε and q, Figure 7(b) shows first that ε has a strong impact on the number of XOR operations that need to be performed. Specifically for increasing values of ε, this number drastically decreases, down to the average number of XOR operations run with LT coding. This behavior seems to result from the combination of two phenomena.

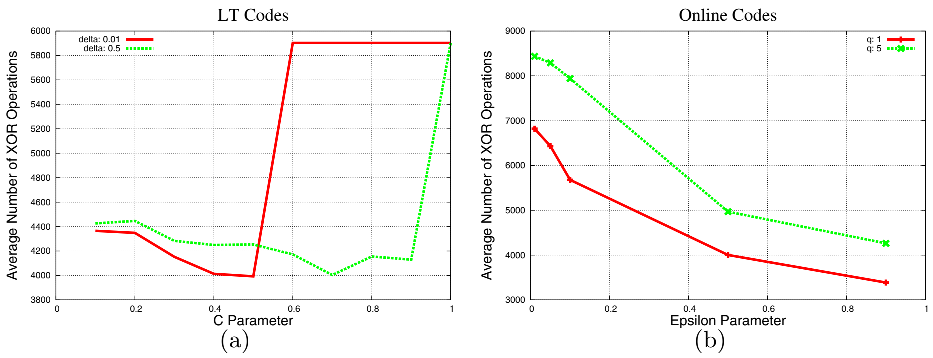


Fig. 7. Average number of XOR operations to successfully recover $k = 100$ input blocks as a function of LT and Online parameters

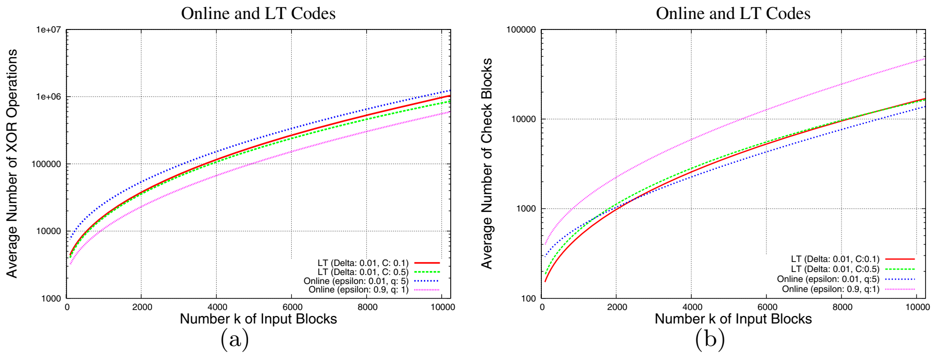


Fig. 8. (a) Average number of XOR operations for respectively LT and Online codes to successfully recover the input data as a function of the size k of the input data. (b) Average number of check blocks for respectively LT and Online codes to successfully recover the input data as a function of the size k of the input data.

For $\varepsilon = 0.01 \ldots 0.1$, the number of check blocks needed to successfully recover the input data decreases (as previously observed in Figure 5(a)), and in the meantime, F equally decreases with an increasing probability of generating degree-one and degree-two check blocks. Thus both phenomena cumulate and give rise to the diminution of the number of XOR operations as observed in Figure 7(b) (recall that degree-one check blocks do not trigger any XOR operations). Now, for $\varepsilon = 0.1 \ldots 0.9$, the number of check blocks increases (as previously observed in Figure 5(a)), and F drastically decreases together with a high probability of generating degree-one and degree-two check blocks. Thus despite the fact that a large number of check blocks need to be decoded, most of them have a degree-one or degree-two, and thus most of them do not trigger XOR operations, which explains the negative gradients of the curves in Figure 7(b). The second observation drawn from this figure is that the number of XOR operations equally depends on q value. Specifically, for large values of q (e.g., $q = 5$) the number of auxiliary blocks is high and their adjacencies degree with the input blocks is small. Thus these auxiliary blocks are involved in the decoding process and thus augment the number of XOR operations. On the other hand, for small values of q (e.g., $q = 1$), the number of auxiliary blocks is small however, these blocks form a quasi-complete bipartite graph with the input blocks, which makes them useless for the decoding process. Consequently, they do not impact the computational cost of Online. Finally, Figures 8(a) and 8(b) depict the computational cost of both LT and Online as a function of the number of input blocks. The main observation is that both LT and Online decoding complexity are linear with k. The second finding is that for increasing values of k, Online decoding complexity is more sensible to parameters variations than LT is.

4 Conclusion

In the present work, we have evaluated both LT and Online rateless codes in terms of recovery and computational performance, and scalability properties. Experiments have confirmed that it is more efficient to collect random check blocks as theoretically predicted than favoring only small degree check blocks. We expect that this study should allow a good insight into the properties of these codes. In particular we have confirmed the good behavior of both coders/decoders when the number of input blocks increases. This is of particular interest for multimedia and storage applications.

References

1. Anceaume, E., Brasiliero, F., Ludinard, R., Ravoaja, A.: Peercube: an hypercube-based p2p overlay robust against collusion and churn. In: Proceedings of the IEEE International Conference on Self Autonomous and Self Organising Systems, SASO (2008)
2. Bhagwan, R., Tati, K., Cheng, Y.C., Savage, S., Voelker, G.M.: Total Recall: System support for automated availability management. In: Proceedings of the USENIX Association Conference on Symposium on Networked Systems Design and Implementation, NSDI (2004)
3. Fiat, A., Saia, J., Young, M.: Making chord robust to byzantine attacks. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 803–814. Springer, Heidelberg (2005)
4. Houri, Y., Jobmann, M., Fuhrmann, T.: Self-organized data redundancy management for peer-to-peer storage systems. In: Proceedings of the 4th IFIP TC 6 International Workshop on Self-Organizing Systems (IWSOS). Springer, Heidelberg (2009)
5. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., et al.: OceanStore: an architecture for global-scale persistent storage. In: ACM SIGARCH Computer Architecture, pp. 190–201 (2000)

6. Luby, M.: LT codes. In: Proceedings of the IEEE International Symposium on Foundations of Computer Science, SFCS (2002)
7. Maymounkov, P.: Online codes. Research Report TR2002-833, New York University (2002)
8. Ribeiro, H.B., Anceaume, E.: DataCube: a P2P persistent storage architecture based on hybrid redundancy schema. In: Proceedings of the IEEE Euromicro International Conference on Parallel, Distributed and Network-Based Computing, PDP (2010)
9. Ribeiro, H.B., Anceaume, E.: Exploiting Rateless Coding in Structured Overlays to achieve Data Persistence. In: Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications, AINA (2010)
10. Rowstron, A., Druschel, P.: Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. ACM SIGOPS Operating System Review 35(5), 188–201 (2001)
11. Russell, J.S.: Report on waves. In: 14th Meeting of the British Association for the Advancement of Science, pp. 311–390 (1844)
12. Shokrollahi, A.: Raptor codes. IEEE/ACM Transactions on Networking, 2551–2567 (2006)
13. Sit, E., Haeberlen, A., Dabek, F., Chun, B.G., Weatherspoon, H., Morris, R., Kaashoek, M.F., Kubiatowicz, J.: Proactive replication for data durability. In: Proceedings of the 5rd International Workshop on Peer-to-Peer Systems, IPTPS 2006 (2006)
14. Tanner, R.: A recursive approach to low complexity codes. IEEE Transactions on Information Theory 27(5), 533–547 (1981)

Dynamically Reconfigurable Filtering Architectures

Mathieu Valero, Luciana Arantes, Maria Gradinariu Potop-Butucaru, and Pierre Sens

LIP6 - University of Paris 6 - INRIA

Abstract. Distributed R-trees (DR-trees) are appealing infrastructures for implementing range queries, content based filtering or k-NN structures since they inherit the features of R-trees such as logarithmic height, bounded number of neighbors and balanced shape. Interestingly, the mapping between the DR-tree logical nodes and the physical nodes has not yet received sufficient attention. In previous works, this mapping was naively defined either by the order physical nodes join/leave the system or by their semantics. Therefore, an important gap in terms of load and latency can be observed while comparing the theoretical work and the simulation/experimental results. This gap is partially due to the placement of virtual nodes. A naive placement that totally ignores the heterogeneity of the network may generate an unbalanced load of the physical system. In order to improve the overall system performances, this paper proposes mechanisms for placement and dynamic migration of virtual nodes that balances the load of the network *without* modifying the DR-tree virtual structure. That is, we reduce the gap between the theoretical results and the practical ones by injecting (at the middleware level) placement and migration strategies for virtual nodes that directly exploit the physical characteristics of the network. Extensive simulation results show that significant performance gain can be obtained with our mechanisms. Moreover, due to its generality, our approach can be easily extended to other overlays or P2P applications (e.g. multi-layer overlays or efficient P2P streaming).

1 Introduction

From the very beginning of the theoretical study of P2P systems, one of the topics that received a tremendous attention is the way these systems are mapped to the real (physical) network. Even early DHT-based systems such as Pastry [19] or CAN [18] included in their design the notion of geographical locality. Later, middlewares designed on top of DHT-based or DHT-free P2P systems exploit various degrees of similarity between peers following criteria such as the geographical vicinity or the semantic of their interests. One of the most relevant example in that sense are content-based publish/subscribe systems. These communication primitives completely decouple the source of events (*a.k.a. publishers*) from their users (*a.k.a. subscribers*). Their efficient implementations in P2P settings are optimized with respect to a broad class of metrics such as latency or load balancing. To this end the similarity between different subscribers is fully exploited and various logical infrastructures have been recently proposed. Most exploited architectures are tree-based due to their innate adaptability to easy filter. In the design of tree-based publish/subscribe there is a trade-off between the maintenance of an optimized tree infrastructure following various criteria (e.g. bounded degree, max/min number of internal nodes or leaves) and the placement of logical nodes on physical machines in order to optimize the overall system load or latency.

Our paper follows this research direction by addressing the placement of virtual nodes in tree-filtering architectures. In particular, we address the case of DR-trees overlays where the quality of service of the overlay greatly depends on the way virtual nodes are mapped on the physical nodes. DR-trees [2] are a distributed P2P version of R-trees [11] which are used to handle objects with a poly-rectangular representation. DR-trees are P2P overlays with a bounded degree and search/diffusion time logarithmic in size of the network. By their natural construction they are adapted to represent subscriptions with rectangle shape in content based systems. The efficient construction of a DR-tree overlay raises several problems. In particular, it should be noted that the design of DR-trees may bring a single physical machine to be responsible for several virtual nodes. Therefore, a wrong choice in the placement of virtual nodes may have a huge impact on overall system performances; in terms of latency, bandwidth etc...

Our contribution tends to address this issue. We investigate the problem of placement and migration of virtual nodes. We define valid migrations (with respect to the logical topology of the DR-trees) and, using some techniques borrowed from transactional systems, we propose strategies for solving conflicts generated by concurrent migrations.

2 Related Work

The placement of virtual nodes has been evoked in several works addressing different ad hoc issues. In publish-subscribe systems such as Meghdoot [10], Mirinae [7], Rebeca [20], SCRIBE [6] or Sub-2-Sub [17], virtual nodes correspond to subscriptions and they are mapped on the physical node that created them. In BATON [13] and VBI [14], two AVL based frameworks, virtual nodes are divided in two categories: leaves and internal nodes. The former are used for storage while the latter are used for routing. A structural property of AVL ensures that there there are roughly as much leaves as internal nodes: each physical node holds one leaf and one internal node. In a DR-tree, due to its degree ($m : M$, with $m > 1$), there are fewer internal nodes than leaves; each physical node holds exactly one leaf and may hold one or more internal nodes.

Many publish/subscribe are based on multicast trees where network characteristics are exploited to build efficient multicast structures in terms of latency and/or bandwidth ([3,1,8,5]). Our approach does not require any extra overlays or structures. Network characteristics are taken into account through a mechanism of virtual node migrations; this mechanism was not conceived for a particular evaluation metric. Metrics can be defined independently making our work more general.

Brushwood [4] is a kd-tree based overlay targeting locality preservation and load balancing. Each physical node holds one virtual node which corresponds to a k-dimensional hyperplane. When a physical node joins the network, it is routed to a physical node. If the joined physical node is overloaded, it will split its hyperplane and delegate half of it to the new node. In addition, Brushwood provides a sporadical load evaluation mechanism: overloaded physical nodes may force underloaded ones to reinsert themself and thus benefit from the join mechanism.

Chordal graph [15] is a range queriable overlay. Efficiency of queries is measured in terms of distance (that could be expressed in terms of latency, bandwidth etc...) between physical nodes. Similarly to SkipNet [12,9], each physical node belongs to different rings. For each ring, the physical node holds a range of values which can be dynamically resized to balance load. During joins, physical nodes are routed according to their relative distance. Conceptually, hyperplanes and ranges are virtual nodes. While Chordal graph [15] and Brushwood [4] modify virtual nodes during hyperplane splitting or range scaling while our approach neither modifies virtual nodes nor the DR-tree structure.

3 Background

In this section we recall some generic definitions and the main characteristics of the DR-trees [2] overlay. Moreover, we discuss the main issues related to the virtual nodes distribution.

3.1 Distributed R-trees

R-trees were first introduced in [11] as height-balanced tree handling objects whose representation can be circumscribed in a poly-space rectangle. Each leaf-node in the tree is an array of pointers to spatial objects. A R-tree is characterized by the following structural properties:

– Every non-leaf node has a maximum of M and at least m entries where $m \leq M/2$, except for the root.
– The minimum number of entries in the root node is two, unless it is a leaf node. In this case, it may contain zero or one entry.
– All the leaf nodes are at the same level.

Distributed R-trees (DR-trees) introduced in [2] extend the R-tree index structures where peers are self-organized in a balanced virtual tree overlay based on semantic relations The structure preserves the R-trees index structure features: bounded degree per node and search time logarithmic in the size of the network. Moreover, the proposed overlay copes with the dynamism of the system.

Physical machines connected to the system wille be further referred as *p-nodes* (shortcut for physical nodes). A DR-tree is a virtual structure distributed over a set of p-nodes. In the following, terms related to DR-tree will be prefixed with “v-”. Thus, DR-trees nodes will be called *v-nodes* (shortcut for virtual nodes). The root of the DR-tree is called the *v-root* while the leaves of the DR-tree are called *v-leaves*. Except the v-root, each v-node n has a v-father (*v-father(n)*), and, if it is not a v-leaf, some v-children (*v-children(n)*). These nodes are denoted $v-neighbors$ of n.

The physical interaction graph defined by the mapping of a DR-tree to $p-nodes$ of the system is a communication graph where there is a $p-edge$ (p,q), $p \neq q$, in the physical interaction graph if: there is a v-edge (s,t) in the DR-tree, p is the p-node holding v-node s, and q is the p-node holding v-node t.

Figure 1 shows a representation of a DR-tree composed of v-nodes $\{n0,\ldots,n12\}$ mapped on p-nodes $\{p1,\ldots,p9\}$. Dashed boxes represent nodes distribution. There is a p-edge $(p1,p5)$ in the interaction graph because there is a v-edge $(n0,n6)$ in the DR-tree, $p1$ is the p-node holding v-node $n0$, and $p5$ is the p-node holding v-node $n6$.

The key points in the construction of a DR-Tree are the join/leave procedures. When a p-node joins the system, it creates a v-leaf. Then the p-node contacts another p-node to insert its v-leaf in the existing DR-tree. During this insertion, some v-nodes may split and then Algorithm 1 is executed.

Algorithm 1. *void* onSplit(n:VNode)

```
1: **if** $n.isVRoot()$ **then**
2:     $newVRoot = n.createVNode()$          ▷ $newVRoot$ is held by the same p-node than $n$
3:     $n.v-father = newVRoot$
4: **end if**
5: $m = selectChildIn(n.v-children)$
6: $newVNode = m.createVNode()$          ▷ $newVNode$ is held by the same p-node than $m$
7: $n.v-children, newVNode.v-children = divide(n.v-children)$
8: $newVNode.v-father = n.v-father$
```

Distribution Invariants: The following two properties are invariant in the implementation of DR-tree proposed in [2]:

- $Inv1$: each p-node holds exactly one v-leaf;
- $Inv2$: if p-node p holds v-node n, either n is a v-leaf or p holds exactly one v-children of n.

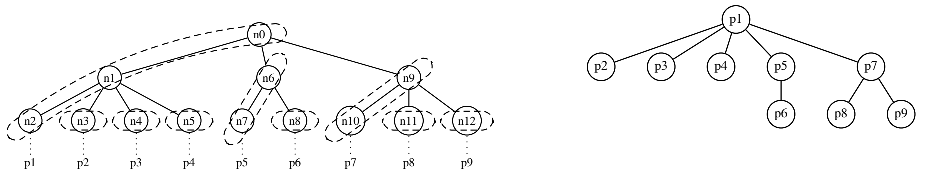
We denote the *top* and *bottom* v-node of a p-node the v-node which is at the top and bottom of the chain of v-nodes kept by the p-node respectively. The above invariants ensure that the communication graph is a tree:

- The p-root is the p-node holding the v-root;
- A p-node p is the p-father of the p-node q (*p-father(q)*) if p holds the v-father of the v-node at the top of the chain of v-nodes held by q.

For instance in Figure 1a, *p-father(p5) = p-father(p7) = p1*. The above distribution invariants also guarantee that p-nodes have a bounded number of p-neighbors. In a system with N p-nodes and a DR-tree with degree $m-M$, the DR-tree height is $log_m(N)$; the p-root holds $log_m(N)$ v-nodes. Since each v-node has up to M v-neighbors, the p-root may have up to $M * log_m(N)$ p-neighbors.

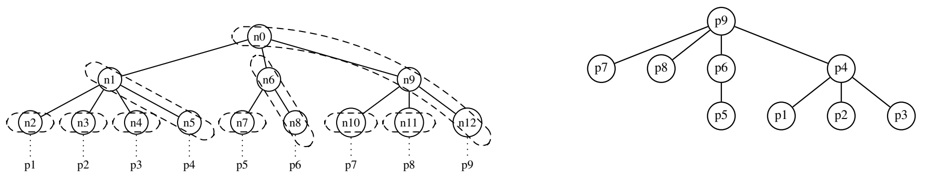
3.2 Virtual Nodes Distribution

A DR-tree is a logical structure distributed across a set of physical machines. That is, each v-node is assigned to a p-node of the system. Figures 1 and 2 show the same DR-tree differently distributed over the same set of physical nodes $\{p1,\ldots,p9\}$. The



(a) A distribution of a given R-tree (b) Corresponding physical interaction graph

Fig. 1. A distribution of a given DR-tree and its corresponding physical interaction graph



(a) A second distribution of the same R-tree (b) Corresponding physical interaction graph

Fig. 2. A second distribution of the same DR-tree leading to a different physical interaction graph

distribution of DR-tree nodes determines the communication interactions between the physical nodes and thus has a strong impact on system performances.

In [2] the distribution of the DR-tree depends both on the join order of machines and on the implementation of the join/split procedures (Algorithm 1). This approach has two main drawbacks. Firstly, the characteristics of the physical machines are not taken into account in the distribution of the DR-tree nodes. Secondly, this distribution is static. Virtual nodes are placed at their creation or during join/split operations. Their location is not changed even if system performances degrade. The first point is problematic in heterogeneous networks, where system performances are highly related to the distribution of the DR-tree root (and its "close" neighborhood). For example, if we evaluate the quality of the system in terms of bandwidth, if $p1$ has a bad one and $p9$ a good one, the mapping proposed in Figure 2a is better than the one proposed in Figure 1a. Static distribution is problematic if the quality of the communications in the physical interactions graph evolves over time. Therefore, virtual nodes that are close to the root of the DR-tree and that are placed on the best possible machines at a given time may induce poor system performances if their guest machines become less performant.

To address these issues, we propose a mechanism for dynamic migration of DR-tree nodes over the physical network. It allows to dynamically modify the DR-tree distribution (without any modification of the logical structure) in order to match the performance changes of the physical system. Our migration mechanism uses feed-backs from some cost functions (e.g. load of nodes) based on the physical interaction graph and also exploits the virtual relations defined by the DR-tree logical structure.

4 Migration Mechanism

In this section we analyze the migration of a v-nodes while the DR-tree overlay evolves over time. We start by explaining which v-nodes are candidate to migrate and which destination p-nodes can accept the former with respect to distribution invariants described in section 3.1. Then, we present our migration protocol and discuss when it is triggered.

In the following, we denote $p \xrightarrow{n} q$ the migration of the v-node n from p-node p to p-node q.

4.1 Migration Policy

Our migration policy prevents migrations that would violate the two invariants of DR-tree distribution. Note that the invariants can be violated by the wrong choice of both the v-node to migrate and the destination p-node where the v-node will be placed.

The p-node p can migrate its v-node n provided that the distribution invariants will still hold if p no longer keeps n, i.e., if the migration of n takes place.

Following migrations of the v-node n would violate one of the two invariants:

- if p holds exactly one v-node n: if p migrates n, p would no longer hold exactly one leaf ($Inv1$);
- if p holds at least two v-nodes: if p migrates its *bottom* v-node, p would no longer hold exactly one leaf ($Inv1$); if p migrates a v-node n which is neither its *bottom* nor its *top* v-node, p would no longer hold exactly one child of v-father(n) ($Inv2$).

However, if p decides to migrate its *top* v-node n, the distribution invariants continue to be verified if the *top* v-node of the destination p-node is the v-child of n. Then, we define a *migrable* v-node as follows:

Definition 1. *A v-node n of p-node p is migrable if p holds at least two v-nodes and n is the top v-node of p.*

For instance, in Figure 1a, only $n0$, $n6$, and $n9$ are *migrable*.

Let's now discuss how to chose a destination p-node, denoted a *valid* destination, which ensures that if the migration of n happens the two invariants will still be verified.

Consider that n is a *migrable* v-node of p and let q be a candidate destination p-node to receive n. Following cases would violate one of the two invariants:

- if q holds no v-neighbor of n: n would not be a v-leaf of q, and q would hold no v-children(n) ($Inv2$);
- if q holds v-father of n: n would not be a v-leaf of q, and q would hold two v-children(n) ($Inv2$).

However, if q holds $m \in$ v-children(n) and if n was migrated to q, then n would become q's *top* v-node. Therefore, we define valid destination of a *migrable* v-node n as follows:

Definition 2. *A p-node q is a valid destination for v-node n held by p, if q holds a v-children of n.*

In Figure 1a, $p1$ can migrate $n0$. Valid destinations are p-nodes which hold a v-children of $n0$: $p5$ and $p7$.

Since the degree of DR-tree is m-M (with $m \geq 2$), the p-root may chose between 1 and M-1 migrations while a p-node holding more than one v-node may chose between m-1 and M-1 migrations.

4.2 Migration Conflict Solver

Our migration policy guarantees a "local" coherency of migrations. However, two concurrent migrations could lead to invalid configurations. For instance, in Figure 1a, $p1$ can decide to execute $p1 \xrightarrow{n0} p5$ while $p5$ concurrently decides to execute $p5 \xrightarrow{n6} p6$. These two migrations are possible according to our migration policy. On the other hand, executing them concurrently will lead to a configuration where the set of v-nodes held by $p5$ is $\{n0, n7\}$, which violates $Inv2$: $n0$ is not a v-leaf but $p5$ holds no v-children(n0).

When a v-node n is migrated from p-node p to p-node q, p and q are obviously concerned in the migration protocol since they exchange some information for ensuring the migration policy (q can accept or not the migration). In order to avoid incoherent configurations due to concurrent migrations as the one described above, p-nodes holding some v-neighbors of n must also be involved in the migration protocol. Therefore, the principle of our migration protocol is that a p-node p that wants to perform $m = p \xrightarrow{n} q$ should ask permission for it to p-nodes that could concurrently execute some migration conflicting with m. The protocol is basically a distributed transaction where the destination p-node and all p-nodes involved in possible concurrent migrations must give their agreement in order to commit the migration; otherwise it is aborted.

Defining which p-nodes could execute a conflicting migration with $p \xrightarrow{n} q$ is in close relation with our migration policy. As the latter restricts migration, a p-node can only receive migration requests from its p-father. Therefore, migrations that could conflict with $p \xrightarrow{n} q$ are:

- $p-father(p) \xrightarrow{v-father(n)} p$
- $q \xrightarrow{m \in v-children(n)} r$ (where r holds a v-children of q)

There is some locality in potential conflicts: when p-node p tries to perform a migration, concurrent conflicts may happen with its p-father or with nodes having p as p-father.

In order to prevent the concurrent execution of conflicting migrations, our protocol adopts the following strategy: whenever p wants to execute $p \xrightarrow{n} q$, the latter is performed if p is the p-root; otherwise p must have the permission of its p-father before performing $p \xrightarrow{n} q$.

Figure 3 illustrates our migration protocol for $p \xrightarrow{n} q$ which is in fact a best-effort protocol. p is a non root p-node.

try($p \xrightarrow{n} q$) is invoked by the migration planner process (described bellow) when the latter wishes to migrate the v-node n from p-node p to the p-node q.

The migration request will be dropped (DROP CASE) if either p is already executing the migration protocol due to a previous v-node migration request or the $p \xrightarrow{n} q$ lasts too long (to avoid disturbing too much the application that run on top of it). Dropping the request will prevent further conflicts. Notice that migrations are expected to be sporadic and not very frequent for a given p-node. Thus, the drop of a migration request will be quite rare.

If $p \xrightarrow{n} q$ is in conflict with another concurrent migration that p has already allowed, the migration protocol is aborted (ABORT CASE 1). In other words, if p has given its permission to $q \xrightarrow{m \in v-children(n)} r$, p must abort $p \xrightarrow{n} q$ in order to avoid the concurrent executions of these two conflicting migrations. On the other hand, if there is no conflict, p invokes **permission(**$p \xrightarrow{n} q$**)** asking its p-father if the latter is not trying to perform any migration that would conflict with $p \xrightarrow{n} q$. If it is the case, p will receive a negative answer (ABORT CASE 2); otherwise, p-father(p) adds $p \xrightarrow{n} q$ to its set of granted migrations and returns a positive answer to p which means that it allows p to execute the migration for which it asked permission. Remark that ABORT CASE 2 happens when p-node attempts to execute $p \xrightarrow{n} q$ but its p-father does not gives it permission for such a migration, i.e., p-father(p) is concurrently executing $p - father(p) \xrightarrow{v-father(n)} p$.

Upon receiving its p-father's permission, p invokes **execute(**$p \xrightarrow{n} q$**)** for actually performing the migration $p \xrightarrow{n} q$. If some error happens during $p \xrightarrow{n} q$ (e.g. technical issues, timeouts, etc.), the migration protocol aborts (ABORT CASE 3). Otherwise, the migration is accomplished.

Finally, **complete(**$p \xrightarrow{n} q$**)** is invoked by p in order to inform its p-father that the migration $p \xrightarrow{n} q$, which it has allowed, was correctly performed. p-father(p) then removes $p \xrightarrow{n} q$ from its set of granted migrations.

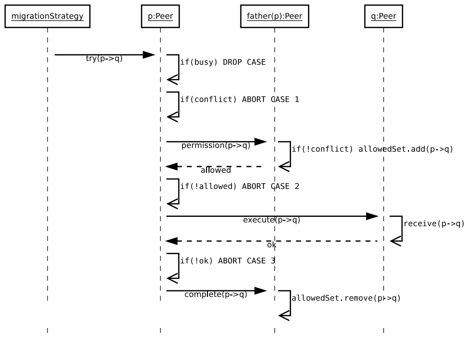


Fig. 3. A straightforward execution of the migration of n from p to q

4.3 Migration Planner

Migration planner defines the exact time the migration should be executed, i.e., when the migration protocol must be invoked. Such an invocation can either take place periodically or whenever there is an event that changes the logical structure of the DR-Tree. Furthermore, an evaluation of the added value of the migration will have on the system must be taken into account in both cases.

In the first case, the migration planner can be implemented as a separated process. Some cost functions (e.g., load of nodes) may be periodically evaluated and when a given value is reached, the planner process is woken up in order to invoke the migration conflict solver. In the second case, the invocation of the migration conflict solver is triggered in reaction to some event that changes the DR-Tree. For instance, when a v-node split occurs, the migration planner should call the migration conflict solver protocol in order to try to map the new v-nodes to the most suitable p-nodes. This approach can also be exploited for the initial placement of the DR-tree.

The migration planner will invoke the migration provided this one improves the performance of the system. Thus, a cost function is evaluated periodically or whenever a DR-Tree event might induce a migration. If this function signals that a proposed migration will degrade the system, the migration will not take place.

A cost function can concern one or more different metrics (e.g., message traffic, capacity of machines, etc.) and $p \xrightarrow{n} q$ involves p-nodes holding v-neighbors of n, as described in section 4.1. The *cost* function called by p is thus evaluated based on the information about the cost of p's p-neighbor. p can dynamically update this information by spoofing or sporadically evaluating the network traffic. An example of *cost* function is presented in Section 5.

The lack of global knowledge of a p-node cost function (e.g. it just exchanges information with its neighbors at the physical interaction graph) may limit the effectiveness of our migration mechanism, i.e., some performance enhanced configurations might not be reachable. For instance, let suppose that in the configuration of Figure 3, $p2$ has the best bandwidth; it should hold v-nodes of higher levels of the DR-tree since the latter are usually more loaded nodes than v-leaves. However, if, according to $p1$'s cost function, both $p1 \xrightarrow{n0} p5$ and $p1 \xrightarrow{n0} p7$ would degrade the bandwidth of the communication graph, $p1$ does not migrate $n0$. Therefore, while $p1$ holds $n0$, $p1 \xrightarrow{n1} p2$ is forbidden since it violates the distribution invariants. The local view of cost functions will not allow the migration of v-nodes to $p2$, even if such migrations would would enhance the bandwidth of the communication graph.

5 Evaluation

Our evaluation experiments were conducted on a discrete ad-hoc simulator. At the start of a cycle, a p-node may check if it can try to improve the performance of the system by migrating its *migrable* v-node, if it has one. To this end, the p-node calls a score function which evaluates the benefit of a possible migration of this v-node.

A migration $m{=}p \xrightarrow{n} q$ concerns p-nodes holding v-neighbors of n; let note this set of p-nodes $concerned(m)$. We assume that p has some knowledge about the *cost* between its p-neighbors. To decide whether m is worth or not, we use the following *score* function:

$$score(m) = \sum_{r}^{concerned(m)} cost(p,r) - cost(q,r)$$

If p computes that $score(m) > 0$ then, according to p, m may enhance system performances. Having scores of possible migration of n, p will try to execute the migration with the best positive score. In our experiments, the metric related to the cost function is the latency between p-nodes.

One thousand simulations were performed with different configurations of DR-tree, with the following parameters:

- 500 p-nodes
- DR-tree degree with m = 2 and M = 4
- each experiment lasts 500 cycles
- the probability that a p-node checks for migrations is 1/10

DR-trees were populated with v-leaves represented by 50*50 2D rectangles randomly distributed in a 1024*1024 map. In order to simulate latency between p-nodes, we use the Meridian data set [16]; it is a latency matrix that reflects the median of round-trip times between 2500x2500 physical nodes spreads on Internet. For our experiments, we extracted a 500x500 sub-matrix.

5.1 Impact of Migrations in All DR-tree Configurations

Impact on latency. We firstly studied the impact of migrations on the latency between two p-nodes of the interaction graph.

Figure 4a shows the average latency gain due to migrations. Simulations with different DR-tree configurations were ordered by increasing average latency without migration (X-axis). Y-axis corresponds to the average latency in millisecond in the communication graph when the system is stabilized, i.e., no more migration takes place.

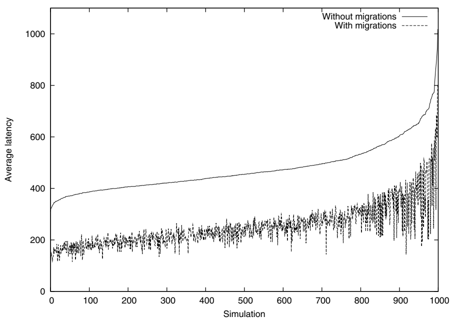
The average latency obtained without migrations -and thus without taking latency heterogeneity into account- is highly dependent on the mapping of v-root and its close neighborhood. In a small number of cases, this mapping was well-suited (resp. bad-suited), leading to an average latency of 300ms (resp. 1000ms). However, in almost 80% of simulations, the average latency was between 350ms and 600ms.

As we can observe in the same figure, migration of v-nodes clearly enhances average latency. The peaks of the curve can be similarly explained: in a small number of cases, many and/or very effective (resp. few and/or less effective) migrations were applied, leading to an average latency of 150ms (resp. 800ms). In most cases, the gain is around 50%.

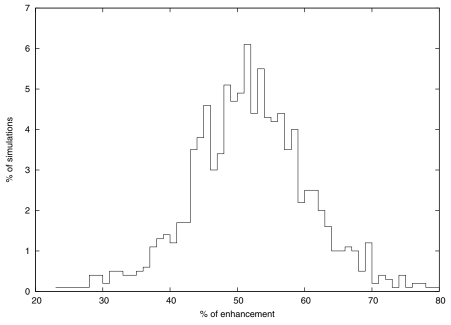
Figure 4b shows the gain distribution. X-axis is the percentage of enhancement reached during a simulation. Y-axis is the percentage of simulations where x gain has been reached. The gain distribution is quite gaussian: 78.7% of simulations raise a gain between 40% and 60%.

Stabilization time. Figure 4c shows the distribution of the stabilization time. X-axis is the last cycle where a migration was executed while Y-axis is the percentage of simulations where the last migration was executed at time x.

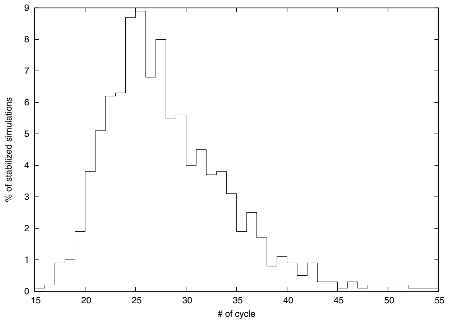
The distribution of stabilization time is also quite gaussian: 92.9% of simulations converge in a number of cycles between 20 and 40. Even with p-nodes “rarely” checking if they can perform migrations, stabilization is reached very fast: simulations last 500 cycles and 96.1% of them converge in less than 40 cycles (100% in less than 55 cycles).



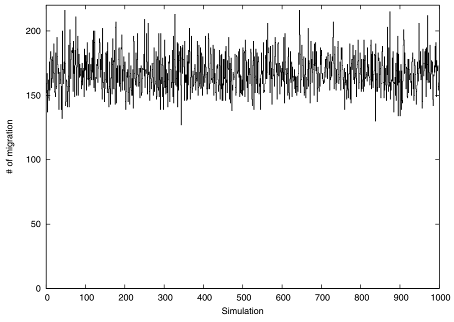
(a) Average latency enhancement



(b) Average latency enhancement distribution



(c) Stabilization time distribution



(d) Number of migrations per simulation

min	max	avg	stddev
15	55	27.494	5.84209

(e) Stabilization time distribution stat

min	max	avg	stddev
127	216	167.793	14.530

(f) Number of migrations per simulation stat

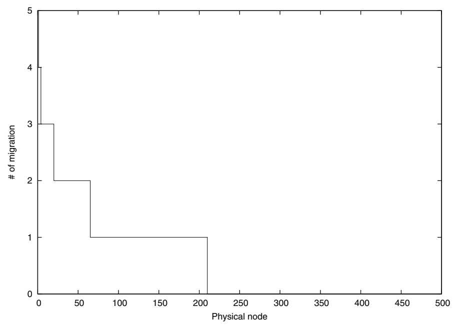
Fig. 4. Measurements on 1000 simulations

Overall number of migrations. Figure 4d shows the number of migrations executed per simulation.

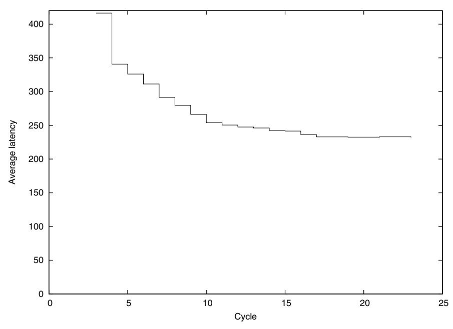
We observe a relatively stable number of migrations around a third of p-nodes with a low standard deviation. This suggests that different configurations of DR-tree with the same degree has low impact on the overall number of migrations. In fact, this number depends in fact on the distribution of latencies between p-nodes which is fixed in our experiments.

5.2 Analysis of Migrations in a Given DR-tree Configuration

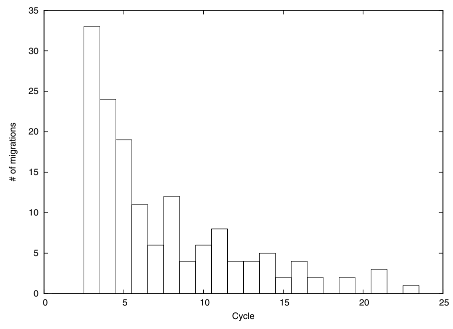
Among the previous simulations, we have chosen one where the corresponding DR-tree configuration presents a distribution enhancement of 50% (Figure 4b), a stabilization time of 25 cycles (Figure 4c), and 173 migrations executions (Figure 4d). Evaluation results presented in the following are based on this simulation.



(a) Number of migrations per p-node



(b) Average latency evolution over time



(c) Number of migrations per cycle

Fig. 5. Measurements on a very "average" simulation

In Figure 5a, we can observe the number of migration per p-node during the referenced simulation. More than half of p-nodes do not participate in any migration and no p-node participates in more than five migrations. Furthermore, the distribution of per p-node migrations is relatively well-balanced since no p-node is overloaded by a high number of migrations and only 4 % of nodes perform more than two migrations.

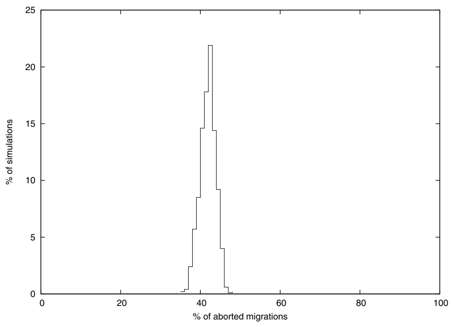
Figure 5b shows the evolution of average latency between p-nodes in the interaction graph till stabilization time. All p-nodes start migration planner at the first cycle of the simulation. However, the first migrations actually occurs at the fourth cycle since our migration protocol takes four cycles to fully commit a v-node migration. Due to our migration protocol too, during the first cycles, migrations of high level p-nodes (and thus v-nodes closer to v-leaves) are likely to be aborted as their respective fathers are also likely to be executing other migrations. Therefore, the first executed migrations are more likely to concern lower level p-nodes (and thus low level v-nodes, close to the v-root) than higher level ones. Furthermore, the further from the v-root a v-node is, the longer the path to the v-root is, and thus its migration has a higher impact in the overall average latency. After these first migrations, the others are more likely to be small adjustments just inducing local communication enhancements which thus do not reduce average latency.

The number of migrations executed per cycle for the same simulation is given in Figure 5c. Y-axis is the number of executed migrations. The results shown in this figure confirm our previous analysis of Figure 5b: since all p-nodes start the migration planner during first cycle, many of them execute migrations during the fourth cycles. Since all p-node join the system during the first cycle, the higher the number of cycles executed, the smaller the number of executed migrations.

5.3 Abort Rate

Our migration protocol is based on a best effort approach which implies that attempts of v-node migration may be aborted. Moreover, abortion of migrations has a cost due to the messages exchanged by the involved p-nodes till the protocol is aborted.

Figure 6 shows the abort rate distribution. X-axis is the percentage of aborted migrations while Y-axis is the percentage of simulations having *x%* of aborted migrations.



min	max	avg	stddev
51.929%	64.591%	58.130%	1.969

Fig. 6. Abort rate distribution

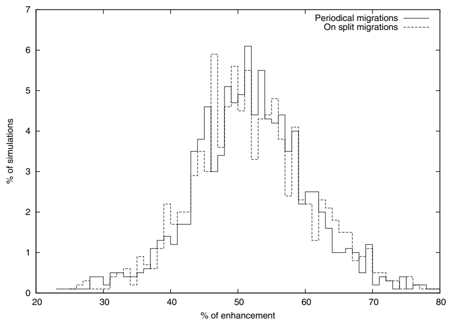
The rate of aborted migration is related to both the frequency of p-nodes attempts for migrating a v-node and the DR-tree degree. The higher this frequency is, the higher chances are that a p-node and its p-father try to execute concurrently conflicting migrations are. On the other hand, the higher the degree of the DR-tree is, the higher the chances are that a p-node can execute a migration and thus abort some of its p-neighbors migration attempts.

5.4 Migration Scheduling Impact

In this section, we compare two migration scheduling strategies for executing the migration planner: *periodical* strategy where the planner is executed periodically, and *triggered* strategy where the planner is executed whenever a v-node is split.

Each simulation is basically composed of two main phases: system building and system "lifetime". The former starts at the simulation initialization and ends when the last p-node has joined the system while the latter starts after the last p-node has joined the system and ends at the simulation termination. In the periodically (resp. triggered) strategy, migrations only takes place during the system "lifetime" (resp. building);

Figure [7] shows the distribution gain of the two migration scheduling. X-axis is the percentage of enhancement reached during a simulation while Y-axis is the percentage of simulations where *x* gain has been reached. We can observe that both strategies are rather similar in terms of gain.



	min	max	avg	stddev
Periodical	-23.173	-80.161	-51.998	8.460
Split triggered	-25.241	-80.168	-51.979	8.823

Fig. 7. Migration scheduling impact on average latency enhancement

6 Conclusion

Our article has shown that it is very interesting to exploit the relation between the logical structure of a DR-tree [2] and its corresponding physical interaction graph. By expressing some invariants, a dynamic migration mechanism can modify the distribution of DR-tree v-nodes over the physical network without modifying its logical structure. Evaluation results have confirmed that even a very simple best-effort dynamic migration protocol substantially improves system performance in terms of latency. Finally, we should point out that our dynamic v-node migration mechanism could be applied to other logical structures. Concepts such as v-nodes, p-nodes, distribution invariants, migration policy, migration conflict management, migration planner, etc. can be easily generalized in order to satisfy other logical structures requirements.

References

1. Baehni, S., Eugster, P.T., Guerraoui, R.: Data-aware multicast. In: DSN, p. 233 (2004)
2. Bianchi, S., Datta, A.K., Felber, P., Gradinariu, M.: Stabilizing peer-to-peer spatial filters. In: ICDCS 2007, p. 27 (2007)
3. Paris, C., Vana, K.: A topologically-aware overlay tree for efficient and low-latency media streaming. In: QSHINE, pp. 383–399 (2009)
4. Zhang, R.Y.W.C., Krishnamurthy, A.: Brushwood: Distributed trees in peer-to-peer systems, pp. 47–57 (2005)
5. Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A.I.T., Singh, A.: Splitstream: High-bandwidth content distribution in cooperative environments. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 292–303. Springer, Heidelberg (2003)

6. Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Constructing scalable overlays for pub-sub with many topics. In: PODC, pp. 109–118 (2007)
7. Choi, Y., Park, D.: Mirinae: A peer-to-peer overlay network for large-scale content-based publish/subscribe systems. In: NOSSDAV, pp. 105–110 (2005)
8. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kouznetsov, P., Kermarrec, A.-M.: Lightweight probabilistic broadcast. ACM Trans. Comput. Syst. 21(4), 341–374 (2003)
9. Guerraoui, R., Handurukande, S.B., Huguenin, K., Kermarrec, A.-M., Fessant, F.L., Riviere, E.: Gosskip, an efficient, fault-tolerant and self organizing overlay using gossip-based construction and skip-lists principles. In: Peer-to-Peer Computing, pp. 12–22 (2006)
10. Gupta, A., Sahin, O.D., Agrawal, D., Abbadi, A.E.: Meghdoot: Content-based publish/Subscribe over P2P networks. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 254–273. Springer, Heidelberg (2004)
11. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: ACM SIGMOD, pp. 47–57 (1984)
12. Harvey, N.J.A., Munro, J.I.: Deterministic skipnet. Inf. Process. Lett. 90(4), 205–208 (2004)
13. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: a balanced tree structure for peer-to-peer networks. In: VLDB, pp. 661–672 (2005)
14. Jagadish, H.V., Ooi, B.C., Vu, Q.H., Zhang, R., Zhou, A.: Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In: ICDE, p. 34 (2006)
15. Joung, Y.-J.: Approaching neighbor proximity and load balance for range query in p2p networks. Comput. Netw. 52(7), 1451–1472 (2008)
16. Madhyastha, H.V., Anderson, T., Krishnamurthy, A., Spring, N., Venkataramani, A.: A structural approach to latency prediction. In: IMC, pp. 99–104 (2006)
17. Pujol Ahullo, J., Garcia Lopez, P., Gomez Skarmeta, A.F.: Towards a lightweight content-based publish/subscribe services for peer-to-peer systems. Int. J. Grid Util. Comput. 1(3), 239–251 (2009)
18. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: SIGCOMM, pp. 161–172 (2001)
19. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
20. Terpstra, W.W., Behnel, S., Fiege, L., Zeidler, A., Buchmann, A.P.: A peer-to-peer approach to content-based publish/subscribe. In: DEBS, pp. 1–8 (2003)

A Quantitative Analysis of Redundancy Schemes for Peer-to-Peer Storage Systems

Yaser Houri, Johanna Amann, and Thomas Fuhrmann

Technische Universität München
{houri,ja,fuhrmann}@so.in.tum.de

Abstract. Fully decentralized peer-to-peer (P2P) storage systems lack the reliability guarantees that centralized systems can give. They need to rely on the system's statistical properties, only. Nevertheless, such probabilistic guarantees can lead to highly reliable systems. Moreover, their statistical nature makes P2P storage systems an ideal supplement to centralized storage systems, because they fail in entirely different circumstances than centralized systems.

In this paper, we investigate the behavior of different replication and erasure code schemes as peers fail. We calculate the data loss probability and the repairing delay, which is caused by the peers' limited bandwidth. Using a Weibull model to describe peer behavior, we show that there are four different loss processes that affect the availability and durability of the data: initial loss, diurnal loss, early loss, and longterm loss. They need to be treated differently to obtain optimal results. Based on this insight we give general recommendations for the design of redundancy schemes in P2P storage systems.

1 Introduction

Peer-to-peer systems can provide inexpensive, scalable, and reliable distributed storage. They can serve a wide field of potential applications. Nomadic users, for example, can store their personal data in a location transparent manner. Private users can share their resources and reliably store their personal data independently, i. e. without having to rely on service providers such as Google or Amazon.

P2P storage systems must store the data redundantly, because peers may fail or leave the system at any time. Moreover, as we show, the initial creation of redundancy is necessary but not sufficient. The system must repair lost redundancy over time. It must do so in time, i. e. before the redundancy is depleted and the data is permanently lost.

Many peer-to-peer storage systems have been developed in the recent years. Most of them provide reliable storage on top of unreliably collaborating peers. To this end, they use different redundancy techniques, typically replication, erasure coding or combinations of those. They also use different maintenance policies to ensure data durability.

Providing and maintaining redundancy consumes network bandwidth and storage space. Today's hard disks provide a huge storage space, but network bandwidth is typically still a scarce resource for the home user and all the more for the nomadic user. As we will show, the network bandwidth is indeed the limiting resource and should thus be used wisely.

Several studies [8, 11, 16] have tried to determine when the repairing process should be invoked. But to the best of our knowledge no one has considered the peers' bandwidth limitation in this context. It causes a delay until the repairing process completes. During this time more losses can occur, and eventually, the repair process might even fail.

In this paper, we investigate the behavior of different replication and erasure codes schemes when peers become unavailable. We calculate the bandwidth costs and the repairing delay caused by the peers' bandwidth limitation. Depending on our calculated parameters (*data loss rate*, *bandwidth consumption*, and *repairing delay*), we can choose the appropriate redundancy scheme and the proper time when to invoke the repairing process for our application scenario.

The remainder of this paper is organized as follows. In section 2 we discuss the relevant related work. In section 3 we analyze the behavior of different redundancy schemes. Finally, we conclude with a summary of our findings in section 4.

2 Related Work

In the course of the recent years many P2P storage systems have been proposed, which use different techniques to ensure reliable storage. Their authors discuss the issues of choosing the right data redundancy strategy, fragment placement, and redundancy maintenance, both analytically and through simulation, mainly to determine when it becomes necessary to replenish lost redundancy to avoid permanent data loss.

DHash [7] and pStore[3] both use replication to ensure data availability and durability. DHash places replica on selected peers and uses an *eager repair policy* to maintain that redundancy. Using an eager repair policy wastes storage space and bandwidth when the temporarily unavailable peers return. Unlike DHash, pStore[3] places replicas on random peers. Tempo [18] repairs the redundancy proactively, which overly consumes bandwidth and storage space.

TotalRecall [4] and OceanStore [14] both use erasure coding to reduce storage overhead. They place the data randomly on the participating peers. Unlike OceanStore, TotalRecall uses a *lazy repair policy*, which allows the reintegration of the temporarily unavailable fragments. Carbonite [6] extends this policy to allow full reintegration of the temporarily unavailable fragments. It uses a specific multicast mechanism to monitor the availability of the fragments.

Using a lazy repair policy can reduce the maintenance bandwidth, but it can also cause data loss if peers do not return as expected. Our paper aims at clarifying how much eagerness and laziness a redundancy scheme should have.

Besides the choice of the right repair policy, it is also important to consider the effect caused by the peers' limited bandwidth. Data availability and durability can not be retained when repair process does not finish before too much redundancy has been lost. To the best of our knowledge, our paper is the first to address this effect in detail.

The authors in [4, 7, 15, 21] compared replication and erasure codes in terms of storage overhead. They showed that erasure codes occupy less storage space than replication while retaining the same reliability level. Lin et al. [15] concluded that replication strategy is more appropriate for systems with low peers' availability.

Rodrigues et al. [17] and Utard et al. [20] argued that the maintenance costs for erasure codes are higher than replication which is a limiting factor for the scalability of P2P storage system [5]. They suggested a hybrid solution by maintaining a copy of the data object at one peer, while storing the fragments on other peers. Their solution creates a bottleneck due to the large number of fragments that should be replaced when peers leave the system. Moreover, ensuring data objects availability adds great complexity to the system, because it should maintain two types of redundancy.

Motivated by network coding [2, 13], Dimakis et al. developed a solution to overcome the complication of the hybrid strategy [9]. They applied the random linear network coding approach of Ho et al. [13] to a *Maximum-Distance Separable* (MDS) erasure code for storing data. The performance of their proposed scheme proved to be worse than erasure codes unless the new peers keep all the data they download.

Acedański et al. showed mathematically and through simulation [1] that random linear coding strategy delivers a higher reliability level than erasure codes while occupying much less storage space. They suggested to cut a file into m pieces and store k random linear combination of these pieces with their associated vectors in each peer. A similar strategy was developed by Gkantsidis et al. in [12] for content distribution networks. But in both papers, the authors neglected the cost for repairing the redundancy. Therefore their results come in favor of random linear coding.

Duminuco et al. [10] showed that using linear coding causes high costs for redundancy maintenance compared to simple replication. They introduced a new coding scheme called *Hierarchical Codes* which offers a trade-off between storage efficiency and repairing costs. Their coding scheme has much lower reliability level compared to erasure codes.

3 Redundancy Scheme Analysis

Our work aims at finding the most appropriate redundancy scheme for peer-to-peer storage systems. In this paper, we consider the most common redundancy techniques in storage systems: replication and erasure coding. Our analysis reflects both, the inevitable churn among the peers and the limited bandwidth of the peers. We compare the data loss rates of the different redundancy schemes, their maintenance costs in terms of bandwidth consumption, and the corresponding repair time depending on the peers' available bandwidth. (We omit the network latency, because it is typically small as compared to the delay caused by the bandwidth limitation.)

In this paper we study the problem mainly analytically. We consider erasure coding, where a data block is split into m fragments, which are then encoded into n fragments. We also consider replication, which can be viewed as special case with $m = 1$. We assume a large system, so that we can model peer failures by simple random sampling with replacement. In particular, we assume the number of peers N to be much larger than the number of redundancy fragments n per data block, i. e. $N \gg n$. This is a weak assumption that should hold for all practical purposes.

First, we analyze a kind of static scenario, where we do not consider when the peers join and leave the system. We rather assume that a given fraction of peers has left, and we need to repair the lost redundancy. This models a case where suddenly a large number of peers fail in correlation, e. g., because of network problems. Luckily, severe network outages that cause such correlated failures are rare and transient. Thus, we do not seek to propose redundancy schemes that guard against such failures. (I. e. we do not consider it worth the additional bandwidth and storage cost.)

We then rather focus on the dynamic process of peer churn. To this end, we combine the results from the static analysis with a dynamic model, which we derive from a measurement trace of a peer-to-peer network. Thereby, we obtain the time-variant loss probabilities and the required repair traffic bandwidth.

Besides our analytical work, we have also confirmed our analysis with simulations. As expected, we have obtained results that deviate only within the statistical range error. Thus we do not discuss these simulations extensively here.

3.1 Data Loss Probability

Each peer in our model can be either be available (with probability p) or fail (with probability $1 - p$). Moreover, we assume that the underlying random process is independent of the peer and its content. I. e. we assume that there is no adversary that picks the peers to explicitly kill a data block. Thus, we can model the peers as independent identically distributed (iid) random variables X_i that are distributed according to a *Bernoulli distribution.*

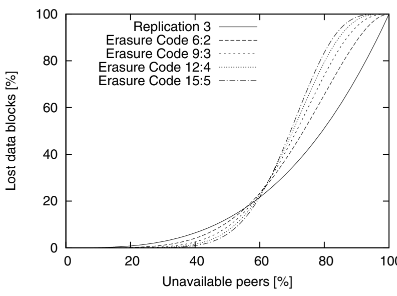
The probability to lose exactly k out of the n fragments or replicas of a data block follows a binomial distribution:

$$P_k = \binom{n}{k}(1-p)^k p^{n-k}$$

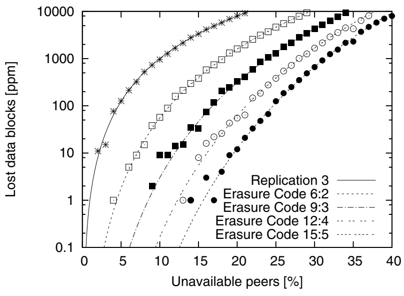
The probability to irrecoverably lose data, i. e. the probability to lose more than m of the n fragments is

$$\begin{aligned} P &= \sum_{k=m+1}^{n} P_k \\ &= \sum_{k=m+1}^{n} \frac{n!}{k!(n-k)!}(1-p)^k p^{n-k} \\ &= I_{1-p}(m+1, n-m) \end{aligned}$$

where I_{1-p} is the incomplete beta function.



(a) Replication vs. Erasure codes



(b) A closer look at low loss probabilities

Fig. 1. Data Loss with Replication versus Erasure Coding

Figure 1(a) shows the resulting behavior: When using a replication scheme, the number of unavailable data blocks rises only slowly with the number of unavailable peers. The curves for erasure coding rise more steeply. The larger the number of fragments per block, the steeper the gradient. Hence, erasure coding leads to lower data loss rates when only a few peers have become unavailable, whereas replication can better handle situations where a large number of peers have failed.

However, this plot is misleading because it draws the attention to the behavior at large loss probabilities. Given that a reliable storage system must not lose data at all, we had rather focus on very small loss probabilities. Figure 1(b) thus shows a respective close-up. As we can see, here erasure codes outperform replication. A 15:5 erasure coding scheme loses less than one 'part per million' (ppm), even when about 15% of the peers have become unavailable.

3.2 Repair Costs

In practice, most of the peers in a P2P network leave the system again soon after they have joined. Thus, to ensure the system's reliability and prevent data loss, it is important to timely replenish the redundancy that has been lost. But since some of the peers return after a while, replenishing the redundancy too early unnecessarily wastes bandwidth.

Let us first analyze the static case: Assume that a fraction x of all peers has suddenly left the system, and we want to immediately replenish the thereby lost redundancy. What amount of data do we need to send over the network for the different redundancy schemes?

For our calculation, we again assume statistical independence, i. e. we assume that a peer does not store more than one replica or redundancy fragment of the same data block.

Moreover, we assume that one of the remaining peers of each redundancy group does the repair, i. e. replenishes the redundancy. In practice, this requires coordination among the peers, which contributes to the communication overhead. For simplicity, we neglect this overhead, here. We also neglect the overhead that the respective peer causes in discovering that redundancy has been lost. However, in both cases the overhead is small as compared to the repair traffic, because we seek to probe and repair at the right time, i. e. when there is a significant probability that redundancy has been lost. Thus, there is only a small number of probes per repair activity, and we can neglect the coordination and probing overhead as compared to the transfer of the data itself.

As already said, the probability to lose k out of n fragments follows a binomial distribution. To replenish the redundancy, the repairing peer has to first load $m-1$ other fragments, and then store k of the regenerated fragments in the network. (A replication scheme has $m=1$, i. e. the peer does not need to load any data.)

Each fragment has the size m^{-1}. If $k=0$, there is no need to replenish the redundancy. If $k > n-m$, there is not enough redundancy left to successfully perform the repair process.

Hence, the resulting bandwidth cost C is

$$\begin{aligned} C &= \frac{m-1}{m}\sum_{k=1}^{n-m} P_k + \frac{1}{m}\sum_{k=1}^{n-m} kP_k \\ &= \frac{1}{m}\sum_{k=1}^{n-m}(m-1+k)P_k \end{aligned}$$

Here, $C=1$ means that the system transfers an amount of data that is equal to the stored amount of data. I. e. if a peer has stored 1 GB in the system, $C=1$ means that it must fetch and write another 1 GB of redundancy data.

Figure 2 shows the resulting effect. The bandwidth costs rise steeply and peak between 50% and 60% unavailable peers. Beyond that, the costs fall again, because the system is increasingly unable to replenish the lost redundancy. If all peers have become unavailable, there is no bandwidth cost at all, because all data has been irrecoverably lost.

We see that the costs rise more quickly for erasure coding than for replication. The reason is that erasure codes require the peers to download other fragments before they can replenish the redundancy. Furthermore, the costs peak higher for erasure codes than for replication. The reason is that erasure codes withstand the loss of peers better than replication, i. e. replication has to give up earlier.

Some authors have proposed that there is one peer that stores all fragments of an erasure coding scheme to be prepared in case it needs to repair the redundancy later [5]. Such a redundancy scheme is a mixture of replication and erasure coding. In particular, it introduces an asymmetry among the peers, because that peer must not fail. If the system has to guard against failures of that peer too, the redundancy scheme is a again a replication scheme.

3.3 Repair Time

When peers become unavailable, the remaining peers need to replenish the lost redundancy to avoid permanent data loss. Given that peers have only a limited

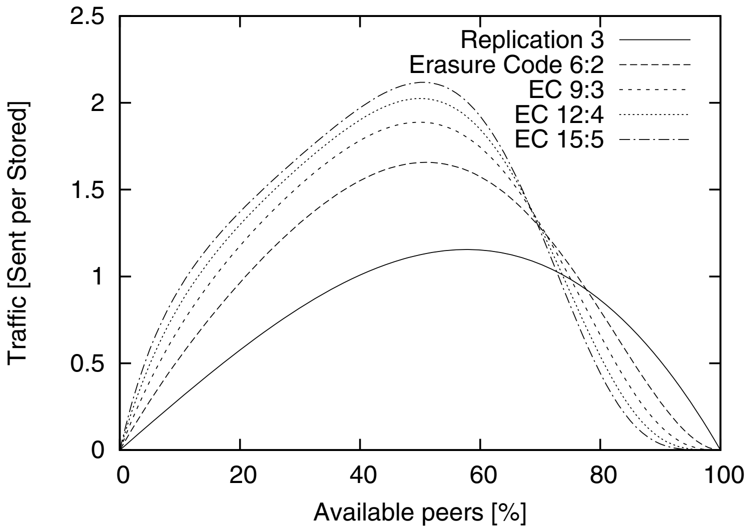


Fig. 2. Cost to Replenish the Redundancy

bandwidth for download and upload, replenishing the lost redundancy may require a significant amount of time. During this time, more peers will become unavailable, so that the system might fail to provide durability of the stored data.

In general, peer availability has often been found to follow a Weibull distribution [19], i. e.

$$p(t) \propto t^{-\alpha} \exp(-\beta t^{1-\alpha})$$

For $\alpha = 0$ the distribution becomes an exponential distribution. For $\beta = 0$ the distribution becomes a power-law distribution.

Figure 3 shows an autocorrelation analysis of a trace from the Kademlia network [19], which confirms the principal assumption of a Weibull distribution. As we can see, about 15% of the peers stay for less than one hour in the system. About half of the peers stay for less than a day. Moreover, we see a strong diurnal pattern, and weaker weekly pattern. Introducing the diurnal and weekly pattern into the Weibull model, and introducing a time offset to be able to reflect the distribution at $t = 0$, we arrive at the following peer availability function:

$$\begin{aligned} p(t) = {} & \alpha_1 (t+T)^{-\beta_1} e^{\gamma_1 (t+T)^{1-\beta_1}} \\ + \quad & \alpha_2 (t+T)^{-\beta_2} e^{\gamma_2 (t+T)^{1-\beta_2}} \cos(\frac{2\pi t}{24}) \\ + \quad & \alpha_3 (t+T)^{-\beta_3} e^{\gamma_3 (t+T)^{1-\beta_3}} \cos(\frac{2\pi t}{7 \cdot 24}) \end{aligned}$$

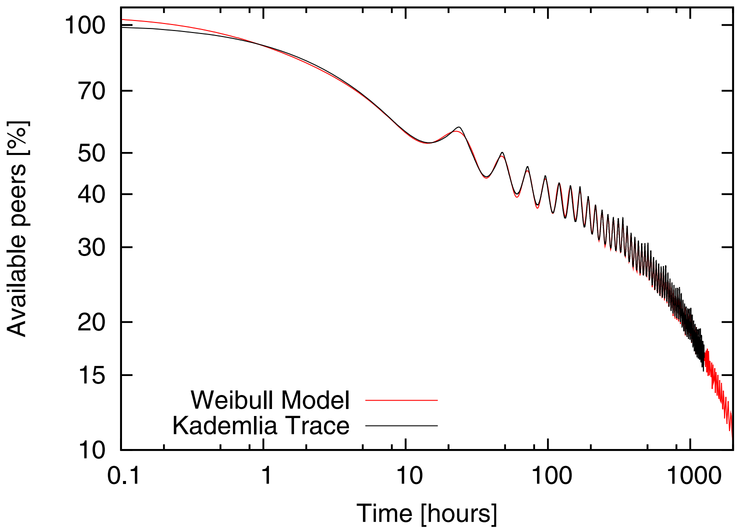


Fig. 3. Autocorrelation of peer availability

Table 1 gives the fitted parameters.

Table 1. Parameters for the Weibull Model

$$T = 1.00040 \pm 3.87\% \text{ hours}$$

$$\alpha_1 = 0.111133 \pm 0.18\%$$
$$\beta_1 = 0.001198 \pm 0.07\%$$
$$\gamma_1 = 0.728504 \pm 0.09\%$$

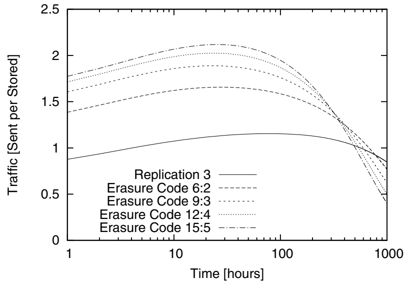
$$\alpha_2 = 0.105977 \pm 2.37\%$$
$$\beta_2 = 0.001463 \pm 0.95\%$$
$$\gamma_2 = 0.062237 \pm 1.12\%$$

$$\alpha_3 = 0.677504 \pm 2.21\%$$
$$\beta_3 = 0.044204 \pm 14.73\%$$
$$\gamma_3 = 0.278142 \pm 2.85\%$$

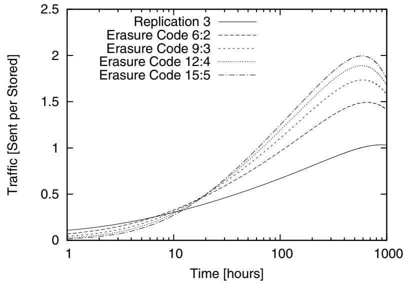
The Kademlia trace and the Weibull model indicate that we have to differentiate between four different loss components: initial loss, early loss, diurnal loss, and longterm loss.

When a peer stores data in the network, about half of the redundancy is lost within the first 24 hours. If we tried to replenish the redundancy with an erasure coding scheme, the resulting traffic would immediately exceed the originally stored amount of data (cf. fig. 4(a)). The loss probabilities are immediately above one percent (cf. fig. 4(c)).

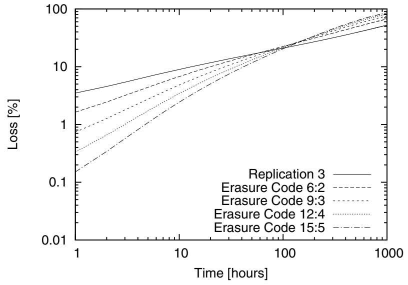
In practice, the peers cannot send such an amount of repair traffic instantaneously. They would have to equally share their bandwidth between the initially



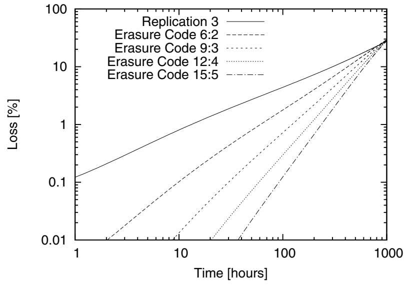
(a) Repair traffic w/o extra setup redundancy



(b) Repair traffic w/ extra setup redundancy



(c) Loss probability w/o extra setup redundancy



(d) Loss probability w/ extra setup redundancy

Fig. 4. Extra redundancy at setup time improves reliability

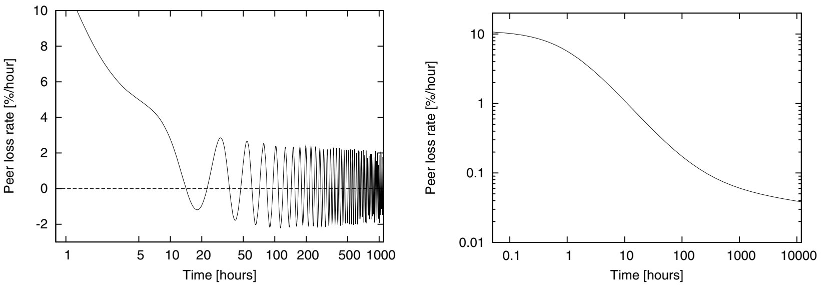
stored data and the repair traffic. Thus, it makes more sense to use the bandwidth to initially store more than the required redundancy to avoid the repair traffic. Thereby, the system can better withstand the initial loss of the replicas and redundancy fragments. After that, the peers do not become unavailable that quickly any more, and the repair process can keep pace.

Figs. 4(b) and 4(d) show the example with twice as much initial redundancy. As we can see, the proposed initial excess redundancy significantly improves the situation. The probability to lose data during the first day is very low. The required repair traffic remains well below a ratio of one.

Nevertheless, the extra redundancy cannot entirely avoid applying a repair process during the 'early days' of the stored data. Depending on the desired reliability, the peers have to probe and replenish the redundancy on a time frame of a few hours. As we will see, the probing frequency can be reduced over the first days and weeks until it reaches a low, permanent level.

After the first day, the diurnal oscillations begin to dominate the peer churn process (cf. fig. 5). Data can become unavailable for a few hours until enough peers have come back again. If the requesting peer follows the diurnal pattern, i. e. if it stores and retrieves data always at the same time of day, it will not notice the diurnal process. Otherwise, i. e. if the data has to be available all the time, extra repair effort might be required.

As we can see from fig. 5(a), the diurnal churn rate is similar to the loss rate during the first day. However, the fraction of nodes that follow the diurnal process is small (at least in the measured Kademlia network). Thus, most of the redundancy will be stored on the non-diurnal peers. Furthermore, since most of the diurnal peers return after a day, the extra repair is typically required only during the first one or two days. i. e. diurnal replenishment populates the peers at different phases of the diurnal cycle and can stop afterwards.



(a) With diurnal churn component (b) Without diurnal churn component

Fig. 5. Peer churn rates in the Weibull model

In the long run the replenishment process has to follow the regular pattern that is dominated by the power-law component of the Weibull model. As we can see from fig. 5(b), the loss rate drops about two orders of magnitude over the first days and weeks. Accordingly, the redundancy maintenance rate can also be reduced in a similar way, i. e. the probing and repair intervals can be extended following a power-law.

After a few weeks, the exponential component of the Weibull model dominates the loss process, and the redundancy maintenance rate must be kept at a low, but constant level.

4 Conclusion

Reliability is important for peer-to-peer storage systems. But so far, the design of these systems was mainly guided by static considerations: When is the provided redundancy depleted and needs to be repaired?

In this paper we discuss the question with the dynamical aspect in mind: When and how must the redundancy be repaired given that the peers have only a limited bandwidth available? Analyzing data from a Kademlia trace and its resulting Weibull model we find four different loss components: A large fraction of peers leaves within the first hours. The corresponding initial loss is best anticipated by an increased initial redundancy. During the first days and weeks the loss rate follows a power-law. It should thus be probed and repaired in increasing intervals. If the peers need to guard against diurnal oscillations, they must create a small amount of additional redundancy during the first one or two days. After a few weeks, the loss is dominated by an exponential process, i. e. the redundancy must be probed and repaired at a low but constant rate.

References

1. Acedański, S., Deb, S., Médard, M., Kötter, R.: How good is random linear coding based distributed networked storage. In: Proc. of the 1st Workshop on Network Coding, Theory and Applications (NetCod), Riva del Garda, Italy (April 2005)
2. Ahlswede, R., Cai, N., Li, S.y.R., Yeung, R.W., Member, S.: Network information flow. IEEE Transactions on Information Theory 46, 1204–1216 (2000)
3. Batten, C., Barr, K., Saraf, A., Trepetin, S.: pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science (October 2002)
4. Bhagwan, R., Tati, K., Cheng, Y.C., Savage, S., Voelker, G.M.: Total recall: System support for automated availability management. In: Proc. of the 1st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2004 (2004)
5. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: pick two. In: Proc. of the 9th USENIX Conference on Hot Topics in Operating Systems (HOTOS 2003), Lihue, Hawaii (2003)
6. Chun, B.G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M.F., Kubiatowicz, J., Morris, R.: Efficient replica maintenance for distributed storage systems. In: Proc. of the 3rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2006 (2006)
7. Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M.F., Morris, R.: Designing a DHT for low latency and high throughput. In: Proc. of the 1st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2004 (2004)
8. Datta, A., Aberer, K.: Internet-scale storage systems under churn – a study of the steady-state using markov models. In: Proc. of the 6th IEEE Intl. Conference on Peer-to-Peer Computing (P2P 2006), Washington, DC (2006)
9. Dimakis, R.G., Godfrey, P.B., Wainwright, M.J., Ramch, K.: Network coding for distributed storage systems. In: Proc. of the 26th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2007 (2007)
10. Duminuco, A., Biersack, E.: Hierarchical codes: How to make erasure codes attractive for peer-to-peer storage systems. In: Proc. of the 8th International Conference on Peer-to-Peer Computing (P2P 2008), Aachen, Germany (September 2008)
11. Duminuco, A., Biersack, E., En-Najjary, T.: Proactive replication in distributed storage systems using machine availability estimation. In: Proc. of the 3rd ACM Intl. Conference on Emerging Networking Experiments and Technologies (CoNEXT 2007), New York, NY (2007)
12. Gkantsidis, C., Rodriguez, P.R.: Network coding for large scale content distribution. In: Proc. of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005), vol. 4, pp. 2235–2245 (March 2005)
13. Ho, T., Mdard, M., Koetter, R., Karger, D.R., Member, A., Effros, M., Member, S., Member, S., Member, S., Shi, J., Leong, B.: A random linear network coding approach to multicast. IEEE Trans. Inform. Theory 52, 4413–4430 (2006)

14. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: OceanStore: An architecture for global-scale persistent storage. In: Proc. of the 9th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2000 (2000)
15. Lin, W.K., Chiu, D.M., Lee, Y.B.: Erasure code replication revisited. In: Proc. of the 4th International Conference on Peer-to-Peer Computing (P2P 2004), Zurich, Switzerland (August 2004)
16. Ramabhadran, S.: Analysis of long-running replicated systems. In: Proc. of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2006), Barcelona, Spain (2006)
17. Rodrigues, R., Liskov, B.: High availability in DHTs: Erasure coding vs. replication. In: Castro, M., van Renesse, R. (eds.) IPTPS 2005. LNCS, vol. 3640, pp. 226–239. Springer, Heidelberg (2005)
18. Sit, E., Haeberlen, A., Dabek, F.,, B.: g. Chun, H. Weatherspoon, R. Morris, M.F. Kaashoek, J. Kubiatowicz. Proactive replication for data durability. In: Proc. of the 5th Intl. Workshop on Peer-to-Peer Systems (IPTPS 2006), Santa Barbara, CA (February 2006)
19. Steiner, M., En-Najjary, T., Biersack, E.W.: Long term study of peer behavior in the KAD DHT. IEEE/ACM Transactions on Networking 17(5) (October 2009)
20. Utard, G., Vernois, A.: Data durability in peer to peer storage systems. In: Proc. of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), Chicago, Illinois (April 2004)
21. Weatherspoon, H., Kubiatowicz, J.: Erasure Coding vs. Replication: A Quantitative Comparison. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, p. 328. Springer, Heidelberg (2002)

A Framework for Secure and Private P2P Publish/Subscribe

Samuel Bernard, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil

UPMC Univ. Paris 6 LIP6/CNRS UMR 7606
samuel.bernard@lip6.fr,
maria.gradinariu@lip6.fr,
sebastien.tixeuil@lip6.fr

Abstract. We propose a novel and totally decentralized strategy for private and secure data exchange in peer-to-peer systems. Our scheme is particularly appealing for point-to-point exchanges and use zero-knowledge mechanisms to preserve privacy. Furthermore, we show how to plug our private and secure data exchange module in existing publish/subscribe architectures. Our proposal enriches the original system with security and privacy making it resilient to a broad class of attacks (e.g. brute-force, eavesdroppers, man-in-the middle or malicious insiders). Additionally, the original properties of the publish/subscribe system are preserved without any degradation. A nice feature of our proposal is the reduce message cost: only one extra message is sent for every message sent in the original system. Note that our contribution is more conceptual than experimental and can be easily exploited by new emergent areas such as P2P Internet Games or Social Networks where a major trend is to achieve a secure and private communication without relying on any fixed infrastructure or centralized authority.

1 Introduction

Multi-player Internet Games or Social Networks provide a wide field of applications that has been very seldom explored by the distributed computing community. Recently, architecture designers in these systems used distributed middleware and P2P architectures [9,23,11] that do not fully exploit the advances in traditional distributed computing area. Surprisingly, their motivation is acknowledged to the need of privacy and security that is not always a design priority in distributed computing. The most meaningful examples are publish/subscribe systems. The publish/subscribe paradigm is an effective technique for building distributed applications in which information has to be disseminated from *publishers* (event producers) to *subscribers* (event consumers). Users express their interests in receiving certain types of events by submitting a filter on the event content, called a *subscription*. When a new event is generated and *published*, the publish/subscribe infrastructure is responsible for checking the event against all current subscriptions and delivering it to all users whose subscriptions match the event. Content-based publish/subscribe systems allow complex filters on the event content, enabling the use of constraints such as ranges, prefixes, and suffixes.

Research in Publish/Subscribe systems has been highly active in recent years in the distributed computing area. The most relevant results deal with the design of infrastructures and routing algorithms, having scalability as a major goal. Theoretical work has also appeared, aiming at formally specifying the behaviour of publish/subscribe systems. In such works a publish/subscribe system is seen as an abstract distributed system whose behaviour is described in terms of liveness and safety properties. However, actual deployment of existing publish/subscribe architectures is currently limited by their lack of security and privacy capabilities. Enhancing a publish/subscribe system with these capabilities allows an easier deployment and a more flexible adaptation in a larger spectrum of applications.

Interestingly, secure information dissemination and routing in P2P systems benefited recently from the full attention of the distributed computing community [29,21,13,7]. The proposed solutions can be easily included in existing publish/subscribe architectures in order to make them secure. However, aspects such as privacy and point-to-point and secure data exchange with zero knowledge still need further research.

Wang *et al.*[34] propose an overview of security issues in publish/subscribe systems. The paper states the open problems and further research directions related to publish/subscribe security and privacy. Miklòs[25] and Belokosztolszki *et al.*[8] propose strategies to control subscriptions in order to deploy payment system in broker based publish/subscribe systems. Miklòs trusts the entire event-dispatching network which is improved by Belokosztolszki with different trust level among brokers. Trust management is also discussed in Fiege *et al.*[19] in order to deploy different security policies.

Related to our topic are the security issues in group communication. The similarity comes from the organization of a publish/subscribe systems. In most of the architectures subscribers gather in groups following various criteria (e.g. similarity of their subscriptions). The "communication group" may be the set of subscribers interested in the same event. Since the number of groups may be exponential in the number of subscribers, Opyrchal *et* Prakash [26] propose a dynamic caching algorithm to deal with group keys and limits the number of encryptions (a naive solution is to make brokers encrypt each event with subscriber's private keys). Srivatsa *et* Liu. [31] improve the method by proposing to derive an encryption key into sub-keys which can decrypt a sub-set of messages the main key can. Then they have a tree of keys, each key representing a filter. Finally, Khurana [24] proposes solutions for confidentiality, integrity and authentication for publish/subscribe systems with untrusted brokers. Similarly, Srivatsa and Liu [30] propose a set of pluggable "guards" on untrusted brokers, each one guaranteeing a secure property. They suppose an untrusted event network. These works are seemingly close to our work. However, the main drawback of their proposal is that they need a fixed trusted infrastructure while we propose a completely decentralized solution, making no assumption on the network or the members of the system.

Another related topic is anonymous routing and multicast [18,32]. The above schemes can be applied in a publish/subscribe context but only in a static context since they use secure predefined paths. In our case, we assume the network is completely dynamic therefore we do not rely on any predefined secure path between the publishers and the subscribers. Moreover, our exchange scheme does not focus on the routing itself. Our main focus here is how to cluster peers without revealing their interests. That is, a peer joins a group of similar peers if and only if it has the same interests as the other peers in the group. Then the group will exchange events based on the peer-defined encryption keys.

Our contribution. Our contribution is twofold. We first propose a novel strategy for secure and private data exchange based only on point-to-point interactions. Our scheme is a conditional key exchange protocol: it provides a private shared key if and only if the two participants share the same interest (represented by a value). It does not need a third party and can cope with a broad class of attacks. In particular our scheme is resistant to attacks of the type "man-in-the-middle", i.e. the attacker can listen and alter all the messages sent and received and also forge new illegal messages. Second we show how to plug our exchange scheme in a generic architecture for publish/subscribe systems without any degradation of the original properties of the system.

Our contribution is more conceptual than experimental since we provide a formal framework that helps in extending existing publish/subscribe systems with privacy and security properties at a reduced cost. Note that our framework makes use of only one extra message per application message.

We advocate that our framework by its generality has a broader area of applications than the one considered in the current paper. Two of these possible applications are the design of secure and private distributed architectures for Multi-player Internet Games or Social Networks.

2 Publish/Subscribe Framework

2.1 Publish/Subscribe Model

We consider a finite yet unbounded set of nodes Π. The set is dynamic in the sense that nodes can join or leave at an arbitrary time. Each node is associated with a unique identifier. Within the set Π we distinguish two subsets: the set of publishers Π_p and the set of subscribers Π_s. These two sets can overlap and possibly span the whole Π. Each node in Π maintains a subscription list S_i. A node p_i in Π_p can issue a *publish* operation (an event generation), while a node s_i in Π_s can issue the *subscribe* and *notify* operations. The $subscribe_i(s)$ operation provokes the insertion of the subscription s into the subscription list of some nodes (not necessarily p_i). This operation requires in general some time to complete. A subscription is said to be stable when the corresponding $subscribe$ operation has completed. The *notify* operation sends to the upper layer the received event.

We consider a publish/subscribe data model [22,1] where both subscriptions and events use as building blocks a finite, yet unbounded universe of attributes. Each attribute is defined by a unique name, a type (e.g., integer, string, or enumerated), and a domain (the range of the attribute values). A *topic based* subscription is a predicate over the attributes in the system (possibly one single attribute). A *c*ontent based subscription or *content-based filter* is a conjunction of predicates over the attribute fields.

In [3] the authors state that any publish/subscribe system verifies the four properties below:

Legality. If some node s_i issues $notify_i(e)$, then s_i previously issued a $subscribe_i(f)$, such that e matches the subscription f.

Validity. If some node s_i issues $notify_i(e)$, then there exists some node p_j that previously issued $publish_j(e)$.

Fairness. Every node may publish infinitely often.

Event Liveness. If a node issues $publish_i(e)$, $notify_j(e)$ will be eventually issued by all the nodes s_j such that $subscribe_j(f)$ is stable at the moment of the notification such that e matches the subscription f.

The early publish/subscribe systems were implemented over a network of *brokers* that are responsible for routing information or events between publishers and subscribers [2,14,17]. However, the efficiency of this method has some limitations in dynamic systems where there is no guarantee that brokers will be permanently connected to the system. Therefore, recent publish/subscribe architectures adopt a P2P approach where any node can act as a broker. In these systems subscribers are arranged in a virtual overlay and actively participate to the information filtering. Solving the notification problem comes down to (i) efficiently arranging the virtual overlay of subscribers and propagating events within this overlay and (ii) efficiently matching an event against a large number of subscriptions.

An extensive analysis of existing publish/subscribe systems [12,35,22,16,15,4,33] brings the authors of [5] to the conclusion that the implementation of any publish/subscribe system can be abstracted to two oracles: one helps the efficient subscription while the second targets the efficient event dissemination. Intuitively, a node that wishes to subscribe with a filter, invokes the *subscription oracle* to find either its position on the subscribers overlay or the broker in charge of its subscription. The subscription oracle provides a *routing* method in charge of revealing the next node to contact in order to get closer to the goal position. A publisher invokes the *dissemination oracle* in order to get partial lists of nodes to whom the publisher should forward its event. This oracle provides a *forward* method. As advocated in [5], an efficient implementation of the forward method should be able to cover all the interested subscribers (that is, all the nodes interested in the event) while minimizing the number of the oracle invocations.

In [5] the authors propose an oracle based generic architecture that satisfies the publish/subscribe properties specified above (i.e. legality, validity, fairness and event liveness). One open question the current work responds is how to extend the generic architecture proposed in [5] to implement secure and private publish/subscribe systems. Obviously the oracles identified in the original generic architecture should be extended in order to deal with the attacks described below.

2.2 Attacks Model

Implementing private and secure Publish/Subscribe systems comes to enhance the subscription and dissemination oracle with additional power in order to tolerate the following attacks:

Brute-force attacks. The attacker will use all the computable power he has to break the security of the system.

Eavesdroppers. The attacker can listen all the messages sent and received.

Man-in-the-middle. The attacker can listen and alter all messages sent and received. He can also forge new illegal messages.

Malicious Insiders. The attacker is a member of the system, participating to the protocols but deviating from them in order to break the security.

Interestingly all these attacks are effective during point-to-point information exchange. Therefore, implementing secure and private publish/subscribe systems reduces to the secure and private point-to-point data exchange. In the following we propose a novel strategy for data exchange that tolerates the attacks specified above. Furthermore, we show how to plug the secure and private data exchange module into the generic publish/subscribe architecture of [5].

3 Secure and Private Data Exchange

In the distributed implementations of publish/subscribe systems nodes exchange information (i) to detect if they have common interest, then cluster in order to make efficient the events dissemination or (ii) for filtering purposes. Each data exchange can be therefore decomposed in two phases: the *test phase* and the *secret key exchange phase*. The *test phase* allows interested parties to verify if they share the same interest. During this phase none of the participants should learn the topic or the filter of the other parties. Moreover an eavesdropper or a third-party must not be able to learn the topics or filters exchanged and neither the result of the exchange. The *key exchange phase* starts only if the test phase is positive. That is, if the exchange participants have the same interests then they need a shared secret key in order to communicate securely.

3.1 Secure and Private Test and Key Exchange

In the following we consider two actors: A with a value a and B with a value b. A and B would like to test if a and b are equal without revealing their value. To solve this problem one may use the following algorithm: A and B hash their values plus a salt value and send it. Then they compute the hash of their value plus the other's salt value. If there is an equality with the received hash then the values are equal. The main drawback of this simple protocol is that one can try to revert the hash function by brute-force (by hashing every possible value): after the exchange nothing prevents B (respectively A) to try other values than b (resp a) to guess a (resp b). This brute-force attack is feasible when the set of possible values is relatively small as in the current topic based publish/subscribe systems.

We propose a novel algorithm **SPTest** resilient to this attack, shown as Algorithm 1. It is particularly efficient as the both phase (test and key exchange) are done simultaneously. It uses a commutative cryptographic function and a secure hash function. A commutative cryptographic function f is a strongly one-way function [20] (easy to compute but hard to invert) such as for any private keys k_a, k_b and any input x we have: $f_{k_a}(f_{k_b}(x)) = f_{k_b}(f_{k_a}(x))$.

As a commutative cryptographic function, we suggest to use the difficulty of discrete logarithms computation on a simple group like $\mathbb{Z}_p$ where p is a large prime number (note that any discrete group could work) as no efficient (polynomial) algorithm is known to solve this problem [28]. To make this function injective we also need that the key k is smaller than p. Also the exchanged message must be coded by a positive integer a between 2 and $p - 1$. Finally based on the Diffie-Hellman scheme [28], our function f is $f_k(a) = a^k \mod p$ where k is the private key and x the value to encrypt. This choice is not innocent, since if $a = b$ then the shared secret key will be $a^{k_a * k_b}$

mod p which is also the shared key in the Diffie-Hellman key exchange protocol [28]. However, contrary to the Diffie-Hellman scheme, in our scheme a and b are private hence it is protected against man-in-the-middle attacks (see lemma 6). Finally, as a hash function, we use the unbroken SHA-2 family hash function [27].

Note 1. The correctness of **SPTest** is not limited by a particular function: any commutative cryptographic function or secure hash function could be used instead of the one we suggest.

Algorithm 1. Secure private equality test and key exchange algorithm (SPTest)

```
processes $A$, $B$
private values
  $a$ : value of process $A$,    $k_a$ : key of process $A$
  $b$ : value of process $B$,    $k_b$ : key of process $B$
public function
  $f_k(x)$ : commutative cryptographic function using private key $k$ on value $x$
  $h$ : cryptographic hash function
action
  1. $A$ computes $ak = f_{k_a}(a)$ and send it to $B$
     $B$ computes $bk = f_{k_b}(b)$ and send it to $A$
  2. $A$ computes $ha = h(ak + f_{k_a}(bk))$ and send it to $B$
     $B$ computes $hb = h(bk + f_{k_b}(ak))$ and send it to $A$
  3. $A$ computes $h(bk + f_{k_a}(bk))$ and compare it to $hb$ equals iff $a = b$
     $B$ computes $h(ak + f_{k_b}(ak))$ and compare it to $ha$ equals iff $a = b$
  4. If $a = b$ then $f_{k_a}(bk) = f_{k_b}(ak)$ is a shared secret key between $A$ and $B$
```

3.2 Correctness and Complexity of SPTest

In the following we prove that the algorithm **SPTest** tests if the input values are equal. Moreover we show that the algorithm allows the participants to share a secret key if their input values are equal. Additionally we prove that the algorithm tolerates the attacks described in Section 2.2.

Lemma 1 (Equality test). *Algorithm 1 tests if $a = b$ and when the algorithm stops both A and B will know the result of the test.*

Proof. The proof follows from the commutativity of the cryptographic function f. That is, for any x and any keys k_a, k_b, $f_{k_a}(f_{k_b}(x)) = f_{k_a}(f_{k_b}(x))$. Therefore, if $a = b$ then $h(bk+f_{k_a}(bk)) = hb = h(bk+f_{k_b}(ak))$ computed at A returns true and symmetrically $h(ak + f_{k_b}(ak)) = ha = h(ak + f_{k_a}(bk))$ computed at B returns true as well. It follows that A and B can decide that they share the same private value. Additionally, A and B can use the key $f_{k_a}(f_{k_b}(b)) = f_{k_b}(f_{k_a}(a))$ as shared secret key for further communication.

Note that during the exchange $f_{k_a}(f_{k_b}(b))$ or $f_{k_a}(f_{k_b}(a))$ should not be disclosed since eavesdroppers must not learn the test result. Therefore A and B compute the hash of those values added to a deterministic salt they both know which is $f_{k_a}(a)$ or $f_{k_b}(b)$.

Even if the salt is used to have $ha \neq hb$ when $a = b$, any known salt can be used (please refer to [28] for further details).

Note also that a hash function is not injective so there is a probability that $h(bk + f_{k_a}(bk)) = hb$ even if $a \neq b$ because of a collision in the hash function. However with a 256-bits hash function, this is negligible. Moreover if $a \neq b$ then A and B will not have a valid secret key and will not be able to further communicate.

Lemma 2 (Key exchange). *$SPTest$ shown as Algorithm 1 allows A and B to exchange a secret key if $a = b$.*

Proof. If $a = b$ then A and B both know the value $f_{k_a}(f_{k_b}(x)) = f_{k_b}(f_{k_a}(x))$ which is secret and may be used as a private secure key.

In the following we prove that our algorithm tolerates the class of attacks specified in Section 2.2.

Lemma 3 (Brute-force attack tolerance). *The messages encrypted by f_k cannot be decrypted without the key k and the hash function is not invertible.*

Proof. By hypothesis, f is such a function. And in particular this is the case for the proposed function $f_k(a) = a^k \mod p$: there is no known algorithm that finds efficiently k given only a, p, and $a^k \mod p$. Then by definition, a secure hash function is not invertible and the proposed SHA-2 function has not been broken. As point by note 1, if one of the proposed function were no longer secure, it could be replaced without modifying our scheme.

Lemma 4 (Eavesdroppers attack tolerance). *Eavesdroppers cannot learn a, b nor if $a = b$.*

Proof. First an eavesdropper cannot learn a nor b because he cannot compute them from ak, bk, ha or hb. Since ak and bk are different even if $a = b$ then ha and hb are always different and the eavesdropper does not know if $a = b$. So he cannot know the result of the test.

Lemma 5 (Insiders attack tolerance). *The probability a malicious insider learns the other's value is less than $1/n$ when the size of a group is n.*

Proof. The algorithm is symmetric so consider A as the attacker.

If A follows the protocol and behaves as an eavesdropper, A learns nothing related to b if $a \neq b$.

Assume A lies on a. Thus A may be able to make a guess on b. If A is right then it will know b. But at each exchange he can make only one guess. That is, A need B to participate for each guess thus making impossible a brute-force attack: B may have a limit on the number of **SPTest** it performs and may refuse to answer (or it can lie) to A if A asks for too much **SPTest**. Therefore, in one single exchange A has $1/n$ chance to learn b. Notice that no known algorithm can prevent this.

Assume A lies on ha in order to corrupt the result of the test. To make a false negative, A just has to send a different ha. B will compute $h(ak + f_{k_b}(ak))$ and will find it is different from ha so B decides $a \neq b$. Notice that A can still know the good result but in any case does not know b (if $a \neq b$).

Furthermore, since A does not know the private key of B, k_b, A cannot compute the result of $h(ak + f_{k_b}(ak))$. Therefore, A cannot send a false ha which may make B believe that $a = b$. So no false positive can be forged. Even if B thinks that is shares b with A, b is not revealed and A will not be able to read B messages (because they do not have the same shared key).

Lemma 6 (Man-in-the-middle attack tolerance). *In the case of n possible values (topics) the probability a man-in-the-middle learns a or b is less than $1/n$.*

Proof. A man-in-the-middle attacker can intercept every message of A or B and can alter, drop them or even create new messages. If he does not modify any messages then it acts as an eavesdropper and following Lemma 4: it learns nothing.

If the attacker modifies some messages, it can act like A or B or both in a test between A and B with just a difference, it does not know neither a nor b. Thus the attacks it can perform to discover a or b are the same as the one performed by a malicious member of the system (see Lemma 5). However, in order to leer the result of the test $a = b$ the attacker has to successfully guess a and b which is impossible (since it is unable to compute the hash function).

Lemma 7 (Security of the shared secret key). *The exchanged secret key is safe and allows A and B to exchange confidential messages.*

Proof. The shared key is $f_{k_a}(bk) = f_{k_b}(ak)$. Both ak and bk are transmitted unencrypted and have to be considered public. At contrary k_a and k_b are private keys so preventing other actors than A or B to compute $f_{k_a}(bk)$ or $f_{k_b}(ak)$.

With our suggested function f, when $a = b$ the shared key is $a^{k_a * k_b} \mod p$ and the information transiting the network are $a^{k_a} \mod p$ and $a^{k_b} \mod p$ thus making our key exchange part of the Diffie-Hellman algorithm [28]. However, Diffie-Hellman algorithm is vulnerable to a Man-in-the-middle attack because the generator (represented by a in our algorithm) is public. In our case, a is private so this attack is no longer working. Overall, A and B can exchange confidential messages when $a = b$.

Remark 1. Note that since the hash function is not reversible then it is not possible to gain information on the key from the step 3 of Algorithm 1.

In the following we evaluate the complexity of our algorithm both in terms of additional message exchanged and the computational power needed in order to implement the scheme.

Lemma 8 (Complexity). *The complexity in number of messages of Algorithm 1 is 2 per participant. Each participant computes 2 hashes and 2 f functions.*

Proof. The protocol is symmetric between A and B. At step 1, A computes one f function and sends one message. At step 2, it computes one f, one hash and sends one message. At step 3, it computes just one hash. Step 4 adds no transmission or functional computation.

4 Secure and Private Publish/Subscribe

In the following we extend the Publish/Subscribe specification proposed in Section 2.1 with security and privacy properties. Therefore, in systems prone to attacks, we expect published/subscribe communication primitive to verify the following additional properties. Obviously, these properties append to the legality, validity, fairness and event liveness properties defined in Section 2.1:

Nodes privacy. No one should be able to learn the filters or the topics of a node if the node does not reveal it explicitly or if they do not have it in common.
Messages privacy. No one should be able to read and modify messages in a group if he is not a member of this group.
Messages security. No one should be able to insert fake or corrupted messages in a group if he is not a member of this group.
Group privacy. No one should be able to completely map a group, thus learn who belongs to this group and how its members are connected.
Group security. No one could join a group if he does not share the same topic or filter as the rest of the group.

These properties heavily rely on the notion of *group*: the set of peers sharing the same filter in the case of content based systems or topic in topic based systems. Note that the *group privacy property* holds only for outsiders. In the next section we will discuss some issues related to how to extend this property to the insiders. In the following we exclusively address the topic based systems.

4.1 Private Subscription/Dissemination Oracles

In [5] the authors, after extensively studying the most relevant P2P publish/subscribe systems, propose a generic oracle-based architecture that basically reduces the publish/subscribe system to the implementation of two abstractions: the *subscription oracle* and the *dissemination oracle*. The subscription oracle is in charge of efficiently arranging subscriptions in a P2P overlay while the dissemination oracle is in charge of efficiently match events against a large number of subscriptions. The subscription oracle, when invoked by a subscriber, returns a contact point in the virtual overlay. The dissemination oracle, when invoked by a publisher, returns a subset of nodes interested in the event. The generic implementation of the architecture proposed in [5] was targeted to efficiency in terms of reduced message complexity. However, aspects like security and privacy were left open. In the sequel we extend the work by enhancing the oracle-based generic architecture for publish/subscribe systems introduced in [5] with privacy and security features. We also show that adding the new properties, the original properties suffer no degradation and the message complexity increase with a constant factor.

Private Subscription Oracle. In the original architecture the subscription oracle provides the `Route` method. A subscriber invokes the Route method indicating its identifier and its subscription. The method eventually returns a connection point in the overlay defined by the subscribers. The connection point has to be either a broker or a sibling for the invoking subscriber. The original implementation of this oracle makes its invocation recursive. That is, at each invocation the subscriber gets closer to its broker or sibling. This type of invocation can be observed in systems such as Meghdoot [22] or Sub-to-Sub [33]. In tree-based architectures such as [12,10] a subscriber travels downward the tree overlay until it finds its group or the most appropriate parent. In such architectures a subscription oracle will provide the subscriber with the next node in its walk.

Note that for privacy reasons a subscriber invoking a private and secure subscription oracle should not reveal its interests. Therefore in our extended architecture the only information a subscriber provides to the oracle is its own node identifier, *pid*. The only information returned by the oracle is only a node identifier. This node can be the parent of the subscriber in a tree-overlay, the identity of a broker in charge of the subscription or a node on a DHT in charge of storing the subscription. Note that without additional information on the filter of the subscriber the oracle has low accuracy which may increase the latency of the system.

Our oracle provides the `Private Route` method and uses the transitive invocation of the Private Route defined below.

Definition 1 (Light Transitive Invocation). *A sequence of invocations of the* `Private Route` *method in which all the invocations in the* j^{th} *call are done with the same node identifier returned by the* $(j-1)^{th}$ *invocation is called* a transitive invocation of `Private Route`.

Whenever the Private Route method is transitively invoked it should converge. Two cases may appear: (1) the subscriber finds a sibling or the broker in charge of its filter and stops the invocation of the oracle. (2) the subscriber has no sibling or broker in the system and stops the invocation. Formally, the convergence of the transitive invocation is defined as follows:

Definition 2 (Convergence of the transitive invocation). *The Light transitive invocation of the* `Private Route` *converges iff either (i) there exists an integer* l *such that the Light transitive invocation of* `Private Route` *consists of exactly* l *invocations or (ii)there exists an integer* l *such that the Light transitive invocation of* `Private Route` *consists of at least* l *invocations and returns the set of nodes* Π.

Note that for privacy and security reasons the subscriber may continue to invoke the subscription oracle even if it finds its broker or sibling. In this way, an observer (internal or external of the system) cannot deduce the interests of the subscriber by analyzing the length of its walk in the overlay. Therefore, in safe systems l may be set to infinite.

An effective implementation of a *private subscription oracle* should satisfy the following two properties:

Convergence: For every initial node identifier, every Light transitive invocation of the `Private Route` method converges.

Subscription Privacy: For every initial node identifier, pid, in every transitive invocation of `Private Route` method, the pid is the only revealed information.

A naive implementation of the subscription oracle can be done via a random walk. That is, any time a subscriber invokes the oracle it gets a random node in the overlay. Then the subscriber tests via the algorithm **SPTest** if the returned node is its sibling. When the result of the test is positive the subscriber adds the returned node in its neighbours table. Otherwise it invokes the oracle in order to get another node. This new node can be a random neighbour of the node returned in the previous invocation. Hence, the recursive invocation can run forever or can be pruned when the node fills its neighbours table.

A DHT-based implementation of this oracle can take advantage of the already existing routing methods in DHTs. When the oracle is invoked for the first time it provides the caller with a random node on the DHT. Then the caller verifies if the given node is its sibling via the **SPTest** algorithm. Then it stops the invocation if it finds a sibling or recursively invokes the oracle until the overlay is completed visited (in Chord-based architectures a round trip around the ring terminates the oracle invocation while in CAN-based architectures for example a walk following a virtual Eulerian circuit would be sufficient).

In DHT-free architectures a DFS or BFS traversal of the overlay can also be simple implementations for our subscription oracle. Note that several recent works address these traversals in dynamic settings. Please refer to [6] for further details.

Private Dissemination Oracle. In the original architecture the dissemination oracle is in charge of efficiency matching an event against a large number of subscriptions. This oracle supports the *F*orward method that takes as input a node identifier(initially the publisher identifier) and the event to be dispatched. The Forward returns a partial list of nodes to which the event should be propagated. Obviously, in order to disseminate the event to all the subscribers in the system the Forward method has to be recursively invoked.

Note that in a private system the publisher cannot disclose its publication topics therefore our *private dissemination oracle* supports the *Private Forward* method that takes as input only a node identifier:

PID-LIST `Private Forward`(**in** PID *pid*)

The `Private Forward` method returns a list of nodes to which the current generated event should be propagated. Note that for privacy and security reasons the returned list of nodes should also include nodes that are not interested in the event in order to give no hint to the caller related to the real interests of the returned nodes.

In order to be effective, an implementation of the private and secure dissemination oracle is expected to provide the eventually full coverage property. That is, the union of lists returned by the `Private Forward` method during any pruned recursive invocation chain for an event e is a super-set of the nodes that during this invocation chain have a stable subscription with a filter f such that e matches f and the privacy property below:

Definition 3 (Dissemination Privacy). *For any published event* e*, the only disclosed information (revealed or learnt information) during the recursive invocation of* `Private Forward` *method are* pid *and* e*.* e *will be disclosed only to the nodes that have subscribed with filters that match the event.*

As for the case of the private and secure subscription oracle, the private and secure dissemination can be easily implemented in a broad class of systems using similar policies as described for the case of private and secure subscriptions. Gossip seems to be one the most appealing since popular publish/subscribe systems use this technique. The BFS traversals of dynamic overlays is also an alternative to gossip-based inconveniences such as explosion in terms of message complexity. Alternatively, DHTs via their predefined structure are good candidates for efficient implementations of a dissemination oracle. That is, a simple diffusion in a ring, tree of grid solve the problem.

4.2 Secure and Private Publish/Subscribe Implementation

A simple and generic implementation of a notification service based on the private and secure subscription and dissemination oracles described in the previous section can be as follows: The interface to the notification system at each node p_i remembers all the filters to which p_i is subscribed (that is, the application at p_i issued a subscribe with no following unsubscribe). When a node p_i invokes the `Subscribe`(p_i) method, the publish/subscribe service interface contacts the subscription oracle to obtain a connection point, p_j. Then p_i and p_j start a private exchange in order to test if they have or not the same interest. To this end, algorithm **SPTest** is used for the topic based systems. Note that this algorithm also defines a private secret key which can be further used for events dissemination. p_i continues to invoke the oracle until it finds a similar node (a node with the same topic). Note that even after finding a similar node, p_i may decide to invoke the oracle in order to not reveal to the oracle any information related to its interests. In our generic implementation the infinite loop of the subscription phase captures the later case.

When a node p_i invokes the `Publish`(e) method, the service's interface first contacts the dissemination oracle to obtain a list of nodes to which the event should be forwarded; p_i then forwards the encrypted event to all these nodes, by a point-to-point strategy. Every node p_j that receives such an encrypted event e for the first time uses its private key in order to decrypt the message and then checks all the local subscription filters. If e matches any of them, then e is notified locally. Also, p_j invokes (recursively) the `Private Forward` method to obtain the next list of nodes, and forwards e to them.

5 Security Trade-Off Analysis

In the previous section we presented a framework for a secure publish/subscribe system but we did not discuss the cost of the privacy. This cost is not fixed and depends on the strength of the privacy. More the system enforces privacy more the search of compatible nodes is inefficient. Suppose we normalize topics to a fixed length (like 128 bits), for instance by a secure hash function.

First, if nodes do not reveal a single bit of their topic at the oracle, the oracle cannot distinguish them and thus cannot provide an efficient search. The result is that in the worst case, a node has to test every node of the system to find the one it is looking for. 0 bit revealed equals a worst case complexity of n searches.

Secondly, nodes may reveal some bits of their topic. Each bit they reveal can be used by the oracle to select likely-better nodes for a topic match. For each bit revealed, the size of the set of potential match is in average divided by two. But in the same way, the privacy of the node is also divided by two, that is it is known it may belong to only half of the set of topics.

So it is possible to control the trade-off between privacy and performance by setting the number, and which bits, nodes have to reveal. These values are not necessarily global as each node may choose them. For instance, a thousand of nodes may share k revealed bits but are using a lot of different topics. This prevents each one to have their topic be easily guess. At contrary, with the same combination of revealed bits, there may be a small group of nodes sharing the same topic. Knowing theses bits leads to know their topic. So this small group may choose to reveal an other combination of bits to hide in a larger portion of the system.

6 Conclusions and Discussions

We addressed secure and private data exchange in P2P networks. We propose a novel data exchange scheme that outperforms the Diffie and Hellman scheme with respect to its resilience to man in the middle attacks. Our scheme is appealing in systems where the main priority is to ensure privacy and security in point-to-point interactions without relying on any fixed infrastructure or central authority. A second contribution of this paper is the proposal of a generic framework to extend architectures for publish/subscribe systems with secure and private properties. Interestingly the cost of our extension is only one extra message for any original application message.

Our generic framework opens several research directions. A first important direction is how to protect a group of subscribers from intruders. That is, the door of a group of similar peers rely on their topic. If this topic is easy to guess, then attackers may successfully join the group. In this case, they will be able to read every message that is transmitted inside the group. Interestingly, with our scheme anonymity is still possible and the attackers would not know who is the source of an event. A possible solution to bypass this problem is to have different trust degrees inside a group and to forward only secret messages to trusted nodes. Another problem that remains open is to bypass the coalition of attackers which may prevent someone to connect to a group by altering its equality test making it falsely negative. A third direction would be to extend the study to content based systems.

References

1. Aguilera, M., Strom, R., Sturman, D., Astley, M., Chandra, T.: Matching events in a content-based subscription system. In: Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC 1999), pp. 53–61 (1999)
2. Altinel, M., Franklin, M.: Efficient filtering of XML documents for selective dissemination of information. In: Proceedings of the 26th International Conference on Very Large Databases (VLDB 2000), pp. 53–64 (2000)
3. Anceaume, E., Datta, A.K., Gradinariu, M., Simon, G.: Publish/Subscribe Scheme for Mobile Networks. In: Proc. of the Workshop on Principles on Mobile Computing, POMC 2002 (2002)
4. Anceaume, E., Datta, A., Gradinariu, M., Simon, G., Virgillito, A.: A semantic overlay for self*- peer-to-peer publish subscribe. In: Proceedings of the 26th International Conference on Distributed Computing Systems, ICDCS 2006 (2006)

5. Anceaume, E., Friedman, R., Gradinariu, M., Roy, M.: An architecture for dynamic scalable self-managed persistent objects. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3291, pp. 1445–1462. Springer, Heidelberg (2004)
6. Baldoni, R., Bertier, M., Raynal, M., Piergiovanni, S.T.: Looking for a definition of dynamic distributed systems. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 1–14. Springer, Heidelberg (2007)
7. Baldoni, R., Doria, L., Lodi, G., Querzoni, L.: Managing reputation in contract-based distributed systems. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009. LNCS, vol. 5870, pp. 760–772. Springer, Heidelberg (2009)
8. Belokosztolszki, A., Eyers, D.M., Pietzuch, P.R., Bacon, J., Moody, K.: Role-based access control for publish/subscribe middleware architectures. In: DEBS 2003: Proceedings of the 2nd international workshop on Distributed event-based systems, pp. 1–8. ACM, New York (2003)
9. Bharambe, A., Pang, J., Seshan, S.: Colyseus: a distributed architecture for online multiplayer games. In: NSDI 2006: Proceedings of the 3rd conference on Networked Systems Design & Implementation, pp. 12–12 (2006)
10. Bianchi, S., Felber, P., Potop-Butucaru, M.G.: Stabilizing distributed r-trees for peer-to-peer content routing. IEEE Transactions on Parallel and Distributed Systems 99
11. Botev, J., Hohfeld, A., Schloss, H., Scholtes, I., Sturm, P., Esch, M.: The hyperverse - concepts for a federated and torrent-based "3d web". Int. J. Adv. Media Commun. 2(4) (2008)
12. Castro, M., Druschel, P., Kermarrec, A.M., Rowston, A.: Scribe: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communications 20(8) (October 2002)
13. Champel, M.L., Kermarrec, A.M., Scouarnec, N.L.: Fog: Fighting the achilles' heel of gossip protocols with fountain codes. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 180–194. Springer, Heidelberg (2009)
14. Chan, C.Y., Felber, P., Garofalakis, M., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. VLDB Journal, Special Issue on XML 1(4), 354–379 (2002)
15. Chand, R., Felber, P.: Semantic peer-to-peer overlays for publish/subscribe networks. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 1194–1204. Springer, Heidelberg (2005)
16. Costa, P., Migliavacca, M., Picco, G., Cugola, G.: Epidemic algorithms for reliable content-based publish/subscribe: An evaluation. In: Proc. of the 24th International Conference on Distributed Computing Systems, ICDCS 2004 (2004)
17. Diao, Y., Fischer, P., Franklin, M., To, R.: YFilter: Efficient and scalable filtering of XML documents. In: Proceedings of the 18th International Conference on Data Engineering, ICDE 2002 (2002)
18. Dolev, S., Ostrobsky, R.: Xor-trees for efficient anonymous multicast and reception. ACM Trans. Inf. Syst. Secur. 3(2), 63–84 (2000)
19. Fiege, L., Zeidler, A., Buchmann, A., Darmstadt, T.: Security aspects in publish/subscribe systems. In: Third Intl. Workshop on Distributed Event-based Systems (DEBS 2004). IEEE, Los Alamitos (2004)
20. Goldreich, O.: Foundations of cryptography. Basic Tools, vol. 1. Cambridge University Press, Cambridge (2007)
21. Guerraoui, R., Huguenin, K., Kermarrec, A.M., Monod, M.: Brief announcement: Towards secured distributed polling in social networks. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 241–242. Springer, Heidelberg (2009)
22. Gupta, A., Sahin, O., Agrawal, D., Abbadi, A.E.: Meghdoot: Content-based publish:subscribe over p2p networks. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 254–273. Springer, Heidelberg (2004)

23. Keller, J., Simon, G.: Solipsis: A massively multi-participant virtual world. In: PDPTA, pp. 262–268 (2003)
24. Khurana, H.: Scalable security and accounting services for content-based publish/subscribe systems. In: SAC 2005: Proceedings of the 2005 ACM symposium on Applied computing, pp. 801–807. ACM, New York (2005)
25. Miklos, Z.: Towards an access control mechanism for wide-area publish/subscribe systems. In: Proceedings of 22nd International Conference on Distributed Computing Systems Workshops, pp. 516–521 (2002)
26. Opyrchal, L., Prakash, A.: Secure distribution of events in content-based publish subscribe systems. In: SSYM 2001: Proceedings of the 10th conference on USENIX Security Symposium, pp. 21–21. USENIX Association, Berkeley (2001)
27. Sanadhya, S.K., Sarkar, P.: New collision attacks against up to 24-step sha-2. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 91–103. Springer, Heidelberg (2008)
28. Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd edn. Wiley, Chichester (2007)
29. Serbu, S., Riviere, E., Felber, P.: Network-friendly gossiping. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 655–669. Springer, Heidelberg (2009)
30. Srivatsa, M., Liu, L.: Securing publish-subscribe overlay services with eventguard. In: CCS 2005: Proceedings of the 12th ACM conference on Computer and communications security, pp. 289–298. ACM, New York (2005)
31. Srivatsa, M., Liu, L.: Secure event dissemination in publish-subscribe networks. In: ICDCS 2007: Proceedings of the 27th International Conference on Distributed Computing Systems, p. 22. IEEE Computer Society, Washington (2007)
32. Syverson, P., Reed, M., Goldschlag, D.: Onion Routing access configurations. In: Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 2000), vol. 1, pp. 34–40 (2000)
33. Voulgaris, S., Rivire, E., Kermarrec, A., van Steen, M.: Sub-2-Sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In: Proceedings of the 5th International Workshop on Peer-to-Peer Systems, IPTPS 2006 (2006)
34. Wang, C., Carzaniga, A., Evans, D., Wolf, A.: Security issues and requirements for internet-scale publish-subscribe systems. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS 2002, pp. 3940–3947 (January 2002)
35. Zhuang, S.Q., Zhao, B.Y., Joseph, A.D., Katz, R., Kubiatowicz, J.: Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In: Proc. of the Int. Workshop on Network and OS Support for Digital Audio and Video (2001)

Snap-Stabilizing Linear Message Forwarding*

Alain Cournier[1], Swan Dubois[2], Anissa Lamani[1],
Franck Petit[2], and Vincent Villain[1]

[1] MIS, Université of Picardie Jules Verne, France
[2] LiP6/CNRS/INRIA-REGAL, Université Pierre et Marie Curie - Paris 6, France

Abstract. In this paper, we present the first snap-stabilizing message forwarding protocol that uses a number of buffers per node being independent of any global parameter, that is 4 buffers per link. The protocol works on a linear chain of nodes, that is possibly an overlay on a large-scale and dynamic system, *e.g.,* Peer-to-Peer systems, Grids, etc. Provided that the topology remains a linear chain and that nodes join and leave "neatly", the protocol tolerates topology changes. We expect that this protocol will be the base to get similar results on more general topologies.

Keywords: Dynamic Networks, Message Forwarding, Peer-to-Peer, Scalability, Snap-stabilization.

1 Introduction

These last few years have seen the development of large-scale distributed systems. Peer-to-peer (P2P) architectures belong to this category. They usually offer computational services or storage facilities. Two of the most challenging issues in the development of such large-scale distributed systems are to come up with scalability and dynamic of the network. *Scalability* is achieved by designing protocols with performances growing sub-linearly with the number of nodes (or, processors, participants). *Dynamic Network* refers to distributed systems in which topological changes can occur, *i.e.,* nodes may join or leave the system.

Self-stabilization [1] is a general technique to design distributed systems that can tolerate arbitrary transient faults. Self-stabilization is also well-known to be suitable for dynamic systems. This is particularly relevant whenever the distributed (self-stabilizing) protocol does not require any global parameters, like the number of nodes (n) or the diameter (D) of the network. With such a self-stabilizing protocol, it is not required to change global parameters in the program (n, D, etc) when nodes join or leave the system. Note that this property is also very desirable to achieve scalability.

The *end-to-end communication* problem consists in delivery in finite time across the network of a sequence of data items generated at a node called the sender, to a designated node called the receiver. This problem is generally split

* This work is supported by ANR SPADES grant.

into the two following problems: (*i*) the *routing* problem, *i.e.,* the determination of the path followed by the messages to reach their destinations; (*ii*) the *message forwarding* problem that consists in the management of network resources in order to forward messages. The former problem is strongly related to the problem of spanning tree construction. Numerous self-stabilizing solutions exist for this problem, *e.g.,* [2–4].

In this paper, we concentrate on the latter problem, *i.e.,* the message forwarding problem. More precisely, it consists in the design of a protocol managing the mechanism allowing the message to move from a node to another on the path from the sender A to the receiver B. To enable such a mechanism, each node on the path from A to B has a reserved memory space called buffers. With a finite number of buffers, the message forwarding problem consists in avoiding deadlocks and livelocks (even assuming correct routing tables). Self-stabilizing solutions for the message forwarding problem are proposed in [5, 6]. Our goal is to provide a snap-stabilizing solution for this problem. A *snap-stabilizing protocol* [7] guarantees that, starting from any configuration, it always behaves according to its specification, *i.e.,* it is a self-stabilizing algorithm which is optimal in terms of stabilization time since it stabilizes in 0 steps. Considering the message-forwarding problem, combined with a self-stabilizing routing protocol, snap-stabilization brings the desirable property that every message sent by the sender is delivered in finite time to the receiver. By contrast, any self-stabilizing (but not snap-stabilizing) solution for this problem ensures the same property, "eventually".

The problem of minimizing the number of required buffers on each node is a crucial issue for both dynamic and scalability. The first snap-stabilizing solution for this problem can be found in [8]. Using n buffers per node, this solution is not suitable for large-scale system. The number of buffers is reduced to D in [9], which improves the scalability aspect. However, it works by reserving the entire sequence of buffers leading from the sender to the receiver. Furthermore, to tolerate the network dynamic, each time a topology change occurs in the system, both of them would have to rebuild required data structures, maybe on the cost of loosing the snap-stabilization property.

In this paper, we present a snap-stabilizing message forwarding protocol that uses a number of buffers per node being independent of any global parameter, that is 4 buffers per link. The protocol works on a linear chain of nodes, that is possibly an overlay on a large-scale and dynamic system *e.g.,* Peer-to-Peer systems, Grids, etc. Provided that (*i*) the topology remains a linear chain and (*ii*) that nodes join and leave "neatly", the protocol tolerates topology changes. By "*neatly*", we mean that when a node leaves the system, it makes sure that the messages it has to send are transmitted, *i.e.,* all its buffers are free. We expect that this protocol will be the base to get similar results on more general topologies.

The paper is structured as follow: In Section 2, we define our model and some useful terms that are used afterwards. In Section 3, we first give an informal overview of our algorithm, followed by its formal description. In Section 4, we prove the correctness of our algorithm. Network dynamic is discussed in Section 5. We conclude the paper in Section 6.

2 Model and Definitions

Network. We consider a network as an undirected connected graph $G = (V, E)$ where V is the set of nodes (processors) and E is the set of bidirectional communication links. A link (p, q) exists if and only if the two processors p and q are neighbours. Note that, every processor is able to distinguish all its links. To simplify the presentation we refer to the link (p, q) by the label q in the code of p. In our case we consider that the network is a chain of n processors.

Computational model. We consider in our work the classical local shared memory model introduced by Dijkstra [10] known as the state model. In this model communications between neighbours are modelled by direct reading of variables instead of exchange of messages. The program of every processor consists in a set of shared variables (henceforth referred to as variable) and a finite number of actions. Each processor can write in its own variables and read its own variables and those of its neighbours. Each action is constituted as follow:

$$< \textit{Label} >::< \textit{Guard} > \rightarrow < \textit{Statement} >$$

The guard of an action is a boolean expression involving the variables of p and its neighbours. The statement is an action which updates one or more variables of p. Note that an action can be executed only if its guard is true. Each execution is decomposed into steps.

The state of a processor is defined by the value of its variables. The state of a system is the product of the states of all processors. The local state refers to the state of a processor and the global state to the state of the system.

We denote by C the set of all configurations of the system. Let $y \in C$ and A an action of p ($p \in V$). A is *enabled* for p in y if and only if the guard of A is satisfied by p in y. Processor p is enabled in y if and only if at least one action is enabled at p in y. Let P be a distributed protocol which is a collection of binary transition relations denoted by $\rightarrow$, on C. An execution of a protocol P is a maximal sequence of configurations $e = y_0 y_1 ... y_i y_{i+1} \ldots$ such that, $\forall$ $i \geq 0$, $y_i \rightarrow y_{i+1}$ (called a step) if y_{i+1} exists, else y_i is a terminal configuration. *Maximality* means that the sequence is either finite (and no action of P is enabled in the terminal configuration) or infinite. All executions considered here are assumed to be maximal. ξ is the set of all executions of P. Each step consists on two sequential phases atomically executed: (i) Every processor evaluates its guard; (ii) One or more enabled processors execute at least one of their actions that are enabled in each algorithm. When the two phases are done, the next step begins. This execution model is known as the *distributed daemon* [11]. We assume that the daemon is *weakly fair*, meaning that if a processor p is continuously *enabled*, then p will be eventually chosen by the daemon to execute an action.

In this paper, we use a composition of protocols. We assume that the above statement (*ii*) is applicable to every protocol. In other words, each time an enabled processor p is selected by the daemon, p executes the enabled actions of every protocol.

Snap-Stabilization. Let Γ be a task, and S_Γ a specification of Γ . A protocol P is snap-stabilizing for S_Γ if and only if $\forall \Gamma \in \xi$, Γ satisfies S_Γ.

Message Forwarding Problem. Messages transit in the network in the Store and Forward model *i.e.,* they are stored temporally in each processor before being transmitted. Once the message is transmitted it can be deleted from the previous processor. Note that in order to store messages, each processor use a space memory called buffer. We assume in our case that each buffer can store a whole message and each message needs only one buffer to be stored.

It is clear that each processor uses a finite number of buffers for the message forwarding. Thus the aim is to bound these resources avoiding deadlocks (a configuration is which in every execution some messages can not be transmitted) and starvation (a configuration from which, in every execution, some processors are no longer able to generate messages). Thus some control mechanisms must be introduced in order to avoid these kind of situations.

The message forwarding problem is formally specified as follows:

Specification 1 (SP). *A protocol P satisfies SP if and only if the following two requirements are satisfied in every execution of P:*

1. *every message can be generated in finite time;*
2. *every valid message (generated by a processor) is delivered to its destination once and only once in finite time.*

Buffer Graph. In order to conceive our snap-stabilizing algorithm we will use a structure called Buffer Graph introduced in [12]. A Buffer Graph is defined as a directed graph where nodes are a subset of the buffers of the network and links are arcs connecting some pairs of buffers, indicating permitted message flow from one buffer to another one. Arcs are permitted only between buffers in the same node, or between buffers in distinct nodes which are connected by communication link.

Let us define our buffer graph (refer to Figure 1):
Each processor p has four buffers, two for each link (p, q) such as $q \in N_p$ (except for the processors that are at the extremity of the chain that have only two buffers, since they have only one link). Each processor has two input buffers denoted by $IN_p(q)$, $IN_p(q')$ and two output buffers denoted by $OUT_p(q)$, $OUT_p(q')$ such as $q, q' \in N_p$ (N_p is the set of identities of the neighbours of p) and $q \neq q'$ (one for each neighbour). The generation of a message is always done in the output buffer of the link (p, q) so that, according to the routing tables, q is the next processor for the message in order to reach the destination. Let us refer to $nb(m, b)$ as the next buffer of Message m stored in b, $b \in \{IN_p(q) \vee OUT_p(q)\}$, $q \in N_p$. We have the following properties:

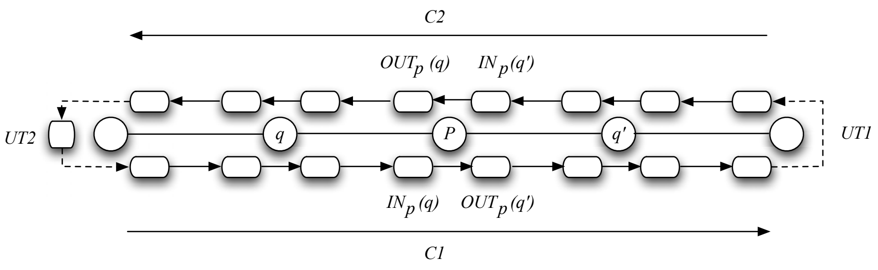


Fig. 1. Buffer Graph

1. $nb(m, IN_p(q)) = OUT_q(p)$, *i.e.,* the next buffer of the message m that is in the input buffer of p on the link (p, q) is the output buffer of q connected to p,
2. $nb(m, OUT_p(q)) = IN_q(p)$, *i.e.,* the next buffer of the message m that is in the output buffer of p on the link (p, q) is the input buffer of q connected to p.

3 Message Forwarding

In this section, we first give the idea of our snap-stabilizing message forwarding algorithm in the informal overview, then we give the formal description followed by the correctness proofs.

3.1 Overview of the Algorithm

In this section, we provide an informal description of our snap-stabilizing message forwarding algorithm that tolerates the corruption of the routing tables in the initial configuration.

To ease the reading of the section, we assume that there is no message in the system whose destination is not in the system. This restriction is not a problem as we will see in Section 5.

We assume that there is a self-stabilizing algorithm, *Rtables*, that calculates the routing tables and runs simultaneously to our algorithm. We assume that our algorithm has access to the routing tables via the function $Next_p(d)$ which returns the identity of the neighbour to which p must forward the message to reach the destination d. To reach our purpose we define a buffer graph on the chain which consists of two chains, one in each direction ($C1$ and $C2$ refer to Figure 1).

The overall idea of the algorithm is as follows: When a processor wants to generate a message, it consults the routing tables to determine the next neighbour by which the message will transit in order to reach the destination. Note that the generation is always done in the output buffers. Once the message is on the chain, it follows the buffer chain (according to the direction of the buffer graph).

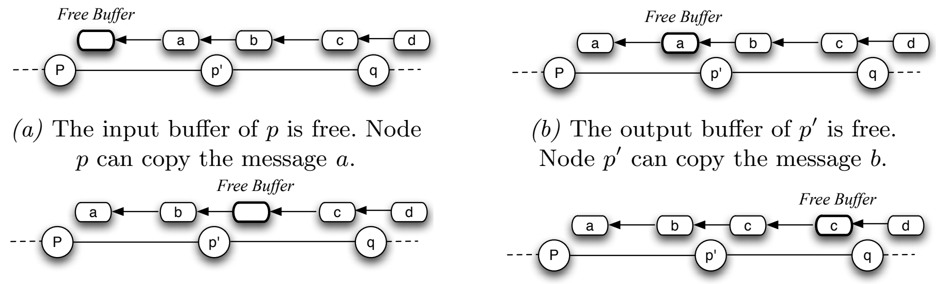
To avoid duplicated deliveries, each message is alternatively labelled by a color. If the messages can progress enough in the system (move) then it will either meet its destination and hence it will be consumed in finite time or it will reach the input buffer of one of the processors that are at the extremity of the chain. In the latter case, if the processor that is at the extremity of the chain is not the destination then, that means that the message was in the wrong direction. The idea is to change the direction of the message by copying it in the output buffer of the same processor (directly (UT1) or using the extra buffer (UT2), refer to Figure 1). Let p_0 be the processor that is at the extremity of the chain that has an internal buffer that we call Extra buffer.

Note that if the routing tables are stabilized and if all the messages are in the right direction then all the messages can move on $C1$ or $C2$ only and no deadlock happens. However, in the opposite case (the routing tables are not stabilized or some messages are in the wrong direction), deadlocks may happen if no control is introduced. For instance, suppose that in the initial configuration all the buffers, uncluding the extra buffer of $UT2$, contain different messages such that no message can be consumed. It is clear that in this case no message can move and the system is deadlocked. Thus, in order to solve this problem we have to delete at least one message. However, since we want a snap-stabilizing solution we cannot delete a message that has been generated. Thus, we have to introduce some control mechanisms in order to avoid this situation to appear dynamically (after the first configuration). In our case we decided to use the PIF algorithm that comprises two main phases: Broadcast (Flooding phase) and Feedback (acknowledgement phase) to control and avoid deadlock situations.

Before we explain how the PIF algorithm is used, let us focus on the message progression again. A buffer is said to be *free* if and only if it is empty (it contains no message) or contains the same message as the input buffer before it in the buffer graph buffer. For instance, if $IN_p(q) = OUT_q(p)$ then $OUT_q(p)$ is a free buffer. In the opposite case, a buffer is said to *busy.* The transmission of messages produces the filling and the cleaning of each buffer, *i.e.,* each buffer is alternatively free and busy. This mechanism clearly induces that *free slots* move into the buffer graph, a free slot corresponding to a free buffer at a given instant. The movement of free slots is shown in Figure 2[1]. Notice that the free slots move in the opposite direction of the message progression. This is the key feature on which the PIF control is based.

When there is a message that is in the wrong direction in the input buffer of the processor p_0, p_0 copies this message in its extra buffer releasing its input buffer and it initiates a PIF wave at the same time. The aim of the PIF waves is to escort the free slot that is in the input buffer of p_0 in order to bring it in the output buffer of p_0. Hence the message in the extra buffer can be copied in the output buffer and becomes in the right direction. Once the PIF wave is initiated no message can be generated on this free slot, at each time the Broadcast progresses on the chain the free slot moves as well following the PIF wave (the free slot moves by transmitting messages on $C1$ (refer to Figure 1). In

[1] Note that in the algorithm, the actions (b) and (c) are executed in the same step.



(a) The input buffer of p is free. Node p can copy the message a.

(b) The output buffer of p' is free. Node p' can copy the message b.

(c) The input buffer of p' is free. Node p' can copy the message c.

(d) The output buffer of q is free. Node q can copy the message d.

Fig. 2. An example showing the free slot movement

the worst case, the free slot is the only one, hence by moving the output buffer of the other extremity of the chain p becomes free. Depending on the destination of the message that is in the input buffer of p, either this message is consumed or copied in the output buffer of p. In both cases the input buffer of p contains a free slot.

In the same manner during the feedback phase, the free slot that is in the input buffer of the extremity p will progress at the same time as the feedback of the PIF wave. Note that this time the free slot moves on $C2$ (see Figure 1). Hence at the end of the PIF wave the output buffer that comes just after the extra buffer contains a free slot. Thus, the message that is in the extra buffer can be copied in this buffer and deleted from the extra buffer. Note that since the aim of the PIF wave is to bring the free slot in the output buffer of p_0 then when the PIF wave meets a processor that has a free buffer on $C2$ the PIF wave stops escorting the previous free slot and starts the feedback phase with this second free slot (it escorts the new free slot on $C2$). Thus, it is not necessary to reach the other extremity of the chain.

Now, in the case where there is a message in the extra buffer of p_0 such as no PIF wave is executed then we are sure that this message is an invalid message and can be deleted. In the same manner if there is a PIF wave that is executed such that at the end of the PIF wave the output buffer of p_0 is not free then like in the previous case we are sure that the message that is in the extra buffer is invalid and thus, can be deleted. Thus when all the buffers are full such as all the messages are different and cannot be consumed, then the extra buffer of p_0 will be released.

Note that in the description of our algorithm, we assumed the presence of a special processor p_0. This processor has an Extra buffer used to change the direction of messages that are in the input buffer of p_0 however their destination is different from p_0. In addition it has the ability to initiate a PIF wave. Note also that the other processors of the chain do not know where this special processor is. A symmetric solution can also be used (the two processors that are at the extremity of the chain execute the same algorithm) and hence both have an extra buffer and can initiate a PIF wave. The two PIF wave initiated at each extremity of the chain use different variable and are totally independent.

3.2 Formal Description of the Algorithm

We first define in this section the different data and variables that are used in our algorithm. Next, we present the PIF algorithm and give a formal description of the linear snap-stabilizing message forwarding algorithm.

Character `'?'` in the predicates and the algorithms means *any value*.

- **Data**
  - n is a natural integer equal to the number of processors of the chain.
  - $I = \{0, ..., n-1\}$ is the set of processors' identities of the chain.
  - N_p is the set of identities of the neighbours of the processor p.
- **Message**
  - (m, d, c): m contains the message by itself, *i.e.,* the data carried from the sender to the recipient, and $d \in I$ is the identity of the message recipient. In addition to m and d each message carries an extra field, c, which is a color number in $\{0, 1\}$ alternatively given to the messages to avoid duplicated deliveries.
- **Variable**
  - *In the forwarding algorithm*
    - $IN_p(q)$: The input buffer of p associated to the link (p, q).
    - $OUT_p(q)$: The output buffer of p associated to the link (p, q).
    - EXT_p: The Extra buffer of processor p which is at the extremity of the chain.
  - *In the PIF algorithm*
    - $S_p = (B \vee F \vee C, q)$ refers to the state of processor p (B, F, and C refer to Broadcast, Feedback, and Clean, respectively), q is a pointer to a neighbour of p.

- **Input/Output**
  - $Request_p$: Boolean, allows the communication with the higher layer, it is set to true by the application and false by the forwarding protocol.
  - $PIF\text{-}Request_p$: Boolean, allows the communication between the PIF and the forwarding algorithm, it is set to true by the forwarding algorithm and false by the PIF algorithm.
  - The variables of the PIF algorithm are the input of the forwarding algorithm.

- **Procedures**
  - $Next_p(d)$: refers to the neighbour of p given by the routing table for the destination d.

- $Deliver_p(m)$: delivers the message m to the higher layer of p.
- $Choice(c)$: chooses a color for the message m which is different from the color of the message that are in the buffers connected to the one that will contain m.

– **Predicates**

- $Consumption_p(q, m)$: $IN_p(q) = (m, d, c) \wedge d = p \wedge OUT_q(p) \neq (m, d, c)$
- $leaf_p(q)$: $S_q = (B, ?) \wedge (\forall\, q' \in N_p/\{q\}, S_{q'} \neq (B, p) \wedge (consumption_p(q) \vee OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p)))$.
- $NO\text{-}PIF_p$: $S_p = (C, NULL) \wedge \forall q \in N_p, S_q \neq (B, ?)$.
- $init\text{-}PIF$: $S_p = (C, NULL) \wedge (\forall q \in N_p, S_q = (C, NULL)) \wedge PIF\text{-}Request_p = true$.
- $Inter\text{-}trans_p(q)$: $IN_p(q) = (m, d, c) \wedge d \neq p \wedge OUT_q(p) \neq IN_p(q) \wedge (\exists q' \in N_p/\{q\}, OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p))$.
- $internal_p(q)$: $p \neq p_0 \wedge \neg\, leaf_p(q)$.
- $Road\text{-}Change_p(m)$: $p = p_0 \wedge IN_p(q) = (m, d, c) \wedge d \neq p \wedge EXT_p = \epsilon \wedge OUT_q(p) \neq IN_p(q)$.
- $\forall$ $TAction \in C, B$, we define $TAction\text{-}initiator_p$ the predicate: $p = p_0 \wedge$ (the garde of TAction in p is enabled).
- $\forall$ $Tproc \in \{internal, leaf\}$ and $TAction \in \{B, F\}$, $T\text{-}Action\text{-}Tproc_p(q)$ is defined by the predicate: $Tproc_p(q)$ is true $\wedge$ TAction of p is enabled.
- $PIF\text{-}Synchro_p(q)$: $(B_q\text{-}internal_p \vee F_q\text{-}leaf_p \vee F_q\text{-}internal_p) \wedge S_q = (B, ?)$.

– We define a fair pointer that chooses the actions that will be performed on the output buffer of a processor p. (Generation of a message or an internal transmission).

Algorithm 1. PIF

```
– For the initiator ($p_0$)
    • B-Action:: $init\text{-}PIF \rightarrow S_p := (B, -1)$, $PIF\text{-}Request_p := false$.
    • C-Action:: $S_p = (B, -1) \wedge \forall q \in N_p, S_q = (F, ?) \rightarrow S_p := (C, NULL)$.

– For the leaf processors: $leaf_p(q) = true \vee |N_p| = 1$
    • F-Action:: $S_p = (C, NULL) \rightarrow S_p := (F, q)$.
    • C-Action:: $S_p = (F, ?) \wedge \forall q \in N_p, S_q = (F \vee C, ?) \rightarrow S_p := (C, NULL)$.

– For the processors
    • B-Action:: $\exists! q \in N_p, S_q = (B, ?) \wedge S_p = (C, ?) \wedge \forall q' \in N_p/\{q\}, S_{q'} = (C, ?) \rightarrow$
      $S_p := (B, q)$.
    • F-Action:: $S_p = (B, q) \wedge S_q = (B, ?) \wedge \forall q' \in N_p/\{q\}, S_{q'} = (F, ?) \rightarrow S_p := (F, q)$.
    • C-Action:: $S_p = (F, ?) \wedge \forall q' \in N_p, S_{q'} = (F \vee C, ?) \rightarrow S_p := (C, NULL)$.

– Correction (For any processor)
    • $S_p = (B, q) \wedge S_q = (F \vee C, ?) \rightarrow S_p := (C, NULL)$.
    • $leaf_p(q) \wedge S_p = (B, q) \rightarrow S_p := (F, q)$.
```

Algorithm 2. Message Forwarding

- **Message generation (For every processor)**
  R1:: $Request_p \wedge Next_p(d) = q \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge NO\text{-}PIF_p \rightarrow OUT_p(q) := (m, d, choice(c)),\ Request_p := false.$

- **Message consumption (For every processor)**
  R2:: $\exists q \in N_p,\ \exists m \in M;\ Consumption_p(q, m) \rightarrow deliver_p(m),\ IN_p(q) := OUT_q(p).$

- **Internal transmission (For processors having 2 neighbors)**
  R3:: $\exists q \in N_p,\ \exists m \in M,\ \exists d \in I;\ Inter\text{-}trans_p(q, m, d) \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow OUT_p(q') := (m, d, choice(c)),\ IN_p(q) := OUT_q(p).$

- **Message transmission from q to p (For processors having 2 neighbors)**
  R4:: $IN_p(q) = \epsilon \wedge OUT_q(p) \neq \epsilon \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow IN_p(q) := OUT_q(p).$

- **Erasing a message after its transmission (For processors having 2 neighbors)**
  R5:: $\exists q \in N_p,\ OUT_p(q) = IN_q(p) \wedge (\forall q' \in N_p \setminus \{q\},\ IN_p(q') = \epsilon) \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow OUT_p(q) := \epsilon,\ IN_p(q') := OUT_{q'}(p).$

- **Erasing a message after its transmission (For the extremities)**
  R5':: $N_p = \{q\} \wedge OUT_p(q) = IN_q(p) \wedge IN_p(q) = \epsilon \wedge ((p = p_0) \Rightarrow (EXT_p = \epsilon)) \wedge (NO\text{-}PIF_p \vee PIF\text{-}Synchro_p(q)) \rightarrow OUT_p(q) := \epsilon,\ IN_p(q) := OUT_q(p).$

- **Road change (For the extremities)**
  - **R6**:: $Road\text{-}Change_p(m) \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \rightarrow OUT_p(q) := (m, d, choice(c)),\ IN_p(q) := OUT_q(p).$
  - **R7**:: $Road\text{-}Change_p(m) \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge PIF\text{-}Request_p = false \rightarrow PIF\text{-}Request_p := true.$
  - **R8**:: $Road\text{-}Change_p(m) \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge PIF\text{-}Request_p \wedge B\text{-}initiator \rightarrow EXT_p := IN_p(q),\ IN_p(q) := OUT_q(p).$
  - **R9**:: $p = p_0 \wedge EXT_p \neq \epsilon \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge C\text{-}Initiator \rightarrow OUT_p(q) := EXT_p,\ EXT_p := \epsilon.$
  - **R10**:: $p = p_0 \wedge EXT_p \neq \epsilon \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge C\text{-}Initiator \rightarrow EXT_p := \epsilon.$
  - **R11**:: $|N_p| = 1 \wedge p \neq 0 \wedge IN_p(q) = (m, d, c) \wedge d \neq p \wedge OUT_p(q) = \epsilon \wedge OUT_q(p) \neq IN_p(q) \rightarrow OUT_p(q) := (m, d, choice(c)),\ IN_p(q) := OUT_q(p).$

- **Correction (For p_0)**
- **R12**:: $p = p_0 \wedge EXT_p \neq \epsilon \wedge S_p \neq (B, -1) \rightarrow EXT_p = \epsilon.$
- **R13**:: $p = p_0 \wedge S_p = (B, ?) \wedge PIF\text{-}Request = true \rightarrow PIF\text{-}Request = false.$
- **R14**:: $p = p_0 \wedge S_p = (C, ?) \wedge PIF\text{-}Request = true \wedge [(IN_p(q) = (m, d, c) \wedge d = p) \vee IN_p(q) = \epsilon] \rightarrow PIF\text{-}Request = false.$

4 Proof of Correctness

In this section, we prove the correctness of our algorithm—due to the lack of space, the formal proofs are omitted[2]. We first show that starting from an arbitrary configuration, our protocol is deadlock free. Next, we show that no node can be starved of generating a new message. Next, we show the snap-stabilizing property of our solution by showing that, starting from any arbitrary configuration and even if the routing tables are not stabilized, every valid message is delivered to its destination once and only once in finite time.

Let us first state the following lemma:

Lemma 1. *The PIF protocol (Algorithm 1) is snap-stabilizing.*

[2] The complete proof can be found in `http://arxiv4.library.cornell.edu/abs/1006.3432`

The proof of Lemma 1 is based on the fact that the PIF algorithm introduced here is similar to the one proposed in [7]. The only effect of the message forwarding algorithm on the PIF algorithm (w.r.t. [7]) is that a leaf is no more only defined in terms of a topology property. Here, a leaf is a dynamic property of any node. It is easy to check that this change keeps the property of snap-stabilization.

We now show (Lemma 2) that the extra buffer located at p_0 cannot be infinitely continuously busy. As explained in Section 3, this solves the problem of deadlocks. We know from Lemma 1 that each time p_0 launches a PIF wave, then this wave terminates. When this happens, there are two cases: If the output buffer of p_0 is free, then message in the extra buffer is copied in this buffer. Otherwise (the output buffer is busy), the message in the extra buffer is deleted. In both cases, the extra buffer becomes free (a free slot is created).

Lemma 2. *If the extra buffer of the processor p_0 (EXT_{p_0}) which is at the extremity of the chain contains a message then this buffer becomes free after a finite time.*

We deduce from Lemma 2 that if the routing tables are not stabilized and if there is a message locking the input buffer of p_0, then this message is eventually copied in the extra buffer. Since the latter is infinitely often empty (Lemma 2 again), the following lemma is proven by induction:

Lemma 3. *All the messages progress in the system even if the routing tables are not stabilized.*

Let us call a *valid PIF* wave every PIF wave that is initiated by the processor p_0 at the same time as executing $R8$.

Lemma 4. *For every valid PIF wave, when the C-Action is executed in the initiator either $OUT_p(q) = IN_q(p)$ or $OUT_p(q) = \epsilon$.*

Proof Outline. The idea of the proof is as follows:

- We prove first that during the broadcast phase there is a synchrony between the PIF and the forwarding algorithm. Note that when the message that was in the input buffer of the initiator is copied in the extra buffer, the input buffer becomes free. The free slot in that buffer progresses in the chain at the same time as the broadcast of the PIF wave.
- Once the PIF reaches a leaf, a new buffer becomes free in $C2$ (refer to Figure 1).
- As in the broadcast phase, there is a synchrony between the PIF and the forwarding algorithm during the feedback phase. (The feedback will escort the new free slot on $C2$ to the output buffer of p_0.) □

In the remainder, we say that a message is in a *suitable* buffer if the buffer is on the right direction to its destination. A message is said to be deleted if it is removed from the system without being delivered.

Let us consider messages that are not deleted only. Let m be such a message. According to Lemma 3, m progresses in the system (no deadlock happens and no message stays in the same buffer indefinitely). So, if m is in a buffer that is not suitable for it, then m progresses in the system according to the buffer graph. Thus, it eventually reaches an extremity, which changes its direction. Hence, m is ensured to reach its destination, leading to the following lemma:

Lemma 5. *For every message that is not in a suitable buffer, it will undergo exactly a single route change if it not deleted before.*

Once the routing tables are stabilized, every new message is generated in a suitable buffer. So, it is clear from Lemma 5 that the number of messages that are not in a suitable buffer strictly decreases. The next lemma follows:

Lemma 6. *When the routing tables are stabilized and after a finite time, all the messages are in buffers that are suitable for them.*

From there, it is important to show that any processor can generate a message in finite time. From Lemma 6, all the messages are in suitable buffers in finite time. Since the PIF waves are used for route changes only, then:

Lemma 7. *When the routing tables are stabilized and all the messages are in suitable buffer, no PIF wave is initiated.*

From this point, the fair pointer mechanism cannot be disrupted by the PIF waves anymore. So, the fairness of message generation guarantees the following lemma:

Lemma 8. *every message can be generated in finite time under a weakly fair daemon.*

Due to the color management (Function $Choice(c)$), the next lemma follows:

Lemma 9. *The forwarding protocol never duplicates a valid message even if Rtables runs simultaneously.*

From Lemma 8, every message can be generated in finite time. From the PIF mechanism and its synchronization with the forwarding protocol the only message that can be deleted is the message that was in the extra buffer at the initial configuration. Thus:

Lemma 10. *Every valid message (that is generated by a processor) is never deleted unless it is delivered to its destination even if Rtables runs simultaneously.*

From Lemma 8, every message can be generated in finite time. From Lemma 10, every valid message is never deleted unless it is delivered to its destination even if *Rtables* runs simultaneously. From Lemma 9, no valid message is duplicated. Hence, the following theorem holds:

Theorem 1. *The proposed algorithm (Algorithms 1 and 2) is a snap-stabilizing message forwarding algorithm (satisfying SP) under a weakly fair daemon.*

Note that for any processor p, the protocol delivers at most $4n - 3$ invalid messages. Indeed, the system contains only $4n - 3$ buffers and in the worst case, initially, all the buffers are busy with different invalid messages (that were not generated).

5 Network Dynamic

In dynamic environments, processors may leave or join the network at any time. To keep our solution snap-stabilizing we assume that there are no crashes and if a processor wants to leave the network (disconnect), it releases its buffers (it sends all the messages it has to send and to wait for their reception by its neighbours) and accepts no more message before leaving.

In this discussion we assume that the rebuilt network is still a chain. It is fundamental to see that in dynamic systems the problem of keeping messages for ghost destinations with the hope they will join the network again and the lack of congestion are contradictory. If there is no bound on the number of leavings and joins this problem does not admit any solution. The only way is to redefine the problem in dynamic settings. For example we can modify the second point of the specification *(SP)* as follows: A valid message m generated by the processor p to the destination q is delivered to q in finite time if m, p and q are continuously in the same connected component during the forwarding of the message m. Even if that could appear very strong, this kind of hypothesis is often implied in practice. However we can remark that this new specification is equivalent to SP in static environments. Our algorithm can easily be adapted in order to be snap-stabilizing for this new specification in dynamic chains.

Thus, we can now delete some messages as follows: we suppose that every message has an additional boolean field initially set to false. When a message reaches an extremity which is not its destination we have two cases: *(i)* The value of the boolean is false, then the processor sets it to true and sends it in the opposite direction. *(ii)* The value of the boolean is true, then the processor deletes it (in this case, if the message is valid, it crossed all the processors of the chain without meeting its destination).

Finally, in order to avoid starvation of some processors, the speed of joins and leavings of the processors has to be slow enough to avoid a sequence of PIF waves that could prevent some processors to generate some messages.

6 Conclusion

In this paper, we presented the first snap-stabilizing message forwarding protocol that uses a number of buffers per node being independent of any global parameter. Our protocol works on a linear chain and uses only 4 buffers per link. It tolerates topology changes (provided that the topology remains a linear chain).

This is a preliminary version to get the same result on more general topologies. In particular, by combining a snap-stabilizing message forwarding protocol with any self-stabilizing overlay protocols (*e.g.,* [13] for DHT or [14–16] for *tries*), we would get a solution ensuring users to get right answers by querying the overlay architecture.

References

1. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
2. Huang, S.T., Chen, N.S.: A self-stabilizing algorithm for constructing breadth-first trees. Inf. Process. Lett. 41(2), 109–117 (1992)
3. Kosowski, A., Kuszner, L.: A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 75–82. Springer, Heidelberg (2006)
4. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 184–198. Springer, Heidelberg (2003)
5. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilizing end-to-end communication. Journal of High Speed Networks 5(4), 365–381 (1996)
6. Kushilevitz, E., Ostrovsky, R., Rosén, A.: Log-space polynomial end-to-end communication. In: STOC 1995: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, pp. 559–568. ACM, New York (1995)
7. Cournier, A., Dubois, S., Villain, V.: Snap-stabilization and PIF in tree networks. Distributed Computing 20(1), 3–19 (2007)
8. Cournier, A., Dubois, S., Villain, V.: A snap-stabilizing point-to-point communication protocol in message-switched networks. In: 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009), pp. 1–11 (2009)
9. Cournier, A., Dubois, S., Villain, V.: How to improve snap-stabilizing point-to-point communication space complexity? In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 195–208. Springer, Heidelberg (2009)
10. Edsger, W., Dijkstra: Self-stabilizing systems in spite of distributed control. ACM Commum. 17(11), 643–644 (1974)
11. Burns, J., Gouda, M., Miller, R.: On relaxing interleaving assumptions. In: Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89 (1989)
12. Merlin, P.M., Schweitzer, P.J.: Deadlock avoidance in store-and-forward networks. In: Jerusalem Conference on Information Technology, pp. 577–581 (1978)
13. Bertier, M., Bonnet, F., Kermarrec, A.M., Leroy, V., Peri, S., Raynal, M.: D2HT: the best of both worlds, Integrating RPS and DHT. In: European Dependable Computing Conference (2010)
14. Aspnes, J., Shah, G.: Skip Graphs. In: Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 384–393 (January 2003)
15. Caron, E., Desprez, F., Petit, F., Tedeschi, C.: Snap-stabilizing Prefix Tree for Peer-to-Peer Systems. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 82–96. Springer, Heidelberg (2007)
16. Caron, E., Datta, A., Petit, F., Tedeschi, C.: Self-stabilization in tree-structured P2P service discovery systems. In: 27th International Symposium on Reliable Distributed Systems (SRDS 2008), pp. 207–216. IEEE, Los Alamitos (2008)

Vulnerability Analysis of High Dimensional Complex Systems

Vedant Misra, Dion Harmon, and Yaneer Bar-Yam

New England Complex Systems Institute
{vedant,dion,yaneer}@necsi.edu
http://www.necsi.edu

Abstract. Complex systems experience dramatic changes in behavior and can undergo transitions from functional to dysfunctional states. An unstable system is prone to dysfunctional collective cascades that result from self-reinforcing behaviors within the system. Because many human and technological civilian and military systems today are complex systems, understanding their susceptibility to collective failure is a critical problem. Understanding vulnerability in complex systems requires an approach that characterizes the coupled behaviors at multiple scales of cascading failures. We used neuromorphic methods, which are modeled on the pattern-recognition circuitry of the brain and can find patterns in high-dimensional data at multiple scales, to develop a procedure for identifying the vulnerabilities of complex systems. This procedure was tested on microdynamic Internet2 network data. The result was a generic pipeline for identifying extreme events in high dimensional datasets.

Keywords: complex systems, vulnerability detection, stability and instability, high-dimensional, dimensionality reduction, neuromorphic methods, self-stabilizing systems.

1 Introduction

High dimensional complex systems are comprised of large numbers of interdependent elements [9]. When high dimensional systems perform critical tasks, the task is shared by and dynamically allocated among the components. The ability to distribute function dynamically enables robust and self-stabilizing function in a highly variable environment, but breaks down when collective loads are excessive, or when local failures or allocation process failures lead to cascading failures of large parts of the system as a whole. Thus, interdependence is necessary for function, but at the same time leads to dysfunctions associated with collective breakdowns. Because collective failures are dynamic and emergent, it is essential to identify when they occur and how to prevent them for the effective operation of a large number of critical systems.

Predicting the conditions of collective failures typically requires extensive study of the system and an understanding of both general dynamical characteristics and specific structural details. This is apparent in the limited prediction ability of such well-known collective failures as traffic jams and gridlock in road and highway systems. Similar issues arise in many much less visible systems, including power grids; water supply systems; communication networks (the internet); transportation networks (airlines, trains, shipping, etc); the global financial system; manufacturing, food and other commodity supply systems; and social networks and organizations. The potential impact of catastrophic failures in such systems has led to interest in developing detailed models of the systems, but not principles for evaluating system vulnerability [16,37].

Generally, interactions among a system's elements can generate collective dysfunctions, and operating conditions can trigger dramatic changes in the system's overall behavior, such as cascading failures. When a system is highly susceptible to behavioral changes of this sort, it is functionally unstable [20,29].

Vulnerable systems are likely to transition from stability to instability. Like a pencil on its tip, a vulnerable system will collapse if it experiences a sufficiently large deviation. By contrast, a stable system can restore itself to its equilibrium state when perturbed, like a pendulum. A system that is normally stable can become functionally unstable due to changes in global conditions or in relationships between the system's constituent elements.

Understanding the vulnerabilities of complex systems is a critical societal problem because of the many human and technological systems today that rely on distributed function and that can be characterized as high dimensional complex systems. Currently, responses to failure are reactive instead of proactive because we do not have a generic pipeline for analyzing high dimensional systems and anticipating their vulnerabilities. The goal of this paper is to develop a method for characterizing and anticipating extreme behavior and system failure, and to test it on a specific case study.

2 Internet2

Transitions from stability to instability are manifest in the Internet, which makes it a suitable prototype case for studying the dynamical properties of high dimensional systems [11,21,30,32,36]. A central function of the Internet is to enable any node to communicate with any other node transparently and without significant delays or lost communication. The Internet is designed as a self-stabiizing system [6], returning by itself to normal operation despite data errors and equipment failure [31] and despite dynamical deviations from functional states [14]. Nonetheless, the Internet architecture sometimes exhibits collective behaviors that make transparent end-to-end connectivity impossible. Such aggregate collective phenomena include cascading failures [20,22,29], the largest of which have been associated with worm attacks [12,35,38], and "route flapping," which occurs when a router fluctuates quickly between routes without settling into an effective routing pattern [24]. Other such phenomena include bottlenecks, storms, and collective oscillations[10,17,25].

A suitable prototype case for studying the dynamical properties of the Internet is the Internet2 network, backbone hubs of which are depicted in Figure 2. Internet2 is a collaboration of research institutions and companies that require

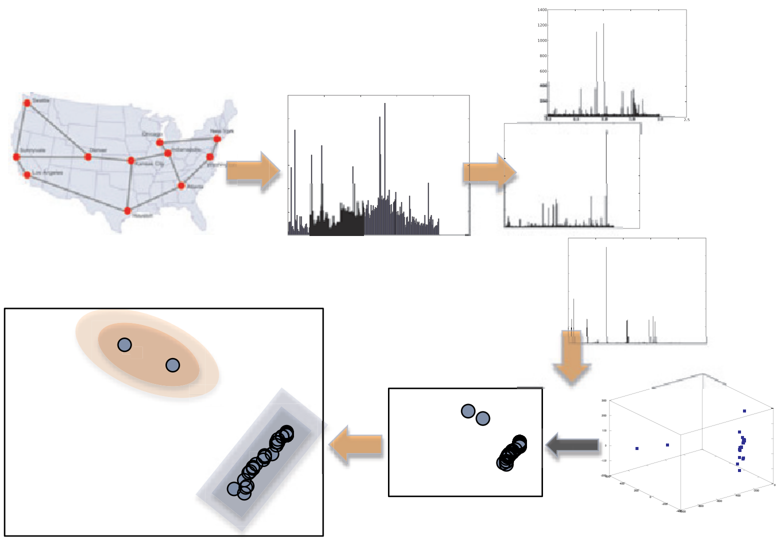


Fig. 1. Depiction of the pipeline. Each orange arrow represents one phase of the four-step process. Step 1, the sensor process, converts high dimensional heterogeneous data into a structured data stream representation. Step 2, the attention process, determines an attention trigger, extracts high-dimensional event data, and applies an alignment algorithm to align events in time. The result is a high-dimensional matrix. Step 3, the pattern process, employs pattern-discovery algorithms (gray arrow) to convert the high dimensional input into a lower-dimensional representation. Step 4, the interpretation process, characterizes the domains in the lower-dimensional representation space and makes it possible to distinguish normal system operations from system vulnerability or failure.

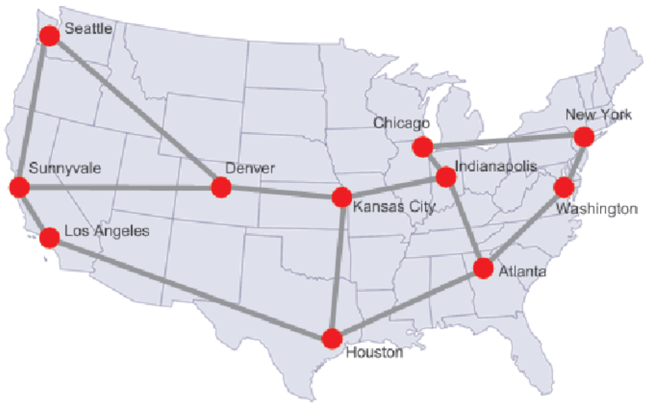


Fig. 2. Backbone hubs on the Internet2 network and traffic flow links between them

high-speed network infrastructure for communication. The Internet2 network is similar to the Internet in design and function, but smaller. While Internet2 is partially isolated from the Internet, it uses the same protocols for routing and is large enough to manifest collective dysfunction [18]. By design, massive volumes of network data can be collected through protocols built into the network, so extensive data about traffic on Internet2 has been archived [1,2,3,4,5,13]. The availability of historical data makes Internet2 a suitable laboratory for studying the collective failure of high-dimensional systems [23,33,39].

Internet2 data archives include logs of routing changes in the network issued by its communications protocol, Border Gateway Protocol (BGP) the same protocol used by routers on the Internet [34]. Under BGP, each node sends updates to its neighbors about which routes are most efficient for transmitting data. As the traffic demand changes, routes can become overloaded. Delays are detected by network routers that read messages from other routers. When delays are detected, BGP messages are sent between routers so that they change their routing tables [34].

Consider what might happen during a speech given by the President that is broadcast via a live video feed from Washington and is of interest to many people near Kansas City. Data packets may be transmitted from Washington, through Atlanta and Houston, to Kansas City (see Fig. 2). The resulting spike in network traffic may impede traffic from Indianapolis to Washington, which also passes through Atlanta.

To overcome this problem, Indianapolis traffic may be rerouted through Chicago and New York. A message from Atlanta to Indianapolis forcing this routing change constitutes a BGP update. The number of updates per minute varies from as little as a few dozen to several thousand depending upon the volume and nature of network activity. System failure occurs if the network experiences unusually high update volumes without settling into an effective routing pattern.

A simple example of system failure is one that occurs in self-generated traffic and route oscillations, where if one route is overloaded, the system dynamically reroutes traffic. However, rerouted traffic may cause overloading and delays in the new route while leaving the older route underutilized. Subsequent rerouting may exacerbate this effect by inducing routing oscillations that never achieve effective system utilization.

Update logs can in principle enable an observer to understand the network's dynamics. However, a single update, or even a large number of updates, are not indicative of failure. Aggregate behaviors must be characterized using patterns in the BGP traffic that enable us to distinguish poor resource utilization and failed communications from effective use of bandwidth and successful communications.

BGP updates are one of several types of records in the Internet2 archives. A central problem in developing a model that reveals the network's collective behaviors is determining which data best represent the system and which can be ignored. Additionally, understanding vulnerabilities in Internet2 requires an approach that recognizes the consequences of dependencies between nodes.

Traditional analysis, which focuses on individual variables and pair-wise correlations, is not sufficient to capture the system's collective behaviors and does little to help discriminate between useful and irrelevant data streams. Furthermore, collective behaviors at multiple scales should be described by k-fold correlations [7,8,15,26,27] that would be difficult to evaluate directly.

3 Neuromorphic Method

We have developed a process for identifying extreme behavior in high-dimensional systems using neuromorphic pattern discovery methods. This process characterizes the differences between patterns of collective behavior and uses them to recognize instability.

Neuromorphic pattern discovery methods are designed to mimic the nervous system's pattern-recognition circuitry using computer algorithms. Our approach consists of four stages: sensor, attention (event detection), pattern finding, and interpretation (classification). Each of these stages is analogous to a specific neurobiological function.

This report describes the successful implementation of our approach but does not describe the multiple methods that have been studied in order to develop this approach [9,28]. These studies investigated both conceptual and practical aspects of computational analysis. Some of the implementations tested in order to identify the strategy used and its refinement were performed on systems other than the Internet 2 data reported here.

Optimization of the method has been performed at a global rather than a local scale, which ensures that the neuromorphic method retains essential information while eliminating unnecessary or redundant information at each stage of processing. That the method does not require optimization at each stage is critical to its widespread applicability. Thus, in this method, no attempt is made at each phase of the process to isolate a single correct output, because a multiplicity of potential outputs can, after the interpretation process has been applied, result in the same conclusion.

We ensure that the patterns discovered by the process are meaningful by requiring that we retain key representative elements of the data stream. High dimensional data is retained until the penultimate stage. Information selection at earlier stages is designed to retain a representation of the coupled dynamical processes that underly system failure. The relative timing of events among multiple units is a critical aspect of the information retained that is often discarded in other forms of analysis. The relative timing data contains the high order correlations among the components of the system.

We overcome the difficulty faced by pattern recognition methods in resolving patterns where multiple instances of the same phenomenon do not appear the same in the input due to transformations such as time or space translation. To address this limitation, we treat the overall collective dynamics of the system as a single entity. We implement a symmetry-breaking process that aligns the events with each other in time. Such a symmetry-breaking process could also have been done in space, but was not necessary for this application.

3.1 Sensor Process

The sensor process refines large volumes of variously structured raw data into a well-defined and standardized high-dimensional parallel data stream. This is analogous to the brain converting compressions and rarefactions of air molecules against the eardrum, or light waves reaching the retina, into neural signals. The biological examples demonstrate that this stage of the process is system specific– i.e. the nature of the originating data is specific to the system being considered (sound or light) and the purpose of the sensory stage is to use a system-specific mechanism to convert the available information into a formatted data stream.

The 10 TB of available data for Internet2 were refined by a computer program that processed raw network data into a dynamic measure of network interactions while dealing with complications like data inconsistencies and gaps. The available data consist of second-resolution logs of various network statistics, including netflow data containing a record of IP flows passing through each router, throughput data consisting of records of the average rate of successful message delivery, and usage data comprised of logs of system load for individual machines at each node. The sensor program parsed these data and returned a time series of the most representative aspects of the data set for the collective behavior with which we are concerned — a data stream representing the existence of a change in the router table at a particular router of the system.

3.2 Attention Process

The next phase of processing requires that we specify a “trigger,”—a dynamic feature of large excursions that we can use to identify when an extreme event may be happening. The trigger is tuned using historical data to maximize the number of events identified by the event detection process while excluding false positives from the data set.

The trigger is based on an aggregate measure of the system's behavior over space and time – in the case of Internet2, across major backbone nodes. Event data is extracted from the data stream using a program that monitors this aggregate measure. A deviation of the measure from a background value well above its statistical variation signals an event – we looked for deviations larger than 3 standard deviations above the moving average – at which point the event's data stream is extracted. Figure 3 is a visualization of an aggregate measure of the behavior of the system, in which each bar represents the number of update messages per day over seven months. The figure shows that update spikes are an easily-identified first approximation for what might constitute an appropriate trigger.

The next phase is to align the event data; this is an essential part of the attention process because it enables comparison of the intra-event dynamics of different events. An algorithm extracts a 40-hour window of data surround each event and examines it to identify the period within that window that best represents aberrant network activity, and then shifts each window in time according to the location of the most active period. In the example in Figure 4, the windows have been shifted to align the largest spikes within the window.

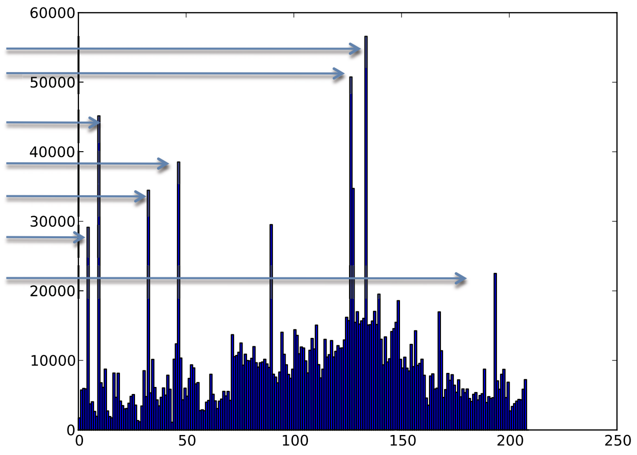


Fig. 3. BGP Updates per day over seven months at an Internet2 node; x-axis is days since the start of 2008. Arrows indicate days that are visually identifiable as "spikes" in the number of BGP updates. The attention process in a neuromorphic algorithm must identify the quantitative signature of such anomalies and use it to extract a data stream that represents the network's dynamical properties.

The alignment process outputs a set of 15,000-element vectors, one vector per event. Each vector represents the behavior at every node over a specific time frame, with the salient features of each event aligned within the output matrix.

3.3 Pattern Process

To identify details of the dynamics of large excursions, we employ a wide array of pattern finding algorithms designed for processing high dimensional, high volume data. Many of these algorithms reduce the dimensionality of the system description by discarding dimensions that are not essential for characterizing the system's overall behavior. A common approach to dimensionality reduction is to assume that the data lie on an embedded non-linear manifold within the high-dimensional space defined by the complete dataset. While some techniques give a mapping from the high dimensional space to the low dimensional space, others provide only a visualization of the low-dimensional data.

Both types of algorithms are designed to maximize coverage of the lower dimensional representation space and minimize the distortion of the projection. Dimensionality reduction algorithms map high-dimensional data vectors ξ_i in an input space of dimension n to lower-dimensional representation vectors x_i in an output space of dimension $m << n$. The algorithms seek to preserve the distances between pairs of points. Given metrics d_ξ and d_x that measure distances between high-dimensional vectors and low-dimensional vectors, respectively, the distances $d_x(x_i, x_j)$ approximate the distances $d_\xi(\xi_i, \xi_j)$. At the same time the algorithms try to maximize a measure of spatial covering so that the

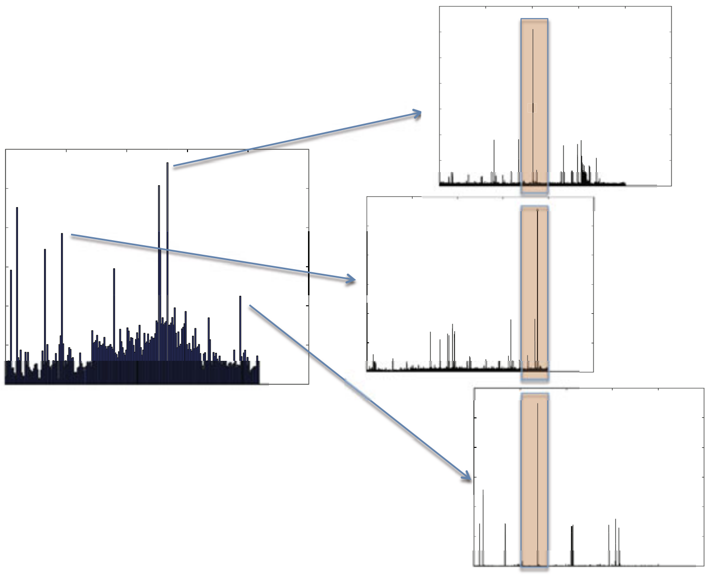


Fig. 4. Aligning events using their dynamic profiles. The window at left depicts the dynamic profile of an Internet2 node over several months. The windows at right depict the profiles of individual days during the month that were flagged during event detection. The alignment process determines what significant characteristic of each event best correlates to significant characteristics of other events and aligns them using the resulting criterion.

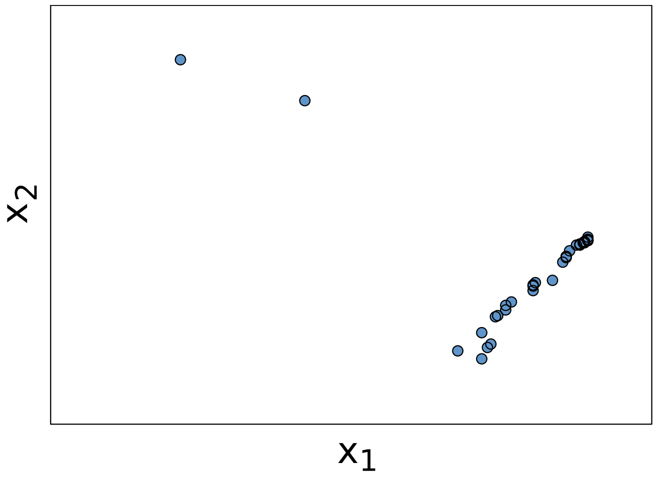


Fig. 5. Dimensionality-reduced representation of event vectors. Each point represents an event. This plot represents an attribute space of event parameters found to be significant by the dimensionality reduction algorithm. Specifically, x_1 and x_2 are the two most prominent lower-dimensional parameters. Note the two extreme events, which appear separate from the large number of rerouting events that did not destabilize the network.

representation vectors x_i represent as much of the high dimension variation in the lower dimension as possible.

One technique for maximizing spatial covering employed by some dimensionality reduction algorithms is to use an intermediary transformation from the input space to a *feature space* in which the underlying structure of the input vectors is more visible. This enables non-linear methods to be incorporated in an otherwise linear process. One such method is Kernel Principal Component Analysis (Kernel PCA), in which the linear operations of PCA are applied to the feature space with nonlinear mapping. Given a set of input data points $\xi_i, i = 1, 2, \ldots, n$ in the n-dimensional input space, we would first nonlinearly transform the i^{th} input vectors ξ_i into a point $\Phi(\xi_i)$ in an N_H dimensional feature space H where each

$$\Phi(\xi_i) = (\phi_1(\xi_i), \ldots, \phi_{N_H}(\xi_i)) \in H, \quad i = 1, 2, \ldots, n. \tag{1}$$

and then use PCA in the feature space H [19]. Carrying out linear PCA in the feature space then yields a presumably lower-dimensional distance-preserving representation of the input vectors x_i, $i = 1, 2, \ldots, m$, with $m < n$. The method we employed was inspired by Kernel PCA; the nonlinear sensor and attention processes primed the input space for dimensionality reduction, after which linear methods were sufficient for identifying structure in the data.

The results of nonlinear dimensionality reduction are visualizations of the high dimensional data in a lower dimensional space that make it possible to uncover patterns within the data using the coordinates of the resulting points in the low dimensional space. Figure 5 depicts a scatter plot generated using the results of dimensionality reduction.

3.4 Interpretation Process

The pattern finding process outputs a representation of the lower-dimensional space to which high-dimensional input was mapped. Just as in a neural processing system, interpration of this lower-dimensional representation must be guided by an understanding of the consequences of previous events, either by studying long term feedback or by training from a previous generation. Similarly, the interpretation of the events in the neuromorphic system can be guided by human interpretation. Since the dimensionality of this output space is small, our own interpretive processes can identify the relevant regions of the space from the historical data.

Extreme events appeared separate from the cluster of background events in the lower-dimensional output space. They are visible in Figure 5 in the upper-left. To determine what property of these events separates them from the trend, we used radar charts that illustrated the node-to-node variation of each event and temporal plots to visualize the dynamics of the activity.

Figure 6 contains two such plots, along with temporal plots and indications of where each event falls in Figure 5. The radar chart insets indicate the magnitude of each event at each node. Clearly visible in the first of the two events, which manifested at every node but was aberrantly large at only one node, are several

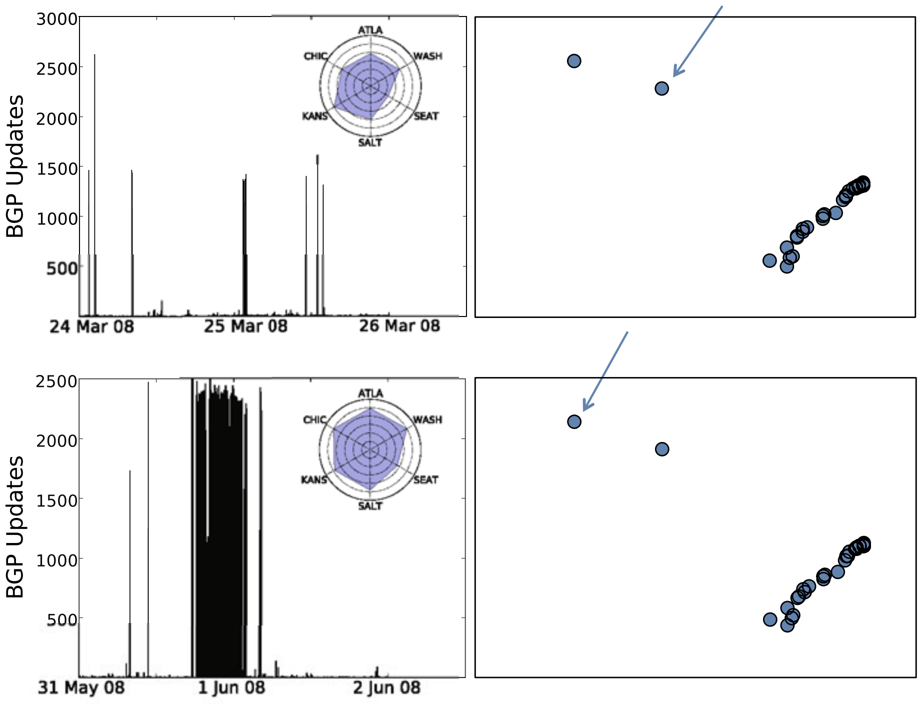


Fig. 6. Dynamic profiles for the two events. The insets indicate the magnitude of each event at each node. Also indicated is where each event appeared on the dimensionality-reduced plot.

distinct spikes in network activity. This repeated and persistent aberrant activity is a signature of systemic instability. The second event depicted consisted of nearly eight hours of large numbers of updates. The entire network was forced to completely rewire itself every 30 seconds. This is precisely the type of system failure our method is designed to detect.

The results of our analysis prompted us to revisit the theoretical nature of vulnerability and failure. Within the context of vulnerable systems, large cascades are common rather than isolated events. System failure is a persistent and recurrent cascade. Thus, both vulnerability and failure can be identified as persistent large deviations from normal behavior.

Our analysis shows that self-stabilizing systems can be vulnerable to collective dysfunctions. While the routing systems can adapt rapidly to changes in the network and the dynamics of demand, there are conditions of the system or the demand on the system that can lead to cascades that cause dysfunction. Recognizing these conditions and detecting extreme events is essential to expanding the domain of effective function.

Our processing pipeline is well-suited to detecting extreme events because of the attentional trigger which aligns events according to their largest excursion.

All events where the large excursion is sufficiently isolated from other large events will appear much more similar to each other than they do to events for which multiple large excursions occur over time and across the network. This ensures that the pattern recognition algorithms will be able to distinguish between the two types.

Our method sheds light on the dynamical characteristics of extreme events and explains why our processing pipeline can distinguish extreme events from those that do not result in system failure. This new insight provides a general explanation of how and why real-time detection of extreme events is possible.

4 Conclusion

We have developed a neuromorphic information processing pipeline that can characterize the vulnerability of complex systems. The process consists of extracting a dynamic measure of network activity and processing the resulting time series to find patterns of collective behavior. The process succeeded in identifying extreme events that are distinct from high demand but otherwise effective system activity. Novel spatiotemporal analysis and dimensionality reduction techniques made this result possible. The pipeline can be used quite generally for analyzing high-dimensional time series and isolating extreme events in real world communication, transportation and economic systems. This system can be combined with real-time system monitoring of data streams to identify dysfunctional behaviors and characterize vulnerabilities or system instabilities as they occur.

Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency, the U. S. Army Research Laboratory, and the U. S. Army Research Office under contract/grant number W911NF-09-1-0165.

References

1. B.G.P.: routing table analysis. modified (August 2006), `http://thyme.apnic.net/`
2. BGPmon: Next generation BGP Monitor, `http://bgpmon.netsec.colostate.edu/`
3. Border gateway protocol (BGP) data collection standard communities, `http://www.bgp4.as/bgp-data-collection-standard-communities` (modified February 23, 2006)
4. New sources of BGP data, `http://inl.info.ucl.ac.be/blogs/08-10-27-new-sources-bgp-data` (modified October 28, 2008)
5. University of Oregon Route Views Project, `http://www.routeviews.org/` (modified January 25, 2005)
6. Adam, C., Stadler, R.: Patterns for routing and self-stabilization. In: Proc IEEE/IFPS NOMS (2004)

7. Bar-Yam, Y.: Multiscale complexity/entropy. Advances in Complex Systems 7, 47–63
8. Bar-Yam, Y.: Multiscale variety in complex systems. Complexity 9(4), 37–45 (2004)
9. Bar-Yam, Y.: Dynamics of Complex Systems. Perseus Press, Cambridge (1997)
10. Barabási, A.L., de Menezes, M., Balensiefer, S., Brockman, J.: Hot spots and universality in network dynamics. The European Physical Journal B 38, 169–175 (2004)
11. Cowie, J., Ogielski, A., Premore, B., Yuan, Y.: Global routing instabilities triggered by Code Red II and Nimda worm attacks
12. Cowie, J., Ogielski, A., Premore, B., Yuan, Y.: Internet worms and global routing instabilities: scalability and traffic control in IP networks II. In: Proc SPIE, vol. 4868, pp. 195–199 (2002)
13. Cymru, T.: BGP monitoring, http://www.team-cymru.org/Monitoring/BGP/ (modified 2010)
14. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
15. Gheorghiu-Svirchevski, S., Bar-Yam, Y.: Multiscale analysis of information correlations in an infinite-range, ferromagnetic ising system. Phys. Rev. E 70(066115) (2004)
16. Hohn, N.: Measuring, understanding, and modelling internet traffic
17. Huberman, B., Lukose, R.: Social dilemmas and internet congestion. Science 277 (1997)
18. Internet2: http://www.internet2.edu
19. Izenman, A.: Modern Multivariate Statistical Techniques. Springer, Heidelberg (2008)
20. Jr., E.C., Ge, Z., Misra, V., Towsley, D.: Network resilience: exploring cascading failures within bgp. In: Proceedings of Allerton Conference on Communications, Computing, and Control (2001)
21. Labovitz, C., Malan, G., Jahanian, F.: Internet routing instability 6(5) (1998)
22. Lakhina, A., Crovella, M., Diot, C.: Mining anomalies using traffic feature distributions. BUCS (002) (2005)
23. M G.:
24. Mao, Z., Govindan, R., Varghese, G., Katz, R.: Route flap damping exacerbates internet routing convergence. In: Proceedings of ACM SIGCOMM, Pittsburgh, PA, USA, pp. 221–233 (2002)
25. de Menezes, M., Barabási, A.L.: Fluctuations in network dynamics. Phys. Rev. Lett. 92(028701) (2008)
26. Metzler, R., Bar-Yam, Y.: Multiscale analysis of correlated gaussians. Phys. Rev. E 71(046114), 2005 (2005)
27. Metzler, R., Bar-Yam, Y., Kardar, M.: Information flow through a chaotic channel: prediction and postdiction at finite resolution. Phys. Rev. E 70(020605) (2004)
28. Misra, V., Harmon, D., de Aguiar, M., Epstein, I., Braha, D., Bar-Yam, Y.: Vulnerability detection in complex systems. Unpublished report (2009)
29. Motter, A., Lai, Y.C.: Cascade-based attacks on complex networks. Phys. Rev. E 66(065102) (2002)
30. Nicol, D.: Challenges in using simulation to explain global routing instabilities. In: Conference on Grand Challenges in Simulation (2002)
31. Perlman, R.: Interconnections: Bridges, Routers, Switches and Internetworking Protocols, 2nd edn. Addison-Wesley Longman, Amsterdam (2001)
32. Valverde, S., Internet's, R.S.: critical path horizon. The European Physical Journal B 38(2) (2004)

33. Siganos, G., Faloutsos, M.: Detection of BGP routing misbehavior against cyberterrorism. In: IEEE Military Communications Conference (2005)
34. Smith, R.: The dynamics of internet traffic: self-similarity, self-organization, and complex phenomena. ArXiV:0806.3374 (2008)
35. Wang, L., Zhao, X., Pei, D., Bush, R., Massey, D., Mankin, A., Wu, S., Zhang, L.: Observation and analysis of BGP behavior under stress. In: Proceedings of the 2nd ACM SIGCOMM workshop of internet measurement (2002)
36. Wang, Y.: Protecting mission critical networks. Seminar on Network Security Publications in Telecommunications Software and Multimedia (2001)
37. Yuan, J., Mills, K.: Macroscopic dynamics in large-scale data networks. Complex Dynamics in Communication Networks (2005)
38. Zou, C.C., Gong, W., Towsley, D.: Code red worm propagation modeling and analysis. In: Proceedings of the 9th ACM conference on Computer and communications security (2002)
39. Zou, C., Gao, L., Gong, W., Towsley, D.: Monitoring and early warning for internet worms. In: Proceedings of the 10th ACM conference on Computer and communications security (2003)

Storage Capacity of Labeled Graphs

Dana Angluin[1,⋆], James Aspnes[1,⋆⋆], Rida A. Bazzi[2], Jiang Chen[3], David Eisenstat[4], and Goran Konjevod[2]

[1] Department of Computer Science, Yale University
[2] Department of Computer Science and Engineering, Arizona State University
[3] Google
[4] Department of Computer Science, Brown University

Abstract. We consider the question of how much information can be stored by labeling the vertices of a connected undirected graph G using a constant-size set of labels, when isomorphic labelings are not distinguishable. An exact information-theoretic bound is easily obtained by counting the number of isomorphism classes of labelings of G, which we call the **information-theoretic capacity** of the graph. More interesting is the **effective capacity** of members of some class of graphs, the number of states distinguishable by a Turing machine that uses the labeled graph itself in place of the usual linear tape. We show that the effective capacity equals the information-theoretic capacity up to constant factors for trees, random graphs with polynomial edge probabilities, and bounded-degree graphs.

1 Introduction

We consider what happens if we replace the linear tape of a standard Turing machine with some fixed finite connected graph. This gives us a way to represent self-organizing systems consisting of many communicating finite-state machines, where at any time, one machine (the location of the Turing machine head) takes a leadership role. Our main question is how much computing power such machines can cooperate to achieve. The answer depends on the inherent storage capacity of the graph, a function of its size (bigger gives more space) and symmetries (more symmetries makes the space harder to exploit).

In more detail, a **graph Turing machine** consists of an undirected connected graph G, each of whose nodes holds a symbol from some finite alphabet, together with a finite-state controller that can move around the graph and update the symbols written on nodes. Because there is no built-in sense of direction on an arbitrary graph, the left and right moves of a standard Turing machine controller are replaced by moves to adjacent graph nodes with a given symbol. If there is no such adjacent graph node, the move operation fails, which allows the controller

⋆ Supported in part by NSF grant CCF-0916389.
⋆⋆ Supported in part by NSF grants CNS-0435201 and CCF-0916389.

to test its immediate neighborhood for the absence of particular symbols. If there is more than one such node, which node the controller moves to is chosen arbitrarily. (A more formal definition of the model is given in Section 3.)

The intent of this model is to represent what computations are feasible in various classes of simple distributed systems made up of a network of finite-state machines. Inclusion of an explicit head that can move nondeterministically to adjacent nodes (thus breaking at least local symmetries in the graph) makes the model slightly stronger than similar models from the self-stabilization literature (e.g., Dijkstra's original model in [8]) or population protocols [5]; we discuss the connection between our model and these other models in Section 2.

The main limitation on what a graph Turing machine can compute appears to be the intrinsic **storage capacity** of its graph. For some graphs (paths, for example) the storage capacity is essentially equivalent to a Turing machine tape of the same size. For others (cliques, stars, some trees), the usable storage capacity may be much less, because symmetries within the graph make it difficult to distinguish different nodes with the same labeling. We define a notion of **information-theoretic capacity** of a graph (Section 4.1) that captures the number of distinguishable classes of labelings of the graph. Essentially this comes down to counting equivalence classes of labelings under automorphisms of the graph; it is related to the notion of the **distinguishing number** of a graph, which we discuss further in Section 2.3.

The information-theoretic capacity puts an upper bound on the **effective capacity** of the graph, the amount of storage that it provides to the graph Turing machine head (defined formally in Section 4.2). Extracting usable capacity requires not only that labelings of the graph are distinguishable in principle but that they are distinguishable to the finite-state controller in a way that allows it to simulate a classic Turing machine tape. We show that an arbitrary graph with n nodes provides at least $\Omega(\log n)$ tape cells worth of effective capacity (which matches the information-theoretic upper bound for cliques and stars, up to constant factors). For specific classes of graphs, including trees (Section 7), random graphs with polynomial edge probabilities (Section 9), and bounded-degree graphs (Section 8), we show that the effective capacity similarly matches the information-theoretic capacity.

Notably, these classes of graphs are ones for which testing graph isomorphism is easy. Whether we can extract the full capacity of a general graph is open, and appears to be related to whether graph isomorphism for arbitrary graphs can be solved in **LOGSPACE**. We discuss this issue in Section 10.

2 Related Work

2.1 Self-stabilizing Models

A graph Turing machine bears a strong resemblance to a network of finite-state machines, which has been the basis for numerous models of distributed computing, especially in the self-stabilization literature. Perhaps closest to the present work is the original self-stabilizing model of Dijkstra [8], where we have a collection of finite-state nodes organized as a finite connected undirected graph, and at each step some node may undergo a transition to a new state that depends on its previous state and the state of its immediate neighbors. The main difference between the graph Turing machine model and this is the existence of a unique head, and even more so, its ability to move to a single neighbor of the current node—these properties break symmetry in ways that are often difficult in classic self-stabilizing systems. A limitation of the graph Turing machine model is the restriction on what the head can sense of adjoining nodes: it cannot distinguish neighbors in the same state, or even detect whether one or many neighbors is in a particular state.

Itkis and Levin [11] give a general method for doing self-stabilizing computations in asynchronous general topology networks. Their model is stronger than ours, in that each node can maintain pointers to its neighbors (in particular, it can distinguish neighbors in the same state). Nonetheless, we have found some of the techniques in their paper useful in obtaining our current results.

2.2 Population Protocols

There is also a close connection between our model and the **population protocol** model [5], in which a collection of finite-state agents interact pairwise, each member of the pair updating its state based on the prior states of both agents (see [6] for a recent survey on this and related models). This is especially true for work on population protocols with restricted communication graphs (for example, [3]). Indeed, it is *almost* possible to simulate a graph Turing machine in a population protocol, simply by moving the state of the head around as part of the state of the node it is placed on, and using interactions with neighbors to sense the local state. The missing piece in the population protocol model is that there is no mechanism for detecting the absence of a particular state in the immediate neighborhood. Although a fairness condition implies that every neighbor will make itself known eventually, the head node has no way to tell if this has happened yet. Urn automata [4], a precursor to the population protocol model in which a finite-state controller manages the population, also have some similarities to graph Turing machines, especially in the combination of a classical Turing-machine controller with an unusual data store.

The **community protocol** model of Guerraoui and Ruppert [9,10] extends population protocols by allowing agents to store a constant number of pointers to other agents that can only be used in limited ways. Despite these restrictions, Guerraoui and Ruppert show that community protocols with n agents can simulate **storage modification machines** as defined by Schönhage [16], which consist of a dynamic graph on n nodes updated by a finite-state controller. Such machines can in turn simulate standard Turing machines with $O(n \log n)$ space. The community protocol and storage modification machine models are both stronger than our graph Turing machines because they allow for a dynamic graph, while our machines have to work with the graph they are given.

2.3 Distinguishing Number

The **distinguishing number** [2] $d(G)$ of a graph G is the minimum number of colors needed to color the vertices of G so that G has no color-preserving automorphisms.

If the distinguishing number of a class of graphs is bounded, then we can in principle color the nodes with a distinguishing coloring that uniquely identifies each node based on its position in the graph (though it still may require substantial work to identify a particular node). With a large enough alphabet, we can use a second component of the state to store the contents of a Turing machine tape cell. This would give an information-theoretic capacity for the graph of $\Theta(n)$.

Albertson and Collins [2] show that any graph has distinguishing number $O(\log(|\operatorname{Aut}(G)|))$. This implies that the information-theoretic capacity of the class of graphs with constant-sized automorphism groups is $\Theta(n)$ (the effective capacity may be smaller in some cases). Thus graphs with low information-theoretic capacity will have large automorphism groups, i.e., lots of symmetry.

Computing distinguishing number exactly appears to be difficult. Some improved characterizations may be found in [15,1].

3 Graph Turing Machines

Formally, a **graph Turing machine** is specified by a 4-tuple (Σ, Q, q_0, δ) where Σ is a finite **alphabet** of **tape symbols**, Q is a finite set of **controller states**, $q_0 \in Q$ is the **initial controller state**, and $\delta : Q \times \Sigma \times \mathcal{P}(\Sigma) \to (Q \cup \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Sigma \times \Sigma$ is the **transition function**. We assume that the alphabet Σ contains the special **blank** symbol $-$. The graph G on which the machine runs and the initial position of the controller $v_0 \in V(G)$ are supplied separately.

The first argument of the transition function δ is the current state of the controller, the second argument is the symbol on the current node, and the third gives the set of symbols that appear on one or more of the neighbors of the current node. The output of δ gives the new state of the controller, the symbol to write to the current node, and the symbol indicating which adjacent node to move to.

The special states q_{accept} and q_{reject} are accepting and rejecting **halting states**, respectively; if the machine enters one of these two halting states, there is no move to a neighboring node. For transitions that do not enter a halting state, we require that the target symbol be present in the immediate neighborhood (i.e., that it is chosen from the set of neighboring symbols). Implicit in this rule is that, in the unusual event that G contains only a single node v_0 and the set of symbols on neighboring nodes is empty, the machine must halt immediately.

A **configuration** of a graph Turing machine (Σ, Q, q_0, δ) running on a graph G is a triple $(q, v, s) \in (Q \cup \{q_{\text{accept}}, q_{\text{reject}}\}) \times V(G) \times \Sigma^{V(G)}$ where q is the current state of the controller, v is its current position, and s specifies the current tape symbol s_v on each node v of G. A **halting configuration** is a configuration in which the controller state is either q_{accept} or q_{reject}; in the former case it is an **accepting configuration** and in the latter a **rejecting configuration**.

We consider (G, v_0) to be the input to the graph Turing machine, where G is a graph and $v_0 \in V(G)$ is the initial node. Given an input (G, v_0), the **initial configuration** of the machine is $(q_0, v_0, \{-\})^{V(G)}$, i.e., the configuration in which the controller starts on node v_0 in state q_0 and all nodes contain the blank symbol. As with standard Turing machines, we write $M(G, v_0)$ for the machine M operating on input (G, v_0).

Given a non-halting configuration (q, v, s), let

$$(q', \sigma_1, \sigma_2) = \delta\left(q, s_v, \{s_u : (u, v) \in E(G)\}\right).$$

There is a **transition** from (q, v, s) to (q', v', s') if (a) $s'_v = \sigma_1$, (b) $s'_u = s_u$ for all $u \in V(G) - \{v\}$, and (c) $s'_{v'} = s_{v'} = \sigma_2$. Note that there may be more than one such transition if there is more than one neighbor v' with $s_{v'} = \sigma_2$, Note further that there are no transitions from a halting configuration.

Given input (G, v_0), a **computation path** is a sequence of configurations $C_0, C_1, \ldots$ where C_0 is the initial configuration and there is a transition from C_i to C_{i+1} for each i. A graph Turing machine **halts** on input (G, v_0) if every computation path is finite. A graph Turing machine **accepts** (**rejects**) input (G, v_0) if every computation path is finite and ends in an accepting (rejecting) configuration. The **running time** of a graph Turing machine with input (G, v_0) is the maximum length of any computation path, or ∞ if no such maximum exists.

Though most computations of graph Turing machines are inherently nondeterministic, we will call a graph Turing machine **deterministic** if for any input (G, v_0) it either accepts on all computation paths or rejects on all computation paths. The justification for this unusual usage is that for a deterministic graph Turing machine, the choice of which of several alternative nodes to move to can be made arbitrarily—possibly even according to some deterministic tie-breaking rule (whose inclusion would complicate the model.)

We say that a graph Turing machine M_1 with input (G_1, v_1) **simulates** a graph Turing machine M_2 with input (G_2, v_2) if there is a mapping from configurations of $M_1(G_1, v_1)$ to configurations of $M_2(G_2, v_2)$ such that every transition of $M_1(G_1, v_1)$ maps to either a transition of $M_2(G_2, v_2)$ or to a no-op. Often we will have $(G_1, v_1) = (G_2, v_2)$, with the main differences between M_1 and M_2 being that M_2 is a graph Turing machine extended in some way, such as by adding multiple heads or more built-in storage. The particular case of simulating a standard Turing machine will be used to define effective capacity in Section 4.2.

4 Storage Capacity of Graphs

In this section, we consider the question of how much information can be stored in a given graph. We first look at the information-theoretic capacity bound (Section 4.1), then consider how much of this potential capacity can actually be extracted (Section 4.2).

4.1 Information-Theoretic Capacity

The information-theoretic capacity of a graph is just the base 2 logarithm of the number of distinguishable labelings of its nodes, where two labelings are distinguishable if there is no automorphism of the graph that carries one to the other and equivalent otherwise. This quantity is in principle computable using Burnside's Lemma; the number of distinguishable labelings is

$$L(G) = |X/\operatorname{Aut} G| = \frac{1}{|\operatorname{Aut}(G)|} \sum_{g \in \operatorname{Aut}(G)} |X^g|,$$

where X is the set of all labelings, $X/\operatorname{Aut}(G)$ is the quotient set of equivalence classes of labelings under automorphisms in G, and X^g is the set of labelings preserved by a particular automorphism g. The information-theoretic capacity I_G of G is then the base 2 logarithm $\lg L(G)$ of this quantity.

In practice, computing the number of distinguishable labelings will be easiest for classes of graphs that have no non-trivial automorphisms, or for which the set of automorphisms has a particularly simple structure, such as cliques, stars, or trees. For example, any permutation of the nodes of a clique, or any permutation of the non-central nodes of a star, is an automorphism of the graph. We can map one labeling to another by a color-preserving permutation precisely when each has the same number of nodes with each color (in the case of a star, when this property holds for the leaves and the central nodes have the same color). It follows that an equivalence class can be specified by counting the number of nodes with each color (plus $O(1)$ bits for the central node for a star). In either case we get $\Theta(\log n)$ bits of information.

Graphs with constant distinguishing number (see Section 2.3) or for which a small number of carefully-colored nodes eliminate color-preserving automorphisms will have information-theoretic capacity $\Theta(n)$. An example would be a path; by fixing distinct colors of the endpoints, no color-preserving automorphisms remain.

The information-theoretic capacity of general trees depends heavily on the structure of the tree: whether it looks more like a star, with many automorphisms, or a path, with few. We discuss this issue in detail in Section 7.

In general, we can bound the information-theoretic capacity of any graph with n nodes by $O(n)$; this is just the number of bits needed to represent all possible labelings without considering equivalence.

The usefulness of the information-theoretic capacity is that it puts an upper bound on how much state can be stored in the graph. Call two configurations of a graph Turing machine **equivalent** if

1. The head is in the same state in both configurations.
2. There is a label-preserving automorphism of G that carries the position of the head in the first configuration to the position of the head in the second configuration.

It is not hard to see that equivalent states have equivalent successors, since the same automorphism can be used after a transition as long as we are careful to make the heads move to matching locations. It follows that for the purpose of simulating a graph Turing machine, we need only record its state up to equivalence.

Theorem 1. *Fix a graph Turing machine, and suppose it is used to simulate a standard Turing machine. When running on graph G with n nodes and information-theoretic capacity I_G, the simulation has at most $O(I_G)$ space.*

Proof. We can describe a state of the graph Turing machine up to equivalence by specifying (a) some member of a class of equivalent graph labelings (I_G bits); (b) the state of the finite-state controller ($O(1)$ bits); and (c) the position of the finite-state controller ($\log n$ bits). Summing these quantities gives $O(1) + \log n + I_G$ bits, which translates into at most $O(\log n + I_G)$ tape cells for the simulated machine. But now observe that any graph has $I_G = \Omega(\log n)$ (provided the alphabet size is at least 2), since we can obtain at least $n + 1$ distinct automorphism classes by labeling k nodes with one symbol and $n - k$ with another, where k ranges from 0 to n. So $O(\log n + I_G) = O(I_G)$. □

4.2 Effective Capacity

Our intent is that the **effective capacity** of a graph is the size of the largest standard Turing machine tape that can be simulated using the graph. However, we can in principle make this size arbitrarily large for any fixed graph by increasing the size of the alphabet and the number of states in the finite-state controller. To avoid this problem, we define effective capacity only for classes of graphs.

Definition 1. *A class of graphs $\mathcal{G}$ has effective capacity $f(G)$ if there is a function f such that for any standard Turing machine M, there is a graph Turing machine M' where for any G in $\mathcal{G}$, and any vertex v_0 of G, $M'(G, v_0)$ simulates M running on an initially blank tape with $f(G)$ tape cells.*

Note that because we have not specified alphabet sizes, effective capacity is defined only up to constants. Furthermore, any particular construction can only demonstrate a lower bound on effective capacity. For example, we can show that the class of all graphs has effective capacity $\Omega(\log n)$, where n is the number of nodes in the graph. The reason for this is we can use the nodes in the graph as a unary counter, then use a standard construction [14] to simulate a $\log(n)$-space Turing machine. However, this does not exclude the possibility that some subclass of the class of all graphs has higher effective capacity, or that there might be a construction that obtains $\Omega(\log n)$ space on general graphs while doing better on some specific graphs.

An example of a class of graphs with high effective capacity are paths. The essential idea is that we can use a path directly to simulate a standard Turing machine tape, with each node in the path representing one cell in the tape. A minor complication is that a standard Turing machine can tell its left from its right while a graph Turing machine can only do so if the neighbors of the current cell have different labels. But we can handle this by adding an extra field in each node that holds repeating values $\{0, 1, 2\}$ (this is the slope mechanism from [11]), with the left neighbor of a node with value x being the one with $(x - 1) \bmod 3$ and the right being the one with $(x + 1) \bmod 3$. This extra information triples the size of the alphabet, but that is permitted by the definition.

On the other hand, we can't do any better than $\Theta(n)$. It is immediate from Definition 1 and Theorem 1 that no class of graphs has an effective capacity that exceeds the information-theoretic capacity by more than a constant factor.

Definition 1 also does not include a time bound. A natural restriction would be to consider **polynomially-bounded effective capacity**, where the simulation can use at most a polynomial number of steps for each step of the simulated machine. In our constructions, we are more interested in showing possibility rather than specific time bounds, but we will state time bounds when we can.

5 Graph Traversal

A fundamental tool for doing computation with a graph Turing machine is the ability to traverse every node in the graph. In this section, we show how this can be done regardless of the structure of the graph.

We adapt depth-first search to our needs. Depth-first search requires a stack, which we can represent by marking the nodes that are on the stack. Unfortunately, because the graph may contain cycles, a simple mark does not suffice to indicate unambiguously a stack node's parent. If each node were labeled with its distance modulo three from the root, then the correct parent node would be evident from the labels. In order to establish these distance labels, we use depth-first traversal repeatedly on the previously labeled portion of the graph to expand the search by one step at a time from the root, as in breadth-first search.

For the depth-first search method, we assume that every node has the fields color (which may be black, white or gray) and depth (which may be ∞, 0, 1 or 2). The depth field is calculated modulo 3 to keep the size of the state finite; this approach is similar to the "centered slope" technique used in [11]. The variable head refers to the node that is the current position of the graph Turing machine. In the initial configuration in which every node contains the blank symbol, we have $v.\mathsf{depth} = \infty$ and $v.\mathsf{color} = \mathsf{white}$ for every node v. In order to initiate the process of assigning the depth labels, we set head.color to white and head.depth to 0.

Because depth-first search will be used in the process of establishing the correct depth labels, its invariant refers to C, the portion of the graph that has been successfully depth labeled so far. A node v is in C if and only if $v.\mathsf{depth} \neq \infty$. The depth-first search algorithm is a framework to allow the performance of some action at every node of C. The action can occur in pre-order or post-order with respect to the search.

During the depth-first search, a node v is **finished** if v.color = black or v.depth = ∞. A per-node action is **safe** if it respects the invariant concerning the depth field, leaves the head where it found it, and alters the color or depth fields of a node only when that node is finished and remains finished after the alteration.

```
Invariant: letting $C := \{v \mid v.\mathsf{depth} \neq \infty\}$, the induced subgraph on $C$ is
           connected, $\mathsf{root} \in C$, and all nodes $v \in C$ have
           $v.\mathsf{depth} = d(\mathsf{root}, v) \bmod 3$
Precondition: $\mathsf{head} = \mathsf{root}$, and all nodes $v$ with $v.\mathsf{depth} \neq \infty$ have
              $v.\mathsf{color} = \mathsf{white}$
1  done := false;
2  repeat
3      if head.color = white then
4          head.color := gray;
           // perform a safe per-node action (preorder)
5      if head has a neighbor $w$ with $w$.color = white and
       $w$.depth = (head.depth + 1) mod 3 then head := $w$;
6      else
7          head.color := black;
           // perform a safe per-node action (postorder)
8          if head has a neighbor $v$ with $v$.color = gray and
           $v$.depth = (head.depth − 1) mod 3 then head := $v$;
9          else done := true;
10 until done;
11 exchange the roles of white and black;
Postcondition: the precondition holds
```

Algorithm 1. Depth-first traversal of a distance-labeled subgraph

In each phase of the breadth-first algorithm to establish the depth labels we use depth-first search to visit every node v in C and perform the safe per-node operation of changing the depth field of every neighbor w of v with w.depth = ∞ to (v.depth+1) mod 3. If at least one node has its depth field modified, then C has expanded, and the next phase of the breadth-first expansion ensues. Otherwise, the process of assigning depth labels to all nodes accessible from the root is complete.

To establish the correctness of the depth-first search, we observe that at any time, the induced subgraph on the nodes that are white or gray is connected, and the gray nodes are the nodes along a shortest path from the root to the location of the head or a predecessor of it. Because the head never moves to a black node, it is clear that the only way a node fails to have a white or gray neighbor is if it is the root and it is the last node to be finished. This is the safety property; the liveness property is that we continue to make progress. If we continue to go down, eventually there must be a node with no white or gray children, which will be made black. This in turn establishes the correctness of

```
   Precondition: all nodes $v$ have $v$.depth $= \infty$
1  head.color := white;
2  head.depth := 0;
3  repeat
4      $C := \{v \mid v.\text{depth} \neq \infty\}$;
       // $C$ consists of all nodes within distance $k$ of the starting
          node, where $k$ is the number of completed iterations
5      foreach $v \in C$ do
6          foreach $w$ in $v$'s neighbors with $w$.depth $= \infty$ do
7              $w$.color := black;
8              $w$.depth := ($v$.depth + 1) mod 3;
9  until no node $w$ with $w$.depth $= \infty$ is encountered;
```

Algorithm 2. Breadth-first assignment of distance labels for Algorithm 1

the breadth-first depth labeling process: at each phase all the currently labeled nodes are visited, and any unlabeled neighbors of them are properly labeled. All accessible nodes are properly depth labeled if and only if no neighbors of labeled nodes are unlabeled.

If the graph G has n nodes and m edges accessible from the root, the depth labeling process completes in $O(n^2)$ steps. Once all accessible nodes are depth labeled, then the depth-first search process can visit every node in $O(m)$ steps.

6 Variants of the Model

In the full paper, we show that the graph Turing machine model is robust against minor changes, including:

– Expanding the store on the controller to $O(\log n)$ bits. This space is represented in unary across the nodes of the graph, combining the traversal methods of Section 5 with the classic counter-based Turing machine simulation of Minsky [14].
– Replacing the single head with k heads.
– Removing the controller's ability to see adjacent nodes. Instead, the controller feels its way through the graph "blindly," staying put when it attempts to move to a non-existent neighbor.

These are analogous to classic Turing machine results showing that small changes in the definition do not affect what we can compute.

7 Trees

The question of whether the information-theoretic capacity of a general family of graphs is necessarily achievable effectively seems to be tied up with the problem of graph isomorphism. For trees, the isomorphism problem is simpler and can be solved in polynomial time [13]. Moreover, it is possible to place a total order on classes of isomorphic trees, which can in turn be used to drive a counter simulation that extracts the full storage capacity of the graph, albeit at the expense of an exponential slowdown introduced by the embedded counter machine simulation.

The canonization algorithm of [13] runs in **LOGSPACE**, so it is tempting to use the **LOGSPACE** simulation of Section 6 to execute it directly. Unfortunately, the algorithm assumes that the tree is presented on a read-only work tape using unique identifiers for each node. We don't have this in our model. So instead we describe a new mechanism for comparing labeled trees that works despite this restriction, while allowing us to compute the next labeling of a tree in place. Isomorphism and increment can be shown to run in polynomial time.

We consider rooted trees. In the full paper, we show that this does not change the asymptotic storage capacity of the tree.

7.1 Information-Theoretic Capacity of a Tree

Let T be a tree and let $T_1, T_2, \ldots, T_d$ be the subtrees rooted at the children of the root of T. Let f_k map a tree to the number of inequivalent labelings over an alphabet of size $k \geq 1$. Then we have the recurrence

$$f_k(T) = k \prod_i \binom{c_i + f_k(U_i) - 1}{f_k(U_i) - 1}$$

where $U_1, U_2, \ldots, U_\ell$ are the non-isomorphic classes of the T_js, and c_i is the multiplicity of U_i among the T_js. Note that the base case is provided by the tree with one node, whose root has no children.

7.2 Comparing Trees

We define an ordering on (isomorphism classes of) labeled rooted trees by induction. Let T be a tree with root label ℓ and immediate subtrees $T_1, \ldots, T_c$; let T' be a tree with root label ℓ' and immediate subtrees $T'_1, \ldots, T'_d$. If $\ell < \ell'$, then $T < T'$. If $\ell > \ell'$, then $T > T'$. Otherwise, let $C = \{T_1, \ldots, T_c\}$ and $D = \{T'_1, \ldots, T'_d\}$ be multisets. If $C = D$, then $T = T'$, where subtree equality is defined inductively. If $\max(C - D) > \max(D - C)$, then $T > T'$. Otherwise, $\max(D - C) > \max(C - D)$, and $T < T'$. It is straightforward to verify that this relation $\leq$ on labeled rooted trees is reflexive, transitive, and antisymmetric up to isomorphism.

In the full paper, we present an algorithm that computes this total ordering. Here, we give only a brief overview of the algorithm. The algorithm assumes a graph Turing machine with two heads, one on tree T and one on tree T'. Each node has a "removed" bit, which is initially unset. The main invariant is that removing all subtrees whose removed bits are set does not affect the order.

The algorithm begins by comparing the root labels. Assuming that they are equal, for each of the immediate subtrees T_i of T, we attempt to find a immediate subtree of T' isomorphic to T_i. If there is such a subtree T'_j, we set the removed bit for both T_i and T'_j; these subtrees offset one another and will not be considered again. If no match can be made, then we set the removed bit for each T'_j that was determined to be less than T_i, as these subtrees cannot be the maximum in the symmetric difference of the two multisets of subtrees. At the end, if all immediate subtrees have been removed, then $T = T'$. Otherwise, if all immediate subtrees of T' have been removed, then $T > T'$, else $T < T'$.

7.3 Implementing Counters with Labeled Trees

In this section we simulate counter machines with operations of clear, increment, and compare counters for equality, which can implement a standard Turing machine, albeit with exponential slowdown [14].

Given a tree T whose nodes can be labeled $0, \dots, k-1$, we represent counter values $0, \dots, f_k(T) - 1$ by the labelings they index in the order defined above. Zero corresponds to the all-zeros labeling, so clearing a register can be accomplished with one traversal. Two counters whose underlying unlabeled trees are isomorphic can be compared using the isomorphism algorithm. The increment operation requires a new algorithm.

We assume that each node of the tree has space to save its old label, and the increment routine will save the previous counter value. To increment a subtree T with overflow, first save the root label and then increment its immediate subtrees (saving their respective values). If every subtree overflowed (i.e., is zero), then increment the root label mod k and overflow if it becomes zero. Otherwise, use the isomorphism checker to find and mark the minimum nonzero immediate subtree M. Restore every tree other than M to its original value and then zero those that are less than M. The call stack for this recursive algorithm is represented using a state label at each node of the tree.

We prove the correctness of the increment algorithm by induction. Suppose we are incrementing a labeled tree T with immediate subtrees $T_1 \geq \ldots \geq T_c$. Let T' be the resulting tree. In case $T_1, \dots, T_c$ are already at their respective maximums, it is straightforward to verify that T' is the successor of T. Otherwise, let U be any relabeling of T such that $U > T$. We show that $T < T' \leq U$ and thus that T' is the successor of T.

Let $T'_1 \geq \cdots \geq T'_c$ be the immediate subtrees of T' and let $U_1 \geq \cdots \geq U_c$ be the immediate subtrees of U. Note that, on account of the labelings, T'_i (respectively U_i) may not correspond to T_i. Given that some T_i is not maximum, then the root labels of T and T' are identical, and the algorithm is able to find a minimum incremented tree T'_j, where we choose j to be as large as possible in case of ties. We have $T_i = T'_i$ if $i < j$, and $T_j < T'_j$ (count the number of trees greater than or equal to T'_j). For all $i > j$, the tree T'_i is zero. If the root label of U is not equal to the root label of T then $T' < U$. Otherwise, let ℓ be the least index for which $T_\ell < U_\ell$. For all $i < \ell$, we have $U_i = T_i$. If $\ell < j$, then $U > T'$. Otherwise, U_ℓ has the same shape (disregarding labels) as some tree $T_i < T'_j$. Since T'_j was the minimum increment, it follows that $T'_j \leq U_\ell$ and thus that $T' \leq U$.

8 Capacity of a Bounded-Degree Graph

For a bounded-degree graph, we can use the mechanism in [3] (which itself derives much of its structure from the previous construction in [11]) with only a few small modifications.

In a graph with degree bound Δ, it is possible to assign each node a label in $\{1, \ldots, \Delta^2 + 1\}$ so that each node's label is unique within a ball of radius 2. This is a **distance two labeling**, and it gives each node the ability to identify its neighbors uniquely. Angluin *et al.*. [3] construct a distance two labeling non-deterministically, by having each node adopt a new label if it detects a second-order neighbor with the same label. In our model, we can construct the labeling deterministically, by iteratively assigning each node a label that does not conflict with a second-order neighbor (it is easy to see that each time we do this, we cannot create any new conflicts, so we converge after $O(n)$ iterations to a correct labeling).

Using such a labeling, it is straightforward to adapt the traversal routine to build a spanning tree, which in turn can simulate a Turing machine tape to provide $\Theta(n)$ bits of effective capacity.

9 Random Graphs

Suppose we consider random graphs G drawn from $G(n, p)$, i.e., a graph on n nodes where each edge uv appears with probability p.[1] Suppose further that p scales as $\Theta(n^{-c})$ for some fixed $0 < c < 1$. Then it is possible to achieve an effective capacity of $\Theta(n)$ with high probability[2] from graphs in this class. Note that graphs in this class are connected with high probability.

The basic idea is that if we can compute a total order on nodes, we can use each node to hold one Turing machine cell, with left and right movements corresponding to moving down or up in the ordering. We compute this ordering by assigning a signature to each node, based on random values stored in its neighborhood. For simplicity, we assume that the Turing machine simulator can generate random values. However, we suspect that a more sophisticated application of the same basic approach could work using only the randomness inherent in the graph.

Details are given in the full paper. The key step in the proof is to show that, if every node in $G(n, p)$ with $p = n^{-c}$ is labeled with a random bit, then the probability that two nodes u and v have the same number of neighbors with 1 bits is $O(n^{(c-1)/2}$, which can be reduced below $O(n^{-3})$ by repeating the construction $k = O(1)$ times. The resulting $k = O(1)$ neighborhood counts then give a unique signature for almost all nodes with high probability, which can be computed and compared easily by a **LOGSPACE** controller.

[1] See [7,12] for an overview of random graphs.

[2] We use **with high probability** to mean that the probability that the event does not occur is $O(n^{-c})$ for any fixed c.

The full result is:

Theorem 2. *A member of the family of random graphs $G(n,p)$ where $p = \Theta(n^{-c})$ for any fixed $0 < c < 1$ has effective capacity $\Theta(n)$ with high probability.*

10 Conclusion

We have defined a new class of graph-based Turing machines, motivated by potential applications in self-organizing systems of finite-state automata. We have shown that this class is robust under natural changes to the model, and that its power is primarily characterized by the effective capacity of the underlying graph, which is the amount of usable storage obtained by writing symbols from a finite alphabet on its nodes. This is at least $\Omega(\log n)$ bits of space for an arbitrary n-node graph, and rises to $\Theta(n)$ bits for bounded-degree graphs and almost all random graphs with polynomial edge probabilities. For trees, the effective capacity ranges from $\Theta(\log n)$ for trees with many symmetries (stars) to $\Theta(n)$ for trees with few (binary trees, paths). In intermediate cases we have shown that we can always get within a constant factor of the full information-theoretic capacity corresponding to the number of non-isomorphic states, although the time complexity of our algorithm could be significantly improved.

The main open problem remaining is whether it is possible to extract the full information-theoretic capacity from an arbitrary graph. This seems closely tied to the problem of computing graph isomorphism, which is not known to be hard, even for **LOGSPACE**. The reason is that distinguishing two different labelings of a graph appears to depend on being able to distinguish between non-isomorphic subgraphs (since this gives a weak form of orientation to the graph). However, the problem is not exactly the same, because we have the ability to supplemental isomorphism testing by using some of our labels as signposts and we do not need a perfect isomorphism tester as long as we can group subgraphs into small equivalence classes. So it may be that extracting the full capacity of an arbitrary graph is possible without solving graph isomorphism in general.

Acknowledgments

The authors would like to thank Yinghua Wu for many useful discussions during the early stages of this work.

References

1. Albertson, M.O.: Distinguishing Cartesian powers of graphs. Electronic Journal of Combinatorics 12, N17 (2005)
2. Albertson, M.O., Collins, K.L.: Symmetry breaking in graphs. Electronic Journal of Combinatorics 3(1), R18 (1996)

3. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In: Proc. 1st IEEE International Conference on Distributed Computing in Sensor Systems: pp. 63–74 (2005)
4. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Urn automata. Technical Report YALEU/DCS/TR-1280, Yale University Department of Computer Science (November 2003)
5. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing 18(4), 235–253 (2006)
6. Aspnes, J., Ruppert, E.: An introduction to population protocols. In: Garbinato, B., Miranda, H., Rodrigues, L. (eds.) Middleware for Network Eccentric and Mobile Applications, pp. 97–120. Springer, Heidelberg (2009)
7. Bollobás, B.: Random Graphs, 2nd edn. Cambridge University Press, Cambridge (2001)
8. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
9. Guerraoui, R., Ruppert, E.: Even small birds are unique: Population protocols with identifiers. Technical Report CSE-2007-04, Department of Computer Science and Engineering, York University (2007)
10. Guerraoui, R., Ruppert, E.: Names trump malice: Tiny mobile agents can tolerate byzantine failures. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 484–495. Springer, Heidelberg (2009)
11. Itkis, G., Levin, L.A.: Fast and lean self-stabilizing asynchronous protocols. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 226–239 (1994)
12. Janson, S., Łuczak, T., Ruciński, A.: Random Graphs. John Wiley & Sons, Chichester (2000)
13. Lindell, S.: A logspace algorithm for tree canonization (extended abstract). In: STOC, pp. 400–404. ACM, New York (1992)
14. Minsky, M.L.: Computation: Finite and infinite machines. Prentice-Hall series in automatic computation. Prentice-Hall, Inc., Englewood Cliffs (1967)
15. Russell, A., Sundaram, R.: A note on the asymptotics and computational complexity of graph distinguishability. Electronic Journal of Combinatorics 5(1), R23 (1998)
16. Schönhage, A.: Storage modification machines. SIAM J. Comput. 9(3), 490–508 (1980)

Safe Flocking in Spite of Actuator Faults

Taylor Johnson and Sayan Mitra

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Abstract. The safe flocking problem requires a collection of N mobile agents to (a) converge to and maintain an equi-spaced lattice formation, (b) arrive at a destination, and (c) always maintain a minimum safe separation. Safe flocking in Euclidean spaces is a well-studied and difficult coordination problem. Motivated by real-world deployment of multi-agent systems, this paper studies one-dimensional safe flocking, where agents are afflicted by *actuator faults*. An actuator fault is a new type of failure that causes an affected agent to be stuck moving with an arbitrary velocity. In this setting, first, a self-stabilizing solution for the problem is presented. This relies on a failure detector for actuator faults. Next, it is shown that certain actuator faults cannot be detected, while others may require $O(N)$ time for detection. Finally, a simple failure detector that achieves the latter bound is presented. Several simulation results are presented for illustrating the effects of failures on the progress towards flocking.

Keywords: failure detector, flocking, safety, stabilization, swarming.

1 Introduction

Safe flocking is a distributed coordination problem that requires a collection of mobile agents situated in a Euclidean space to satisfy three properties, namely to: (a) form and maintain an equi-spaced lattice structure or a *flock*, (b) reach a specified destination or *goal* position, and (c) always maintain a minimum *safe* separation. The origins of this problem can be traced to biological studies aimed at understanding the rules that govern flocking in nature (see [13,11], for example). More recently, recognizing that such understanding could aid the design of autonomous robotic platoons or swarms, the problem as stated above and its variants have been studied in the robotics, control, and multi-agent systems literature (see [7,5,12,8,14] and references therein). Typically, the problem is studied for agents with synchronous communication, without failures, and with double-integrator dynamics—that is, the distributed algorithm sets the acceleration for each agent. To the best of our knowledge, even in this setting, safe-flocking is an open problem, as existing algorithms require unbounded accelerations for guaranteeing safety [12], which cannot be achieved in practice.

In this paper, we study one-dimensional safe-flocking within the realm of synchronous communication, but with a different set of dynamics and failure assumptions. First, we assume rectangular single-integrator dynamics. That is, at the beginning of each round, the algorithm decides a target point u_i for agent i based on messages received from i's neighbors, and agent i moves with bounded speed $\dot{x_i} \in [v_{min}, v_{max}]$ in the direction of u_i for the duration of that round. This simplifies the dynamics and achieving safety becomes relatively easy. Even in this setting however, it is nontrivial to develop and prove that an algorithm provides collision avoidance, as illustrated by an error—a forgotten case for the special dynamics of the rightmost (N^{th}) agent—that we found in the inductive proof of safety in [7]. To fix the error, the algorithm from [7] requires the modification presented later in this paper (Figure 3, Line 31). The model obtained with rectangular dynamics overapproximates any behavior that can be obtained with double integrator dynamics with bounded acceleration. Our algorithm combines the corrected algorithm from [7] with Chandy-Lamport's global snapshot algorithm [3]. The key idea is that each agent periodically computes its target based on messages received from its neighbors, then moves toward this target with some arbitrary but bounded velocity. The targets are computed such that the agents preserve safe separation and eventually form a *weak flock*, which remains invariant, and progress is ensured to a tighter *strong flock*. Once a strong flock is attained, this property can be detected through the use of a distributed snapshot algorithm [3]. Once this is detected, the detecting agent moves toward the destination, sacrificing the strong flock in favor of making progress toward the goal, but still preserving the weak flock.

Unlike the algorithms in [7,5,8,12] that provide convergence to a flock, we require the stronger *termination*. Our algorithm achieves termination through *quantization*: we assume that there exists a constant $\beta > 0$ such that an agent i moves in a particular round if and only if the computed target u_i is more than β away from the current position x_i. We believe that such quantized control is appropriate for realistic actuators, and useful for most power-constrained settings where it is undesirable for the agents to move forever in order to achieve convergence. Quantization affects the type of flock formation that we can achieve and also makes the proof of termination more interesting.

We allow agents to be affected by *actuator faults*. This physically corresponds to, for example, an agent's motors being stuck at an input voltage or a control surface becoming immobile. Actuator faults are permanent and cause the afflicted agents to move forever with a bounded and constant velocity. Actuator faults are a new class of failures that we believe are going to be important in designing and analyzing a wide range of distributed cyber-physical systems [9]. Unlike byzantine faults, behaviors resulting from actuator faults are constrained by physical laws. Also, unlike crash failures which typically thwart progress but not safety, actuator faults can also violate safety. A faulty agent has to be detected (and possibly avoided) by the non-faulty agents. In this paper, we assume that after an actuator fault, an agent continues to communicate and compute, but its actuators continue to move with the arbitrary but constant velocity.

Some attention has been given to failure detection in flocking such as [6], which works with a similar model of actuator faults. While [6] uses the therein developed *motion probes* in failure detection scenarios, no bounds are stated on detection time. Instead, convergence was ensured assuming that failure detection had occurred within some bounded time, while our work states an $O(N)$ detection time bound.

Our flocking algorithm determines *only* the direction in which an agent should move, based on the positions of adjacent agents. The speed with which an agent moves is chosen nondeterministically over a range, making the algorithm implementation independent with respect to the lower-level motion controller. Thus, the intuition behind failure detection is to observe that an agent has moved in the wrong direction. Under some assumptions about the system parameters, a simple lower-bound is established, indicating that no detection algorithm can detect failures in less than $O(N)$ rounds, where N is the number of agents. A failure detector is presented that utilizes this idea in detecting certain classes of failures in $O(N)$ rounds. Unfortunately, certain failures lead to a violation of safety in fewer rounds, so a failure detector which detects failures faster than $O(N)$ rounds is necessary to ensure safety. However, some failures are undetectable, such as an agent failing with zero velocity at the goal, and thus we establish that no such failure detector exists. But, under a restricted class of actuator faults, it is shown that the failure detector with $O(N)$ detection time can be combined with the flocking algorithm to guarantee the required safety and progress properties. This requires non-faulty agents to be able to avoid faulty ones. In one dimension (such as on highways), this is possible if there are multiple *lanes*.

In summary, the key contributions of the paper are the following:

(a) Formal introduction of the notion of actuator faults and stabilization in the face of such faults.
(b) A solution to the one-dimensional safe flocking problem in the face of actuator faults, quantization, and with bounded control. Our solution brings distributed computing ideas (self-stabilization and failure detection) to a distributed control problem.

2 System Model

This section presents a formal model of the distributed flocking algorithm modeled as a discrete transition system, as well as formal specifications of the system properties to be analyzed. For $K \in \mathbb{N}$, $[K] \triangleq \{1, \ldots, K\}$ and for a set S $S_\perp \triangleq S \cup \{\perp\}$. A *discrete transition system* $\mathcal{A}$ is a tuple $\langle X, Q, Q_0, A, \rightarrow\rangle$, where (i) X is a set of *variables* with associated types, (ii) Q is the set of *states*, which is the set of all possible valuations of the variables in X, (iii) $Q_0 \subseteq Q$ is the set of *start states*, (iv) A is a set of transition *labels*, and (v) $\rightarrow \subseteq Q \times A \times Q$ is a set of *discrete transitions*. An *execution fragment* of A is an (possibly infinite) alternating sequence of states and transition names, $\alpha = \mathbf{x}_0, a_1, \mathbf{x}_1, \ldots$, such that for each index k appearing in α, $(\mathbf{x}_k, a_{k+1}, \mathbf{x}_{k+1}) \in \rightarrow$. An *execution* is an execution fragment with $\mathbf{x}_0 \in Q_0$.

A state $\mathbf{x}$ is *reachable* if there exists a finite execution that ends in $\mathbf{x}$. A *stable* predicate $S \subseteq Q$ is a set of states closed under $\rightarrow$. If a stable predicate S contains Q_0, then it is called an *invariant* predicate and the reachable states of $\mathcal{A}$ are contained in S. A *safety* property specified by a predicate $S \subseteq Q$ is satisfied by $\mathcal{A}$ if all of its reachable states are contained in S. Self-stabilization is a property of non-masking fault tolerance which guarantees that once new failures cease to occur, the system eventually returns to a legal state [4]. In this paper, we model actuator faults by transitions with the special label fail. Given $G \subseteq Q$, $\mathcal{A}$ *self-stabilizes* to G if (a) G is a stable predicate for $\mathcal{A}$ along execution fragments without fail-transitions, and (b) from every reachable state of $\mathcal{A}$ (including states reached via fail transitions), every *fail*-free execution fragment eventually reaches G.

2.1 Model of Safe Flocking System

The distributed system consists of a set of N mobile *agents* physically positioned on N_L infinite, parallel *lanes*. The system can be thought of as a collection of cars in the lanes on a highway. Refer to Figure 1 for clarity, and Figure 2 shows the system making progress (reaching the origin as a flock), without indicating lanes, but note that agents 1 through 5 move to lane 2 around round 375 to avoid the failed agent 6. We assume synchrony and the communication graph is complete, regardless of lanes[1]. That is, agents have synchronized clocks, message delays are bounded, and computations are instantaneous. At each round, each agent exchanges messages bearing state information with everyone, and note that this means agents in different lanes communicate. Agents then update their software state and (nondeterministically) choose their velocities, which they operate with until the beginning of the next round. Under these assumptions, it is convenient to model the system as a collection of discrete transition systems that interact through shared variables. Let $ID \triangleq [N]$ be the set of unique agent identifiers and $LD \triangleq [N_L]$ be the set of lane identifiers. The following positive constants are used throughout the paper: (a) r_s: minimum required inter-agent gap or *safety distance* in the absence of failures, (b) r_r: reduced safety distance in the presence of failures, (c) r_f: desired maximum inter-agent gap which defines a flock, (d) δ: flocking tolerance parameter—that is, the maximum deviation from r_f agents may be spaced and constitute a *flock*, (e) β: quantization parameter, used to prevent agents from moving if the algorithm decides too small a movement so that eventually the algorithm terminates, and (f) v_{min}, v_{max}: minimum and maximum velocities.

State Variables. The discrete transition system corresponding to $\mathsf{Agent_i}$ has the following *private* variables with the *type followed by initial value in parentheses*: (a) *gsf* (Boolean; *false*): indicates whether the stable predicate detected by the global snapshot is satisfied or not, (b) *sr* (Boolean; *false*): indicates whether the global snapshot algorithm has been initiated, (c) *failed* (Boolean; *false*): indicates whether or not agent i has failed, (d) vf ($\mathbb{R}$; $\bot$): velocity with which agent i has failed, and (e) L and R ($ID_\bot$): identifiers of the nearest left and right agents to agent i. The following *shared* variables are controlled by agent i, but can also be read by others (*type followed by initial value in parentheses*): (a) x and xo ($\mathbb{R}$): current position and position from the previous round of agent i, (b) u and uo ($\mathbb{R}$; x and xo): target position and target position from the previous round of agent i, (c) *lane* (LD; 1): the lane currently occupied by agent i, (d) *Suspected* ($ID_\bot$; $\emptyset$): set of neighbors that agent i believes to have failed. These variables are *shared* in the following sense: At the beginning of each round k, their values are broadcast by $\mathsf{Agent_i}$ and are used by other agents to update their states in that round. The discrete transition system modeling the complete ensemble of agents is called System. We refer to states of System with bold letters $\mathbf{x}$, $\mathbf{x}'$, etc., and $\mathsf{Agent_i}$'s individual state components by $\mathbf{x}.x_i$, $\mathbf{x}.u_i$, etc.

Actuator Faults and Failure Detection. The failure of agent i's actuators is modeled by the occurrence of a transition labeled by fail_i. This transition is always enabled unless

[1] This communication assumption is relaxed to nearby neighbors being able to communicate synchronously in [9].

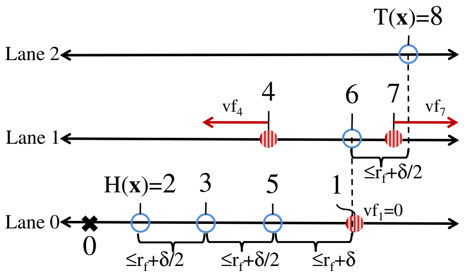


Fig. 1. System at state $\mathbf{x}$ for $N = 8$, $\bar{F}(\mathbf{x}) = \{2, 3, 5, 6, 8\}$, $F(\mathbf{x}) = \{1, 4, 7\}$. Failed actuator velocities are labeled vf_i and eventually 4 and 7 will diverge. Non-faulty agents have avoided failed agents by changing lanes. Note that $L(\mathbf{x}, 6) = 5$. Also, if $4 \in Suspected_6$, then $L_S(\mathbf{x}, 6) = L(6) = 5$, else $L_S(\mathbf{x}, 6) = 4$. Assuming $S(\mathbf{x}) = F(\mathbf{x})$, $Flock_W(\mathbf{x})$, but $\neg Flock_S(\mathbf{x})$, since $|\mathbf{x}.x_6 - \mathbf{x}.x_5 - r_f| \leq \delta$ (and not $\delta/2$).

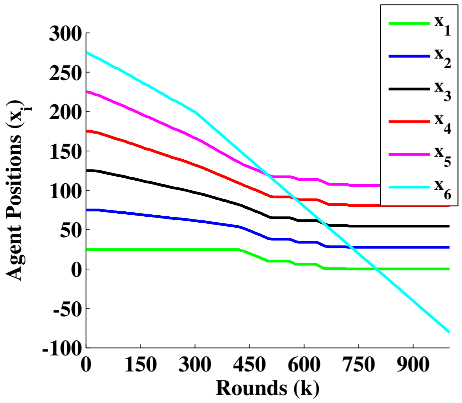


Fig. 2. System progressing: eventually the agents have formed a flock and the failed agent 6 with nonzero velocity has diverged

i has already failed, and as a result of its occurrence, the variable $failed_i$ is set to $true$. An actuator fault causes the affected agent to move forever with a constant but arbitrary *failure velocity*. At state $\mathbf{x}$, $F(\mathbf{x})$ and $\bar{F}(\mathbf{x})$ denote the sets of faulty and non-faulty agent identifiers, respectively.

Agents do not have any direct information regarding the failure of other agents' actuators (i.e., agent i cannot read $failed_j$). Agents rely on timely failure detection to avoid violating safety or drifting away from the goal by following a faulty agent. Failure detection at agent i is abstractly captured by the $Suspected_i$ variable and a transition labeled by $\mathsf{suspect}_i$. The $\mathsf{suspect}_i(j)$ transition models a detection of failure of some agent j by agent i. Failures are irreversible in our model, and thus so are failure detector suspicions. For agent i, at any given state $Suspected_i \subseteq ID$ is the set of agent identifiers that agent i's failure detector suspects as faulty. Agent_j is said to be *suspected* if some agent i suspects it, otherwise it is *unsuspected*. Which particular agent suspects a faulty agent j is somewhat irrelevant. We assume the failure detectors of all agents share information through some background gossip, and when one agent suspects agent i, all other agents also suspect i in the same round[2]. Denote the sets of suspected and unsuspected agents by $S(\mathbf{x})$ and $\bar{S}(\mathbf{x})$, respectively.

The *detection time* is the minimum number of rounds within which every failure is always suspected. In most parts of Section 3 we will assume that there exists a finite detection time k_d for any failure. In Section 3.3, we will discuss specific conditions

[2] This assumption is relaxed to adjacent agents in [9].

under which k_d is in fact finite and then give upper and lower bounds for it. The failure detection strategy used by our flocking algorithm is encoded as the precondition of the suspect transition. Note that the precondition assumes that i has access to some of j's shared variables, namely x_j, xo_j, u_j and uo_j. When the precondition of suspect(j) is satisfied at Figure 3, j is added to $Suspected_i$. This precondition checks that either j moved when it should not have, or that j moved in the wrong direction, away from its computed target. The rationale behind this condition will become clear as we discuss the flocking algorithm.

```
fail_i(v), |v| ≤ v_max
pre ¬ failed
eff failed := true; vf := v

suspect_i(j), j ∈ ID
pre j ∉ Suspected ∧ (if |xo_j − uo_j| ≥ β
  then sgn (x_j − xo_j) ≠ sgn (uo_j − xo_j)
  else |x_j − uo_j| ≠ 0)
eff Suspected := Suspected ∪ {j}

snapStart_i
pre L = ⊥ ∧ ¬sr
eff sr := true // global snapshot invoked

snapEnd_i(GS), GS ∈ {false, true}
eff gsf := GS; // global snapshot returns
  sr := false

update_i
eff uo := u; xo := x
  for each j ∈ ID, Suspected := Suspected ∪ Suspected_j
Mitigate:
  if ¬ failed ∧ (∃ s ∈ Suspected : lane_s = lane)
    ∧ (∃ 𝓛 ∈ LD : ∀ j ∈ ID, (lane_j = 𝓛 ⇒
    x_j ∉ [x − r_s − 2v_max , x + r_s + 2v_max ]))
    then lane := 𝓛 fi
Target:
  if L = ⊥ ∧ gsf then u := x − min{x, δ/2};
    gsf := false
  elseif L = ⊥ then u := x
  elseif R = ⊥ then u := (x_L + x + r_f)/2
  else u := (x_L + x_R)/2 fi
Quant: if |u − x| < β then u := x fi
Move: if failed then x := x + vf
  else x := x + sgn (x − u) choose [v_min , v_max] fi
```

Fig. 3. Agent$_i$'s transitions: failure detection, global snapshots, and target updates

Neighbors. At state $\mathbf{x}$, let $L(\mathbf{x}, i)$ (and symmetrically $R(\mathbf{x}, i)$) be the nearest non-failed agent left (resp. right) of Agent$_i$, with ties broken arbitrarily. If no such agent exists, then $L(\mathbf{x}, i)$ and $R(\mathbf{x}, i)$ are defined as $\perp$. Let $L_S(\mathbf{x}, i)$ (and symmetrically $R_S(\mathbf{x}, i)$) be the nearest unsuspected agent left (resp. right) of Agent$_i$ at state $\mathbf{x}$, or $\perp$ if no such agents exist. An unsuspected Agent$_i$ with both unsuspected left and right neighbors is a *middle agent*. An unsuspected Agent$_i$ without an unsuspected left neighbor is the *head agent*, and is denoted by the singleton $H(\mathbf{x})$. If Agent$_i$ is unsuspected, is not the head, and does not have an unsuspected right neighbor, it is the *tail agent* and is denoted by the singleton $T(\mathbf{x})$.

Flocking Algorithm. The distributed flocking algorithm executed at Agent$_i$ uses two separate processes (threads): (a) a process for taking distributed global snapshots, and (b) a process for updating the target position for Agent$_i$.

The snapStart and snapEnd transitions model the periodic initialization and termination of a distributed global snapshot protocol—such as Chandy and Lamport's snapshot algorithm [3]—by the head agent. This global snapshot is used for detecting a stable global predicate, which in turn influences the target computation for the head agent. Although we have not modeled this explicitly, we assume that the snapStart$_i$ transition is performed periodically by the head agent when the precondition is enabled. If the global predicate holds, then snapEnd$(true)$ occurs, otherwise snapEnd$(false)$ occurs. Chandy-Lamport's algorithm can be applied since (a) we are detecting a stable predicate, (b) the communications graph is complete, and (c) the stable predicate being detected is reachable. Thus, we assume that in any infinite execution, a $\mathsf{snapEnd}_i$ transition occurs within $O(N)$ rounds from the occurrence of the corresponding $\mathsf{snapStart}_i$ transition.

The update transition models the evolution of all (faulty and non-faulty) agents over a synchronous round. It is composed of four subroutines: *Mitigate*, *Target*, *Quant*, and *Move*, which are executed in this sequence for updating the state of System. The entire update is instantaneous and atomic; the subroutines are used for clarity of presentation. To be clear, for $\mathbf{x} \overset{\mathsf{update}}{\rightarrow} \mathbf{x}'$, $\mathbf{x}'$ is obtained by applying each of these subroutines. We refer to the intermediate states after *Mitigate*, *Target*, *Quant*, and *Move* as $\mathbf{x}_M$, $\mathbf{x}_T$, $\mathbf{x}_Q$, and $\mathbf{x}_V$, respectively. That is, $\mathbf{x}_M \triangleq Mitigate(\mathbf{x})$, $\mathbf{x}_T \triangleq Target(\mathbf{x}_M)$, etc., and note $\mathbf{x}' = \mathbf{x}_V = Move(\mathbf{x}_Q)$.

Mitigate is executed by non-faulty agents and may cause them to change lanes, thus restoring safety and progress properties that may be reduced or violated by failures. *Target* determines a new target to move toward. There are three different rules for target computations based on an agent's belief of whether it is a head, middle, or tail agent. For a state $\mathbf{x}$, each middle agent i attempts to maintain the average of the positions of its nearest unsuspected left and right neighbors (Figure 3, Line 32). Assuming that the goal is to the left of the tail agent, the tail agent attempts to maintain r_f distance from its nearest unsuspected left neighbor (Figure 3, Line 31). The head agent periodically invokes a global snapshot and attempts to detect a certain stable global predicate $Flock_S$ (defined below). If this predicate is detected, then the head agent moves towards the goal (Figure 3, Line 29), otherwise it does not change its target u from its current position x. As mentioned before, targets are still computed for faulty agents, but their actuators ignore these new values. *Quant* is the quantization step which prevents targets u_i computed in the *Target* subroutine from being applied to real positions x_i, if the difference between the two is smaller than the *quantization parameter* β. It is worth emphasizing that quantization is a key requirement for any realistic algorithm that actuates the agents to move with bounded velocities. Without quantization, if the computed target is very close to the current position of the agent, then the agent may have to move with arbitrarily small velocity over that round. Finally, *Move* moves agent positions x_i toward the quantized targets. Note that *Move* abstractly captures the physical evolution of the system over a round; that is, it is the time-abstract transition corresponding to physical evolution over an interval of time.

2.2 Key Predicates

We now define a set of predicates on the state space of System that capture the key properties of safe flocking. These will be used for proving that the algorithm described above solves safe flocking in the presence of actuator faults. We start with safety. A state $\mathbf{x}$ of System satisfies *Safety* if the distance between every pair of agents on the same lane is at least the safety distance r_s. Formally, $Safety(\mathbf{x}) \triangleq \forall i, j \in ID, i \neq j, \mathbf{x}.lane_i = \mathbf{x}.lane_j \implies |\mathbf{x}.x_i - \mathbf{x}.x_j| \geq r_s$. When failures occur, a reduced inter-agent gap of r_r will be guaranteed. We call this weaker property *reduced safety*: $Safety_R(\mathbf{x}) \triangleq \forall i \in \bar{F}(\mathbf{x}), \forall j \in ID, i \neq j, \mathbf{x}.lane_i = \mathbf{x}.lane_j \implies |\mathbf{x}.x_i - \mathbf{x}.x_j| \geq r_r$.

An ϵ-flock is where each non-faulty agent with an unsuspected left neighbor (not necessarily in the same lane) is within $r_f \pm \epsilon$ from that neighbor. Formally, $\mathit{Flock}(\mathbf{x}, \epsilon) \triangleq \forall i \in \bar{S}(\mathbf{x}), L_S(\mathbf{x}, i) \neq \bot, \left|\mathbf{x}.x_i - \mathbf{x}.x_{L_S(\mathbf{x},i)} - r_f\right| \leq \epsilon$. In this paper, we will use the *Flock* predicate with two specific values of ϵ, namely δ (the flocking tolerance parameter) and $\frac{\delta}{2}$. The *weak flock* and the *strong flock* predicates are defined as $\mathit{Flock}_W(\mathbf{x}) \triangleq \mathit{Flock}(\mathbf{x}, \delta)$, and $\mathit{Flock}_S(\mathbf{x}) \triangleq \mathit{Flock}(\mathbf{x}, \frac{\delta}{2})$, respectively.

Related to quantization, we have the *no big moves (NBM)* predicate, where none of the agents (except possibly the head agent) have any valid moves, because their computed targets are less than β (quantization constant) away from their current positions. $\mathit{NBM}(\mathbf{x}) \triangleq \forall i \in \bar{F}(\mathbf{x}), L_S(\mathbf{x}, i) \neq \bot, |\mathbf{x}_T.u_i - \mathbf{x}.x_i| \leq \beta$, where $\mathbf{x}_T$ is the state following the application of *Target* subroutine to $\mathbf{x}$. The *Goal* predicate is satisfied at states where the head agent is within β distance of the goal (assumed to be the origin without loss of generality), that is, $\mathit{Goal}(\mathbf{x}) \triangleq \mathbf{x}.x_{H(\mathbf{x})} \in [0, \beta)$. Finally, a state satisfies the *Terminal* predicate if it satisfies both *Goal* and *NBM*.

3 Analysis

The main result of the paper (Theorem 1) is that the algorithm in Figure 3 achieves safe flocking in spite of failures provided: (a) there exists a failure detector that detects actuator faults *sufficiently fast*, and (b) each non-faulty agent has *enough room* to jump to some lane to safely avoid faulty agents and eventually make progress. For the first part of our analysis, we will simply assume that any failure is detected within k_d rounds. In Section 3.3, we shall examine conditions under which k_d is finite and state its lower and upper bounds. Assumption (b) is trivially satisfied if the number of lanes is greater than the total number of failures; but it is also satisfied with fewer lanes, provided the failures are sufficiently apart in space. There are two space requirements for Assumption (b): the first ensures safety and the second ensure progress by preventing "walls" of faulty agents from existing forever and ensuring that infinitely often all non-faulty agents may make progress.

Theorem 1. *Suppose there exists a failure detector which suspects any actuator fault within k_d rounds. Suppose further that $v_{max} \leq (r_s - r_r)/(2k_d)$. Let $\alpha = \mathbf{x}_0, \ldots, \mathbf{x}_p, \mathbf{x}_{p+1}$ be an execution where x_p is the state after the last* fail *transition. Let $\alpha_{ff} = \mathbf{x}_{p+1}, \ldots,$ be the fail-free suffix of α. Let f be the number of actuator faults. Suppose either (a) $N_L > f$,* or *(b) $N_L \leq f$ and along α_{ff}, $\forall \mathbf{x} \in \alpha_{ff}$, $\exists \mathcal{L} \in LD$ such that $\forall i \in \bar{F}(\mathbf{x})$, $\forall j \in F(\mathbf{x})$, $\mathbf{x}.lane_j \neq \mathcal{L}$ and $|\mathbf{x}.x_i - \mathbf{x}.x_j| > r_s + 2v_{max}k_d$, and also that infinitely often, $\forall m, n \in F(\mathbf{x})$, $m \neq n$, $|\mathbf{x}.x_m - \mathbf{x}.x_n| > r_s + 2v_{max}$. Then, (a) Every state in α satisfies the reduced safety property, Safety_R, and (b) Eventually Terminal and Flock_S are satisfied.*

In what follows, we state and informally discuss a sequence of lemmas that culminate in Theorem 1. Under the assumptions and analysis of this section, the following relationships are satisfied: $\mathit{NBM} \subset \mathit{Flock}_S \subset \mathit{Flock}_W \subset \mathit{Safety} \subset \mathit{Safety}_R$. Detailed proofs of the lemmas appear in the technical report [10]. We begin with some assumptions.

Assumptions. Except where noted in Section 3.3, the remainder of the paper utilizes the assumptions of Theorem 1. Additionally, these assumptions are required throughout the paper: (a) $N_L \geq 2$: there are at least 2 lanes, (b) $r_r < r_s < r_f$: the reduced safety gap r_r required under failures is strictly less than the safety gap r_s in the absence of failures, which in turn is strictly less than the flocking distance, (c) $0 < v_{min} \leq v_{max} \leq \beta \leq \delta/(4N)$, and (d) the communication graph of the non-faulty agents is always fully connected, so the graph of non-faulty agents cannot partition. Assumption (c) bounds the minimum and maximum velocities, although they may be equal. It then upper bounds the maximum velocity to be less than or equal to the quantization parameter β. This is necessary to prevent a violation of safety due to overshooting computed targets. Finally, β is upper bounded such that $NBM \subseteq Flock_S$. Intuitively, the bound on β is to ensure that errors from flocking due to quantization do not accumulate along the flock from the head to the tail. This is used to show that eventually $Flock_S$ is satisfied by showing eventually NBM is reached.

3.1 Safety

First, we establish that System satisfies the safety part of the safe flocking problem. The following lemma states that in each round, each agent moves by at most v_{max}, and follows immediately from the specification of System.

Lemma 1. *For any two states* $\mathbf{x}, \mathbf{x}'$ *of* System*, if* $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ *for some transition* a*, then for each agent* $i \in ID$, $|\mathbf{x}'.x_i - \mathbf{x}.x_i| \leq v_{max}$.

The next lemma establishes that, upon changes in which other agents an agent i uses to compute its target position, safety is not violated.

Lemma 2. *For any execution* α*, for states* $\mathbf{x}, \mathbf{x}' \in \alpha$ *such that* $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ *for any* $a \in A$*,* $\forall i, j \in ID$*, if* $L_S(\mathbf{x}, i) \neq j$ *and* $R_S(\mathbf{x}, j) \neq i$ *and* $L_S(\mathbf{x}', i) = j$ *and* $R_S(\mathbf{x}', j) = i$ *and* $\mathbf{x}.x_{R_S(\mathbf{x},j)} - \mathbf{x}.x_{L_S(\mathbf{x},i)} \geq c$*, then* $\mathbf{x}'.x_{R_S(\mathbf{x}',j)} - \mathbf{x}'.x_{L_S(\mathbf{x}',i)} \geq c$*, for any* $c > 0$.

Invariant 1 shows the spacing between any two non-faulty agents in any lane is always at least r_r, and the spacing between any non-faulty agent and any other agent in the same lane is at least r_r. There is no result on the spacing between any two faulty agents—they may collide. The proof is by induction.

Invariant 1. *For any reachable state* $\mathbf{x}$*,* $Safety_R(\mathbf{x})$.

3.2 Progress

The progress analysis works with fail-free executions, that is, there are no further fail_i transitions. Note that this does not mean $F(\mathbf{x}) = \emptyset$, only that along such executions $|F(\mathbf{x})|$ does not change. This is a standard assumption used to show convergence from an arbitrary state back to a stable set [1], albeit we note that we are dealing with permanent faults instead of transient ones. In this case, the stable set eventually reached are states where *Terminal* is satisfied. However, note that the first state in such an execution is not entirely arbitrary, as Section 3.1 established that such states satisfy at least $Safety_R$, and all the following analysis relies on this assumption.

First observe that, like safety, progress may be violated by failures. Any failed agent with nonzero velocity diverges by the definition of velocities in Figure 3, Line 34. This observation also highlights why $Flock$ is quantified over agents with identifiers in the set of suspected agents $\bar{S}(\mathbf{x})$ and not the set of failed agents $\bar{F}(\mathbf{x})$ or all agents ID—if it were quantified over ID, at no future point could $Flock(\mathbf{x})$ be attained if a failed agent has diverged. Zero velocity failures may also cause progress to be violated, where a “wall” of non-moving failed agents may be created, but such situations are excluded by the second part of Assumption (b) in Theorem 1.

Progress along Fail-Free Executions. In the remainder of this section, we show that once new actuator faults cease occurring, System eventually reaches a state satisfying $Terminal$. This is a convergence proof and we will use a Lyapunov-like function to prove this property. The remainder of this section applies to any infinite fail-free execution fragment, so fix such a fragment α_{ff}.

These descriptions of error dynamics are used in the analysis:

$$e(\mathbf{x}, i) \triangleq \begin{cases} |\mathbf{x}.x_i - \mathbf{x}.x_{\mathbf{x}.L_i} - r_f| & \text{if } i \text{ is a middle or a tail agent,} \\ 0 & \text{otherwise,} \end{cases}$$

$$eu(\mathbf{x}, i) \triangleq \begin{cases} |\mathbf{x}.u_i - \mathbf{x}.u_{\mathbf{x}.L_i} - r_f| & \text{if } i \text{ is a middle or a tail agent,} \\ 0 & \text{otherwise.} \end{cases}$$

Here $e(\mathbf{x}, i)$ gives the error with respect to r_f of Agent$_i$ and its non-suspected left neighbor and $eu(\mathbf{x}, i)$, with respect to target positions $\mathbf{x}.u_i$ rather than physical positions $\mathbf{x}.x_i$.

Now, we make the simple observation from Line 35 of Figure 3 that if a non-faulty agent i moves in some round, then it moves by at least a positive amount v_{min}. Observe that an agent may not move in a round if the conditional in Figure 3, Line 33 is satisfied, but this does not imply $v_{min} = 0$. Then, Lemma 3 states that from any reachable state $\mathbf{x}$ which does not satisfy NBM, the maximum error over all non-faulty agents in non-increasing. This is shown by first noting that only the update transition can cause any change of $e(\mathbf{x}, i)$ or $eu(\mathbf{x}, i)$, and then analyzing the change in value of $eu(\mathbf{x}, i)$ for each of the computations of u_i in the $Target$ subroutine of the update transition. Then it is shown that applying the $Quant$ subroutine cannot cause any $eu(\mathbf{x}, i)$ to increase, and finally computing x_i in the $Move$ subroutine does not increase any $e(\mathbf{x}, i)$.

Lemma 3. *For reachable states $\mathbf{x}, \mathbf{x}'$, if $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ and $\mathbf{x} \notin NBM$, for some $a \in A$, then* $\max_{i \in \bar{F}(\mathbf{x})} e(\mathbf{x}', i) \leq \max_{i \in \bar{F}(\mathbf{x})} e(\mathbf{x}, i)$.

Next, Lemma 4 shows sets of states satisfying NBM are invariant, a state satisfying NBM is reached, and gives a bound on the number of rounds required to reach such a state. Define the candidate Lyapunov function as $V(\mathbf{x}) \triangleq \sum_{i \in \bar{F}(\mathbf{x})} e(\mathbf{x}, i)$. Define the maximum value the candidate Lyapunov function obtained over any state $\mathbf{x} \in \alpha_{\mathit{ff}}$ satisfying NBM as $\gamma \triangleq \sup_{\mathbf{x} \in NBM} V(\mathbf{x})$.

Lemma 4. *Let $\mathbf{x}_k$ be the first state of α_{ff}, and let the head agent's position be fixed. If $V(\mathbf{x}_k) > \gamma$, then the* update *transition decreases $V(\mathbf{x}_k)$ by at least a positive* *constant ψ. Furthermore, there exists a finite round c such that $V(\mathbf{x}_c) \leq \gamma$, where $\mathbf{x}_c \in NBM(\mathbf{x})$ and $k < c \leq \lceil (V(\mathbf{x}_k) - \gamma)/\psi \rceil$, where $\psi = v_{min}$.*

Lemma 4 stated a bound on the time it takes for System to reach the set of states satisfying NBM. However, to satisfy $Flock_S(\mathbf{x})$, all $\mathbf{x} \in NBM$ must be inside the set of states that satisfy $Flock_S$, and the following lemma states this. From any state $\mathbf{x}$ that does not satisfy $Flock_S(\mathbf{x})$, there exists an agent that computes a control that will satisfy the quantization constraint and hence make a move towards NBM. This follows from the assumption that $\beta \leq \delta/(4N)$.

Lemma 5. *If $Flock_S(\mathbf{x})$, then $V(\mathbf{x}) \leq \sum_{i \in \bar{F}(\mathbf{x})} e(\mathbf{x}, i) = (\delta \left|\bar{F}(\mathbf{x})\right|)/4$.*

Now we observe that $Flock_W$ is a stable predicate, that is, that once a weak flock is formed, it remains invariant. This result follows from analyzing the $Target$ subroutine which computes the new targets for the agents in each round. Note that the head agent moves by a fixed distance $\frac{\delta}{2}$, only when $Flock_S$ holds, which guarantees that $Flock_W$ is maintained even though $Flock_S$ may be violated. This establishes that for any reachable state $\mathbf{x}'$, if $V(\mathbf{x}') > V(\mathbf{x})$, then $V(\mathbf{x}') < (\delta \left|\bar{F}(\mathbf{x})\right|)/2$.

Lemma 6. *$Flock_W$ is a stable predicate.*

The following corollary follows from Lemma 4, as $Flock_S(\mathbf{x})$ is violated after becoming satisfied only if the head agent moves, in which case $\mathbf{x}'.x_{H(\mathbf{x}')} < \mathbf{x}.x_{H(\mathbf{x})}$, which causes $V(\mathbf{x}') \geq V(\mathbf{x})$.

Corollary 1. *For $\mathbf{x} \in \alpha_{ff}$ such that, if $Flock_S(\mathbf{x})$, $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ $\forall a \in A$, and $\mathbf{x}.x_{H(\mathbf{x})} = \mathbf{x}'.x_{H(\mathbf{x}')}$, then $Flock_S(\mathbf{x}')$.*

The following lemma—with Assumption (b) of Theorem 1 that gives eventually a state is reached such that non-faulty agents may pass faulty agents—is sufficient to prove that $Terminal$ is eventually satisfied in spite of failures. After this number of rounds, no agent $j \in \bar{F}(\mathbf{x})$ believes any $i \in F(\mathbf{x})$ is its left or right neighbor, and thereby any failed agents diverge safely along their individual lanes if $|\mathbf{x}.v_i| > 0$ by the observation that failed agents with nonzero velocity diverge. Particularly, after some agent j has been suspected by all non-faulty agents, the $Mitigate$ subroutine of the update transition shows that the non-faulty agents will move to a different lane at the next round. This shows that mitigation takes at most one additional round after detection, since we have assumed in Theorem 1 that there is always free space on some lane. This implies that so long as a failed agent is detected prior to safety being violated, only one additional round is required to mitigate, so the time of mitigation is a constant factor added to the time to suspect, resulting in the constant c being linear in the number of agents.

Lemma 7. *For any fail-free execution fragment α_{ff}, if $\mathbf{x}.failed_i$ at some state $\mathbf{x} \in \alpha_{ff}$, then for a state $\mathbf{x}' \in \alpha_{ff}$ at least c rounds from $\mathbf{x}$, $\forall j \in ID.\mathbf{x}'.L_j \neq i \wedge \mathbf{x}'.R_j \neq i$.*

The next theorem shows that System eventually reaches the goal as a strong flock, that is, there is a finite round t such that $Terminal(\mathbf{x}_t)$ and $Flock_S(\mathbf{x}_t)$ and shows that System is self-stabilizing when combined with a failure detector.

Theorem 2. *Let α_{ff} be written $\mathbf{x}_0, \mathbf{x}_1, \ldots$. Consider the infinite sequence of pairs $\langle \mathbf{x}_0.x_{H(\mathbf{x}_0)}, V(\mathbf{x}_0) \rangle$, $\langle \mathbf{x}_1.x_{H(\mathbf{x}_1)}, V(\mathbf{x}_1) \rangle$, $\ldots$, $\langle \mathbf{x}_t.x_{H(\mathbf{x}_t)}, V(\mathbf{x}_t) \rangle$, $\ldots$. Then, there* *exists t at most $\left\lceil \frac{(V(\mathbf{x}_0)-|\bar{F}(\mathbf{x})|\delta/4)}{v_{min}} \right\rceil + \left\lceil \frac{|\bar{F}(\mathbf{x})|\delta/4}{v_{min}} \right\rceil \max\{1, \frac{\mathbf{x}_0.x_{H(\mathbf{x}_0)}}{v_{min}} O(N)\}$ rounds from $\mathbf{x}_0$ in α_{ff}, such that: (a) $\mathbf{x}_t.x_{H(\mathbf{x}_t)} = \mathbf{x}_{t+1}.x_{H(\mathbf{x}_{t+1})}$, (b) $V(\mathbf{x}_t) = V(\mathbf{x}_{t+1})$, (c) $\mathbf{x}_t.x_{H(\mathbf{x}_t)} \in [0, \beta]$, (d) $V(\mathbf{x}_t) \leq |\bar{F}(\mathbf{x})| \frac{\delta}{4}$, (e) Terminal$(\mathbf{x}_t)$, and (f) Flocks$_S(\mathbf{x}_t)$.*

3.3 Failure Detection

In the earlier analysis we assumed that it is possible to detect all actuator faults within finite number of rounds k_d. Unfortunately this is not true, as there exist failures which cannot be detected at all. A trivial example of such an undetectable failures is the failure of a node with 0 velocity at a terminal state, that is, a state at which all the agents are at the goal in a flock and therefore are static. While such failures were undetectable in any number of rounds, these failures do not violate *Safety* or *Terminal*. It turns out that only failures which cause a violation of safety or progress may be detected.

Lower-Bound on Detection Time. While the occurrence of $\mathsf{fail}_i(v)$ may never be detected in some cases as just illustrated, we show a lower-bound on the detection time for all $\mathsf{fail}_i(v)$ transitions that can be detected. The following lower-bound applies for executions beginning from states that do not *a priori* satisfy $Terminal$. It says that a failed agent mimicked the actions of its correct non-faulty behavior in such a way that despite the failure, System still progressed to NBM as was intended. From an arbitrary state, it takes $O(N)$ rounds to converge to a state satisfying NBM by Lemma 4.

Lemma 8. *The detection time lower-bound for any detectable actuator fault is $O(N)$.*

Next we show that the the failure detection mechanism incorporated in Figure 3 does not produce any false positives.

Lemma 9. *In any reachable state $\mathbf{x}$, $\forall j \in \mathbf{x}.Suspected_i \Rightarrow \mathbf{x}.failed_j$.*

The next lemma shows a partial *completeness* property [2] of the failure detection mechanism incorporated in Figure 3.

Lemma 10. *Suppose that $\mathbf{x}$ is a state in the fail-free execution fragment α_{ff} such that $\exists$ $j \in F(\mathbf{x})$, $\exists\, i \in ID$, and j is not suspected by i. Suppose that either (a) $|\mathbf{x}.xo_j - \mathbf{x}.uo_j| \leq \beta$ and $|\mathbf{x}.x_j - \mathbf{x}.uo_j| \neq 0$, or (b) $|\mathbf{x}.xo_j - \mathbf{x}.uo_j| > \beta$ and $\operatorname{sgn}(\mathbf{x}.x_j - \mathbf{x}.xo_j) \neq \operatorname{sgn}(\mathbf{x}.uo_j - \mathbf{x}.xo_j)$. Then, $\mathbf{x} \overset{\mathsf{suspect}_i(j)}{\rightarrow} \mathbf{x}'$.*

Now we show an upper-bound on the number of rounds to detect any failure which may be detected using the failure detection mechanism incorporated in Figure 3 by applying Lemma 8 with Lemmata 9 and 10, and that agents share suspected sets in Figure 3, Line 21. This states an $O(N)$ upper-bound on the detection time of our failure detector and shows that eventually all non-faulty agents know the set of failed agents.

Corollary 2. *For any state $\mathbf{x}_k \in \alpha_{\mathit{ff}}$ such that $\mathbf{x}_k \notin Terminal$, there exists a round $\mathbf{x}_s$ in α_{ff} such that $\forall i \in \bar{F}(\mathbf{x}_s)$, $\mathbf{x}_s.Suspected_i = F(\mathbf{x})$ and $k - s$ is $O(N)$.*

3.4 Simulations

Simulation studies were performed, where flocking convergence time (as by Lemma 4), goal convergence time (as by Theorem 2), and failure detection time (as by Corollary 2) were of interest. Unless otherwise noted, the parameters are chosen as $N = 6$, $N_L = 2$, $r_s = 20$, $r_f = 40$, $\delta = 10$, $\beta = \delta/(4N)$, $v_{min} = \frac{\beta}{2}$, $v_{max} = \beta$, the head agent starts with position at r_f, and the goal is chosen as the origin. Figure 4 shows the value of the Lyapunov function V and maximum agent error from flocking, e_{max}. The initial state is that each agent is spaced by r_s from its left neighbor. Observe that while moving towards the goal, $Flock_S$ is repeatedly satisfied and violated, with invariance of $Flock_W$.

Figure 5 shows that for a fixed value of v_{min}, the time to convergence to NBM is linear in the number of agents. This choice of fixed v_{min} must be for the largest number of agents, 12 in this case, as v_{min} is upper bounded by $\beta = \frac{\delta}{4N}$ which is a function of N. As v_{min} is varied the inverse relationship with N is observed, resulting in a roughly quadratic growth of convergence time to NBM. This illustrates linear convergence time as well as linear detection time, as this is bounded by the convergence time from Corollary 2. The initial state was for expansion, so each agent was spaced at r_s from its left neighbor.

In all single-failure simulations, a trend was observed on the detection time. When failing each agent individually, and with all else held constant (initial conditions, round of failure, etc.), only one of the detection times for failure velocities of $-v_{max}$, 0, or v_{max} is ever larger than one round. The frequent occurrence of a single round detection is interesting. For instance, in the expansion case, each failed agent i except the tail are detected in one round when $vf_i \neq 0$ since a violation of safety occurs. However, detecting that the head agent has failed with zero velocity requires convergence of the system to a strong flock prior to detection, as does detecting that the tail agent failed with v_{max}, as this mimics the desired expansive behavior up to the point where the tail moves beyond the flock. In the contraction case, each failed agent i except the tail is detected in one round when $vf_i \neq 0$, since they are at the center of their neighbors positions, while the tail agent failing with $-v_{max}$ takes many rounds to detect, since it should be

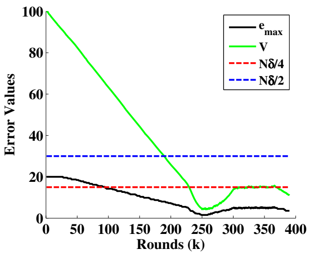


Fig. 4. Expansion simulation showing max error e_{max}, Lyapunov function value V, with weak and strong flocking constants

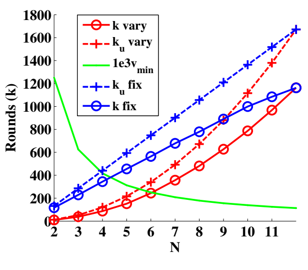


Fig. 5. Rounds k (upper bounded by k_u) to reach *Terminal* versus number of agents N with fixed and varying v_{min}

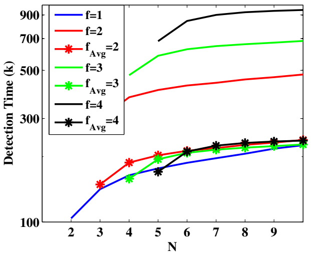


Fig. 6. Multiple failure simulation with f zero velocity failures at round 0 from initial state of $2r_f$ inter-agent spacing

moving towards its left neighbor to cause the contraction. Thus the observation is, for a reachable state $\mathbf{x}$, if $|F(\mathbf{x})| = 1$, let the identifier of the failed agent be i, and consider the three possibilities of $\mathbf{x}.vf_i = 0$, $\mathbf{x}.vf_i \in (0, v_{max}]$, and $\mathbf{x}.vf_i \in [-v_{max}, 0)$. Then along a fail-free execution fragment starting from $\mathbf{x}$, for one of these choices of vf_i, the detection time is greater than 1, and for the other two, the detection time is 1. This illustrates there is only one potentially "bad" mimicking action which allows maintenance of both safety and progress and takes more than one round to detect. The other two failure velocity conditions violate either progress or safety immediately and lead to an immediate detection.

Finally, Figure 6 shows the detection time with varying N and f from a fixed initial condition of inter-agent spacings at $2r_f$. The $f_{Avg} = i$ lines show the total detection time divided by f. Failures were fixed with $vf_i = 0$, failing each combination of agents, so for $f = 2$ and $N = 3$, each combination of $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ were failed individually, and the detection time is the average over the number of these combinations for each choice of f and N. The detection time averaged over the number of failure indicates that the detection time to detect any failure in a multiple failure scenario is on the same order as that in the single failure case. However, the detection time not averaged over the number of failures indicates that the detection time to detect all failures increases linearly in f and on the order of N, as predicated by Corollary 2.

4 Conclusion

This paper presented an algorithm for the safe flocking problem—where the desired properties are safety invariance and eventual progress, that eventually a strong flock is formed and a destination reached by that flock—in spite of permanent actuator faults. An $O(N)$ lower-bound was presented for the detection time of actuator faults, as well as conditions under which the given failure detector can match this bound, although it was established that this is not always possible. The main result was that the algorithm is self-stabilizing when combined with a failure detector. Without the failure detector, the system would not be able to maintain safety as agents could collide, nor make progress to states satisfying flocking or the destination, since failed agents may diverge, causing their neighbors to follow and diverge as well. Simulation results served to reiterate the formal analysis, and demonstrated the influence of certain factors—such as multiple failures—on the failure detection time.

References

1. Arora, A., Gouda, M.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Trans. Softw. Eng. 19, 1015–1027 (1993)
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
3. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)
4. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
5. Fax, J., Murray, R.: Information flow and cooperative control of vehicle formations. IEEE Trans. Autom. Control 49(9), 1465–1476 (2004)

6. Franceschelli, M., Egerstedt, M., Giua, A.: Motion probes for fault detection and recovery in networked control systems. In: American Control Conference 2008, pp. 4358–4363 (2008)
7. Gazi, V., Passino, K.M.: Stability of a one-dimensional discrete-time asynchronous swarm. IEEE Trans. Syst., Man, Cybern. B 35(4), 834–841 (2005)
8. Jadbabaie, A., Lin, J., Morse, A.: Coordination of groups of mobile autonomous agents using nearest neighbor rules. IEEE Trans. Autom. Control 48(6), 988–1001 (2003)
9. Johnson, T.: Fault-Tolerant Distributed Cyber-Physical Systems: Two Case Studies. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801 (May 2010)
10. Johnson, T., Mitra, S.: Safe and stabilizing distributed flocking in spite of actuator faults. Tech. Rep. UILU-ENG-10-2204 (CRHC-10-02), University of Illinois at Urbana-Champaign, Urbana, IL (May 2010)
11. Okubo, A.: Dynamical aspects of animal grouping: Swarms, schools, flocks, and herds. Adv. Biophys. 22, 1–94 (1986)
12. Olfati-Saber, R.: Flocking for multi-agent dynamic systems: algorithms and theory. IEEE Trans. Autom. Control 51(3), 401–420 (2006)
13. Shaw, E.: Fish in schools. Natural History 84(8), 40–45 (1975)
14. Tsitsiklis, J., Bertsekas, D., Athans, M.: Distributed asynchronous deterministic and stochastic gradient optimization algorithms. IEEE Trans. Autom. Control 31(9), 803–812 (1986)

Author Index