Xavier Défago
Franck Petit
Vincent Villain (Eds.)

LNCS 6976

Stabilization, Safety, and Security of Distributed Systems

13th International Symposium, SSS 2011
Grenoble, France, October 2011
Proceedings

Springer

Lecture Notes in Computer Science 6976

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

Xavier Défago Franck Petit
Vincent Villain (Eds.)

Stabilization, Safety, and Security of Distributed Systems

13th International Symposium, SSS 2011
Grenoble, France, October 10-12, 2011
Proceedings

Springer

Volume Editors

Xavier Défago
Japan Advanced Institute of Science and Technology (JAIST)
School of Information Science
Ishikawa, Japan
E-mail: defago@jaist.ac.jp

Franck Petit
LIP6/INRIA/UPMC Sorbonne Universities
Paris, France
E-mail: franck.petit@lip6.fr

Vincent Villain
MIS, University of Picardie Jules Verne
Amiens, France
E-mail: vincent.villain@u-picardie.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24549-7 e-ISBN 978-3-642-24550-3
DOI 10.1007/978-3-642-24550-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011937547

CR Subject Classification (1998): C.2.4, C.3, F.1, F.2.2, K.6.5

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

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

Printed on acid-free paper

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

Preface

The papers in this volume were presented at the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), held on October 10–12, 2011 in Grenoble, France.

SSS is an international forum for researchers and practitioners in the design and development of distributed systems with self-* attributes, such as self-stabilization, self-configuration, self-organization, self-management, self-healing, self-optimization, self-adaptiveness, self-repair, self-protection, etc. Many researchers are now focusing on bringing self-* properties into distributed systems. They mainly aim to tolerate different kinds of undesirable phenomena without human intervention. Moreover, distributed systems are now at a crucial point in their evolution, marked by the increasing importance of flexibility, as is the case in peer-to-peer networks, large-scale wireless sensor networks, mobile ad-hoc networks, cloud computing, robotic networks, etc. Also, new applications with self-* requirements are currently coming up in different fields such as grid and web services, banking and e-commerce, e-health and robotics, aerospace and avionics, automotive, industrial process control, etc.

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, in 2003, the title of the forum was changed 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 five SSS conferences were held in Dallas (2006), Paris (2007), Detroit (2008), Lyon (2009), and NewYork (2010).

This year the Program Committee was organized into several tracks reflecting most topics related to self-* systems. The tracks were: (*i*) Ad-Hoc, Sensor, and Dynamic Networks, (*ii*) Fault-Tolerance and Dependable Systems, (*iii*) Overlay and Peer-to-Peer Networks, (*iv*) Safety and Verification, (*v*) Security, (*vi*) Self-Organizing and Autonomic Systems, and (*vii*) Self-Stabilization.

We received 79 submissions from 19 countries. Each submission was reviewed by at least three Program Committee members with the help of external reviewers. Out of the 79 submitted papers, 29 papers were selected for presentation. The symposium also included 10 brief announcements. Selected papers from the symposium will be published in a special issue of *Theoretical Computer Science* (TCS). This year, we were very fortunate to have two distinguished invited speakers: Nicola Santoro and Toshimitsu Masuzawa.

Among the 29 selected papers, we considered 3 papers for special awards. The best paper award was given to Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju for "Building Self-Stabilizing Overlay Networks with the Transitive Cloture Framework". The best student paper award was shared by Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler for "Corona: A Stabilizing Deterministic Message-Passing Skip List", and Damien Imbs and Michel Raynal for "The Weakest Failure Detector to Implement a Register in Asynchronous Systems with Hybrid Communication".

On behalf of the Program Committee, we would like to thank all the authors who submitted their work to SSS. We sincerely acknowledge the tremendous time and effort the Program Vice Chairs and the Program Committee members invested in the symposium. We are grateful to the external reviewers for their valuable and insightful comments. We also thank the members of the Steering Committee for their invaluable advice. We are grateful to the Organizing Committee members for their time and invaluable effort that greatly contributed to the success of this symposium.

Organizing this event would not have been possible without the support of the following organizations: ANR SPADES, CNRS, Elsevier, Grenoble Institute of Technology (INP), INRIA, Japan Advanced Institute of Science and Technology (JAIST), Lab. MIS (University of Picardie Jules Verne), Springer (LNCS), University Joseph Fourier (UJF), and VERIMAG. Finally, the process of paper submission, selection, and compilation in the proceedings was greatly simplified by the strong and friendly interface of the *EasyChair* system (`http://www.easychair.org`).

Dedication. We were supposed to organize SSS 2011 at Shinagawa (Tokyo), but after the terrible earthquake and tsunami that affected Japan and its people, the consequences on Fukushima Daiichi nuclear power station led us reluctantly to relocate the symposium to another country. On behalf of our community, we would like to dedicate SSS 2011 to Japan and its people.

October 2011

Xavier Défago
Franck Petit
Vincent Villain

Conference Organization

General Chair

Vincent Villain	University of Picardie Jules Verne

Program Chairs

Xavier Défago	Japan Advanced Institute of Science and Technology
Franck Petit	LIP6 / INRIA / UPMC Sorbonne Universities

Program Vice-chairs

Shlomi Dolev	Ben-Gurion University of the Negev
Colette Johnen	University of Bordeaux 1
Stefan Schmid	TU Berlin / Deutsche Telekom Lab
Alexander Shvartsman	University of Connecticut
Paul Spirakis	University of Patras
Tatsuhiro Tsuchiya	Osaka University
Philippas Tsigas	Chalmers University
Koichi Wada	Nagoya Institute of Technology
Roger Wattenhofer	Swiss Federal Institute of Technology in Zurich

Local Arrangements Chair

Karine Altisen	VERIMAG / Grenoble Institute of Technology
Eddy Caron	LIP / INRIA / ENS de Lyon
Frédéric Desprez	LIP / INRIA
Stéphane Devismes	VERIMAG / University Joseph Fourier
Alain Girault	INRIA
Pascal Lafourcade	VERIMAG / University Joseph Fourier

Publicity Chairs

Doina Bein	The Pennsylvania State University
Lélia Blin	LIP6 CNRS
Naohiro Hayashibara	Kyoto Sangyo University

Program Committee

Ad-Hoc, Sensor, and Dynamic Networks
Chair: Koichi Wada

Zohir Bouzid	LIP6 / UPMC Sorbonne Universities
Jiannong Cao	Hong Kong Polytechnic University
Wei Chen	Tennessee state University
Paola Flocchini	University of Ottawa
Pierre Fraigniaud	CNRS / University Paris Diderot
Yoshiaki Katayama	Nagoya Institute of Technology
Shay Kutten	Technion Israel Institute of Technology
Nikola Milosavljević	Max-Planck-Institut für Informatik
Ravi Prakash	University of Texas at Dallas
Matthieu Roy	LAAS-CNRS

Fault-Tolerance and Dependable Systems
Chairs: Shlomi Dolev and Alexander Shvartsman

Brian Coan	Telcordia Technologies
Mohamed Gouda	National Science Foundation
Maria Gradinariu Potop-Butucaru	LIP6 / INRIA / UPMC Sorbonne Universities
Rachid Guerraoui	EPFL
Toshimitsu Masuzawa	Osaka University
Nathalie Mitton	INRIA
Mark Tuttle	Intel Corporation
Paul Vitanyi	CWI / University of Amsterdam

Overlay and Peer-to-Peer Networks
Chairs: Stefan Schmid and Roger Wattenhofer

Sonja Buchegger	KTH
Anwitaman Datta	NTU Singapore
Isabelle Guérin-Lassous	Université Lyon / LIP
Riko Jacob	TU München
Miroslaw Korzeniowski	Wroclaw University of Technology
Mikhail Nesterenko	Kent State University
Andrea Richa	ASU
Christian Schindelhauer	University of Freiburg
Jukka Suomela	HIIT / University of Helsinki
Sébastien Tixeuil	LIP6 / UPMC Sorbonne Universities

Safety and Verification
Chair: Tatsuhiro Tsuchiya

Cyrille Valentin Artho	AIST
Borzoo Bonakdarpour	University of Waterloo

Vijay Garg	University of Texas at Austin
Sandeep Kulkarni	Michigan State University
Stephan Merz	INRIA
Sayan Mitra	University of Illinois at Urbana-Champaign
Kazuhiro Ogata	JAIST
Oliver Theel	University of Oldenburg
Keiichi Yasumoto	Nara Institute of Science and Technology

Security
Chair: Philippas Tsigas

Christian Cachin	IBM Research - Zurich
Felix Freiling	University of Erlangen
Evangelos Markatos	FORTH
Atsuko Miyaji	Japan Advanced Institute of Science and Technology
Angelos Stavrou	George Mason University
Moti Yung	Google / Columbia University

Self-Organizing and Autonomic Networks
Chair: Paul Spirakis

Ioannis Chatzigiannakis	CTI / University of Patras
Lefteris Kirousis	University of Patras
Evangelos Kranakis	Carleton University
Danny Krizanc	Wesleyan University
Marios Mavronicolas	University of Cyprus
Giuseppe Persiano	Università di Salerno
Nicola Santoro	Carleton University
Elad Michael Schiller	Chalmers University of Technology

Self-Stabilization
Chair: Colette Johnen

Joffroy Beauquier	LRI-CNRS / Université Paris Sud
Doina Bein	The Pennsylvania State University
Lélia Blin	LIP6 CNRS
Jorge Cobb	The University of Texas at Dallas
Ajoy K. Datta	University of Nevada, Las Vegas
Stéphane Devismes	VERIMAG / University Joseph Fourier
Danny Dolev	Hebrew University
Sukumar Ghosh	University of Iowa
Mohamed Gouda	National Science Foundation
Ted Herman	University of Iowa
Mehmet H. Karaata	Kuwait University
Fukuhito Ooshita	Osaka University
Laurence Pilard	PRISM / CNRS / UVSQ

Roger Wattenhofer	Swiss Federal Institute of Technology in Zurich
Masafumi Yamashita	Kyushu University

Steering Committee

Anish Arora	The Ohio State University
Ajoy K. Datta	University of Nevada, Las Vegas
Shlomi Dolev	Ben-Gurion University of the Negev
Sukumar Ghosh (*chair*)	University of Iowa
Mohamed Gouda	National Science Foundation
Ted Herman	University of Iowa
Toshimitsu Masuzawa	Osaka University
Vincent Villain	University of Picardie Jules Verne

Additional Reviewers

Orestis Akribopoulos
Christian Attiogbe
Bharath Balasubramanian
Thibault Bernard
Janna Burman
Ken Calvert
Johanne Cohen
Shantanu Das
Abhishek Dhama
Josep Diaz
Faith Ellen
Arnaud Fontaine
Rachid Hadid
Stefan Hallerstede
Michiko Inoue
Sayaka Kamei
Matthew Kellett
Ajay Kshemkalyani
Petr Kuznetsov
Neeraj Mittal
Stavros Nikolaou
Sotiris Nikoletseas
Kazumasa Omote
Giuseppe Prencipe
Roopsha Samanta
Christian Storm
Morihiko Tamai
Toshiaki Tanaka
Yukiko Yamauchi

Table of Contents

Silence Is Golden: Self-stabilizing Protocols Communication-Efficient after Convergence

Toshimitsu Masuzawa

Graduate School of Information Science and Technology,
Osaka University, Suita 565-0871, Japan

1 Motivation

Self-stabilization is a general paradigm to provide forward recovery capabilities to distributed systems. A self-stabilizing protocol can eventually recover its intended behavior even when starting from an arbitrary initial configuration, and thus, it has high adaptability to transient faults (e.g., process state corruptions and message corruptions) and network topology changes. The high adaptability is usually acquired at the cost of efficiency. A crucial difference in cost between self-stabilizing and non-self-stabilizing protocols lies in the cost of communication after reaching a desired configuration. It is quite evident for static problems, e.g., spanning-tree construction. Self-stabilizing protocols cannot allow any process to terminate its communication even after converging to a desired configuration (where a solution of the problem is already obtained), while non-self-stabilizing ones can eventually allow every process to terminate all the activity.

Actually, most of self-stabilizing protocols require every pair of neighboring processes to communicate with each other repeatedly and forever even after convergence to desired configurations. This leads high communication load and makes the protocols unacceptable in some situations. Especially, in practical applications, we can expect that self-stabilizing protocols show their intended behavior most of the time during their execution. Deviation from the intended behavior may occur because of transient faults or topology changes but it is infrequent. Thus, the efficiency after convergence (or during the intended behavior) is more important than that during convergence. Nevertheless, only a few recent papers [1,2,3,4] are dedicated to improving communication efficiency after convergence.

In this paper, we introduce some complexity measures reflecting communication-efficiency after convergence of self-stabilizing protocols. We present some results on possibility and impossibility of communication-efficient protocols for fundamental problems.

2 Point-to-Point Communication Model

Devismes *et al.*[2] and Masuzawa *et al.*[4] introduced some complexity measures for communication-efficiency after convergence of self-stabilizing protocols. The measures are targeting the point-to-point communication model.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 1–3, 2011.

A self-stabilizing protocol $\mathcal{A}$ is $\diamond$-*h-communication-efficient* if every (infinite) execution of $\mathcal{A}$ has a suffix where at most h ordered pairs of processes communicate with each other at every (asynchronous or synchronous) round. The communication-efficiency reflects the total communication load after convergence but does not consider congestion of the communication. The local-communication-efficiency is a measure for the local communication load after convergence; a self-stabilizing protocol $\mathcal{A}$ is $\diamond$-*k-locally-communication-efficient* if every (infinite) execution of $\mathcal{A}$ has a suffix where every process receives information from at most k of its neighbors at every round.

The (local-)communication-efficiency reflects the communication load at every round, and the communicating process pairs can vary from round to round. The concept of $\diamond$-*k-stability* is introduced to denote the stability of communicating process pairs; a self-stabilizing protocol $\mathcal{A}$ is $\diamond$-*k-stable* if every (infinite) execution of $\mathcal{A}$ has a suffix where each process receives information from at most k neighbors during the execution suffix. From the definition, $\diamond$-k-stability implies $\diamond$-k-local-communication-efficiency.

The followings are some of the results presented in [2,4].

(1) For the problem class containing the maximal independent set problem, there is no $(\Delta - 1)$-stable protocol, where Δ is the maximum degree of processes. This implies that at least one process has to keep communicating with all of its neighbors.
(2) For the maximal independent set problem, there is a $\diamond$-1-locally-communication-efficient protocol such that processes not in the independent set receives information only from one of its neighbors in a suffix of any execution.
(3) For the spanning tree construction, when a unique root is designated, there exists a $\diamond$-1-stable self-stabilizing protocol that allows each process, after convergence, to receive information only from its parent.
(4) For the spanning tree construction, when each process has a unique identifier but a unique root is not designated, no protocol can allow even a single pair of neighbors to eventually stop communication between them (i.e., $\diamond$-$o(m)$-communication-efficiency is unattainable, where m is the number of links).
(5) For the spanning tree construction, when each process has a unique identifier and knows an upper bound N ($n \leq N < 2n$) of n a priori (but a unique root is not designated), there exists a $\diamond$-$2(n-1)$-communication-efficient protocol that allows each process, after convergence, to receive information only from its parent and children. The restriction $N < 2n$ on the upper bound N is the weakest in the sense that $\diamond$-$o(m)$-communication-efficiency becomes unattainable when $N = 2n$.

The results (3) and (4) bring out the contrast between self-stabilization and communication-efficient self-stabilization: existence of a unique root is sufficient for attaining communication-efficiency but existence of unique process identifiers is not, however, self-stabilizing (but not communication-efficient) spanning-tree construction is possible with either assumption.

3 Local-Broadcast Communication Model

Communication-efficiency is more important in wireless networks such as wireless sensor networks, since energy consumption is critical in these networks. In wireless networks, local-broadcast is commonly used for communication; once a node sends a message, all of its neighbors receive the message. In the distributed system with local-broadcast communication, reducing the number of processes that repeatedly broadcast messages is important. Recently, we started studying the communication-efficiency in the distributed system model[5].

A self-stabilizing protocol $\mathcal{A}$ is $\diamond$-*k-broadcast-stable* if every execution of $\mathcal{A}$ has a suffix where only k processes locally broadcast messages. Another measure is the (average) number of processes at every round that locally broadcast messages. A self-stabilizing protocol $\mathcal{A}$ is $\diamond$-*k-average-broadcast-efficient* if every execution of $\mathcal{A}$ has a suffix where only k processes locally broadcast messages at every round on average.

The followings are the results for the synchronous systems presented in [5].

(1) For the minimal connected dominating set problem and the spanning tree construction problem, no self-stabilizing protocol allows even a single process to stop broadcasting after convergence, i.e., $\diamond$-$(n-1)$-broadcast-stability cannot be attained.
(2) For the maximal independent set problem and the minimal dominating set problem, there exist self-stabilizing protocols such that only the processes in the set keep broadcasting after convergence.
(3) For all the problems in the above, any $\diamond$-$o(n)$-average-broadcast-efficient protocol has no upper bound on its convergence time (or the number of rounds required to reach a desired configuration).
(4) For the spanning tree construction problem, there exists a $\diamond$-1-average-broadcast-efficient protocol when each process knows the number n of processes in the system.

References

1. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 219–233. Springer, Heidelberg (2007)
2. Devismes, S., Masuzawa, T., Tixeuil, S.: Communication efficiency in self-stabilizing silent protocols. In: Proc. the 29th ICDCS, pp. 474–481 (2009)
3. Kutten, S., Zinenko, D.: Low communication self-stabilization through randomization. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 465–479. Springer, Heidelberg (2010)
4. Masuzawa, T., Izumi, T., Katayama, Y., Wada, K.: Brief announcement: Communication-efficient self-stabilizing protocols for spanning-tree construction. In: Proc. the 13th OPODIS, pp. 219–224 (2009)
5. Takimoto, T., Ooshita, F., Kakugawa, H., and Masuzawa, T.: Communication-efficient self-stabilizing protocols in wireless networks. In: Proc. the 7th WTCS (to appear) (in Japanese)

Computing in Time-Varying Networks

Nicola Santoro

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

There has been recently a large number of investigations devoted to the study of infrastructure-less highly dynamic networks. These include most types of *highly-mobile ad hoc networks*, such as pedestrian or vehicular networks, where the network's topology may change dramatically over time due to the movement of the nodes; *sensor networks* with sleep scheduling, where links only exist when two neighbouring sensors are awake and have power; and low-density ad hoc networks made up of satellites, where nodes are most of the time isolated and must rely on a store-carry-forward mechanism for their communications. These highly dynamic networks, variously called *delay-tolerant*, *disruptive-tolerant*, *challenged*, *opportunistic*, have in common that the assumption of *connectivity* does not necessarily hold, at least with the usual meaning of *contemporaneous end-to-end multi-hop paths* between any pair of nodes. The network may actually be disconnected at every time instant. Still, communication routes may be available over time and space, and make broadcast, routing, and distributed computing feasible.

Not surprisingly, an extensive amount of research has been devoted, mostly by the engineering community, to the problems of broadcast and routing in such highly dynamical environment. As part of these research efforts, a number of important concepts have been identified, and occasionally expressed within a more general scheme. Interestingly, closely related insights have been obtained in the investigations being carried out in some apparently unrelated areas of dynamic systems. This is for example the case of the study of complex real-world networks ranging from neuroscience or biology to transportation systems or social studies, e.g., the characterization of the interaction patterns emerging in a social network. In several cases, differently named concepts identified by different researchers are actually one and the same concept. Indeed, the concepts discovered in all these investigations can be viewed as parts of the same conceptual universe; and the formalisms proposed so far to express some specific concepts can be viewed as fragments of a larger formal description of this universe. A common point in all these areas is that the system structure - the network topology - varies in time. Furthermore the rate and/or degree of the changes is generally too high to be reasonably modeled in terms of network faults or failures: in these systems changes are not anomalies but rather integral part of the nature of the system.

This talk describes the current research effort to integrate the vast collection of concepts, formalisms, and results found. It also reviews the status of the research on distributed computing in time-varying networks, outlining the challanges, difficulties and promising directions.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, p. 4, 2011.

The K-Observer Problem in Computer Networks

H.B. Acharya[1], Taehwan Choi[1],
Rida A. Bazzi[2], and Mohamed G. Gouda[1,3]

[1] The University of Texas at Austin, USA
{acharya,ctlight}@cs.utexas.edu
[2] Arizona State University, USA
[3] The National Science Foundation, USA

Abstract. For any non-negative integer K, a *K-observer* P of a network N is a set of nodes in N such that each message, that travels at least K hops in N, is handled (and so observed) by at least one node in P. A K-observer P of a network N is *minimum* iff the number of nodes in P is less than or equal the number of nodes in every K-observer of N. The nodes in a minimum K-observer of a network N can be used to monitor the message traffic in network N, detect denial-of-service attacks, and act as firewalls to identify and discard attack messages. This paper considers the problem of constructing a minimum K-observer for any given network. We show that the problem is NP-hard for general networks, and give linear-time algorithms for constructing minimum or near-minimum K-observers for special classes of networks: trees, rings, L-rings, and large grids.

1 Introduction

Every node in a computer network performs a number of traditional tasks: generating messages, routing and forwarding messages, and consuming messages. Beside these traditional tasks, some nodes are designated, in a network, to perform additional tasks: observing and collecting statistics concerning the message traffic that goes through each designated node and filtering the message traffic that goes through each designated node. We refer to those nodes that are designated, in a computer network, to perform these additional tasks as *network observers*.

This paper discusses the problem of how to select the nodes to be designated network observers in a computer network. We start the discussion by proposing a criterion, named minimum K-observers for some non-negative integer K, which can be used to identify the network observers in a computer network.

A K-observer P of a network N is a set of nodes in N such that each message, that travels at least K hops in N, is handled (and so observed) by at least one node in P.

Clearly, this definition of a K-observer P depends on the chosen value of K. On one hand, if the chosen value of K is small (with respect to the total number

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 5–18, 2011.

of nodes n in N), then P is a large set containing most of the nodes in N. For example, if $K = 0$, then P is the set of all nodes in N. On the other hand, if the chosen value of K is large (with respect to the total number of nodes n in N), then P can be a small set containing few nodes in N. For example, if $K = n-1$, then P can be a singleton containing only one node (any node) in N. Also, if $K \geq n$, then P can be the empty set.

Note that if a set P is a K-observer of a network N, then adding more nodes of N to P does not change the status of P of being a K-observer of N. This note suggests that we should be more interested in minimum K-observers, rather than in K-observers, as defined next.

A K-observer P of a network N is *minimum* iff the number of nodes in P is less than or equal the number of nodes in every K-observer of N.

Based on this discussion, in order to identify the network observers in a computer network N, one needs to construct a minimum K-observer P of network N, for some chosen K, and use the nodes in the constructed P as the network observers in N.

The problem of constructing a minimum K-observer in a network is related, though not identical, to several established problems in network design:

1. Constructing a node cover in a network
   The *Node Cover Problem* is to find a minimum number of nodes such that each link is incident to at least one node in the network [9]. The K-observer problem is more generalized than the *Node Cover Problem* such that the Node Cover Problem is the special case of the K-observer problem when $K = 1$. Armbruster [3] discusses the Node Cover Problem for sources and destinations of all paths in the network whereas the K-observer problem generalizes the Node Cover Problem for a path of length K in the network.
2. Creating a backbone for communication
   A very important problem for wireless networks, the *Connected Dominating Set Problem*, is to find a set of nodes that are connected, such that each node in the network either belongs to this set of nodes or is one hop from it [19]. A connected dominating set is used for virtual backbones in wireless networks. Although mobile networks do not have physical backbones, virtual backbones can be formed to help communication in wireless ad-hoc networks [2], [17], [12].
3. Placing guards in an art gallery
   The *Art Gallery Problem* is to determine the number of guards necessary to cover an art gallery, such that every point in the art gallery is guarded by at least one observer [7]. This problem is equivalent to the *Coverage Problem* in the context of wireless sensor networks [10], wireless ad-hoc networks, and wireless sensor ad-hoc networks [13]. Moreover, this problem is equivalent to the *Dominating Set Problem* if guards must be placed on nodes, and only nodes need to be guarded [9].

4. The problem of facility location in a network
   The *Facility Location Problem* is to find a place for a facility such that the distances from customers are minimized [18]. This problem can be regarded as the *Set Cover Problem* [11]; it has been discussed in various contexts such as the placement of monitoring nodes [4], web server replicas [14], and overlay nodes [16].

The rest of the paper is organized as follows. We begin by proving that our K-observer problem is NP hard for general networks in Section 2. We then explore solutions for some special cases: tree networks in Section 3, ring networks in Section 4, and grid networks in Section 5. We go on to discuss some possible applications for the K-observer problem, and conclude with a few remarks.

2 The K-Observer Problem

A *network* N is an undirected graph (V, E), where V is a nonempty set of *nodes* and E is a set of undirected *links*. Each *link* in E is a set of two distinct nodes in V. A link $\{u, v\}$ in a network N is said to be *incident* at nodes u and v in N.

A *path* in a network N is a nonempty sequence $(u_1, u_2, \ldots, u_r)$ of distinct nodes in N such that each pair $\{u_i, u_{i+1}\}$ of consecutive nodes in the sequence constitutes a link in network N.

The *length* of a path $(u_1, u_2, \ldots, u_r)$ in a network is $r - 1$. For example, the length of the path (u_1) is 0, the length of the path (u_1, u_2) is 1, and so on.

Let P denote a set of nodes in a network N and let K be a non-negative integer. Set P is called K-*observer* of N iff every path of length at least K in N has at least one node in P.

A K-observer P of a network N is called *minimal* iff for each node u in P, N has a path q of length at least K such that P and q share only one node and their shared node is u.

A minimal K-observer P of a network N is called *minimum* iff for every minimal K-observer Q of network N, the number of nodes in Q is at least the number of nodes in P.

Let K be a non-negative integer. The K-*observer problem* is to design an algorithm that takes as input any network N and produces as output a minimum K-observer of network N.

An algorithm that solves the K-observer problem, when $K = 0$, can be designed as follows. This algorithm takes any network N as input and produces the set of all nodes in N as output. The correctness of this algorithm is based on the observation that the only 0-observer (and so the only minimum 0-observer) of a network N is the set of all nodes in N.

Unfortunately, the following theorem states that the K-observer problem, for any given constant $K > 0$, is NP-hard, so any algorithm to solve this problem is very likely to be expensive.

Theorem 1. *The K-observer problem, for any given constant value of K that is greater than 0, is NP hard.*

Proof. We begin our proof by noting that, for the case where $K = 1$, the K-observer problem reduces to choosing a minimum number of nodes such that every path of 1 or more hops i.e. every edge or simple path in the graph has at least one chosen node. This is the well-known minimum vertex cover problem, which is of course NP-hard [9].

We will now prove the NP-hardness of the general case by contradiction. If the K-observer problem is solvable in polynomial time for any value of K greater than 1 (say $K = k$), then minimum vertex cover is also solvable in polynomial time.

Suppose the K-observer problem is polynomial-time solvable for $K = k$. Given a network N, we can find its minimum vertex cover as follows.

To each node u of N, we attach a "string" of nodes $u_1, u_2, ...u_{k-1}$. These nodes have no edges incident on them, except the edges that link them to form the string. We call this modified network, consisting of N as well as the new nodes and edges added to form strings, network N'. The node on a string that belongs to N is called the base node of the string.

We now run the polynomial-time algorithm to find the minimum k-observer of N'. Note the size of N' is k times, i.e. a constant times the size of N, so this runs in time polynomial in the size of N also.

Further, we note that as this is the minimum k-observer, it will not contain more than one member of each 'string' $(u, u_1, ...u_{k-1})$. (If there is more than one member, the one farther away from the base node i.e. u can be unmarked without breaking the condition that all paths of length k or more have at least one marked node, so the solution is not minimal and not minimum.) We now apply a simple linear time transformation: if any node in a string is marked, we unmark it and mark the base node of the string.

The set of marked nodes is a minimum K-observer of N', because it is still a K-observer (any path using edges from the original N that had a marked node on it, still has a marked node on it, and any path not using edges from N has a maximum length of $k-1$) and it has the same number of nodes as the minimum K-observer found by our polynomial-time algorithm. It is also a vertex cover of N (because if there is an edge $\{u, v\}$ in N such that neither u nor v is marked, then there is a path $\{u, v, ...v_{k-1}\}$ of length k in N' without any marked nodes, which is impossible). Thus, we have a vertex cover of size m', where m' is the size of the minimum k-observer of N'.

Now we note that any vertex cover of N is a k-observer of N', as any path of length k or more must use at least one edge in N and thus contain a marked node. So if the size of the minimum vertex cover of N is m and the minimum k-observer of N' is m', we have $m \geq m'$. (In other words, the minimum vertex cover of N being a k-observer of N' is at least as large as the minimum k-observer of N'.)

Hence the vertex cover we compute, being of size m' and thus $\leq m$, is in fact the minimum vertex cover (because it is a vertex cover, upper bounded

by the size of the minimum vertex cover). Thus, if the K-observer problem is solvable for general graphs for $K = k > 1$, we have a polynomial time algorithm for the minimum vertex cover problem, which is NP hard; in other words, the K-observer problem is NP hard for all $K > 0$.

Having shown that the K-observer problem is NP hard for general networks, we present next polynomial time (in fact linear time) algorithms for solving this problem for special classes of networks such as tree networks, ring networks, and grid networks.

3 *K*-Observers of Tree Networks

In this chapter, we solve the K-observer problem for tree networks that have no cycles. Figure 1 shows an example of a tree network T that has 13 nodes, named node 0 to node 12, and 12 links.

Next, we describe an algorithm that computes a minimum K-observer for a tree network. This algorithm takes as input a tree network T and a positive integer K and returns as output a minimum K-observer P of network T. This algorithm consists of the following four steps:

Step 1:
Choose any node in T to be the root and add directions to the links in T to make T a directed tree where the root is a sink node.
Step 2:
Define for each node x in T, a variable named len_x whose range of values is $0 \ldots K$.
Step 3:
For each node x in T, where the values of the len variables of all predecessor nodes of x have already been computed, compute the value of len_x as follows:

$len_x := 0$	if x has no predecessor y whose $len_y < K$
$:= len_y + 1$	if x has exactly one predecessor y whose $len_y < K$
$:= len_y + 1$	if x has two or more predecessors $\{y, z, \ldots\}$ whose $len's < K$ and len_y is the maximum len among those predecessors and len_z is the second maximum len among those predecessors and $(len_y + len_z + 2) < K$
$:= K$	if x has two or more predecessors $\{y, z, \ldots\}$ whose $len's < K$ and len_y is the maximum len among those predecessors and len_z is the second maximum among those predecessors and $(len_y + len_z + 2) >= K$

Step 4:
A minimum K-observer of tree network T is the set of every node x in T where $len_x = K$.

Theorem 2. *If this algorithm is applied to a tree network T, and a positive integer K, then the computed set by this algorithm is a minimum K-observer of network T.*

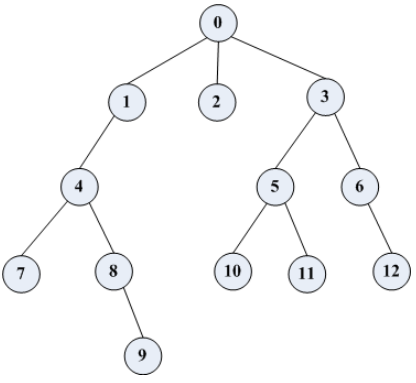


Fig. 1. Example of a tree network T

To see how this algorithm works, consider the tree network T in Figure 1. In step 1, we choose node 5 to be the root and we make the root a sink node such that T becomes a directed tree as shown in Figure 2(a). In step 2, we define a variable named len_x for every node x in T. In step 3, we compute len_x for every node x in T. In step 4, we show that the 2-observer of minimal cardinality of T is $\{4, 0, 3, 5\}$ in Figure 2(b). In addition to that, we show that the 3-observer of minimal cardinality of T is $\{4, 3\}$ in Figure 2(c).

Interestingly, the algorithm works irrespective of the choice of the root. Even if a leaf node is chosen as the root, the algorithm still correctly computes minimum K-observers in tree networks.

The time complexity of this algorithm is linear such that it is proportional to the number of nodes in the input tree network.

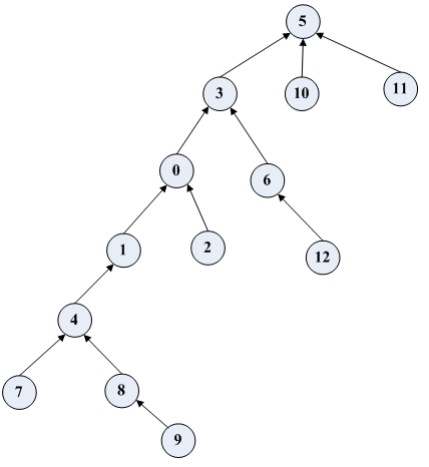
This algorithm as described above is centralized. But a distributed version of this algorithm can be described as follows:

Step 1:
Each node x, that has exactly one neighboring node y in the tree network T, knows that it is a leaf in T and so it assigns its variable len_x the value 0 and sends the value of its len_x to node y.

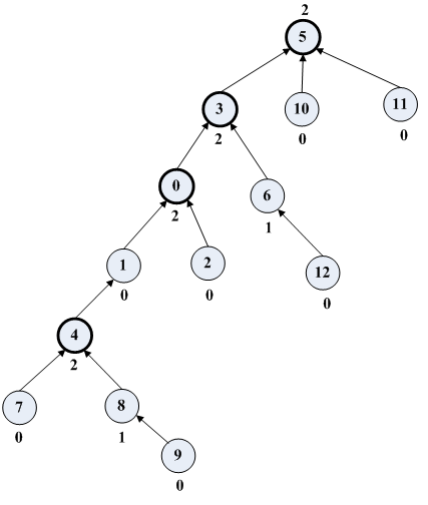
Step 2:
After a node y receives the value of len_x from every neighboring node x except one, say node z, then node y computes the value of its variable len_y (as described in the centralized version of the algorithm) and sends the computed value of len_y to node z.

Step 3:
If a node z receives the value of len_y from every neighboring node y, then node z recognizes that it is the root and computes the value of its variable len_z(as described in the centralized version of the algorithm).

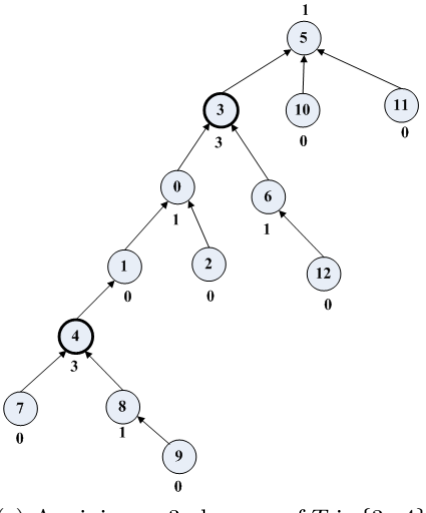
Step 4:
Every node x, where the value of len_x in K, knows that it is in the computed minimum K-observer P of the tree network T.



(a) Making T a directed tree by choosing node 5 to be the root



(b) A minimum 2-observer of T is $\{0,\ 3,\ 4,\ 5\}$



(c) A minimum 3-observer of T is $\{3,\ 4\}$

Fig. 2. Computing minimum K-observers of T

It is possible that during the execution of this distributed algorithm, two adjacent nodes y and z compute the values of their respective len variables and send their computed values to one another. Thus, each of these two nodes first sends its len value to the other node then receives the len value of the other node. In this case, the two nodes y and z behave differently depending on whether or not index y is larger than index z as follows:

i. The node with the larger index, say node y, ignores the value of variable len_z that it receives from $node_z$ and keeps the value of its variable len_y unchanged.
ii. The node with the smaller index, node z, recognizes that it is the root and uses the value of variable len_y received from node y to re-compute the value of its variable len_z.

4 *K*-Observers of Ring Networks

In this chapter, we solve the K-observer problem for ring networks. Figure 3 shows a *ring network* with n nodes, named $u_1, u_2, \ldots, u_n$, and n links.

Next, we describe an algorithm that computes a minimum K-observer for a ring network. This algorithm takes as input a ring network R and a positive integer K and returns as output a minimum K-observer P of network R. This algorithm consists of the following three steps:

Step 1:
Initially, $P := \emptyset$ (the empty set)
Step 2:
if R has at most K nodes **then**
return the empty K-observer P
terminate the algorithm
else
/* R has at least $K + 1$ nodes */
continue the algorithm
end if
Step 3:
remove any node, say node u, and its two incident links from network R
/* the resulting network is a tree */
apply the algorithm in Chapter 3 to compute a minimum K-observer Q of the resulting tree network
$P := Q \cup \{u\}$
return P and terminate the algorithm

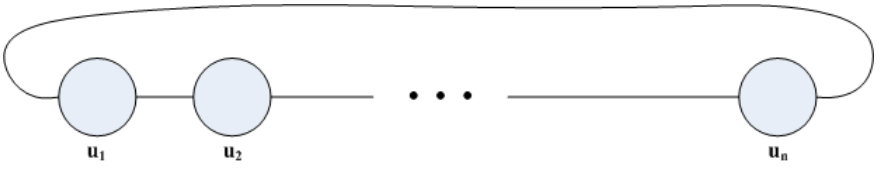


Fig. 3. A ring network of n nodes

Theorem 3. *If this algorithm is applied to a ring network R, and a positive integer K, then the computed set P by this algorithm is a minimum K-observer of network R.*

Note that the time complexity of this algorithm is linear in the number of nodes in the input ring network R.

Note also that if Step 2 is removed from the above algorithm, then the computed K-observer P of ring R may no longer be minimum. In this case, however, the number of nodes in P is no more than one over the number of nodes in a minimum K-observer of R. This suggests the following definition.

Let L be a non-negative integer. A K-observer P of a network N is called *L-bounded* iff the number of nodes in P is no more than L of the number of nodes in a minimum K-observer of network N. (Note that a minimum K-observer of a network N can now be regarded as a 0-bounded K-observer of N.)

Next, we describe a linear time algorithm that can be used to construct an L-bounded K-observer for a special class of networks called L-rings.

Let L be a positive integer. A network RR is called an *L-ring* iff RR has exactly L ring sub-networks. Note that if every ring sub-network in an L-ring is collapsed into a single super node then the resulting network is a tree.

An algorithm, for constructing an L-bounded K-observer P for any given L-ring RR, is as follows. First, identify a node in every ring sub-network in the given L-ring RR. Let $u_1, \ldots, u_x$ denote the identified nodes, where $x \leq L$. Second, remove the identified nodes and their incident links from the given L-ring RR. Note that the resulting network is a tree. Third, apply the algorithm in Chapter 3 to compute the minimum K-observer Q of the resulting tree. Compute the L-bounded K-observer P of the given L-ring RR as follows: $P := Q \cup \{u_1, \ldots, u_L\}$.

5 *K*-Observers of Large Grid Networks

In this chapter, we discuss how to construct minimal K-observers for large grid networks.

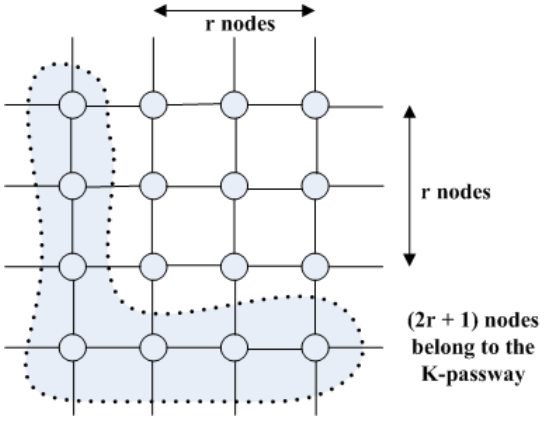


Fig. 4. A section of $(r + 1) \times (r + 1)$ nodes in a large grid network

Let d be a large positive integer. A *d-large grid network* is a network that has d^2 nodes partitioned into 4 *corner nodes*, $4d-8$ *border nodes*, and *middle nodes*. Each corner node has 2 incident links, each border node has 3 incident links, and each middle node has 4 incident links.

Next, we present a construction of a minimal K-observer of a d-large grid network D. An interesting point about this construction is that the number of nodes in the constructed K-observer is $O(d)$ when the value of K is $O(d^2)$.

Our construction takes as input a d-large grid network D and a positive integer K and produces as output a minimal K-observer P of network D. The number of nodes in the constructed P is $O(d^2/\lfloor\sqrt{K}\rfloor)$. The construction proceeds in the following three steps.

Step 1:
Initially, P is the empty set.
Step 2:
Partition the nodes in network D into sections, where each section has $(r+1)^2$ nodes as shown in Figure 4. In Step 3 below, we argue that the value of r is $\lfloor\sqrt{K}\rfloor$. From each section, add the $2r+1$ nodes, that form an L-shape in the section, into set P. Finally, remove the corner node of the L-shape from P.
Step 3:
Any path in network D, whose nodes are not in set P, must be confined to a single section in D. Thus, the length of the longest path in D, whose nodes are not in P, is r^2-1. Therefore, in order to make P a K-observer of D, the value of r needs to be $\lfloor\sqrt{K}\rfloor$.

Theorem 4. *If this algorithm is applied to a d-large grid network D and a positive integer K, then the computed set P by this algorithm is a minimal K-observer of network D.*

Note that the time complexity of this algorithm is linear in the number of nodes in the input d-large grid network.

So far, we could only prove that the computed K-observer P of network D is minimal (not minimum). Luckily, we show next that the number of nodes in the computed P is relatively small when K is $O(d^2)$. But first we need to adopt the following notation.

$|P|$ is the number of nodes in computed set P.
$|D|$ is number of nodes in network D.
$section.P$ is the number of nodes from one section in computed P.
$section.D$ is the number of nodes in one section in network D.

Hence,

$$\begin{aligned} |P| &= (section.P * |D|)/section.D \\ &= ((2r+1) * d^2)/(r+1)^2 \\ &\approx O(d^2/r) \\ &\approx O(d^2/\lfloor\sqrt{K}\rfloor) \end{aligned}$$

Therefore, if K is $O(d^2)$, for example $K = d^2/10$, then $|P|$ is $O(d)$.

6 Applications

The concept of minimal K-observers of a network is a fundamental idea, as it is a generalization of the concept of node cover. This is illustrated by the many applications for which this concept can be put to good use. We mention some of these applications here.

First, minimal K-observers can be used in constructing optimal connectivity paths in wireless sensor networks, wireless ad-hoc networks, and wireless ad-hoc sensor networks. For example, sensor nodes in wireless sensor networks need to maintain connectivity, but use as little energy as possible [6]. A solution to the problem would be to designate some particular nodes as targets, so the other nodes need only get a message to the nearest target node; these special nodes may have access to high power or bandwidth, and can ensure the message is rapidly delivered to its final destination. If we select nodes that constitute a minimal K-observer of the network, and deploy target nodes at these positions, we can guarantee that we choose an optimal number of target nodes to ensure that every "low-power" node has a nearby target (within K hops). Similarly, it may be interesting to choose nodes constituting the minimal K-observer of a content distribution network or of a disruption tolerant network, and place caches at these nodes; each cache can serve the nodes in its "zone", so with a small number of caches we ensure that every node in the network has access to a nearby cache.

Secondly, a minimal K-observer of a network can be used to measure and monitor the network traffic, as it can be considered to form a good number of well spread-out points at which to measure the parameters of the network, such as packet loss and packet delay. Such an arrangement of monitors can also be used for fine-tuning a network; for example, a service provider might deploy monitors at the nodes of a K-observer in order to identify usage accounting or bottlenecks.

Thirdly, minimal K-observers can be used to detect and prevent malicious attacks in the Internet. Thus, if we solve the K-observer problem for a computer network, it can help us to find the optimal places to place firewalls or filters in the network [1], to prevent IP prefix hijacking [15], or to defend against distributed denial of service (DDOS) attacks [3]. Placing "sentry" nodes forming a minimal K-observer ensures that no localized attack can cascade through the network without being detected. Determining the minimal K-observer of a network helps set up many countermeasures to limit the damage done by an attacker. Moreover, it can be used for digital forensics, to track down the adversary after an attack.

It is natural to ask, given the scope of the K-observer problem, whether there is any way to get around the fact that it is intractable in general. In this regard, we note that there exist interesting network topologies (such as trees, rings, and grids) for which the problem is tractable, so we argue that it should be possible to take advantage of various such special cases and compute minimal K-observers (or good approximations thereof) for many networks of practical importance. The problem of identifying more networks for which finding the

minimal K-observer is tractable, as well as finding good heuristics and approximation algorithms, leaves scope for considerable future research.

The other question that must be answered for practical use of the K-observer problem is, "what is the proper choice of K for this application?" It is clear that, the smaller the value of K, the better the coverage (outreach, sensitivity etc.) will be, but as it will require more observer nodes, the solution will be more expensive. The choice of K is dependent upon the domain. In our future work, we intend to extend our study to 'adaptive' systems, where there are many observers but only a small number (K-observer for large K) are active by default; in case an interesting phenomenon is detected, more and more observers are activated (reducing the value of K), improving the resolution precisely when better coverage is required.

7 Concluding Remarks

This paper introduces the concept of minimum K-observers of computer networks. Our primary idea is that the nodes in a minimum K-observer of a network N can be employed as observers of network N. The problem of constructing a minimum K-observer for a general network is NP-hard (Section 2), but we demonstrate that this problem can be solved in linear-time for some special classes of networks: trees (in Section 3), rings (in Section 4), and large grids (in Section 5). Whether the K-observer problem can be solved for Internet-like networks, as specified in [8] and [5], remains an open, but important, question. The K-observer problem is useful in deciding the placement of firewalls, sentry nodes to detect attacks, and so on, and it would be an important contribution to develop algorithms to solve the K-observer problem (at least near-optimally) for networks in the Internet.

There is scope for considerable further work in identifying interesting classes of network for which the K-observer problem is tractable. An important open question that follows from our work is how to devise good heuristics and approximation algorithms for the K-observer problem. Furthermore, it may be noted that we have developed our theory for the K-observer problem under the assumption that the network is stable in time and does not change; it would be interesting to investigate solutions to the problem – and, indeed, to check if the problem is solvable – for time-dependent networks, such as mobile and vehicular networks.

We find that the potential applications for the K-observer problem can be extended from the optimal connectivity for energy efficiency in wireless networks to the efficient node placement for traffic and to the detection and the prevention of network attacks in Section 6. We foresee that the K-observer problem will be helpful to understand and solve many problems in computer networks.

Acnowledgements. The authors would like to express their gratitude to the anonymous reviewer who helped find two major improvements to the paper, and to Mr Keshav Dhandhania of MIT for his help.

References

1. Acharya, H.B., Joshi, A., Gouda, M.G.: Firewall modules and modular firewalls. In: Proceedings of the 18TH IEEE International Conference on Network Protocols, ICNP 2010 (2010)
2. Alzoubi, K.M., Wan, P.-J., Frieder, O.: Message-optimal connected dominating sets in mobile ad hoc networks. In: Proceedings of the 3rd ACM International Symposium on Mobile ad Hoc Networking & Computing, MobiHoc 2002, pp. 157–164. ACM, New York (2002)
3. Armbruster, B., Smith, J.C., Park, K.: A packet filter placement problem with application to defense against spoofed denial of service attacks. European Journal of Operational Research (2007)
4. Breslau, L., Diakonikolas, I., Duffield, N., Gu, Y., Hajiaghayi, M., Johnson, D.S., Karloff, H., Resende, M., Sen, S.: Disjoint-path facility location: Theory and practice. In: Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments, ALENEX (2011)
5. Calvert, K., Doar, M.B., Nexion, A., Zegura, E.W.: Modeling internet topology. IEEE Communications Magazine 35, 160–163 (1997)
6. Chen, B., Jamieson, K., Balakrishnan, H., Morris, R.: Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. Wirel. Netw. 8, 481–494 (2002)
7. Chvátal, V.: A combinatorial theorem in plane geometry. Journal of Combinatorial Theory, Series B, 39–41 (1975)
8. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On power-law relationships of the internet topology. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 1999, pp. 251–262. ACM, New York (1999)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
10. Huang, C.-F., Tseng, Y.-C.: The coverage problem in a wireless sensor network. In: Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications, WSNA 2003, pp. 115–121. ACM, New York (2003)
11. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
12. Lee, S.-J., Su, W., Gerla, M.: Wireless ad hoc multicast routing with mobility prediction. Mob. Netw. Appl. 6, 351–360 (2001)
13. Meguerdichian, S., Koushanfar, F., Potkonjak, M., Srivastava, M.B.: Coverage problems in wireless ad-hoc sensor networks. In: Prodeedings of IEEE INFOCOM, pp. 1380–1387 (2001)
14. Qiu, L., Padmanabhan, V.N., Voelker, G.M.: On the placement of web server replicas. In: Proceedings of IEEE INFOCOM, pp. 1587–1596 (2001)
15. Qiu, T., Ji, L., Pei, D., Wang, J., Xu, J.: Towerdefense: Deployment strategies for battling against ip prefix hijacking. In: Proceedings of the 18th IEEE International Conference on Network Protocols, ICNP (October 2010)
16. Roy, S., Pucha, H., Zhang, Z., Hu, Y.C., Qiu, L.: Overlay node placement: Analysis, algorithms and impact on applications. In: International Conference on Distributed Computing Systems, p. 53 (2007)

17. Wan, P.-J., Alzoubi, K.M., Frieder, O.: Distributed construction of connected dominating set in wireless ad hoc networks. Mob. Netw. Appl. 9, 141–149 (2004)
18. Weber, A.: Theory of the Location of Industries. The University of Chicago Press, Chicago (1909)
19. Wu, J., Li, H.: On calculating connected dominating set for efficient routing in ad hoc wireless networks. In: Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, DIALM 1999, pp. 7–14. ACM, New York (1999)

Pragmatic Self-stabilization of Atomic Memory in Message-Passing Systems*

Noga Alon[1], Hagit Attiya[2], Shlomi Dolev[3], Swan Dubois[4], Maria Potop-Butucaru[4], and Sébastien Tixeuil[4]

[1] Sackler School of Mathematics and Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, 69978, Israel
[2] Department of Computer Science, Technion, 32000, Israel
[3] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il
[4] LIP6, Universite Pierre et Marie Curie, Paris 6/INRIA, 7606, France

Abstract. A fault-tolerant and stabilizing simulation of an atomic register is presented. The simulation works in asynchronous message-passing systems, and allows a minority of processes to crash. The simulation stabilizes in a pragmatic manner, by reaching a long execution in which it runs correctly. A key element in the simulation is a new combinatorial construction of a bounded labeling scheme accommodating arbitrary labels, including those not generated by the scheme itself.

1 Introduction

Distributed systems have become an integral part of virtually all computing systems, especially those of large scale. These systems must provide high availability and reliability in the presence of failures, which could be either permanent or transient. A core abstraction for many distributed algorithms *simulates shared memory* [3]; this abstraction allows to take algorithms designed for shared memory, and port them to asynchronous message-passing systems, even in the presence of failures. There has been significant work on creating such simulations, under various types of permanent failures, as well as on exploiting this abstraction in order to derive algorithms for message-passing systems. (See a recent survey [2].)

All these works, however, only consider permanent failures, neglecting to incorporate mechanisms for handling *transient* failures. Such failures may result from incorrect initialization of the system, or from temporary violations of the assumptions made by

* Research of the first author supported in part by an ERC advanced grant, by a USA-Israeli BSF grant, by the Israel Science Foundation. Research of the second author supported in part by the *Israel Science Foundation* (grants number 953/06 and 1227/10). The work started while the second author was visiting EPFL, and the third author was a visiting professor at LIP6. Research supported in part by the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Microsoft, Deutsche Telekom, US Air-Force, *Israel Science Foundation* (grant number 428/11), and Rita Altura Trust Chair in Computer Sciences.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 19–31, 2011.

the system designer, for example the assumption that a corrupted message is always identified by an error detection code. The ability to automatically resume normal operation following transient failures, namely to be *self-stabilizing* [9], is an essential property that should be integrated into the design and implementation of systems.

This paper presents a stabilizing simulation of an atomic register in asynchronous message-passing systems where a minority of processors may crash. The simulation is based on reads and writes to a (majority) quorum in a system with a fully connected graph topology. A key component of the simulation is a new bounded labeling scheme that needs no initialization, as well as a method for using it when communication links and processes are started at an arbitrary state. To the best of our knowledge, our scheme is the first constructive labeling scheme presenting the above properties.

Overview of our simulation. Attiya, Bar-Noy and Dolev [3] showed how to simulate a single-writer multi-reader (SWMR) atomic register in a message-passing system, supporting two procedures, read and write, for accessing the register. This simple simulation is based on a quorum approach: In a write operation, the writer makes sure that a quorum of processors (consisting of a majority of the processors, in its simplest variant) store its latest value. In a read operation, a reader contacts a quorum of processors, and obtains the latest values they store for the register; in order to ensure that other readers do not miss this value, the reader also makes sure that a quorum stores its return value.

A key ingredient of this scheme is the ability to distinguish between older and newer values of the register; this is achieved by attaching a *sequence number* to each register value. In its simplest form, the sequence number is an unbounded integer, which is increased whenever the writer generates a new value. This solution is appropriate for an *initialized* system, which starts in a consistent configuration, in which all sequence numbers are zero, and are only incremented by the writer or forwarded as is by readers. Pragmatically, a 64-bit sequence number will not wrap around for a number of writes that lasts longer than the life-span of any reasonable system.

However, when there are transient failures in the system, as is the case in the context of self-stabilization, the simulation starts at an uninitialized state, where sequence numbers are not necessarily all zero. It is possible that, due to a transient failure, the sequence numbers hold maximal values when the simulation starts running, and thus, will wrap around very quickly. Traditionally, techniques like distributed reset [5, 6] are used to overcome this problem. However, in asynchronous crash-prone environments the reset may not terminate waiting for the crashed processes to participate. Hence, a reset invocation will not ensure that the sequence numbers are set to zero.

Our solution is to partition the execution of the simulation into *epochs*, namely periods during which the sequence numbers are supposed not to wrap around. Whenever a "corrupted" sequence number is discovered, a new epoch is started, overriding all previous epochs; this repeats until no more corrupted sequence numbers are hidden in the system, and the system stabilizes. In a steady state, after the system stabilizes, it remains in the same epoch (at least until the sequence number wrap around, which is unlikely to happen).

This raises, naturally, the question of identifying epochs. The natural idea, of using integers, is bound to run into the same problems as for the sequence numbers. Instead, we use a *bounded labeling scheme* [14, 18] for the epochs; this is a function for generating

labels (in a bounded domain), that guarantees that two labels can be compared to determine the largest among them. Existing labeling schemes, however, assume that labels have specific initial values, and that new labels are introduced only by means of the label generation function. In contrast, transient failures, of the kind the self-stabilizing simulation must withstand, can create incomparable labels, so it is impossible to tell which is the largest among them or to pick a new label that is bigger than all of them.

To address this difficulty, we introduce a bounded labeling scheme that allows to define a label larger than *any set* of labels, provided that its size is bounded. We assume links have bounded capacity, and hence the number of epoch labels initially hidden in the system is bounded.

The writer tracks the set of epoch labels it has seen recently; whenever the writer discovers that its current epoch label is not the largest, or is incomparable to some existing epoch label, the writer generates a new epoch label that is larger than all the epoch labels it has. The number of bits required to represent an epoch label depends on m, the maximal size of the set, and it is in $O(m \log m)$. We ensure that the size of the set is proportional to the total capacity of the communication links, namely, $O(cn^2)$, where c is the bound on the capacity of each link (expressed in number of messages) and n is the number of processors, and hence, each epoch label requires $O((cn^2(\log n + \log c))$ bits.

It is possible to reduce this complexity, making c constant, using a self-stabilizing data-link protocols for communication among the processors for bounded capacity links over FIFO and non-FIFO communication links [10, 15].[1]

We show that, after a bounded number of write operations, the results of reads and writes can be totally ordered in a manner that respects the real-time order of non-overlapping operations, so that the sequence of operations satisfies the semantics of a SWMR register. This holds until the sequence numbers wrap around, as can happen when the unbounded simulation of [3] is deployed in realistic systems, where all values are bounded.

Note that the original design of [3] copes with non-FIFO and unreliable links. We assume that our atomic register simulation runs on top of an optimal stabilizing data-link layer that emulates a reliable FIFO communication channel over unreliable capacity bounded non-FIFO channels [10].

Related work. Self-stabilizing simulation of an single-writer *single-reader* atomic shared register in a message-passing system was presented in [12]. This simulation does not tolerate processor crashes. More recent papers [11, 19] focused on self-stabilizing simulation of shared registers from weaker shared registers. Self-stabilizing timestamps implementations using SWMR atomic registers were suggested in [1, 13]. These simulations already assume the existence of a shared memory, while, in contrast, we simulate a shared SWMR atomic register in a message-passing system.

2 Preliminaries

A *message-passing system* consists of n *processors*, $p_0, p_1, p_2, \ldots, p_{n-1}$, connected by *communication links* through which messages are sent and received. We assume that

[1] Note that these protocols are also snap-stabilizing—starting in an arbitrary configuration, the first invoked send operation succeeds to deliver the message.

the underlying communication graph is completely connected, namely, every pair of processors, p_i and p_j, have a communication link of bounded capacity c.

A processor is modeled as a state machine that executes *steps*. In each step, the processor changes its state, and executes a single communication operation, which is either a *send* message operation or a *receive* message operation. The communication operation changes the state of an attached link, in the obvious manner.

The system *configuration* is a vector of n states, a state for each processors and $2(n^2 - n)$ queues, each bounded by a constant capacity c. Note that in the scope of self-stabilization, where the system copes with an arbitrary starting configuration, there is no deterministic data-link simulation that use bounded memory when the capacity of links is unbounded [12]. Note further that non-FIFO communication links can be accommodated by mimicking FIFO delivery [10].

An *execution* is a sequence of configurations and steps, $E = (C_1, a_1, C_2, a_2 \ldots)$ such that C_i, $i > 1$, is obtained by applying a_{i-1} to C_{i-1}, where a_{i-1} is a step of a single processor, p_j, in the system. Thus, the vector of states, except the state of p_j, in C_{i-1} and C_i are identical. If the single communication operation in a_{i-1} is a send operation from p_j to processor p_k then s_{jk} in C_i is obtained from s_{jk} in C_{i-1} by enqueuing the message sent in a_{i-1}. If the resulting queue s_{jk} exceeds its size, i.e., $|s_{jk}| = c$, then an arbitrary message is deleted from s_{jk}. The rest of the message queues are unchanged. If the single communication operation in a_{i-1} is a receive operation of a (non null) message M, then M (which is the first message to be dequeued from s_{kj} in C_{i-1}) is removed from s_{kj}, all the other queues are unchanged. A receive operation by p_j from p_k may result in a null message even when the s_{kj} is not empty, thus allowing unbounded delay for any particular message. Message losses are modeled by allowing spontaneous message removals from (any place in) the queue. An edge (i, j) is operational if a message sent infinitely often by p_i is received infinitely often by p_j.

Atomic register. For the simulation of a *single writer multi-reader* (SWMR) atomic register, we assume p_0 is the writer and $p_1, p_2, \ldots, p_{n-1}$ are the readers. There is a procedure for executing a write operation by p_0, and procedures for executing read operations by the readers.

Each invocation of a read or write operation translates into a sequence of computation steps, following the appropriate procedure. Concurrent invocations of read and write operations yield an execution in which the computation steps corresponding to invocations by different processors are interleaved. An operation op_1 *precedes* an operation op_2 in this execution, if op_1 returns before op_2 is invoked. Two operations *overlap* if neither of them precedes the other.

Each interleaved execution of an atomic register is required to be *atomic*, namely, equivalent to an execution in which the operations are executed sequentially and the order of non-overlapping operations is preserved [4]. As advocated in [7], the above definition is equivalent to say that the atomic register has to satisfy the following two properties:

- ***Regularity.*** A read operation returns either the value written by the most recent write operation that completes before the read or a value written by a concurrent write.

– ***No new / old inversions.*** If a read operation R returns the value of a concurrent write operation W, then no read operation that is started after R completes returns the value of a write operation that completes before W starts.

Pragmatically stabilizing atomic register. A message passing system simulates an atomic register is a r-pragmatically stabilizing, if there exist an integer $r' > r$, such that every execution with r' write operations has a segment of execution (fragment) with r write operations that satisfies the atomicity requirements. In particular, a large r implies the existence of a long segment with the desired behavior. In the sequel, when no confusing is possible we refer to r-pragmatically stabilizing simply as pragmatically stabilizing.

Pragmatic stabilization is reminiscent of *pseudo-stabilization* [9] in the sense that an execution has a finite number of specification violation during a long execution; in pseudo-stabilization the length of the long execution is infinite while in pragmatic stabilization the length considered is practically infinite. Roughly speaking, the use of the pigeonhole principle ensures that a partition of r' by the bound on the number of violations ensures the existence of r.

3 Overview of the Algorithm

3.1 The Basic Quorum-Based Simulation

We describe the basic simulation, which follows the quorum-based approach of [3], and ensures that our algorithm tolerates (crash) failures of a minority of the processors.

The simulation relies on a set of *read and write quorums*, each being a majority of processors.[2] The simulation specifies the write and read procedures, in terms of QuorumRead and QuorumWrite operations. The QuorumRead procedure sends a request to every processor, for reading a certain local variable of the processor; the procedure terminates with the obtained values, after receiving answers from processors that form a quorum. Similarly, the QuorumWrite procedure sends a value to every processor to be written to a certain local variable of the processor; it terminates when acknowledgments from a quorum are received. If a processor that is inside QuorumRead or QuorumWrite keeps taking steps, then the procedure terminates (possibly with arbitrary values). Furthermore, if a processor starts QuorumRead procedure execution, then the stabilizing data link [15, 16] ensures that a read of a value returns a value held by the read variable some time during its period; similarly, a QuorumWrite(v) procedure execution, causes v to be written to the variable during its period.

Each processor p_i maintains a variable, $MaxSeq_i$, supposed to be the "largest" sequence number the processor has read, and a value v_i, associated with $MaxSeq_i$, which is supposed to be the value of the implemented register.

The write procedure of a value v starts with a QuorumRead of the $MaxSeq_i$ variables; upon receiving answers $l_1, l_2, \ldots$ from a quorum, the writer picks a sequence number l_m that is larger than $MaxSeq_0$ and $l_1, l_2, \ldots$ by one; the writer assigns l_m to

[2] Standard end-to-end schemes [17] can be used to implement the quorum operation in the case of general communication graph.

$MaxSeq_0$ and calls QuorumWrite with the value $\langle l_m, v\rangle$. Whenever a quorum member p_i receives a QuorumWrite request $\langle l, v\rangle$ for which l is larger than $MaxSeq_i$, p_i assigns l to $MaxSeq_i$ and v to v_i.

The read procedure by p_i starts with a QuorumRead of both the $MaxSeq_j$ and the (associated) v_j variables. When p_i receives answers $\langle l_1, v_1\rangle$, $\langle l_2, v_2\rangle \ldots$ from a quorum, p_i finds the largest epoch label l_m among $MaxSeq_i$, and $l_1, l_2, \ldots$ and then calls QuorumWrite with the value $\langle l_m, v_m\rangle$. This ensures that later read operations will return this, or a later, value of the register. When QuorumWrite terminates, after a write quorum acknowledges, p_i assigns l_m to $MaxSeq_i$ and v_m to v_i and returns v_m as the value read from the register.

Note that the QuorumRead operation, beginning the write procedure of p_0, helps to ensure that $MaxSeq_0$ holds the maximal value, as the writer reads the biggest *accessible* value (directly read by the writer, or propagated to a quorum that will be later read by the writer) in the system during any write.

Let $g(C)$ be the number of distinct values greater than $MaxSeq_0$ that are accessible in some configuration C, and let $C_1, C_2, \ldots$ be the configurations in the execution. Since all the processors, except the writer, only copy values and since p_0 can only increment the value of $MaxSeq_0$ it holds for every $i \geq 1$ that

$$g(C_i) \geq g(C_{i+1}) \ .$$

Furthermore,

$$g(C_i) > g(C_{i+1}) \ ,$$

whenever the writer discovers (when executing step a_i) a value greater than $MaxSeq_0$. Roughly speaking, the faster the writer discovers these values, the earlier the system stabilizes. If the writer does not discover such a value, then the (accessible) portion of the system in which its values are repeatedly written, performs reads and writes correctly.

3.2 Epochs

As described in the introduction, it is possible that the sequence numbers wrap around faster than planned, due to "corrupted" initial values. When the writer discovers that this has happened, it opens a new *epoch*, thereby invalidating all sequence numbers from previous epochs.

Epochs are denoted with labels from a bounded domain, using a *bounded labeling scheme*. Such a scheme provides a function to compute a new label, which is "larger" than any given set of labels.

Definition 1. *A* labeling scheme *over a bounded domain $\mathcal{L}$, provides an antisymmetric comparison predicate $\prec_b$ on $\mathcal{L}$ and a function $\mathbf{Next}_b(S)$ that returns a label in $\mathcal{L}$, given some subset $S \subseteq \mathcal{L}$ of size at most m. It is guaranteed that for every $L \in S$, $L \prec_b \mathbf{Next}_b(S)$.*

Note that the labeling scheme of [18], used in the original atomic memory simulation [3], cannot cope with transient failures. Section 4 describes a bounded labeling

scheme that accommodates badly initialized labels, namely, those not generated by using $\mathbf{Next}$.

This scheme ensures that if the writer eventually learns about all the epoch labels in the system, it will generate an epoch label greater than all of them. After this point, any read that starts after a write of v is completed (written to a quorum) returns v (or a later value), since the writer will use increasing sequence numbers. The eventual convergence of the labeling scheme depends on invoking $\mathbf{Next}_b$ with a parameter S that is a superset of the epoch labels in the system. Estimating this set is another challenge for the simulation, as described next.

Guessing Game. We explain the intuition of this part of the simulation through the following two-player *guessing game*, between a *finder*, representing the writer, and a *hider*, representing an adversary controlling the system.

- The hider maintains a set of labels $\mathcal{H}$, whose size is at most m (a parameter fixed later).
- The finder does not know $\mathcal{H}$, but it needs to generate a label greater than all labels in $\mathcal{H}$.
- The finder generates a label L and if $\mathcal{H}$ contains a label L', such that it does not hold that $L' \prec_b L$ then the hider exposes L' to the finder.
- In this case, the hider may choose to add L to $\mathcal{H}$, however, it must ensure that the size of $\mathcal{H}$ remains at most m, by removing another label. (The finder is unaware of the hiders decision.)
- If the hider does not expose a new label L' from $\mathcal{H}$, the finder wins this iteration and continues to use L.

The strategy of the finder is based on maintaining a FIFO queue of $2m$ labels, meant to track the most recent labels. The queue starts with arbitrary values, and during the course of the game, it holds up to m recent labels produced by the finder, which turned out to be overruled by existing labels (provided by the hider). The queue also holds up to m labels that were revealed to overrule these labels.

Before the finder chooses a new label, it enqueues its previously chosen label and the label received from the hider in response. Enqueuing a label that is already in the queue pushes the label to the front of the queue; if the bound on the size of the queue is reached, then the oldest label in the queue is dequeued. *This semantics of enqueue is used throughout the paper.*

The finder chooses the next label by applying $\mathbf{Next}$, using as parameter the $2m$ labels in the queue. Intuitively, the queue eventually contains a superset of $\mathcal{H}$, and the finder generates a label greater than all the current labels of the hider.

Clearly, when the finder chooses the ith label, $i > 0$, the $2i$ items in the front of the queue consist of the first i labels generated by the finder, and the first i labels revealed by the hider. This is used to show the following property of the game.

Lemma 1. *After at most $m + 1$ labels, the finder generates a label that is larger than all the labels held by the hider.*

Proof. Note that a response cannot expose a label that has been introduced or previously exposed in the game since the finder always choose a label greater than all labels in the

queue. Thus, if the finder does not win when introducing the mth label, all the m labels that the hider had when the game started were exposed and therefore, stored in the queue of the finder together with all the recent m labels introduced by the finder, before the $m+1$st label is chosen. Therefore, the $m+1$st label is larger than every label held by the hider, and the finder wins. □

Note that a step of the hider that exposes more than one label unknown to the finder, accelerates the convergence to a winning stage.

4 A Bounded Labeling Scheme with Uninitialized Values

Let $k > 1$ be an integer, and let $K = k^2 + 1$. We consider the set $X = \{1, 2, .., K\}$ and let $\mathcal{L}$ (the set of labels) be the set of all ordered pairs (s, A) where $s \in X$ is called in the sequel the *Sting* of the label, and $A \subseteq X$ has size k and is called in the sequel the *Antistings* of the label. It follows that $|\mathcal{L}| = \binom{K}{k} K = k^{(1+o(1))k}$.

The comparison operator $\prec_b$ among the bounded labels is defined to be:

$$(s_j, A_j) \prec_b (s_i, A_i) \equiv (s_j \in A_i) \wedge (s_i \notin A_j)$$

Note that this operator is antisymmetric by definition, yet may not be defined for every pair (s_i, A_i) and (s_j, A_j) in $\mathcal{L}$ (e.g., $s_j \in A_i$ and $s_i \in A_j$).

We define now a function to compute, given a subset S of at most k labels of $\mathcal{L}$, a new label which is greater (with respect to $\prec_b$) than every label of S. This function, called $\mathbf{Next}_b$ (see the left side of Figure 1) is as follows. Given a subset of k labels $(s_1, A_1), (s_2, A_2), \ldots, (s_k, A_k)$, we take a label (s_i, A_i) that satisfies:

- s_i is an element of X that is not in the union $A_1 \cup A_2 \cup \ldots \cup A_k$ (as the size of each A_s is k, the size of the union is at most k^2, and since X is of size $k^2 + 1$ such an s_i always exists).
- A_i is a subset of size k of X containing all values $(s_1, s_2, \ldots, s_k)$ (if they are not pairwise distinct, add arbitrary elements of X to get a set of size exactly k).

It is simple to compute A_i and s_i given a set S with k labels, and can be done in time linear in the total length of the labels given, i.e., in $O(k^2)$ time.

Lemma 2. *Given a subset S of k labels from $\mathcal{L}$, $(s_i, A_i) = \mathbf{Next}_b(S)$ satisfies:*

$$\forall (s_j, A_j) \in S, (s_j, A_j) \prec_b (s_i, A_i)$$

Proof. Let (s_j, A_j) be an element of S. By construction, $s_j \in A_i$ and $s_i \notin A_j$, and the result follows from the definition of $\prec_b$. □

Timestamps. Each value is tagged with a *timestamp*—a pair (l, i) where l is a bounded label, and i is a sequence number, and integer between 0 and a fixed bound $r \geq 1$.

The $\mathbf{Next}_e$ operator compares between two timestamps, and is described in the right part of Figure 1. Note that in Line 3 of the code we use $\tilde{S}$ for the set of labels (with sequence numbers removed) that appear in S. The comparison operator $\prec_e$ for timestamps is:

$$(x, i) \prec_e (y, j) \equiv x \prec_b y \vee (x = y \wedge i < j)$$

In the sequel, we use $\prec_b$ to compare timestamps only by their labels (ignoring their sequence numbers).

Next$_b$

input: $S = (s_1, A_1), (s_2, A_2), \ldots, (s_k, A_k)$: labels set
output: (s, A): label
function: For any $\emptyset \neq S \subseteq X$, $pick(S)$ returns arbitrary (later defined for particular cases) element of S
1: $A := \{s_1\} \cup \{s_2\} \cup \ldots \cup \{s_k\}$
2: *while* $|A| \neq k$
3: $\quad A := A \cup \{pick(X \setminus A)\}$
4: $\quad s := pick\,(X \setminus (A \cup A_1 \cup A_2 \cup \ldots \cup A_k))$
5: *return* (s, A)

Next$_e$

input: S: set of k timestamps
output: (l, i): timestamp
1: *if* $\exists (l_0, j_0) \in S$ such that
$\quad \forall (l, j) \in S, (l, j) \neq (l_0, j_0),$
$\quad\quad (l, j) \prec_e (l_0, j_0) \wedge j_0 < r$
2: $\quad$ *then return* $(l_0, j_0 + 1)$
3: *else return* $(\mathbf{Next}_b(\tilde{S}), 0)$

Fig. 1. Next$_b$ and Next$_e$. $\tilde{S}$ is the set of labels appearing in S

5 Putting the Pieces Together

Each processor p_i, holds, in $MaxTS_i$, two fields $\langle ml_i, cl_i \rangle$, where ml_i is the timestamp associated with the last write of a value to the variable v_i and cl_i is a *canceling timestamp* possibly empty ($\perp$), which is not smaller than ml_i in the $\prec_b$ order. The canceling field is used to let the writer (finder in the game) know an evidence on the existence of unknown (non smaller) epoch label. A timestamp (l, i) is an evidence for timestamp (l', j) if and only if $l \not\prec_b l'$. When the writer faces an evidence it changes the current epoch label.

The pseudo code for the read and write procedures appears in Figure 2. Note that in Lines 2 and 10 of the write procedure, an epoch label is enqueued if and only if it is not equal to $MaxTS_0$. Note further, that $Next_e$ in Line 5 of the write procedure, first tries to increment the sequence number of the epoch label in $MaxTS_0$ and if the sequence number already equals to the upper bound r then p_0 enqueues the value of $MaxTS_0$ and uses the updated $epochs$ queue to choose a new value for $MaxTS_0$, which is a new epoch label Next$_b$($epochs$) and sequence number 0.

write$_0(v)$

1:$\langle \langle ml_1, cl_1 \rangle, v_1 \rangle, \langle \langle ml_2, cl_2 \rangle, v_2 \rangle, \cdots :=$QuorumRead
2:$\forall i$, *if* $ml_i \neq MaxTS_0.ml$ *then* enqueue$(epochs, ml_i)$
3:$\forall i$, *if* $cl_i \neq MaxTS_0.ml$ *then* enqueue$(epochs, cl_i)$
4:*if* $\forall\, l \in epochs\ l \preceq_e MaxTS_0.ml$ *then*
5: $\quad MaxTS_0 := \langle Next_e(MaxTS_0.ml \cup epochs), \perp \rangle$
6:*else*
7: $\quad$ enqueue$(epochs, MaxTS_0.ml)$
8: $\quad MaxTS_0 := \langle (Next_b(epochs), 0), \perp \rangle$
9:QuorumWrite$(\langle MaxTS_0, v \rangle)$

Upon a request of QuorumWrite $\langle l, v \rangle$
10:*if* $l \neq MaxTS_0.ml$ *then* enqueue$(epochs, l)$

read

1:$\langle \langle ml_1, cl_1 \rangle, v_1 \rangle, \langle \langle ml_2, cl_2 \rangle, v_2 \rangle, \cdots :=$QuorumRead
2:*if* $\exists m$ such that $cl_m = \perp$ *and*
3: $\quad (\forall\, i \neq m\ ml_i \prec_e ml_m$ and $cl_i \prec_e ml_m)$ then
4: $\quad\quad$ QuorumWrite$\langle ml_m, v_m \rangle$
5: $\quad\quad$ return(v_m)
6:*else* return$(\perp)$

Upon a request of QuorumWrite $\langle l, v \rangle$
7:*if* $MaxTS_i.ml \prec_e l$ and $MaxTS_i.cl \prec_e l$ *then*
8: $\quad MaxTS_i := \langle l, \perp \rangle$
9: $\quad v_i := v$
10:else *if* $l \not\prec_b MaxTS_i.ml$ and $MaxTS_i.ml \neq l$
$\quad\quad$ *then* $MaxTS_i.cl := l$

Fig. 2. write(v) and read

The write of a value v starts with a QuorumRead of the $MaxTS_i$ variables, and upon receiving answers $l_1, l_2, \ldots$ from a quorum, the writer p_0 enqueues the epoch labels of the received ml and non-$\perp$ cl which are not equal to $MaxTS_0$, to the epochs queue (Lines 1-3). The writer then computes $MaxTS_0$ to be the Next_e timestamp, namely if the epoch label of $MaxTS_0$ is the largest in the $epochs$ queue and the sequence number of $MaxTS_0$ less than r, then p_0 increments the sequence number of $MaxTS_0$ by one, leaving the epoch label of $MaxTS_0$ unchanged (Lines 4-5). Otherwise, it is necessary to change the epoch label: p_0 enqueues $MaxTS_0$ to the *epochs* queue and applies $Next_b$ to obtain an epoch label greater than all the ones in the $epochs$ queue; it assigns to $MaxTS_0$ the timestamp made of this epoch label and a zero sequence number (Lines 7-8). Finally, p_0 executes the QuorumWrite procedure with $\langle MaxTS_0, v\rangle$ (Line 9).

Whenever the writer p_0 receives (as a quorum member) a QuorumWrite request containing an epoch label that is not equal to $MaxTS_0$, p_0 enqueues the received epoch label in the *epochs* queue (Line 10). (Recall the rules for enqueuing the queue from Section 3.2.)

The read of a reader p_i starts with a QuorumRead of the $MaxTS_j$ and the (associated) v_j variables (Line 1). When p_i receives answers $\langle\langle ml_1, cl_1\rangle, v_1\rangle, \langle\langle ml_2, cl_2\rangle, v_2\rangle \ldots$ from a quorum, p_i tries to find a maximal timestamp ml_m according to the $\prec_e$ operator from among ml_i, cl_i, ml_1, cl_1, ml_2, cl_2 .... If p_i finds such maximal timestamp ml_m, then p_i executes the QuorumWrite procedure with $\langle ml_m, v_m\rangle$. Once the QuorumWrite terminates (the members of a quorum acknowledged) p_i assigns $MaxTS_i := \langle ml_m, \perp\rangle$, and $v_i := v_m$ and returns v_m as the value read from the register (Lines 2-5). Otherwise, in case no such maximal value ml_m exists, the read is aborted (Line 6).

When a quorum member p_i receives a QuorumWrite request $\langle l, v\rangle$, it checks whether both $MaxTS_i.ml \prec_b l$ and $MaxTS_i.cl \prec_b l$. If this is the case, then p_i assigns $MaxTS_i := \langle l, \perp\rangle$ and $v_i := v$ (Lines 7-9). Otherwise, p_i checks whether $l \not\prec_b MaxTS_i.ml$ and if so assigns $MaxTS_i.cl := l$ (Line 10). Note that $\perp \prec_b l$, for any l.

Diffusing labels over the data-link. Note that we assume an underlying stabilizing data-link protocol [9, 15]. The data-link protocol is used for repeatedly diffusing the value of $MaxTS$ from one processor to another. If the $MaxTS_i.cl$ of a processor p_i is $\perp$ and p_i receives from processor p_j a $MaxTS_j$ such that $MaxTS_j.ml \not\prec_b MaxTS_i.ml$ then p_i assigns $MaxTS_i.cl := MaxTS_j.ml$, otherwise, when $MaxTS_j.cl \not\prec_b MaxTS_i.ml$ then p_i assigns $MaxTS_i.cl := MaxTS_j.cl$ Note also that the writer will enqueue every diffused value that is different from $MaxTS_0.ml$ (similarly to lines 10 of the reader and the writer, where each of $MaxTS_j.ml$ and $MaxTS_j.cl$ are considered l).

6 Outline of Correctness Proof

The correctness of the simulation is implied by the game and our previous observations, which we can now summarize, recapping the arguments explained in the description of the individual components. Note that the writer may enqueue several unknown epochs in a single write operation and only then introduce a greater epoch, such a scenario will

result in a shorter winning strategy in the game as the writer gains more knowledge concerning the existing (hidden) labels before introducing a new epoch.

In the simulation, the finder/writer may introduce new epoch labels even when the hider does not introduce an evidence. We consider a timestamp (l, i) to be an evidence for timestamp (l', j) if and only if $l \not\prec_b l'$. Using a large enough bound r on the sequence number, we ensure that either there is an execution with r writes in which the finder/writer introduces new timestamps with no epoch label change, and therefore with growing sequence numbers, and well-defined timestamp ordering, or a new epoch label is frequently introduced due to the exposure of hidden unknown epoch labels. The last case follows the winning strategy described for the game.

The sequence numbers allow the writer to introduce many timestamps, exponential in the number of bits used to represent r, without storing all of them, as their epoch label is identical. The sequence numbers are a simple extension of the bounded epoch labels just as a least significant digit of a counter; this allows the queues to be proportional to the bounded number of the epoch labels in the system. Thus, either the writer introduces an epoch label greater than any one in the system, and hence will use this epoch label to essentially implement a register for an execution of r writes, or the readers never introduce some existing bigger epoch label letting the writer increment the sequence number practically infinitely often. Note that if the game continues, while the finder is aware of (a superset including) all existing epoch labels and introduces a greater epoch label, there exist an execution of r writes before a new epoch label is introduced.

In the simulation of a SWMR atomic register, following the first write of a timestamp greater than any other timestamp in the system, with a sequence number 0, to a majority quorum, any read in an execution with r writes, will return the last timestamp that has been written to a quorum. In particular, if a reader finds a timestamp introduced by the writer that is larger than all other timestamps but not yet completely written to a majority quorum, the reader assists in completing the write to a majority quorum before returning the read value.

The simulation fails when the set of timestamps does not include a timestamp greater than the rest. That is, read operations may be repeatedly aborted until the writer writes new timestamps. Moreover, a slow reader may store a timestamp unknown to the rest (and in particular to the writer) and eventually introduce the timestamp. In the first case, the convergence of the system is postponed till the writer is aware of a superset of the existing timestamps. In the second case, the system operates correctly, implementing read and write operations, until the timestamp unknown to the rest is introduced.

Each read or write operation requires $O(n)$ messages. The size of the messages is linear in the size of a timestamp, namely the sum of the size of the epoch label and $\log r$. The size of an epoch label is $O(m \log m)$ where m is the size of the *epochs* queue, namely, $O(cn^2)$, where c is the capacity of a communication link.

Note that the size of the *epochs* queue, and with it, the size of an epoch label, is proportional to the number of epoch labels that can be stored in a system configuration. Reducing the link capacity also reduces the number of epoch labels that can be "hidden" in the communication links. This can be achieved by using a stabilizing *data-link* protocol,[10, 15, 16], in a manner similar to the ping-pong mechanism used in [3].

7 Discussion

We have presented a self-stabilizing simulation of a single-writer multi-reader atomic register, in an asynchronous message-passing system in which at most half the processors may crash.

Given our simulation, it is possible to realize a self-stabilizing *replicated state machines* [20]. The self-stabilizing consensus algorithms presented in [13] uses SWMR registers, and our simulation allows to port them to message-passing systems. More generally, our simulation allows the application of any self-stabilizing algorithm that is designed using SWMR registers to work in a message-passing system, where less than the majority of the processors may crash.

Our work leaves open many interesting directions for future research. Note that our algorithms can be initialized [8] to respect the atomicity requirements for the beginning of a practically infinite execution. Still one of the most interesting research directions is to find a stabilizing simulation, which will operate correctly even after sequence numbers wrap around, without an additional convergence period. This seems to mandate a more careful way to track epoch labels, perhaps by incorporating a self-stabilizing analogue of the *viability* construction [3].

Acknowledgments. We thank Ronen Kat and Eli Gafni for helpful discussions.

References

1. Abraham, U.: Self-stabilizing timestamps. Theoretical Computer Science 308(1-3), 449–515 (2003)
2. Attiya, H.: Robust Simulation of Shared Memory: 20 Years After. EATCS Distributed Computing Column (2010)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message-Passing Systems. Journal of the ACM 42(1), 124–142 (1995)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. Wiley Press, Chichester (2004)
5. Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-Stabilization by Local Checking and Global Reset. In: Tel, G., Vitányi, P.M.B. (eds.) WDAG 1994. LNCS, vol. 857, pp. 326–339. Springer, Heidelberg (1994)
6. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Bounding the Unbounded. In: INFOCOM, pp. 776–783 (1994)
7. Baldoni, R., Bonomi, S., Kermarrec, A.-M., Raynal, M.: Implementing a Register in a Dynamic Distributed System. In: ICDCS (2009)
8. Delaet, S., Dolev, S., Peres, O.: Safe and Eventually Safe: Comparing Stabilizing Algorithms and non-Stabilizing Algorithms on a Common Ground. In: Proc. of the 2009 International Conference on Principles of Distributed Systems (OPODIS), pp. 315–329 (2009)
9. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
10. Dolev, S., Dubois, S., Potop-Butucaru, M., Tixeuil, S.: Stabilizing data-link over non-FIFO channels with optimal fault-resilience. Information Processing Letters 111, 912–920 (2011)
11. Dolev, S., Herman, T.: Dijkstra's self-stabilizing algorithm in unsupportive environments. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 67–81. Springer, Heidelberg (2001)

12. Dolev, S., Israeli, A., Moran, S.: Resource Bounds for Self-Stabilizing Message-Driven Protocols. SIAM J. Comput. 26(1), 273–290 (1997)
13. Dolev, S., Kat, R.I., Schiller, E.M.: When Consensus Meets Self-stabilization, Self-stabilizing Failure-Detector, and Replicated State-Machine. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 45–63. Springer, Heidelberg (2006)
14. Dolev, D., Shavit, N.: Bounded Concurrent timestamping. SIAM Journal on Computing 26(2), 418–455 (1997)
15. Dolev, S., Tzachar, N.: Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. Theoretical Computer Science 410(6-7), 514–532 (2009)
16. Dolev, S., Tzachar, N.: Spanders: distributed spanning expanders. In: SAC (2010)
17. Dolev, S., Welch, J.L.: Crash Resilient Communication in Dynamic Networks. IEEE Trans. Computers 46(1), 14–26 (1997)
18. Israeli, A., Li, M.: Bounded timestamps. Distributed Computing 6(4), 205–209 (1993)
19. Johnen, C., Higham, L.: Fault-tolerant Implementations of Regular Registers by Safe Registers with Applications to Networks. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 337–348. Springer, Heidelberg (2008)
20. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
21. Lamport, L.: On Interprocess Communication. Part I: Basic Formalism. Distributed Computing 1(2), 77–85 (1986)

An Algorithm for Implementing BFT Registers in Distributed Systems with Bounded Churn

Roberto Baldoni, Silvia Bonomi, and Amir Soltani Nezhad

Universitá La Sapienza, Via Ariosto 25, I-00185 Roma, Italy
{baldoni,bonomi}@dis.uniroma1.it,
amir.soltaninezhad@gmail.com

Abstract. Distributed storage service is one of the main abstractions provided to the developers of distributed applications due to its capability to hide the complexity generated by the messages exchanged between processes. Many protocols have been proposed to build byzantine-fault-tolerant storage services on top of a message-passing system, but they do not consider the possibility to have servers joining and leaving the computation (*churn* phenomenon). This phenomenon, if not properly mastered, can either block protocols or violate the safety of the storage. In this paper, we address the problem of building of a safe register storage resilient to byzantine failures in a distributed system affected from churn. A protocol implementing a safe register in an eventually synchronous system is proposed and some feasibility constraints on the arrival and departure of the processes are given. The protocol is proved to be correct under the assumption that the constraint on the churn is satisfied.

Keywords: Bounded Churn, Safe Register, Byzantine Failures, Eventually Synchronous System.

1 Introduction

Dependable storage is a pillar of many complex modern software systems (from avionics to cloud computing environments) and byzantine-fault-tolerance (BFT) is one of the main techniques employed to ensure both storage correctness and highly available accesses. Such properties have to be guaranteed despite any types of failures, including malicious ones. Availability is achieved by keeping aligned a fixed number of replicas each one hosted at a separate server.

Looking at large-scale distributed systems such as peer-to-peer systems, interconnected data centers etc., storage implementations have to withstand various types, patterns, degrees and rates of arrival to and departure of processes from the system (i.e. they have to deal with *churn*. As an example, in the context of cloud computing, a storage system is an unmanaged service (e.g. Elastic Block Store of Amazon[1]) ensuring high availability. The storage is implemented through a specific replication pattern where servers hosting replicas are selected

[1] http://aws.amazon.com/ebs/

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 32–46, 2011.

autonomically from the server cloud. From time to time a cloud provider executes maintanance operations on the server cloud, e.g., rollout of security patch operations, that generates a continuous and unpredictable restarts of servers that can take hours [1]. Therefore, a rollout operation translates into servers that join and leave the storage service (i.e., server churn). As a consequence, a correct and highly available storage has to be ready to autonomously tolerate servers churn as well as byzantine server behavior.

Note that, the autonomous behavior of servers, characterizing the churn action, cannot be considered as a byzantine behavior. Byzantine servers, in fact, try to make the storage service deviate from its correct behavior either maliciously or accidentally. On the contrary, server behavior in case of join and leave are well defined: processes are correct, but are temporarily unavailable; as soon as they come back to be available, they start again to work correctly. It is easy to see that if the number of servers leaving the storage service is above a given threshold, data can be lost or compromised, or storage operations cannot terminate.

In this paper, we consider a distributed system that without churn is composed of n servers implementing a storage service, then due to the effect of churn up to J servers can be joining or leaving the service, however such number of serves is guaranteed never be below $n - J$ and eventually tends to come back to n. In this environment, we present a BFT implementation of a safe register, which is able to resist f byzantine failures and a churn of at most J servers (with $J \leq \lfloor \frac{n-5f}{3} \rfloor$). The protocol is based on quorums of size $n - f - J$, and works on top of a very general system model where churn is non-quiescent (i.e., the system model alternates infinitely often periods of no churn and periods of churn), and there is an unknown time t after which communication becomes synchronous for a period long enough to allow the BFT protocol to progress (eventually synchronous system). The algorithm presented in this paper can be also seen as an extension of quorum-based BFT algorithms [15] to ensure tolerance to servers churn.

Let us finally remark that the model of service implementation presented in this paper reflects quite well the structure of service implemented in a cloud environment. In such environment a storage service is configured, by the cloud provider, to work in a normal working situation with a set of n replicas, defined at the beginning of the computation. However, such replicas can be affected by bounded churn due to unpredictable leaves (i.e. crash failures, maintenance operations etc.) and later on, new replicas can be set up by the provider to substitute the ones left, with the aim to resume normal working situation. This is the kind of environment that this paper wants to investigate when considering the presence of byzantine processes.

The rest of the paper is contributed as follows: in Section 3, we define the system model. Section 4 provides the safe register specification while in Section 5, we detail the algorithm and the correctness proofs. Section 2 presents the related works, and finally Section 6 concludes the paper.

2 Related Work

To the best of our knowledge, this is the first work that addresses the contruction of a register resisting byzantine failures and churn in a non-synchronous system based on quorums. In the prior works, we studied the same problem from a structural point of view [7] and in an environment with crash failures [6].

Byzantine Fault Tolerant Systems Based on Quorums. Traditional solutions to build byzantine storage can be divided into two categories: replicated state machines [18] and byzantine quorum systems [9], [15], [16]. Replicated state machines uses $2f + 1$ server replicas and require that every non-faulty replica agrees to process requests in the same order [18]. Quorum systems, introduced by Malkhi-Reiter in [15], do not rely on any form of agreement they need just a sub-sets of the replicas (i.e. *quorums*) to be involved simultaneously. The authors provide a simple wait-freedom implementation of a safe register using $5f$ servers. [4] proposes a protocol for implementing a single-writer and multiple-reader atomic register that holds wait-freedom property with using just $3f + 1$ servers. This is achieved at the cost of longer (two phases) read and write operations. In this paper, our objective has been to design an algorithm that follows the Malkhi-Reiter's approach (i.e., single-phase operations), and that is able to tolerate both f failures and concurrent running join of at most J servers at any time, using less than $5(f + J)$ servers. This number of replicas would have been indeed necessary if we consider churning servers as byzantine processes. Leveraging from the difference between the behavior of a byzantine server and a churning one, the algorithm presented in this paper needs just $5f + 3J$ server replicas.

Registers under Quiescent Churn. In [14], [11] and [10], a Reconfigurable Atomic Memory for Basic Object (RAMBO) is presented. RAMBO works on the top of a distributed system where processes can join and fail by crashing. To guarantee the reliability of data, in spite of network changes, RAMBO replicates data at several network locations and defines *configurations* to manage small and transient changes. For large changes in the set of participant processes, RAMBO defines a *reconfiguration* procedure whose aim is to move the system from an existing configuration to a new one by changing the membership of the read quorums and of the write quorums. Such a reconfiguration is implemented by a distributed consensus algorithm. Thus, the notion of churn is abstracted by a sequence of configurations.

In [2] Aguilera et al. show that a crash resilient atomic register can be realized without consensus and, thus, on a fully asynchronous distributed system provided that the number of reconfigurations is finite and thus the churn is quiescent. Configurations are managed by taking into account all the changes (i.e. join and failure of processes) suggested by the participants and the quorums are represented by any majority of processes. To ensure liveness of read and write operations, the authors assume that the number of reconfigurations is finite and that there is a majority of correct processes in each reconfiguration.

Relationship between the Churn Model and the Crash-Recovery One In crash-recovery model processes may recover after a crash and each process is usually augmented with stable storage and, as in the crash failure model, the set of processes that will be part of the system is known in advance [3]. At a first glance, our churn model could resemble crash-recovery one (i.e., a server that leaves and re-joins the regular register computation could be seen as a crash and a recovery of a process), they differ in several fundamental aspects. In the model presented in this paper: (1) there is no assumption of initial knowledge about the set of processes, which will be part of the computation, (2) processes may join the application at any time, (3) processes may crash and later restart with another identifier an infinite number of times without relying on stable storage, which is an extremely important point when considering servers are virtual machines that can migrate from one physical machine to another one, and then the stable storage of the former machine could not be available anymore. Therefore the model presented in this paper is more general than crash recovery one. Let us finally remark that we are not aware of any BFT protocol working in a crash-recovery environment.

3 System Model

The distributed system is composed of a *universe of clients* U_c (i.e. the clients system) and of a disjoint *universe of servers* U_s (i.e. the servers system). The clients system is composed of a finite arbitrary number of processes (i.e. $U_c = \{c_1, c_2, \ldots c_m\}$) while the servers system is dynamic, i.e. processes may join and leave the system at their will. A server enters the servers system by executing the connect() procedure. Such an operation aims at connecting the new process to both clients and servers that already belong to the system. A server leaves the distributed system by means of the disconnect() operation. In the following, we will assume that the disconnect() operation is a passive operation i.e., processes do not take any specific actions, and they just stop to execute algorithms. In order to model processes continuously arriving to and departing from the servers system, we assume the infinite arrival model (as defined in [17]). The set of processes that can participate in the servers system (also called *server-system population*) is composed of a potentially infinite set of processes $U_s = \{\ldots s_i, s_j, s_k \ldots\}$, each one having a unique identifier (i.e. its index). However, the servers system is composed, at each time, of a finite subset of the server-system population. Initially, every server $s_i \in U_s$ is in the *down* state as soon as s_i invokes the connect() operation, it changes its state from *down* to *up*. When the server s_i disconnects itself from the servers system, it changes again its state coming back to *down*.

Clients and servers can communicate only by exchanging messages through reliable and authenticated FIFO channels. In the following, we assume the existence of a protocol managing the arrival and the departure of servers from the distributed system, such a protocol is also responsible for the connectivity maintenance among the processes belonging to the distributed system. As in [15],

[16], we assume that clients are correct and servers can suffer arbitrary failures. To simplify the presentation, let us assume the existence of a global fictional clock not accessible from processes.

Distributed Computation. Several distributed computations run on top of the distributed system, involve the participation of a subset of the servers set of the servers system. To simplify the presentation, let us assume that there exists only one distributed computation run in our system. We identify as $C_s(t)$ the subset of processes belonging to the servers system U_s that are participating in the distributed computation at time t (i.e. the *server-computation set*). At time t_0, when the server-computation set is set up, n servers belong to the servers computation (i.e. $|C_s(t_0)| = n$). A server s_i, belonging to the servers system that wants to join the distributed computation has to execute the join_Server() operation. Such an operation invoked at some time t is not instantaneous and takes time to be executed; how much this time is, depends on the specific implementation provided for the join_Server() operation. However, from time t, when the server s_i joins the server-computation set, it can receive and process messages sent by any other processes, which are participating in the computation, and it changes its state from *up* to *joining*.

When a server s_j participating in the distributed computation wishes to leave the computation, it stops to execute the server protocols (i.e. the leave_Server operation is passive) and comes back to the *up* state. Without loss of generality, we assume that if a server leaves the computation and later wishes to re-join, it executes again the join_Server() operation with a new identity.

It is important to notice that (i) there may exist processes belonging to the servers system that never join the distributed computation (i.e. they execute the connect() procedure, but they never invoke the join_Server() operation) and (ii) there may exist processes, which even after leaving the servers computation, still remain inside the servers system (i.e. they are correct, but they stop to process messages related to the computation). To this aim, it is important to identify the subset of processes that are actively participating in the distributed computation and the ones that are joining.

Definition 1 (Joining Servers Set). *A server is* joining *from the time it invokes the* join_Server() *operation until the time it terminates such operation. $J(t)$ denotes the set of servers that are execution the* join_Server() *operation at time t.*

In the following, we refer as J the maximum value of $J(t)$ for any t.

Definition 2 (Active Servers Set). *A server is* active *in the distributed computation from the time it returns from the* join_Server() *operation until the time it leaves. $A(t)$ denotes the set of servers that are active at time t, while $A([t, t'])$ denotes the set of servers that are active during the whole interval $[t, t']$ (i.e. $s_i \in A([t, t'])$ iff $s_i \in A(\tau)$ for each $\tau \in [t, t']$).*

A server s_i changes its state from *joining* into *active* as soon as it gets the join_Confirmation event, and remains in such a state until it decides to leave the server-computation set (thus, coming back to the *up* state).

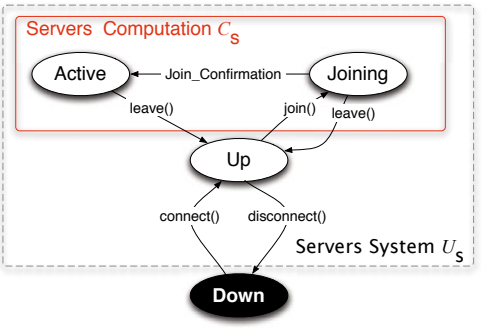


Fig. 1. State-transition diagram of a Correct Server

Note that, at each time t the set of servers participation in the distributed computation is partitioned into active processes and joining processes. i.e.

$$C_s(t) = A(t) \cup J(t)$$

Servers that obey their specification are said to be *correct*. On the contrary, a *faulty* server can deviate arbitrarily from its specification. We assume at most f servers can be faulty at any time during the whole computation[2]. It is important to note that servers know the values f and J, but they are not able to know the subset of C_s representing the faulty processes. In Figure 1 it is shown the state-transition diagram of a correct server.

Non-quiescent Bounded Churn. The servers computation alternates periods of churn and periods of stability. More specifically, there exist some periods T_{churn} in which servers join and leave the computation, then there exist some periods $T_{stability}$ where the computation becomes stable, and no join or leave operations are triggered. However, no assumption is made about how long T_{churn} and $T_{stability}$ are.

At time t_0 all the servers participating in the server computation are active (i.e. $|A(t_0)| = n$). Moreover, we assume that the churn affecting the servers computation is bounded by an integer value $J \geq 0$ and the number of servers participating in the servers computation can change in the interval $[(n - J), n]$ (i.e. $\forall t, |C_s(t)| \in [(n - J), n]$). Finally, we assume that in the distributed computation there are always at least $n - J$ active servers (i.e.$\forall t, |A(t)| \geq n - J$). The above equality implies that the servers computation is configured to work with n servers event though it tolerates that up to J servers can leave and later on they can be replaced by up to J new joining servers. Thus the value J represents the upper bound on the churn.

[2] Note that, f is an upper bound on the number of faulty processes. As a consequence, during the computation, there may exists periods where less than f byzantine servers participate in the computation. Moreover, our assumption does not implies that the set of faulty servers is static but we admit it can change during the whole computation.

Let us finally remark that in this churn model, there is no guarantee that a server remains permanently in the computation and additionally, this model is general enough to encompass both (i) a distributed computation prone to non-quiescent churn i.e., there exists a time t (with $t = t_0$) after which churn holds forever, and (ii) a distributed system prone to quiescent churn i.e., there exists a time t after which stability holds forever.

4 Register Specification

A register is a shared variable accessed by a set of processes, i.e. clients, through two operations, namely read() and write(). Informally, the write() operation updates the value stored in the shared variable while the read() obtains the value contained in the variable (i.e. the last written value). Every operation issued on a register is, generally, not instantaneous and it can be characterized by two events occurring at its boundary: an *invocation* event and a *reply* event. These events occur at two time instants (invocation time and reply time respectively) according to the fictional global time.

An operation *op* is *complete* if both the invocation event and the reply event occur (i.e. the process executing the operation does not crash between the invocation and the reply).

Given two operations *op* and op', their invocation times ($t_B(op)$ and $t_B(op')$) and return times ($t_E(op)$ and $t_E(op')$), we say that *op precedes* op' ($op \prec op'$) iff $t_E(op) < t_B(op')$. If *op* does not precede op', and op' does not precede *op*, then *op* and op' are *concurrent* ($op||op'$). Given a write(v) operation, the value v is said to be written when the operation is complete. In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. In this paper, we will consider a single-writer/multiple-reader safe register which is specified as follows [13][3]:

- Termination: If a correct process (either a client or a server) participating in the computation invokes an operation and does not leave the system, it eventually returns from that operation.
- Validity: a read() not concurrent with any write() returns the last written value before its invocation. In the case of concurrency, a read() may return any value.

As a specialization of the generic model of the computation presented in the previous Section, we consider in this paper a safe register computation, i.e. the join_Server() operation, executed by servers, has the aim to provide new servers with the state of the register. Concerning the departures from the computation, we consider the leave operation as an implicit operation; when a server s_i leaves

[3] Interestingly, safe registers have the same computational power as regular registers and atomic registers. This means that it is possible to implement a multi-writer/multi-reader atomic register from single-writer/single-reader safe registers as shown in [12].

the computation, it just stops to send and process messages related to the register computation. To simplify the notation, whenever not strictly necessary, we use the term join() instead of join_Server().

5 Safe Register Implementation

A register is maintained by the set of active servers. No agreement abstraction is assumed to be available at a server. Clients do not maintain any register information; they can just trigger operations and interact with servers through message exchanges. Moreover, we assume that each server has the same role in the distributed computation (i.e. no server acts as a coordinator) and when it issues a join() operation at some time t, the server does not leave the computation before time $t + 3\delta$.

Eventually Synchronous Communication Model. Due to the impossibility of implementing a register in a fully asynchronous system prone to non-quiescent churn [5], in this paper we will assume a partial synchronous system, i.e. there exists a time t after which a synchrony period holds long enough to ensure the correct progress of protocol implementation. In particular, eventual synchrony implies that each message sent at some time t' after t, by a process p, is delivered within δ time units by every process belonging to the distributed system in the interval $[t', t' + \delta]$.

Quorums. The basic idea of the algorithm is to extend the opaque masking quorums mechanism, defined by Malkhi and Reiter [15], to implement a safe register in a dynamic distributed system with byzantine failures. In particular, both join(), read() and write() operations are executed on quorums of servers participating in the distributed computation of size $n - f - J$. In this section we assume for simplicity that all the processes (both clients and servers) know the values n, f and J. At the end of the Section we discuss how to relax this assumption.

5.1 A Protocol for Eventually Synchronous System

Each reader client c_i maintains the following variables:

- one integer variable, denoted $read\_sn_i$, representing the sequence number to associate to each read() operation. Initially the variable is set to 0.
- a set variable, denoted as $cl\_replies_i$, used to collect answers sent by servers and initially empty.

Moreover, the writer client c_w also maintains:

- two integer variables sn_w and $count_i$, representing respectively the sequence number to associate to each write() operation and the number of tentative write() operations. Initially both the variables are set to 0.

- an array of sets variable, denoted as $write\_ack_i[]$, used to collect the servers that have acknowledged its last write.
- an array of sets variable, denoted as $confirmation_i[]$, used to collect the servers that have confirmed its last write.

Each server s_i has the following local variables.

- A set W_i that stores the writers identifiers.
- Two variables denoted $register_i$ and sn_i; $register_i$ contains the local copy of the safe register, while sn_i is the associated sequence number.
- A boolean $active_i$, initialized to *false*, that is switched to *true* just after s_i has joined the system.
- Two set variables, denoted $replies_i$ and $reply\_to_i$, that are used in the period during which s_i is joining the system. The local variable $replies_i$ contains the 4-tuple $< id, value, sn, r\_sn >$ that s_i has received from other servers during its join period, while $reply\_to_i$ contains the IDs of servers, which are joining the system concurrently with s_i (as far as s_i knows).
- dl_prev_i is a set where (while it is joining the system) p_i records the processes that have acknowledged its inquiry message, while they were not yet active (so, these processes were joining the system too). When it terminates its join operation, p_i has to send them a reply to prevent them to be blocked forever.

In order to simplify the pseudo-code notation, let us consider the function most_frequent($replies$). Such a function is used by both clients and servers to select the most frequent pair $< val, sn >$ occurred in the set $replies_i$. In the case that more than one pair with the same frequency exist, the function returns the pair having the highest sn.

The join() **Operation (Figure 2).** The server s_i broadcasts an INQUIRY ()) message to inform the other servers, which it is entering the distributed computation set, and wants to obtain the value of the safe register (line 03).

Then, after it has received "enough" replies (line 04), s_i selects among the set of received values, the one occurred with the highest frequency (line 05). Moreover, s_i updates its local copy of the register (line 06), it becomes active (line 07), and sends a reply to the processes in the set $reply\_to_i$ (line 08-10). It also sends such a reply message to the servers in its dl_prev_i set, in order to prevent them from waiting forever. In addition to the term $< i, register_i, sn_i >$, a reply message sent to a server s_j, from a server s_i, carries also the read sequence number r_sn that identifies the corresponding request issued by s_j.

When s_i delivers an INQUIRY(j, r_sn), it always sends back a message to p_j. It sends a REPLY() message if it is active (line 13), and a DL_PREV() if it not active yet (line 15). Moreover, in case s_i is not active, it stores the inquiry received from s_j in the $reply\_to_j$ variable, to remember to answer later, as soon as it becomes active (line 14).

When s_i receives a REPLY$(< j, value, sn >, r_sn)$ message from a server s_j, if the reply message is the first answer to its INQUIRY$(i, read_sn)$ message s_i adds $< j, value, sn, 0 >$ to the set of replies that it has received so (line 22). On the

```
operation join(i):
(01)  register_i ← ⊥; sn_i ← −1; active_i ← false; replies_i ← ∅;
(02)  reply_to_i ← ∅; dl_prev_i ← ∅; read_sn_i ← 0;
(03)  broadcast INQUIRY(i, 0);
(04)  wait until(|replies_i| ≥ (n − f − J));
(05)  let < val, sn > ← most_frequent(replies_i);
(06)  if (sn > sn_i) then sn_i ← sn; register_i ← val end if
(07)  active_i ← true;
(08)  for each < j, r_sn > ∈ reply_to_i ∪ dl_prev_i do
(09)            do send REPLY (< i, register_i, sn_i >, r_sn) to p_j
(10)  end for;
(11)  return(ok).
------------------------------------------------------------------------
(12)  when INQUIRY(j, r_sn) is delivered:
(13)        if (active_i) then send REPLY (< i, register_i, sn_i >, r_sn) to p_j
(14)                      else  reply_to_i ← reply_to_i ∪ {< j, r_sn >};
(15)                            send DL_PREV (i, r_sn) to p_j
(16)        end if.

(17)  when REPLY(< j, value, sn >, r_sn) is received:
(18)        if (read_sn_i = r_sn) then
(19)          if (∃ < j, −, −, r_sn > ∈ _replies_i) then
(20)            replies_i ← replies_i/{< j, −, −, r_sn >};
(21)          endif
(22)          replies_i ← replies_i ∪ {< j, val, sn, r_sn >};
(23)        endif

(24)  when DL_PREV(j, r_sn) is received: dl_prev_i ← dl_prev_i ∪ {< j, r_sn >}.
```

Fig. 2. The join() protocol for an eventually synchronous system (code for s_i)

contrary, s_i updates the information already received from s_j with the new value (line 20 - 22).

Finally, when s_i receives a message DL_PREV(j, r_sn), it adds its content to the set dl_prev_i (line 24), in order to remember that it has to send a reply to s_j when it becomes active (lines 08-10).

The read() Operation (Figure 3). The algorithm for the read() operation is a simplified version of the join() algorithm. The main difference between the two algorithms is the "stubborn" re-transmission mechanism used by the read() (lines 03-05). This mechanism is necessary because (i) a read() broadcast message could not be received both by a leaving server and by a joining one and (ii) a reply message sent by a leaving server could not reach the client. This might block the client read protocol that could not reach the expected number of replies $(n - f - J)$ [4]. Resending the read message periodically will ensure that the message eventually reaches enough servers due to the arrival of either a stability period or a synchrony period.

Note that, the same problem does no happen during the join() execution thanks to the DL_PREV mechanism. When a servers s_i joins, in fact, its inquiry is delivered to all the servers belonging to the distributed computation and if

[4] Let us recall that the communication primitives work in a best-effort fashion on top of FIFO channels. Thus, there is no guarantee that a message m sent at some time t by a process p is delivered to other processes in case p leaves the computation.

```
operation read(i):
(01)  read_sn_i ← read_sn_i + 1;
(02)  cl_replies_i ← ∅;
(03)  repeat
(04)      broadcast READ(i, read_sn_i);
(05)  until ((|cl_replies_i| ≥ n − f − J)
(06)  let < val, sn >← most_frequent(cl_replies_i);
(07)  if (sn > sn_i)
(08)      then sn_i ← sn;
(09)           value_i ← val
(10)  end if;
(11)  return(val).
```

```
when REPLY(< j, val, sn >, r_sn) is delivered:
(12)  if (read_sn_i = r_sn) then
(13)     if (∃ < j, −, −, r_sn >∈ cl_replies_i) then
(14)        cl_replies_i ← cl_replies_i/{< j, −, −, r_sn >};
(15)     endif
(16)     cl_replies_i ← cl_replies_i ∪ {< j, val, sn, r_sn >};
(17)  endif
```

(a) Client Protocol

```
when READ(j, r_sn) is delivered:
(01)  if (active_i)
(02)     then send REPLY (< i, value_i, sn_i >, r_sn) to p_j;
(03)     else  reply_to_i ← reply_to_i ∪ {< j, r_sn >};
(04)  end if.
```

(b) Server Protocol

Fig. 3. The read() protocol for an eventually synchronous system

new processes arrive, they become aware about the join of s_i due to the DL_PREV message.

The write() Operation (Figure 4). Similarly to the read() operation, the write() is also implemented by repeating the value dissemination until the writer gets acknowledgements from a quorum of $n - f - J$ processes (lines 06 - 08). In addition, in order to terminate the write(), the client must also receive confirmations that ensure that the $n - f - J$ processes who sent acknowledgements to the writer, have not left the server computation, and still keep the new value. Equivalently, the acknowledgement are effectively sent by a quorum of processes that belongs to the distributed computation for a sufficient long time to correctly represent the new value (lines 09 - 12).

When a message WRITE$(j, < val, sn, num >)$ is delivered to a server s_i, it takes into account the pair (val, sn) if it is more up-to-date than its current pair and only if the message came from an authenticated channel of one writer (line 01). Then, if s_i is active, it sends back an ACK (i, sn) message to the writer (line 06).

```
operation write(v):

(01) sn_i ← sn_i + 1; count_i ← 0;
(02) ∀k : write_ack_i[k] ← ∅;
(03) ∀k : confirmation_i[k] ← ∅;
(04) repeat every (Δ) time units
(05)    count_i ← count_i + 1;
(06)    while (|write_ack_i[count_i]| < n − f − J)) do
(07)       broadcast WRITE(i, < v, sn_i, count_i >);
(08)    endWhile;
(09)    for each (p_j ∈ write_ack_i[count_i]) do
(10)       send CONFIRM (i, sn, count_i) to p_j.
(11)    endFor
(12) until (∃x : |confirmation_i[x]| ≥ n − f − J);
(13) return(ok).
```

```
when ACK(j, sn, num) is received:
(14)   if (sn = sn_i) then
(15)      write_ack_i[num] ← write_ack_i[num] ∪ {j};
(16)   end if.
```

```
when CONFIRM_ACK (j, sn, num) is received:
(17)   if (sn = sn_i) then
(18)      confirmation_i[num] ← confirmation_i[num] ∪ {j};
(19)   end if.
```

(a) Client Protocol

```
when WRITE(j, < val, sn, num >) is delivered:
(01)   if (j ∈ W_i ∧ (sn > sn_i))
(02)      then sn_i ← sn;
(03)         value_i ← val;
(04)   endif;
(05)   if (active_i)
(06)      then send ACK (i, sn, num) to p_j.
(07)   endif
```

```
when CONFIRM (j, sn, num) is received:
(08)   then send CONFIRM_ACK (i, sn, num) to p_j.
```

(b) Server Protocol

Fig. 4. write() protocol for an eventually synchronous system

When the client receives an ACK (j, sn) message from the server s_j, it adds s_j to its set $write\_ack_i$ if this message is an answer to its last write operation(line 09).

Finally, when the client receives a CONFIRM_ACK (j, sn, num), it just takes into account the confirmation received by server s_j.

Correctness Proofs. Due to the lack of space, we report here only the proof showing the relation among the values of n, f and J ensuring the existence of a quorum system. Other proofs can be found in [8].

Definition 3. *A quorum system $\mathcal{Q} \subseteq 2^{|C_s|}$ is a non-empty set of subsets of C_s, every pair of which intersect. Each $Q \in \mathcal{Q}$ is called a* quorum.

Definition 4 (Opaque Masking Quorum). *Let $B \subset C_s$ be the subset of faulty processes participating in the distributed computation. A quorum system $\mathcal{Q}$ is an opaque masking quorum system if:*

- **(P1)** $\forall\ Q_w, Q_r \in \mathcal{Q} : |(Q_w \cap Q_r)/B| \geq |(Q_r \cap B) \cup (Q_r/Q_w)|$
- **(P2)** $\forall\ Q_w, Q_r \in \mathcal{Q} : |(Q_w \cap Q_r)/B| > |(Q_r \cap B)|$
- **(P3)** $\exists\ Q \in \mathcal{Q} : Q \cap B = \emptyset$.

Lemma 1. *Let n be the number of processes participating in the distributed computation at any time t and let f be the maximum number of byzantine processes participating in the computation. If $n \geq 5f + 3J$ then $\mathcal{Q} = \{|Q_i| = n - f - J\}$ is an opaque masking quorum for the safe register computation.*

Proof. Note that, considering a quorum Q_i composed of $n - f - J$ processes, property $P3$ is always guaranteed. Moreover, $P1$ implies $P2$ and thus in the following we will show only $P1$.

Let Q_w and Q_r be respectively two quorums associated to a write(v) operation op and a read()/join() operation op'.

Let $X = (Q_r/Q_w)$ be the set of processes belonging to the computation and not affected from op; the number of this processes is $|X| = n - |\mathcal{Q}_w|$.
The quorum Q_r can be represented as $\mathcal{Q}_r = X \cup (\mathcal{Q}_r \cap \mathcal{Q}_w)$ and considering that X and $\mathcal{Q}_r \cap \mathcal{Q}_w$ are disjoint sets, we can deduce the following: $|\mathcal{Q}_r| = |X| + |\mathcal{Q}_r \cap \mathcal{Q}_w| \Rightarrow |\mathcal{Q}_r \cap \mathcal{Q}_w| = |\mathcal{Q}_r| - |X| = |\mathcal{Q}_r| - n + |\mathcal{Q}_w|$.

Considering that $|\mathcal{Q}_r| = |\mathcal{Q}_r| = n - f - J$, we get $|\mathcal{Q}_r \cap \mathcal{Q}_w| = ((n - f - J) - n + (n - f - J)) = n - 2f - 2J$.

Note that, in the worst case, $(Q_r \cap B) = B$ and it is a disjoint set from (Q_r/Q_w).

As a consequence, $|(Q_r \cap B) \cup (Q_r/Q_w)| = f + n - (n - f - J) = 2f + J$.

Therefore, $|(Q_w \cap Q_r)/B| \geq |(Q_r \cap B) \cup (Q_r/Q_w)| \Rightarrow$
$(n - 2f - 2J) - f \geq 2f + J \Rightarrow n \geq 5f + 3J$. $\square_{Lemma\ 1}$

Theorem 1. Safety. *Let us assume that $n \geq 5f + 3J$. Given the algorithm in Figures 2 - 4, then a* read() *operation that is not concurrent with any* write(), *returns the last value written before the* read() *invocation.*

Lemma 2. *Let us assume that (1) $n \geq 5f + 3J$, and (2) a server that invokes the* join() *operation remains in the system for at least 3δ time units. If a server process s_i invokes the* join() *operation, and does not leave the computation, this join operation terminates.*

Lemma 3. *Let us assume that (1) $n \geq 5f + 3J$, and (2) a server that invokes the* join() *operation remains in the system for at least 3δ time units. If a client c_i invokes a* read() *operation and does not leave the system, this read operation terminates.*

Lemma 4. *Let us assume that (1) $n \geq 5f + 3J$, and (2) a server that invokes the* join() *operation remains in the system for at least 3δ time units. If a client process c_i invokes* write() *and does not leave, this write operation terminates.*

From Lemma 2, Lemma 3 and Lemma 4 we have:

Theorem 2. Termination. *Let us assume that $n \geq 5f+3J$. Given the algorithm in Figures 2 - 4, if a process invokes* join(), read() *or* write (), *and does not leave the system, it terminates its operation.*

On the knowledge of n, k and J. The protocol proposed in this section assumes that all the processes (both clients and servers) participating in the distributed computation know (i) the maximum number of byzantine servers f, (ii) the number n of servers participating in the distributed computation and (iii) the maximum number of non-active servers J. However, such assumptions can be weakened considering n and J only known by servers. These values are, in fact, two configuration parameters of the servers computation defining the robustness of the service with respect to possible churn.

To this aim, it is possible to modify the read() and the write() implementations to let servers dynamic behavior be transparent to clients. In particular a server, before answering to the client, sends a broadcast message to all the other servers, when the server receives $n - f - J$ of such broadcast messages it can reply to the client. The client has to wait only for $2f + 1$ messages, thus it does not need to know n and J. The complete protocol is described in [8].

6 Concluding Remarks

In this paper, we have provided an implementation of a distributed storage in the presence of both servers churn and byzantine servers. In a computation composed of n servers in normal working condition, the protocol is able to tolerate at most J joining servers if $J \leq \lfloor \frac{n-5f}{3} \rfloor$, where f is the maximum number of byzantine servers. The protocol works in an eventually synchronous environment, so it keeps safe during arbitrarily long (but finite) periods of asynchrony and churn, while it is able to quickly terminate as soon as the system gets into synchrony bounds. In this paper for simplicity the protocol assumed n and J be known by clients. In [8], we provide a variation of the protocol where a client needs just to know f while the knowledge about n and J is confined within the servers.

Let us finally remark that we decided to extend Malki-Reiter's protocol for its simplicity and because operations are short in time. Other algorithms (e.g. [16]) reduce indeed the number of servers needed for handling f byzantine failures, however, this is done at the cost of longer (multistep) read and write operations. When facing a dynamic system, such length matter as the leaving of processes during read and write operations can impact both their safety and their liveness as we showed in the paper. We plan to investigate this tradeoff in the future work trying to lower the overall number of replicas needed to cope with churn.

Acknowledgement. This work is partially supported by the European projects SOFIA, GreenerBuildings and SM4All.

References

1. Adya, A., Dunagan, J., Wolman, A.: Centrifuge: integrated lease management and partitioning for cloud services. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI (2010)
2. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. In: Proceedings of 28th Annual ACM Symposium on Principles of Distributed Computing, PODC (2009)
3. Aguilera, M., Chen, W., Toueg, S.: Failure Detection and Consensus in the Crash-recovery Model. Distributed Computing 13(2), 99–125 (2000)
4. Aiyer, A.S., Alvisi, L., Bazzi, R.A.: Bounded Wait-Free Implementation of Optimally resilient Byzantine Storage without (Unproven) Cryptographic assumptions. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 7–19. Springer, Heidelberg (2007)
5. Baldoni, R., Bonomi, S., Kermarrec, A.M., Raynal, M.: Implementing a Register in a Dynamic Distributed System. In: Proceedings of the 29th IEEE International Conference on Distributed Computing Systems, ICDCS (2009)
6. Baldoni, R., Bonomi, S., Raynal, M.: Implementing a Regular Register in an Eventually Synchronous Distributed System prone to Continuous Churn. In: IEEE Transactions on Parallel and Distributed Systems, TPDS (2011), http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.97
7. Baldoni, R., Bonomi, S., Soltani Nezhad, A.: Regular Registers in Dynamic Distributed Systems with Byzantine Processes: Bounds and Performance Analysis Technical report - MIDLAB 3/11 - 2011. A short version of this paper will appear in PODC (2011)
8. Baldoni, R., Bonomi, S., Soltani Nezhad, A.: An Algorithm for implementing BFT Registers in Distributed Systems with Bounded Churn Technical report - MIDLAB 5/11 (2011)
9. Bazzi, R.A.: Synchronous Byzantine Quorum Systems. Distributed Computing 13(1), 45–52 (2000)
10. Chockler, G., Gilbert, S., Gramoli, V., Musial, P.M., Shvartsman, A.: Reconfigurable distributed storage for dynamic networks. Journal Parallel Distributed Computing 69(1), 100–116 (2009)
11. Gilbert, S., Lynch, N., Shvartsman, A.: RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks. In: Proceedings of International Conference on Dependable Systems and Networks, DSN (2003)
12. Haldar, S., Vidyasankar, K.: Constructing 1-writer Multireader Multivalued Atomic Variables from Regular Variables. JACM 42(1), 186–203 (1995)
13. Lamport, L.: On Interprocess Communication, Part 1: Models, Part 2: Algorirhms. Distributed Computing 1(2), 77–101 (1986)
14. Lynch, N., Shvartsman, A.: RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 173–190. Springer, Heidelberg (2002)
15. Malkhi, D., Reiter, M.K.: Byzantine Quorum Systems. Distributed Computing 11(4), 203–213 (1998)
16. Martin, J., Alvisi, L., Dahlin, M.: Minimal Byzantine Storage. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 311–325. Springer, Heidelberg (2002)
17. Merritt, M., Taubenfeld, G.: Computing with Infinitely Many Processes. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 164–178. Springer, Heidelberg (2000)
18. Schneider Fred, B.: Implementing Fault-Tolerant Services Using the State Machine Approach. ACM Computing Surveys 22(4), 299–319 (1990)

Computing Time Complexity of Population Protocols with Cover Times - The ZebraNet Example

Joffroy Beauquier[1,3], Peva Blanchard[1,⋆], Janna Burman[2,⋆⋆], and Sylvie Delaët[1]

[1] LRI, Univ. Paris-Sud 11, Orsay, France
{jb,blanchard,delaet}@lri.fr
[2] MASCOTTE, INRIA, I3S (CNRS/University of Nice Sophia-Antipolis), France
janna.burman@inria.fr
[3] Grand Large project, INRIA Saclay, France

Abstract. *Population protocols* are a communication model for large sensor networks with resource-limited mobile agents. The agents move asynchronously and communicate via pair-wise interactions. The original *fairness* assumption of this model involves a high level of asynchrony and prevents an evaluation of the convergence time of a protocol (via deterministic means). The introduction of some "partial synchrony" in the model, under the form of *cover times*, is an extension that allows evaluating the time complexities.

In this paper, we take advantage of this extension and study a *data collection* protocol used in the ZebraNet project for the wild-life tracking of zebras in a reserve in central Kenya. In ZebraNet, sensors are attached to zebras and the sensed data is collected regularly by a mobile *base station* crossing the area. The data collection protocol of ZebraNet has been analyzed through simulations, but to our knowledge, this is the first time, that a purely analytical study is presented. Our first result is that, in the original protocol, some data may never be delivered to the base station. We then propose two slightly modified and correct protocols and we compute their worst case time complexities. Still, in both cases, the result is far from the optimal.

1 Introduction

Population Protocols (PP) have been introduced [1] as a model of sensor networks consisting of very simple mobile agents. In this model, anonymous mobile agents move asynchronously and any two of them can exchange information and change their states whenever they are chosen by a *scheduler*. When this happens, we say that an *event*, or a *meeting* between two moving agents, happens. Initially, one of the goals of PP was to determine what can be computed in such

⋆ The work of this author was supported by grants from INRIA Saclay.
⋆⋆ The work of this author was supported by the Chateaubriand grant from the French Government.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 47–61, 2011.

a model with a minimal hypothesis. That is why agents are anonymous, move asynchronously and have a small memory. No specific assumption is made on the scheduler, except for a fairness condition that states that an infinitely often reachable configuration is reached infinitely often. It was shown in [3] that the computational power of the model is rather limited. Hence, various extensions were suggested (e.g., [13,7,12,2,5]).

In this paper, we assume a version of the PP model, where an indicator of "speed", a *cover time*, is associated to each agent [5]. A cover time is the minimum number of global events happening in the system for being certain that an agent has met every other agent. A scheduler schedules global events according to the cover times. The assumption that an agent communicates with all other agents periodically, within a finite period, has been experimentally justified for some types of mobility. Indeed, in the case of human or animal mobility within a bounded area or with a "home coming" tendency (the tendency to return to some specific places periodically), the statistical analysis of experimental data sets confirms this assumption (e.g., [14,16,8]). These data sets concern students on a campus [10], participants to a network conference [9] or visitors at Disneyland. All exhibit the fact that the *inter-contact time* (ICT) between two agents, considered as a random variable, follows a truncated Pareto distribution. In particular, this involves that the ICTs, measured in terms of real time, are finite in practice. Thus, they are also finite when measured in events. So is the cover time of an agent, which is the maximum of its ICTs measured in events.

The notion of cover times may be viewed as an introduction of "partial synchrony" assumptions [11] in the original PP model (partial - because the cover times are not assumed to be known by the agents). This extension allows to compute deterministic time complexities expressed in the number of events (also called *event complexities*). This is impossible in the original PP model.

This paper presents, on an example, some techniques for computing the event complexity of population protocols. The example is a slight modification of an existing *data collection* protocol, used by the ZebraNet project [15]. ZebraNet is a project conducted by the Princeton University and deployed in central Kenya. It aims at studying populations of zebras using sensors attached to the animals. This project uses a history-based protocol to deliver the sensed values to a base station. When an agent x has the possibility to relay its data to other agents, it may select the one, y, that has recently met the base station more frequently. The protocol assumes that y will continue meeting the base station frequently in the near future and will deliver data sooner.

The first result in this paper theoretically shows that the original ZebraNet protocol does not ensure the delivery of all the values to the base station. There are infinite executions in which some values cycle between some mobile agents. The fact that about 10% of the sensed values are lost, as exhibited by the simulations in [15], is supported here by a formal explanation. To ensure the delivery without modifying the main structure of the executions, we propose two slightly modified versions respectively called Modified ZebraNet Protocols 1 and 2 (MZP1 and MZP2). We then provide an analysis of their event complexities

thanks to the notion of cover times. In both cases, the worst case complexity is worse than for the algorithm presented in [5] (this algorithm reaches the optimal worst case complexity in general cases).

2 Model and Notations

The model is as in [5]. Let $\mathbf{A}$ be the set of all the agents in the system where $|\mathbf{A}| = \mathbf{n}$ and $\mathbf{n}$ is unknown to the agents. The Base Station (BS) is a distinguishable agent with extended resources and which may be also non-mobile.[1] In contrast with BS, all the other agents are finite-state, anonymous and are referred in the paper as *mobile*. We denote by $\mathbf{A}^*$ the set of mobile agents. Mobile agents are enumerated from 1 to $\mathbf{n} - 1$.

Population protocols can be modeled as transition systems. We adopt the following common definitions (for formal definitions, refer, e.g., to [18]) : *state* of an agent (vector of the values of its variables), *configuration* (a vector of states of all the agents), *transition* (atomic step of two communicating agents and their associated state changes), *execution* (a possibly infinite sequence of configurations related by transitions).

An *event* $(x\ y)$ is a pairwise communication (meeting) of two agents x and y. An event corresponds to a transition. Without loss of generality, we assume that no two events happen simultaneously. A *schedule* is an infinite sequence of events. A schedule, together with an initial configuration, uniquely determines an execution[2]. By abusing the notation, we often write a sequence of events to represent both a schedule and the corresponding execution. Intuitively, it is convenient to see executions as if a scheduler (adversary) "chooses" which two agents participate in the next event. Formally, a scheduler $\mathcal{D}$ is a predicate on schedules. A schedule of $\mathcal{D}$ is a schedule that satisfies the predicate $\mathcal{D}$. For the sake of simplicity, we assume that all agents start an execution *simultaneously* (e.g., on sunrise, according to a clock, or on receipt of a global signal from BS). The *non-simultaneous* start is treated, e.g., in [5,6].

Cover Time Property. In the model, each agent x is associated with a positive integer $\mathbf{cv}_x$, called the *cover time* of x. Agents are not assumed to know the cover times. We denote by $\overline{\mathbf{cv}}$ the vector of agents' cover times and by $\mathbf{cv_{min}}$ (resp. $\mathbf{cv_{max}}$) the minimum (resp. maximum) cover time in $\overline{\mathbf{cv}}$.

Definition (Cover Time Property). *Given a population* $\mathbf{A}$ *of* $\mathbf{n}$ *agents and a vector* $\overline{\mathbf{cv}}$ *of positive integers, a scheduler* $\mathcal{D}$ *(and any of its schedules) is said to satisfy the* cover time property*, if and only if, for every* $x \in \mathbf{A}$*, in any* $\mathbf{cv}_x$ *consecutive events of any schedule of* $\mathcal{D}$*, agent* x *meets every other agent at least once.*

In the paper, we consider only the schedulers that satisfy the cover time property. We say that the cover time vector $\overline{\mathbf{cv}}$ is *uniform* if all its entries are equal, i.e.,

[1] BS is required here only by the nature of the data collection problem.
[2] We only consider deterministic systems.

$\mathbf{cv_{min}} = \mathbf{cv_{max}}$. In this case, we denote by $\mathbf{cv}$ the common value of the agents' cover times.

Data Collection and Convergence. In the context of *data collection*, an *initial configuration* is a configuration in which each mobile agent owns an *input value*. Each input value has to be *delivered* to BS exactly once. When this happens, we say that a *legal* configuration is reached. An execution is said to *converge* if it reaches a legal configuration. The *length* of an execution that converges is the minimum number of events until convergence. The *worst case event complexity* of an algorithm is the maximum length of its executions. A protocol (or an algorithm) is said to converge, if all its executions converge.

When describing an execution, we may annotate each event as follows. The notation $(x\ \underline{y})$ indicates that there is a transfer from x to y. To specify one of the values being transferred, v for example, we note $(x\ \underline{y})^{(v)}$. Note that after $(x\ \underline{y})$, agent x *does not keep any copy* of the transferred values. Also, the notation $(x\ y)$ does not imply that there is no transfer.

For some finite sequences $S_1, S_2, \ldots, S_k$, their concatenation in the given order is denoted by $S_1 \cdot S_2 \cdots S_k$ (or just $S_1 S_2 \ldots S_k$). For any finite sequence S and any positive integer l, the sequence S^l is the sequence obtained by repeating l times the sequence S. In addition, the infinite sequence S^ω denotes the infinite repetition of S.

3 Non Convergence of the Original Protocol

In the original ZebraNet data collection protocol [15] that we consider, an agent chooses, among the agents in its range, the one which is the most likely to meet BS in a near future, and transfers its values to it. In this paper, we chose to use the model with pairwise communications, in contrast to the multi-wise communications possible in ZebraNet. Hence, the ZebraNet Protocol (ZP), Algorithm 1 presented below, is a restricted version of the original ZebraNet protocol. However, as any execution of ZP is also an execution of the original protocol, the non convergence of ZP involves the non convergence of the latter.

In ZP, the state of an agent x is defined by integer variables $accumulation_x$ and $distance_x$, an array of data values $values_x$[3] and an integer constant **decay** that is the same for every agent. The integer variables are initially set to 0. The array $values_x$ holds initially the value provided by the sensor (e.g., temperature or heart-rate). For the sake of simplicity, we assume first that the memory available for each agent is large enough, so that it can store the values of all the others. This assumption prevents memory overflows during transfers[4].

In Algorithm 1, when an agent x meets BS, its variable $accumulation_x$ is incremented and $distance_x$ is reset to 0. When an agent x meets another mobile agent, its variable $distance_x$ is incremented. If $distance_x$ becomes larger than

[3] We do not define the type of these arrays explicitly.

[4] In other words, we assume that agents have an unbounded $O(\mathbf{n})$ memory. The case of bounded memory is discussed in Sec. 6

decay, $accumulation_x$ is decremented and $distance_x$ is reset to 0.[5] When an agent x holds some values in $values_x$ and meets another mobile agent y, if $accumulation_y$ is strictly greater than $accumulation_x$, then agent x transfers all its values to agent y. An agent always transfers all its values when it meets BS.

Algorithm 1. ZebraNet Protocol

when x meets BS **do**
  $<x$ transfers $values_x$ to $BS>$
  $accumulation_x := accumulation_x + 1$
  $distance_x := 0$
end when
when x meets $y \neq BS$ **do**
  if $accumulation_x < accumulation_y \wedge <values_x$ is not empty$>$ **then**
    $<x$ transfers $values_x$ to $y>$
  end if
  $distance_x := distance_x + 1$
  if $distance_x >$ **decay then**
    if $accumulation_x \neq 0$ **then**
      $accumulation_x := accumulation_x - 1$
    end if
    $distance_x := 0$
  end if
end when

It appears that not all executions of ZP converge. Indeed, a value can circulate between mobile agents without ever being delivered to BS.

Theorem 1 (Non Convergence of ZP). *For any population* $\mathbf{A}$ *of* $\mathbf{n} \geq 4$ *agents, for any* **decay** ≥ 1*, there exist a uniform cover time vector* $\overline{\mathbf{cv}}$ *and an execution of* ZP *that does not converge.*

Proof. Consider a population $\mathbf{A}$ of $\mathbf{n} \geq 4$ agents and a constant **decay** ≥ 1. We first define specific sequences of events :

- $U_1 = (1\ BS)(2\ 1)$
- $V = [(2\ 3)\dots(2\ \mathbf{n}-1)] \cdot [(3\ 4)\dots(3\ \mathbf{n}-1)] \cdot \dots \cdot (\mathbf{n}-2\ \mathbf{n}-1)$
  All mobile agents, except for agent 1, meet each other once.
- $W_1 = (1\ 2)\dots(1\ \mathbf{n}-1)$
  Agent 1 meets every other mobile agent once.
- $U_2 = (2\ BS)(1\ 2)$
- $W_2 = (2\ 1)(2\ 3)\dots(2\ \mathbf{n}-1)$
  Agent 2 meets every other mobile agent once.
- $Z = (3\ BS)\dots(\mathbf{n}-1\ BS)$
  All mobile agents, except for agents 1 and 2, meet BS.

[5] For avoiding overflow problems, we assume that the *accumulation* variables are periodically reset to 0.

We choose an integer $\mathbf{g}$ such that $\mathbf{g} \cdot (\mathbf{n} - 3) \geq \mathbf{decay} + 1$. Now we build a schedule $\mathcal{S}$ as follows :

$$X = U_1\ V^{\mathbf{g}}\ W_1^{\mathbf{g}}\ U_2\ W_2^{\mathbf{g}}\ Z$$
$$\mathcal{S} = X^{\omega}$$

By construction, in X, all the agents meet each other at least once. For any mobile agent x, we choose $\mathbf{cv}_x = \mathbf{cv} = |X|$. That implies that $\mathcal{S}$ satisfies the cover time property. Precisely, $\mathbf{cv} = \mathbf{g} \cdot \frac{(\mathbf{n}-3)(\mathbf{n}-2)}{2} + (2\mathbf{g}+1)(\mathbf{n}-2) + 3$.

We claim that the initial value v of agent 2 is never delivered to BS. To see that, consider what happens when the sequence X is applied to an initial configuration $\mathcal{C}_0$. During $U_1 = (1\ BS)(1\ 2)$, agent 1 receives the initial value v of agent 2. During the sequence $V^{\mathbf{g}}$, only agents 2 to $\mathbf{n} - 1$ are involved, thus, at the end, agent 1 still holds v. Then comes the sequence $W_1^{\mathbf{g}}$: agent 1 meets every other mobile agent $\mathbf{g}$ times. Since agents 2 to $\mathbf{n} - 1$ have not met BS yet, their variables *accumulation* equal 0 and agent 1 cannot transfer v to any of them. In addition, since agent 1 is involved in $\mathbf{g} \cdot (\mathbf{n}-2) \geq \mathbf{decay}+1$ (thanks to the choice of $\mathbf{g}$) meetings, the decay mechanism of ZP implies that at the end of $W_1^{\mathbf{g}}$, the variable $accumulation_1$ of agent 1 equals 0.

Therefore, during $U_2 = (2\ BS)(2\ 1)$, agent 1 transfers v to agent 2. In $W_2^{\mathbf{g}}$, agent 2 is involved in $\mathbf{g} \cdot (\mathbf{n}-2) \geq \mathbf{decay}+1$ meetings with other mobile agents. But all their variables *accumulation* equal 0, hence agent 2 keeps v. Note that the decay mechanism implies that at the end of $W_2^{\mathbf{g}}$, the variable $accumulation_2$ of agent 2 equals 0. Finally, during Z, all mobile agents $x \notin \{1, 2\}$ meet BS and increment their variable $accumulation_x$ accordingly. Therefore, the application of the sequence X to an initial configuration $\mathcal{C}_0$ leads to a configuration $\mathcal{C}_1$ that satisfies the property $\mathcal{P}$ defined as follows :

- agent 2 holds its initial value v
- $accumulation_1 = accumulation_2 = 0$
- $\forall x \in \mathbf{A}^* - \{1, 2\}, accumulation_x = 1$

Now, apply X to $\mathcal{C}_1$. At the end of U_1, agent 1 has received v from agent 2 and satisfies $accumulation_1 = 1$. During $V^{\mathbf{g}}$, each mobile agent $x \neq 1$ is involved in $\mathbf{g} \cdot (\mathbf{n} - 3) \geq \mathbf{decay} + 1$ meetings. Therefore, thanks to the decay mechanism, at the end of $V^{\mathbf{g}}$, all the agents, except for agent 1, have their variable *accumulation* equal to 0. Hence during $W_1^{\mathbf{g}}$, agent 1 cannot transfer v to any other mobile agents. In addition, the decay mechanism implies that at the end of $W_1^{\mathbf{g}}$, the variable $accumulation_1$ of agent 1 equals 0. Hence, we see that the same arguments as in the previous paragraph can be applied to the sequence $U_2\ W_2^{\mathbf{g}}\ Z$ that follows. Thus, the application of the sequence X to $\mathcal{C}_1$ leads to a configuration $\mathcal{C}_2$ that also satisfies the property $\mathcal{P}$.

Hence, no matter how many sequences X are applied, the initial value v of agent 2 is never delivered to BS. □

4 Modified ZebraNet Protocol 1

To ensure the convergence, we modify the algorithm by ensuring that a mobile agent that transfers data to another mobile agent can no longer accept data. For this purpose, we add a boolean variable $active_x$, initially set to **true**, that indicates whether agent x is *active* or not, and we impose that only active agents can receive values. Once an active agent has transferred its values to another *mobile* agent, it becomes *inactive*. A formal description of MZP1 is given in Algorithm 2.

Algorithm 2. Modified ZebraNet Protocol 1

when x meets BS **do**
  $<x$ transfers $values_x$ to $BS>$
  $accumulation_x := accumulation_x + 1$
  $distance_x := 0$
end when
when x meets $y \neq BS$ **do**
  if $accumulation_x < accumulation_y \wedge active_y \wedge <values_x$ is not empty$>$ **then**
    $<x$ transfers $values_x$ to $y>$
    $active_x :=$ **false**
  end if
  $distance_x := distance_x + 1$
  if $distance_x >$ **decay then**
    if $accumulation_x \neq 0$ **then**
      $accumulation_x := accumulation_x - 1$
    end if
    $distance_x := 0$
  end if
end when

4.1 Convergence of MZP1

We now show that any execution of MZP1 converges. The proof relies on the fact that the set of active agents cannot increase, so that at some point of any execution, it remains constant. From that point, there is no transfer between two mobile agents, and since all mobile agents eventually meet BS (due to the cover time property), all values are eventually delivered.

Theorem 2 (Convergence of MZP1). MZP1 *converges.*

Proof. Let $\mathcal{E}$ be an execution. We note $ACT(k)$ the set of active agents in the k-th configuration in $\mathcal{E}$. The sequence $(ACT(1), ACT(2), \dots)$ is non-increasing, thus it is eventually constant : $\exists k_0 \in \mathbb{N}, \forall k \geq k_0, ACT(k) = ACT(k_0)$. Starting from the k_0-th configuration, there cannot be any further transfer between two active agents. Otherwise, the set of active agents would decrease. Also, according to Algorithm 2, there cannot be any transfer from an active agent to another

inactive agent, nor from an inactive agent to an inactive agent. In other words, once the set of active agents remains constant, there cannot be any transfer between two mobile agents. Since all mobile agents meet BS in the next $\mathbf{cv_{max}}$ events, all the values are eventually delivered. □

4.2 Upper Bound to the MZP1 Complexity

We compute an upper bound to the number of events needed to collect all the values at the base station. First we define the notion of path.

Definition (Path followed by a value). *Let $\mathcal{E}$ be an execution and v be a value in the system. The* path followed by v in $\mathcal{E}$ *is the sequence (possibly infinite) of mobile agents that successively carry v.*

For example, let x_1 be an agent whose initial value is v. It is possible that x_1 transfers v to some agent x_2, then agent x_2 transfers v to some agent x_3 which finally delivers v to BS. In this case, the path followed by v is $x_1x_2x_3$. Note that, without the *active* variable (e.g. in ZP), agent x_1 and agent x_3 could be the same.

Theorem 3 (Upper Bound - MZP1). *For any population* $\mathbf{A}$ *of* $\mathbf{n} \geq 3$ *agents, for any cover time vector* $\overline{\mathbf{cv}}$*, and for any* **decay** ≥ 1*, any execution of* MZP1 *converges in no more than* $\sum_{x \in A^*} \mathbf{cv}_x$ *events.*

Proof. Let $\mathcal{E}$ be an execution of MZP1. By Theorem 2, $\mathcal{E}$ converges, i.e., all the values are eventually delivered. Let v be an initial value of some agent x_1 such that v is the last delivered value in $\mathcal{E}$. Consider the path π followed by v in $\mathcal{E}$. It is of the form $x_1x_2 \dots x_k$ for some $k \geq 1$, x_k being the agent that delivers v to BS. Since a mobile agent becomes inactive as soon as it transfers some values, all the agents appearing in π are different. Hence, we have $1 \leq k \leq \mathbf{n} - 1$. Then the execution $\mathcal{E}$ can be written as the following sequence of events[6] :

$$\mathcal{E} = \underbrace{\left[\dots (x_1\ \underline{x_2})^{(v)}\right]}_{e_1}\ \underbrace{\left[\dots (x_2\ \underline{x_3})^{(v)}\right]}_{e_2}\ \dots\ \underbrace{\left[\dots (x_{k-1}\ \underline{x_k})^{(v)}\right]}_{e_{k-1}}\ \underbrace{\left[\dots (x_k\ \underline{BS})^{(v)}\right]}_{e_k}\ \dots$$

The subsequence e_i starts after the transfer of v from x_{i-1} to x_i and ends with the transfer of v from x_i to x_{i+1}. At the end of e_k, v is delivered to the base station. For all $1 \leq i \leq k-1$, the length of e_i is upper bounded by $\mathbf{cv}_{x_i}$, because x_i does not meet BS in e_i (at the beginning of e_i, x_i has received v and transfers it to x_{i+1} at the very end of e_i). In addition, the length of e_k is upper bounded by $\mathbf{cv}_{x_k}$, because there the first meeting of x_k with BS necessarily occurs in the first $\mathbf{cv}_{x_k}$ events that follow the reception of v. As a consequence, the value v is delivered to BS in less than $\sum_{x \in \pi} \mathbf{cv}_x \leq \sum_{x \in \mathbf{A}^*} \mathbf{cv}_x$. Since all other values are delivered before v, $\mathcal{E}$ converges in $\sum_{x \in \mathbf{A}^*} \mathbf{cv}_x$ events. □

[6] We remind the reader that this is an abuse of notation, refer to Section 2.

4.3 Lower Bound to MZP1 Complexity

Now we present a lower bound that almost matches the upper bound of the previous section. For the sake of clarity, we assume a uniform cover time vector $\overline{\mathbf{cv}}$. Hence, the upper bound stated in Theorem 3 becomes $(\mathbf{n}-1)\cdot\mathbf{cv}$. In the sequel, we build an execution that converges in $(\mathbf{n}-2)\cdot\mathbf{cv}$, which is close to this upper bound.

Theorem 4 (Lower Bound - MZP1). *For any population* $\mathbf{A}$ *of* $\mathbf{n}\geq 4$ *agents, for any* $\mathbf{decay}\geq 1$*, there exist a uniform cover time vector* $\overline{\mathbf{cv}}$ *and an execution of* MZP1 *that does not converge in strictly less than* $(\mathbf{n}-2)\cdot\mathbf{cv}$ *events.*

Proof. We consider a population $\mathbf{A}$ of $\mathbf{n}\geq 4$ agents and a constant $\mathbf{decay}\geq 1$. Let $\mathbf{g}$ be an integer such that $\mathbf{g}\cdot(\mathbf{n}-3)\geq\mathbf{decay}+1$. We consider a uniform cover time vector $\overline{\mathbf{cv}}$, the value of which is defined later.

We build an execution in which the initial value of agent 1 is successively carried by every other agent. For each $1\leq k\leq\mathbf{n}-2$, we consider a sequence E_k of length $\mathbf{cv}$ in which the value v is transferred from agent k to $k+1$, and another sequence Δ in which agent $\mathbf{n}-1$ delivers v to BS. Since a schedule is an infinite sequence, we also consider a repeating pattern Ω and we define a schedule $\mathcal{S}=E_1E_2\cdots E_{\mathbf{n}-2}\Delta\Omega^\omega$. The difficulty lies in the definition of the sequences E_k,Δ and Ω so that the schedule $\mathcal{S}$ satisfies the cover time property and the value v is delivered at the end of Δ.

For this purpose, we define specific sequences as follows :

- For $1\leq k\leq\mathbf{n}-1$, $U(k)$ is a sequence of events in which all the mobile agents, except for agent k, meet each other once. Hence, each mobile agent (except for agent k) is involved in $\mathbf{n}-3$ meetings. We have $|U(k)|=\frac{(\mathbf{n}-3)(\mathbf{n}-2)}{2}$.
- For $1\leq k\leq\mathbf{n}-1$, $V(k)$ is a sequence in which agent k meets every other mobile agent once. We have $|V(k)|=\mathbf{n}-2$.
- For $1\leq p\leq q\leq\mathbf{n}-1$, $B_q^p=(q\ BS)(q-1\ BS)\ldots(p\ BS)$ is a sequence in which each agent x, from q to p, successively meets BS in this order. We have $|B_q^p|=q-p+1$.
- For $1\leq p\leq q\leq\mathbf{n}-1$, $C_q^p=[(q\ q+1)(q\ BS)]\ldots[(p\ p+1)(p\ BS)]$ is a sequence in which each agent x, from q to p, meets its successor $x+1$ then BS. We have $|C_q^p|=2\cdot(q-p+1)$.

First, we look at what happens when sequences such as $U(k)$ or $V(k)$ are repeatedly applied. In $U(k)^{\mathbf{g}}$, each mobile agent $x\neq k$ is involved in $\mathbf{g}\cdot(\mathbf{n}-3)\geq\mathbf{decay}+1$ meetings. Thus, thanks to the decay mechanism, applying $U(k)^{\mathbf{g}}$ to any configuration of the system makes each non-zero $accumulation_x$, with $x\neq k$, decrease at least by one. The same argument shows that applying $V(k)^{\mathbf{g}}$ to any configuration makes $accumulation_k$ decrease at least by one, unless $accumulation_k$ already equals 0. In other words, the sequences $U(k)^{\mathbf{g}}$ and $V(k)^{\mathbf{g}}$ help resetting the variables $accumulation$.

Now, consider a configuration in which for all $x\in\mathbf{A}^*$, $accumulation_x=0$. In addition, assume that some mobile agent k, such that $1\leq k\leq\mathbf{n}-2$, holds a value

w and that agent $k+1$ is active (i.e., it can receive values). Then it is easy to see that during the sequence $B_{\mathbf{n}-1}^{k+1} \cdot C_k^1 = B_{\mathbf{n}-1}^{k+2}(k+1\ BS)(k\ k+1)(k\ BS)C_{k-1}^1$, agent k transfers w to $k+1$. Moreover, at the end, every $accumulation_x$ (with x a mobile agent) equals 1. In other words, applying $B_{\mathbf{n}-1}^{k+1} \cdot C_k^1$ to the appropriate configuration results in a transfer from agent k to agent $k+1$.

We also define, for each $1 \leq k \leq \mathbf{n}-2$, a "filling" sequence F_k of meetings between mobile agents. We only require that $|F_k| = \mathbf{n}-2-k$ (which implies that $F_{\mathbf{n}-2} = \emptyset$). The purpose of the sequence F_k is to ensure that the length of E_k is constant (independent of k). Now we are ready to define the sequences E_k $(1 \leq k \leq \mathbf{n}-2)$, Δ and Ω :

$$E_k = \underbrace{U(k)^{\mathbf{g}}(k\ k+1)F_k}_{prologue} \cdot \underbrace{B_{\mathbf{n}-1}^{k+1}C_k^1}_{center} \cdot \underbrace{U(k)^{\mathbf{g}}V(k)^{\mathbf{g}}}_{epilogue}$$

$$\Delta = U(\mathbf{n}-1)^{\mathbf{g}} \cdot (\mathbf{n}-1\ BS)$$

$$\Omega = B_{\mathbf{n}-1}^{\mathbf{n}-1}C_{\mathbf{n}-2}^1 \cdot U(\mathbf{n}-1)^{\mathbf{g}}V(\mathbf{n}-1)^{\mathbf{g}} \cdot \Delta$$

Then we set $\mathbf{cv} = |E_k|$. Precisely, we have $\mathbf{cv} = \mathbf{g}\cdot(\mathbf{n}-3)(\mathbf{n}-2)+(\mathbf{g}+2)(\mathbf{n}-2)+2$. Proving that the schedule $\mathcal{S}$ satisfies the cover time property is not difficult but tedious. This proof can be found in the appendix of [4]. Instead, we focus on the circulation of the initial value v of agent 1. Let $\mathcal{C}_1$ be an initial configuration. The *prologue* of E_1 only involves meetings between mobile agents, and, since each mobile agent has its variable *accumulation* equal to 0, there is no transfer. At the end of the *center* of E_1, the previous remarks show that agent 1 has transferred v to agent 2 and each mobile agent x satisfies $accumulation_x = 1$. The *epilogue* of E_1 first begins by $U(2)^{\mathbf{g}}$ at the end of which, each mobile agent x, except for agent 2, has its variable $accumulation_x$ equal to 0. The *epilogue* ends with $V(2)^{\mathbf{g}}$ during which there is no transfer from agent 2 to any other mobile agents (their *accumulation* being equal to 0). Moreover, at the end of E_1, all mobile agents (including agent 2), have their variable *accumulation* equal to 0 and agent 2 holds the initial value v of agent 1. Also, only agent 1 has become inactive. We denote by $\mathcal{C}_2$ the configuration at the end of E_1.

If we focus on the variables *accumulation*, we see that the configuration $\mathcal{C}_2$ is similar to the configuration $\mathcal{C}_1$. Hence, the same arguments show that during E_2, agent 2 transfers v to agent 3. In the resulting configuration $\mathcal{C}_3$, all the agents have their variables *accumulation* equal to 0 again, and the process can be iterated. At the end of $E_{\mathbf{n}-2}$, agent $\mathbf{n}-1$ holds the value v. Therefore, the value v is delivered to BS exactly at the end of $\Delta = U(\mathbf{n}-1)^{\mathbf{g}}(\mathbf{n}-1\ BS)$. In summary, with the schedule $\mathcal{S}$, the algorithm does not converge before the first $(\mathbf{n}-2)\cdot\mathbf{cv}$ events. □

5 Modified ZebraNet Protocol 2

As already explained, the non convergence of ZP is due to the fact that a value can circulate between two or more mobile agents, without ever being delivered to the base station. To prevent that, in MZP1, we imposed that a mobile agent that

transfers some values cannot receive the values later. Another way to prevent cycling of values is to impose that a mobile agent receiving some values cannot transfer them to any other mobile agent later. For this purpose, an *active* bit is also introduced, but with different functionality than in MZP1. The resulting protocol, called MZP2, is given in Algorithm 3.

Algorithm 3. Modified ZebraNet Protocol 2

when x meets BS **do**
  $<x$ transfers its values to $BS>$
  $accumulation_x := accumulation_x + 1$
  $distance_x := 0$
end when
when x meets $y \neq BS$ **do**
  if $accumulation_x < accumulation_y \wedge active_x \wedge <values_x$ is not empty$>$ **then**
    $<x$ transfers its values to $y>$
    $active_y :=$ **false** // *agent y becomes inactive*
  end if
  $distance_x := distance_x + 1$
  if $distance_x >$ **decay then**
    if $accumulation_x \neq 0$ **then**
      $accumulation_x := accumulation_x - 1$
    end if
    $distance_x := 0$
  end if
end when

5.1 Upper Bound to MZP2 Complexity

Theorem 5 (Upper Bound - MZP2). *For any population* $\mathbf{A}$ *of* $\mathbf{n} \geq 1$ *agents, for any cover time vector* $\overline{\mathbf{cv}}$ *and for any* **decay** ≥ 1*, any execution of* MZP2 *converges in less than* $2 \cdot \mathbf{cv_{max}}$ *events.*

Proof. Consider an execution of MZP2 and an agent x with initial value v. During the first $\mathbf{cv_{max}}$ events, there are two possibilities. Either agent x does not transfer v to any other mobile agent then, meeting BS, it delivers v. Or, some mobile agent y has received v from agent x and has become inactive. Hence, agent y cannot transfer v to any other mobile agent, which implies that agent y will transfer v to BS during the next $\mathbf{cv_{max}}$ events. In all cases, v is delivered to the base station in less than $2 \cdot \mathbf{cv_{max}}$ events. Since v can be any value, we see that all values are delivered to the base station in less than $2 \cdot \mathbf{cv_{max}}$ events. □

5.2 Lower Bound to MZP2 Complexity

Theorem 6 (Lower Bound - MZP2). *For any population* $\mathbf{A}$ *of* $\mathbf{n} \geq 4$ *agents and any* **decay** ≥ 1*, there exist a uniform cover time vector* $\overline{\mathbf{cv}}$ *and an execution of* MZP2 *that does not converge in strictly less than* $2 \cdot \mathbf{cv} - 2$ *events.*

Proof. We consider an integer $\mathbf{g}$ such that $\mathbf{g} \cdot (\mathbf{n}-3) \geq \mathbf{decay}+1$, and we define specific sequences as follows :

- $U = (3\ BS) \ldots (\mathbf{n}-1\ BS)$.
  Agents 3 to $\mathbf{n}-1$ meet the base station once.
- $V = [(2\ 3) \ldots (2\ \mathbf{n}-1)] \cdot [(3\ 4) \ldots (3\ \mathbf{n}-1)] \cdot \ldots \cdot (\mathbf{n}-2\ \mathbf{n}-1)$
  In V, all mobile agents, except for agent 1, meet each other once.
- $W = (1\ 3) \ldots (1\ \mathbf{n}-1)$.
  Agent 1 meets every other mobile agent, except for agent 2, exactly once.
- $X = U \cdot V^{\mathbf{g}} \cdot W \cdot (2\ BS)(1\ 2)(1\ BS)$

We build a schedule $\mathcal{S}$ by repeating X infinitely many times : $\mathcal{S} = X^{\omega}$. We choose the same cover time, $\mathbf{cv} = |X|$, for all the agents. A simple calculation shows that $\mathbf{cv} = 2\mathbf{n} - 3 + \mathbf{g} \cdot \frac{(\mathbf{n}-3)(\mathbf{n}-2)}{2}$. It is easy to see that $\mathcal{S}$ satisfies the cover time property.

Now we prove that the execution of MZP2 induced by $\mathcal{S}$ does not converge before the first $2 \cdot \mathbf{cv} - 2$ events. At the end of the first U in $\mathcal{S}$, agents 3 to $\mathbf{n}-1$ have successively met BS and transferred their values to it. Thus, all the variables $accumulation_x$ for $3 \leq x \leq \mathbf{n}-1$ equal 1. Then comes the sequence $V^{\mathbf{g}}$ in which each agent $x \neq 1$ is involved in $\mathbf{g} \cdot (\mathbf{n}-3) \geq \mathbf{decay}+1$ meetings. Hence, thanks to the decay mechanism, at the end of the first $V^{\mathbf{g}}$, every agent x, from 2 to $\mathbf{n}-1$, has its variable $accumulation_x$ reset to 0. As a consequence, there is no transfer from agent 1 to any other mobile agent during the sequence W that follows $V^{\mathbf{g}}$. Then during the sequence $(2\ BS)(1\ 2)(1\ BS)$, agent 2 receives the initial value v of 1. From this point, agent 2 cannot transfer v to any other agent but BS, which is done precisely $\mathbf{cv}$ events later (during the event $(2\ BS)$ in the second X of $\mathcal{S}$). Therefore, the value v is delivered to BS exactly after the $(2 \cdot \mathbf{cv} - 2)$-th events of the schedule. □

6 Bounded Memory

Up to now, we have assumed that mobile agents have an unbounded ($O(\mathbf{n})$) memory. In this section, we discuss the case of bounded memory, i.e., a memory size independent of the number of agents. We assume now that the memory of an agent can hold at most $\mathbf{k}$ values, with $\mathbf{k} \geq 1$. Both MZP1 and MZP2 can be adapted to this case. Indeed, any transfer of values is limited by the available memory and the transfer may be *partial*. During an event, as much as possible values are transferred. Note that all values are equivalent for the data collection problem, thus it is unnecessary to precise which values are actually transferred. In an adapted MZP1, once an agent has transferred some values, even if the transfer is only partial, it becomes inactive and cannot receive other values. For every agent x, the values held by x are stored in a dynamic array $values_x$, whose size is denoted by $size(values_x)$. By definition, we have $size(values_x) \leq \mathbf{k}$. Algorithm 4 presents an adaptation of MZP1, but the same idea can be applied to MZP2. For the sake of clarity, we do not present in the code the management

Algorithm 4. Modified ZebraNet Protocol 1 - Bounded memory

```
when x meets BS do
  <x transfers its values to BS>
  accumulation_x := accumulation_x + 1
  distance_x := 0
end when
when x meets y ≠ BS do
  count := min(size(values_x), k − size(values_y))
  if accumulation_x < accumulation_y ∧ active_y ∧ count > 0 then
    <x transfers count values to y>
    active_x := false
  end if
  distance_x := distance_x + 1
  if distance_x > decay then
    if accumulation_x ≠ 0 then
      accumulation_x := accumulation_x − 1
    end if
    distance_x := 0
  end if
end when
```

of the dynamic array $values_x$. We denote by MZP1-BM (resp. MZP2-BM) the bounded-memory version of MZP1 (resp. MZP2).

It appears that, for both MZP1 and MZP2, the proofs given in the previous sections (Sections 4.1, 4.2, 4.3, 5.1 and 5.2) are still applicable. Indeed, the memory size tightens the constraints on transfers, but do not fundamentally affect the structures of both MZP1 and MZP2. Still, we sketch the proofs for MZP1-BM and MZP2-BM.

Theorem 7 (Bounds to MZP1-BM complexity). *For any population* $\mathbf{A}$ *of* $\mathbf{n} \geq 1$ *agents, for any cover time vector* $\overline{\mathbf{cv}}$*, for any* $\mathbf{decay} \geq 1$*, any execution of* MZP1-BM *converges in less than* $\sum_{x \in \mathbf{A}^*} \mathbf{cv}_x$ *events.*

For any population $\mathbf{A}$ *of* $\mathbf{n} \geq 4$ *agents, for any* $\mathbf{decay} \geq 1$*, there exist a uniform cover time vector* $\overline{\mathbf{cv}}$ *and an execution of* MZP1-BM *that does not converge in strictly less than* $(\mathbf{n} - 2) \cdot \mathbf{cv}$ *events.*

Proof. The fact that MZP1-BM converges is due to the fact that the set of active agents cannot increase. As in MZP1, once the set of active agents remains constant, there cannot be any transfer between any two mobile agents. Since all mobile agents meet BS in the next $\mathbf{cv_{max}}$ events, the protocol converges.

The upper bound to the complexity of MZP1-BM is computed by looking at the path followed by the last delivered value v, i.e., the mobile agents that successively carry v. The memory size does not affect the fact that a mobile agent in this path cannot appear twice, thanks to the bit *active*, nor the fact that a mobile agent x in this path holds v for at most $\mathbf{cv}_x$ consecutive events. Thus any execution of MZP1-BM converges in less than $\sum_{x \in \mathbf{A}^*} \mathbf{cv}_x$ events.

The lower bound to MZP1-BM complexity is obtained thanks to the same schedule described in Section 4.3. Indeed, applying this schedule to an initial configuration gives an execution in which each agent holds at most one value, which is compatible with the assumption $\mathbf{k} \geq 1$. □

Theorem 8 (Bounds to MZP2-BM complexity). *For any population* $\mathbf{A}$ *of* $\mathbf{n} \geq 1$ *agents, for any cover time vector* $\overline{\mathbf{cv}}$*, for any* $\mathbf{decay} \geq 1$*, any execution of* MZP2-BM *converges in less than* $2 \cdot \mathbf{cv_{max}}$ *events.*

For any population $\mathbf{A}$ *of* $\mathbf{n} \geq 4$ *agents, for any* $\mathbf{decay} \geq 1$*, there exist a uniform cover time vector* $\overline{\mathbf{cv}}$ *and an execution of* MZP2-BM *that does not converge in strictly less than resp.* $2 \cdot \mathbf{cv} - 2$ *events.*

Proof. During the first $\mathbf{cv_{max}}$ events, an agent x either transfers its initial value v to BS or to another mobile agent y. In the second case, agent y is then inactive and cannot transfer v to any other agent, but BS, which is done in the next $\mathbf{cv_{max}}$ events. Thus MZP2-BM also converges in less than $2 \cdot \mathbf{cv_{max}}$ events.

The lower bound to MZP2-BM is obtained thanks to the same schedule described in Section 5.2. Indeed, applying this schedule to an initial configuration gives an execution in which each agent holds at most one value, which is compatible with the assumption $\mathbf{k} \geq 1$. □

7 Conclusion

In this paper, we study the ZebraNet data collection protocol in the context of Population Protocols. We show that the original version does not converge in all cases, the problem being the possibility for a value to cycle among the mobile agents without reaching the base station.

To ensure convergence, we propose slightly modified versions of the original protocol, MZP1 and MZP2. Notice that MZP1 is a multi-hop protocol. In contrast, MZP2 is a two-hop one. Hence, MZP1 approximates better the original ZebraNet protocol than MZP2. For both modified versions, the worst case complexity is much worse than for the near optimal data collection protocol presented in [5] (its complexity is less than $2 \cdot \mathbf{cv_{min}}$). However, this protocol assumes that, when two agents meet, both know which of them has a smaller cover time. We do not make such an assumption here, but one could consider that the ZebraNet Protocol is an approximation of the near optimal protocol in the following sense. An agent that has met BS many times in the past, has intuitively to be fast and thus, must have a small cover time. Comparing the values of the accumulation variables, when two agents meet, can be viewed as an approximation of comparing their cover times. This papers shows that this approximation is bad when the worst case complexity is considered. Note that optimal bounds to the worst case complexity can be found in [4]; precisely $\sum_{x \in \mathbf{A}^*} \mathbf{cv}_x - 2 \cdot (\mathbf{n} - 2)$ for MZP1/MZP1-BM and $2 \cdot \mathbf{cv_{max}} - 2$ for MZP2/MZP2-BM. A possible, but surely difficult extension to this work would be to compute the average complexity of the protocols. Perhaps the gap between the protocol in [5] and the protocols

MZP1 and MZP2 is not so large when considering average complexity. Such an analysis would also highlight the role of the memory size.

Another perspective would be to apply our purely analytical methodology to more intricate data collection protocols, as for instance PROPHET [17], for which only simulation results are available. For this protocol, as well as for others, the analytical approach is not supposed to replace simulations, but allows to obtain some information quickly and with less investment.

References

1. Angluin, D., Aspnes, J., Diamadi, Z., Fisher, M., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: PODC, pp. 290–299 (2004)
2. Angluin, D., Aspnes, J., Eisenstat, D.: Fast computation by population protocols with a leader. DC 21(3), 183–199 (2008)
3. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)
4. Beauquier, J., Blanchard, P., Burman, J., Delaet, S.: Exact time complexity of zebranet with cover times. LRI Internal Report (2011)
5. Beauquier, J., Burman, J., Clément, J., Kutten, S.: On utilizing speed in networks of mobile agents. In: PODC, pp. 305–314 (2010)
6. Beauquier, J., Burman, J., Kutten, S.: A self-stabilizing transformer for population protocols with covering. Theor. Comput. Sci. 412(33), 4247–4259 (2011)
7. Beauquier, J., Clément, J., Messika, S., Rosaz, L., Rozoy, B.: Self-stabilizing counting in mobile sensor networks with a base station. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 63–76. Springer, Heidelberg (2007)
8. Cai, H., Eun, D.Y.: Crossing over the bounded domain: from exponential to power-law inter-meeting time in MANET. In: MOBICOM, pp. 159–170 (2007)
9. Chaintreau, A., Hui, P., Crowcroft, J., Diot, C., Gass, R., Scott, J.: Impact of human mobility on opportunistic forwarding algorithms. IEEE Transactions on Mobile Computing 6, 606–620 (2007)
10. College, D.: The dartmouth wireless trace archive (2007)
11. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988)
12. Fischer, M., Jiang, H.: Self-stabilizing leader election in networks of finite-state anonymous agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 395–409. Springer, Heidelberg (2006)
13. Guerraoui, R., Ruppert, E.: Even small birds are unique: Population protocols with identifiers. Technical Report CSE-2007-04. York University (2007)
14. Hong, S., Rhee, I., Kim, S.J., Lee, K., Chong, S.: Routing performance analysis of human-driven delay tolerant networks using the truncated levy walk model. In: MobilityModels, pp. 25–32 (2008)
15. Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L.-S., Rubenstein, D.: Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In: ASPLOS, pp. 96–107 (2002)
16. Karagiannis, T., Boudec, J.L., Vojnovic, M.: Power law and exponential decay of inter contact times between mobile devices. In: MOBICOM, pp. 183–194 (2007)
17. Lindgren, A., Doria, A., Schelén, O.: Probabilistic routing in intermittently connected networks. SIGMOBILE Mob. Comput. Commun. Rev. 7, 19–20 (2003)
18. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press, Cambridge (2001)

Building Self-stabilizing Overlay Networks with the Transitive Closure Framework*

Andrew Berns, Sukumar Ghosh**, and Sriram V. Pemmaraju***

Department of Computer Science
The University of Iowa
14 MacLean Hall
Iowa City, Iowa, USA 52242
{andrew-berns,sukumar-ghosh,sriram-pemmaraju}@uiowa.edu

Abstract. Overlay networks are expected to operate in hostile environments, where node and link failures are commonplace. One way to make overlay networks robust is to design self-stabilizing overlay networks, i.e., overlay networks that can handle node and link failures without any external supervision. In this paper, we first describe a simple framework, which we call the *Transitive Closure Framework* (TCF), for the self-stabilizing construction of an extensive class of overlay networks. Like previous self-stabilizing overlay networks, TCF permits node degrees to grow to $\Omega(n)$, independent of the maximum degree of the target overlay network. However, TCF has several advantages over previous work in this area: (i) it is a "framework" and can be used for the construction of a variety of overlay networks, not just a particular network, (ii) it runs in an optimal number of rounds for a variety of overlay networks, and (iii) it can easily be composed with other non-self-stabilizing protocols that can recover from specific bad initial states in a memory-efficient fashion. We demonstrate the power of our framework by deriving from TCF a simple self-stabilizing protocol for constructing Skip+ graphs (Jacob et al., PODC 2009) which presents optimal convergence time from any configuration, and requires only a $O(1)$ factor of extra memory for handling node Joins.

1 Introduction

An *overlay network* is induced by logical or virtual links constructed over one or more underlying physical links. The use of virtual links enables designers to create any topology regardless of the underlying physical network, allowing the creation of networks with desirable properties, such as low diameter and mean path length (for efficient routing), low degree (for low memory requirements and maintenance overhead), low congestion, etc. Fault tolerance in overlay networks is an important focus for researchers and practitioners alike. Since nodes

* An early version of this work appeared as a Brief Announcement in PODC 2010.
** This work is supported in part by National Science Foundation grant CNS-0956780.
*** This work is supported in part by National Science Foundation grant CCF 0915543.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 62–76, 2011.

and links do not typically exist in stable and controlled environments, overlay networks must be prepared to handle unexpected node and link failures. Traditionally, overlay networks are classified into two categories: structured and unstructured. Unstructured networks have no topological restrictions other than connectedness, and they are relatively simple to manage. Conversely, structured networks have "hard" topological constraints and recovery from bad configurations is a key challenge for such networks. As a "toy" example, consider the LINEAR network that consists of a path of nodes arranged in the order of node identifiers. For the LINEAR network, bad configurations include those in which a node perceives two neighbors both with smaller IDs or those in which a node has three or more neighbors. Such bad configurations may be caused by node or link failures, by a node JOIN or a node LEAVE, or by deliberate actions of nodes trying to derive undue performance benefits for themselves. For these reasons, *self-stabilization* [3] is extremely relevant in overlay network construction. Recently the design of self-stabilizing overlay networks has received considerable attention – examples include algorithms for constructing *double-headed radix trees* [2], the LINEAR network [6], SKIP+ graphs [4], and CHORD-like networks [5]. The goal in these papers is to design self-stabilizing algorithms for overlay network construction; these algorithms run on the individual nodes of a weakly-connected network and, by node actions that include edge-additions and edge-deletions, the network is transformed into a legal overlay network. In a sense, these algorithms are taking a walk through the space of all networks defined by a given set of nodes, starting from a source network that is illegal and ending up at a target network that is legal. This algorithmic process is conceptually no different from, for e.g., starting with a (possibly unbalanced) binary search tree and transforming it into a *balanced* binary search tree (e.g., AVL tree) via a series of local rotations. However, since we are in a distributed setting, it is appropriate that the illegality of a network be detected using only local information and fixed using local actions.

It is worth reemphasizing at the outset that since the edges of an overlay network are virtual and, in a sense, independent of the underlying physical links, these edges can be deleted and inserted as a result of program actions, or as a result of events such as JOINS and LEAVES. This aspect of our model — the fact that the network may change repeatedly as the result of algorithm actions — makes it fundamentally different from other standard models of distributed computation such as $\mathcal{LOCAL}$ and $\mathcal{CONGEST}$ [7]. For example, it is easy to see that $\Omega(\text{Diam}(G))$ is a lower bound on the problem of constructing a minimum spanning tree (MST) of a network G in a distributed setting in the $\mathcal{LOCAL}$ model. However, this lower bound is mainly due to the fact that the underlying network G has to remain static – once we allow program actions to modify the underlying network, this lower bound no longer holds and using techniques described in this paper, it is easy to see that $O(\log n)$ rounds suffice for MST construction, independent of $\text{Diam}(G)$. In the current model of computation, two efficiency metrics seem to make most sense [6]. The first is the traditional worst-case number of rounds needed to terminate or stabilize (i.e., reach the target

overlay network), and the second is the the maximum increase in the degree of a node during algorithm execution. This second measure may be viewed as the amount of "extra memory" that nodes consume during algorithm execution. Another view of this measure is as follows. Typically, overlay networks have small maximum degree (relative to number of nodes in the network) and if we start off with an illegal overlay network in which all degrees are small, then the requirement that the maximum node degree increase be bounded forces the algorithms we construct to travel through the space of only low-degree networks before reaching its destination overlay network. Ideally, from the point of view of scalability, both measures should be sublinear, preferably polylogarithmic, in the number of nodes currently in the network. However, no existing algorithms seem to have achieved this for non-trivial overlay networks. For example, Jacob et al. [4] present a self-stabilizing algorithm for building a SKIP+ graph in $O(\log^2 n)$ rounds (with high probability). However, the worst-case memory requirements of this algorithm are linear, and the algorithm and its proof of correctness are both quite complex.

1.1 Our Contributions

In this paper, we first describe a simple framework, which we call the *Transitive Closure Framework* (TCF), for the self-stabilizing construction of an extensive class of overlay networks. This is a "framework" rather than an algorithm and by instantiating certain subroutines in this framework we can obtain self-stabilizing algorithms for specific overlay networks. Like previous self-stabilizing overlay networks, TCF permits node degrees to grow to $\Omega(n)$, independent of the maximum degree of the target overlay network. However, TCF has several advantages over previous work in this area: (i) it is a "framework" and can be used for the construction of a variety of overlay networks, not just a particular network, (ii) it runs in optimal number of rounds for a variety of overlay networks, and (iii) it can easily be composed with other non-self-stabilizing protocols that can recover from specific bad initial states in a memory-efficient fashion. We elaborate on items (ii) and (iii) below.

- We identify a natural parameter of overlay networks, namely the *detector diameter*. Consider a set of nodes V, a legal overlay network $G = (V, E)$ on V, and a faulty network $G_f = (V, E_f)$ on V. Typically, the diameter of G, denoted $\text{Diam}(G)$, is small relative to $|V|$, whereas $\text{Diam}(G_f)$ may be much larger than $\text{Diam}(G)$. Now let $V' \subseteq V$ denote a set of node that are "detectors," i.e., nodes in G_f whose local states alert them to the fact that the overlay network is faulty (definitions of these notions appear in Sect. 1.2. Assume for now that V' is non-empty). The *detector diameter* $D(n)$ is the maximum distance in G_f between a non-detector node (i.e., a node in $V \setminus V'$) and its closest detector and serves as a measure of how well "detectors" are distributed in G_f. We show that any algorithm obtained from TCF

can transform G_f to G in $O(D(n) + \log n)$ rounds, where $n = |V|$. In other words, the *stabilization time* of algorithms obtained from the TCF is $O(D(n) + \log n)$ rounds. To place this in context, we then show a lower bound: the worst-case self-stabilization time of an algorithm derived from TCF is bounded below by $\Omega(\text{Diam}(G))$. The natural question to ask is: what is the gap between the lower bound $\Omega(\text{Diam}(G))$ and the upper bound $O(D(n)+\log n)$? It may seem as though $D(n)$ can be as large as $\text{Diam}(G_f)$), which, as we mentioned earlier, can be much larger than $\text{Diam}(G)$. For a wide variety of overlay networks, we show that the detector diameter $D(n)$ is no more than $\text{Diam}(G) + 1$, thus showing that the self-stabilization time of algorithms derived from TCF is within an additive logarithmic-factor of the optimal time.

– The above discussion of the self-stabilization time of TCF ignores the maximum node degree increase allowed during recovery by TCF. As mentioned earlier, TCF requires all node degrees to become $\Theta(n)$ during the algorithm execution before the degrees drop down to their final values in the target overlay network G. However, to offset this memory-inefficiency we show that TCF can be easily composed with other, (possibly non-self-stabilizing) overlay network protocols that can deal with specific initial states in a memory-efficient manner. We introduce the Local Repair Framework (LRF), which allows for the efficient repair of certain failures. To demonstrate this, we create a JOIN protocol for SKIP+ graphs that (i) stabilizes from arbitrary initial configurations in $O(D(n) + \log n)$ rounds, while permitting an $O(n)$ degree increase and (ii) stabilizes from a single-JOIN state in $O(\log n)$ rounds, while permitting only an $O(1)$ degree increase.

Finally, we demonstrate the power of our framework by deriving from TCF, a simple self-stabilizing protocol for constructing SKIP+ graphs [4]. The SKIP+ graph is a locally-checkable extension of SKIP graphs [1]. We show that the detector diameter for an n-node SKIP+ graph is $O(\log n)$ and therefore our algorithm runs in $O(\log n)$ rounds, exactly matching the lower bound of $\Omega(\log n)$, which follows from the well-known fact that the diameter of an n-node SKIP+ graph is $\Theta(\log n)$. Since a single-node JOIN operation can be performed in $O(\log n)$ rounds by performing the composition alluded to above, we obtain a self-stabilizing overlay network that (i) stabilizes from arbitrary initial configurations in $O(\log n)$ rounds (which is optimal), while permitting an $O(n)$ degree increase and (ii) stabilizes from a single-JOIN state in $O(\log n)$ rounds, while permitting only an $O(1)$ degree increase.

1.2 Model

Let V be a set of nodes. We suppose that there are two functions $\texttt{id} : V \to \mathbb{Z}^+$ and $\texttt{rs} : V \to \{0,1\}^*$ that associate with each node in V a unique identifier and a random bit string. The association of `id`-values to nodes is adversarial, however it is assumed that the adversary assigns `id`-values to nodes without having access to their `rs`-values. A *family of overlay networks* is defined as a mapping ON :

$\Lambda \rightarrow \mathcal{G}$, where Λ is the set of all triples $\lambda = (V, \texttt{id}, \texttt{rs})$ and $\mathcal{G}$ is the set of all directed graphs. In other words, the family of overlay networks associates a unique directed graph $ON(\lambda) \in \mathcal{G}$ with each labeled set $\lambda = (V, \texttt{id}, \texttt{rs})$ of nodes. At this point, the only assumption we make about ON is that it is well-defined for every member $\lambda \in \Lambda$.

Let E be an arbitrary set of directed edges on V such that the graph $G = (V, E)$ is weakly connected. Our goal is to design an algorithm that starts on any given labeled directed graph $(G = (V, E), \texttt{id}, \texttt{rs})$ and computes the overlay network $ON(\lambda)$, where $\lambda = (V, \texttt{id}, \texttt{rs})$. Such an algorithm is a self-stabilizing algorithm for computing the family of overlay networks ON. We now explain what it means precisely for an algorithm to compute $ON(\lambda)$. Each node $v \in V$ maintains, over the course of the algorithm, a set of out-neighbors $N(v)$, as part of its local state. $N(v)$ is node v's view of the network that it is part of. Initially, the sets $N(v)$, for all nodes $v \in V$ induce E (the input set of directed edges). We use $S(v)$ to denote the local state of a node $v \in V$. $S(v)$ consists of $N(v)$ plus additional local variables that presumably help node v in its computations. We assume a synchronous message-passing model. In each synchronous round, node v reads the messages it received in the previous round, updates $N(v)$ if necessary, and send out messages to all the nodes in $N(v)$. Message sizes are assumed to be unbounded and typically in each round the messages sent by v consist of the `id`-values and `rs`-values of all the nodes $N(v)$. Note that throughout the algorithm, node v can communicate with all of its current out-neighbors, i.e., nodes in $N(v)$ in one round of communication. Since $N(v)$ is continuously changing, which nodes v is able to communicate with in one round is also continuously changing as the algorithm executes. As mentioned before, this aspect of our model makes it fundamentally different from other standard models of distributed computation such as $\mathcal{LOCAL}$ and $\mathcal{CONGEST}$ [7].

Let $N_\lambda(v)$ denote the set of out-neighbors of node v in overlay network $ON(\lambda)$. A node $v \in V$ is said to be *faulty* in a particular round if its current set of out-neighbors $N(v)$ is not equal to $N_\lambda(v)$. The network is said to be *faulty* in a particular round if some node in V is faulty. The goal of our algorithm is to lead the network into a non-faulty state. Note that a faulty node may not know that it is faulty. This is because v is only aware of its current 2-neighborhood and within its 2-neighborhood everything may seem fine. More precisely, let V' denote the current 2-neighborhood of v, let $\texttt{id}'$ and $\texttt{rs}'$ be the restrictions of `id` and `rs` respectively to V', and let λ' denote the triple $(V', \texttt{id}', \texttt{rs}')$. Node v may be able compute its "local" overlay network $ON(\lambda')$ and this may be identical to the edges that v sees in its 2-neighborhood. In such a case, node v has no reason to believe that it is faulty, though it may be faulty because of other nodes in V that are outside its 2-neighborhood.

1.3 SKIP+ Graphs

In this paper, we use Skip+ graphs to illustrate the utility of TCF. We define Skip+ graphs in this section. Skip graphs were first introduced in 2003 by Aspnes and Shah [1] as a fault-tolerant distributed data structure for efficient

searching in peer-to-peer systems. In a Skip graph, each node u has a unique identifier $u.id$, as well as a random sequence, $u.rs$. To help with defining Skip graphs and Skip+ graphs, we provide the following notation, taken with slight modification from [4].

- $pre_i(u)$: for any node u and nonnegative integer i, $pre_i(u)$ denotes the leftmost i bits of $u.rs$
- $pred(u, W)$: for any node u and subset W of nodes, $pred(u, W)$ is the node in the set W with largest id whose id is less than $u.id$.
- $succ(u, W)$: for any node u and subset W of nodes, $succ(u, W)$ is the node in the set W with smallest id whose id is more than $u.id$.

A legal Skip graph consists of levels labeled $0, 1, 2, \ldots$. At each level i, a node u is neighbors with at most two nodes: $pred(u, \{w|pre_i(w) = pre_i(u)\})$ and $succ(u, \{w|pre_i(w) = pre_i(u)\})$, assuming such nodes exist. It is shown in [1] that with high probability (w.h.p.), the degree of each node in a Skip graph is $O(\log n)$ and furthermore there is a simple protocol that can search for a node in $O(\log n)$ rounds. While these are highly desirable properties, a problem for designing self-stabilizing Skip graphs is that Skip graphs are not *locally checkable*. To see this consider Fig. 1. Here, each node believes the topology is correct, when in fact this is not a legal Skip graph, as there should be an edge between the leftmost node (`id` 1) and the rightmost node (`id` 22) nodes in Level 1. However, these two nodes are unware of the existence of each other and other nodes in the vicinity are unable to help.

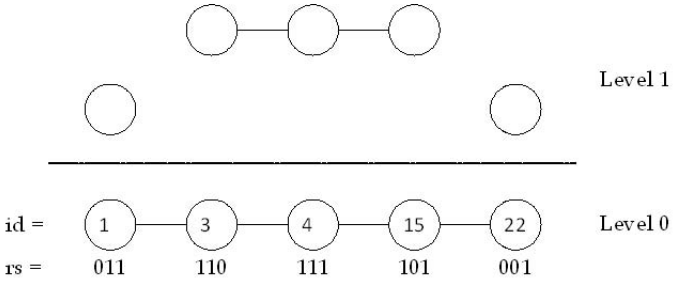


Fig. 1. An illegal Skip graph in which none of the nodes are able to detect a fault

To create a locally-checkable version of the Skip graph, additional links are needed and this leads us to the definition of Skip+ graphs [4]. In a Skip+ graph, as in a Skip graph, each node u has a unique identifier $u.id$, as well as a random sequence $u.rs$. Below we present a few additional definitions that we need for defining Skip+ graphs.

- For node u, nonnegative integer i, and $x \in \{0, 1\}$, $pred_i(u, x) = pred(u, \{w \mid pre_{i+1}(w) = pre_i(u) \cdot x\})$. In words, $pred_i(u, x)$ is the predecessor for u, selected from all nodes who have the same length-i prefix as u and whose $(i + 1)$ bit is x.

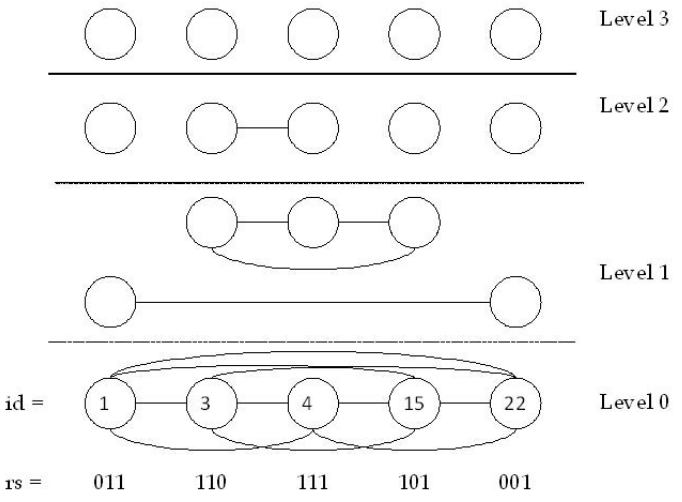


Fig. 2. A Skip+ Graph

- For node u, nonnegative integer i, and $x \in \{0, 1\}$, $succ_i(u, x) = succ(u, \{w \mid pre_{i+1}(w) = pre_i(u) \cdot x\})$.
- For node u and nonnegative integer i, $low_i(u) = \min\{pred_i(u, 0).id, pred_i(u, 1).id\}$.
- For node u and nonnegative integer i, $high_i(u) = \max\{succ_i(u, 0).id, succ_i(u, 1).id\}$.
- $range_i(u) = [low_i(u), high_i(u)]$.

In a legal Skip+ graph, a node u has edges to all nodes v such that $v.id \in range_i(u)$ and $pre_i(u) = pre_i(v)$. These nodes v are referred to as the *level-i neighbors* of u. A legal Skip+ graph is shown in Fig. 2.

2 Transitive Closure Framework

The *Transitive Closure Framework* (TCF) is shown in Program 1. TCF uses a predicate called `DETECT` and a subroutine called `REPAIR` – these need to be instantiated appropriately for specific families of overlay networks. To describe TCF we first introduce some notation and definitions.

Let $\lambda = (V, \mathtt{id}, \mathtt{rs})$ be a labeled set of nodes. Let $G = (V, E)$ be an arbitrary directed graph on the set of nodes V. Recall that $N(v)$ is the set of out-neighbors of v, i.e., $\{w \mid (v, w) \in E\}$. For each node v let $\mathtt{id}_v$ and $\mathtt{rs}_v$ respectively be the restrictions of `id` and `rs` to $N(v)$. Let $N^2(v)$ denote the set of "at most 2-hop out-neighbors" of v, i.e., $\{v\} \cup N(v) \cup \{w \mid (u, w) \in E \text{ and } u \in N(v)\}$. Let $\mathtt{id}_v^2$ and $\mathtt{rs}_v^2$ be the functions `id` and `rs` restricted to the set $N^2(v)$. Then $G_v^2 := ON(N^2(v), \mathtt{id}_v, \mathtt{rs}_v)$ is the legal overlay network on $N^2(v)$. For node v,

based on its local information, this is the "local" version of the overlay network it wants to see. Let E_v be the set of all edges that node v is aware of, i.e., $E_v = \{(v,u)|u \in N(v)\} \cup \{(u,w)|u \in N(v)\}$.

Definition 1. *The* `DETECT` *predicate at a node v, evaluated over λ^2, E_v, is true exactly when $E_v \subseteq E(G_v^2)$; otherwise the* `DETECT` *predicate is false. A node v is called a* detector *if the* `DETECT` *predicate evaluates to true at node v.*

For an example of a `DETECT` predicate, consider the LINEAR graph again [6]. Here, for a given $\lambda = (V, id, rs)$, G_λ is a path $(v_1, v_2, \ldots, v_n)$ formed by nodes in V such that $v_1.id < v_2.id < \cdots < v_n.id$. In this case, the `DETECT` predicate at a node v is true iff the 2-neighborhood of v does not induce a path (with at most 5 nodes) that is sorted by `id`s.

Definition 2. *Given $\lambda = (V,$* `id`*,* `rs`*), the* `REPAIR` *subroutine at node u sets the out-neighborhood of u to $N_\lambda(u)$ in one round.*

Each node u has an associated boolean variable $detect_u$ that becomes true in one of two ways: (i) when the predicate `DETECT` is true for u (Line 4) and (ii) when $detect_v$ is true for some neighbor $v \in N(u)$ in a previous round (Line 10). $detect_u$ becoming true in the second manner causes the spreading of the "$detect = true$" event through the network. As this event spreads through the network, all nodes for which the boolean variable *detect* is true participate in the transitive closure process (Line 9). In other words, each node with $detect = true$ in its neighborhood, expands its out-neighborhood to include all the out-neighbors of its out-neighbors (Line 9). Soon enough, every node has $detect = true$ and the network becomes a clique. At this point (which is detected in Line 5), the ideal neighborhood of each node can be built using the `REPAIR()` subroutine (Line 6).

Because algorithms derived from the TCF are initiated by detectors, the efficiency of these algorithms depends, to a significant extent, on the distribution of detectors in the network. For overlay networks that are not locally checkable (e.g., SKIP graphs) the presence of even a single detector is not guaranteed when the network is faulty – this is in fact the primary motivation of Jacob et al. [4] for defining SKIP+ graphs. To formalize the notion of the distribution of detectors, we define the *detector diameter* $D(n)$ of any family ON of overlay networks as follows.

Fix a family of overlay networks ON. Let $\lambda = (V,$ `id`$,$ `rs`$)$ be a labeled set of nodes and let E be an arbitrary set of directed edges on V such that $G = (V, E)$ is weakly connected. Think of G as representing a possibly faulty overlay network and let $D \subseteq V$ be the set of detectors in G. The nodes in D are independent of any specific algorithm – whether or not a node is a detector depends solely upon the ideal network configuration (i.e., $ON(\lambda)$) and the current network configuration (i.e., G). Note that D may be empty, either because $G = ON(\lambda)$ or because even if $G \neq ON(\lambda)$, no node is able to detect this. If we assume that ON is a family of locally checkable overlay networks then $G \neq ON(\lambda)$ implies that $D \neq \emptyset$. The *detector diameter of G with respect to λ*, denoted $D_\lambda(G)$, is the maximum hop distance in G between any node in V and the closest detector. The

Program 1. Transitive Closure Framework

Program for process u
Variables: neighborhood $N(u)$, Boolean $detect_u$

in each round do
1. Send $N(u)$, $\texttt{id}_u$, $\texttt{rs}_u$ to every neighbor $v \in N(u)$
2. Receive $N(v)$, $\texttt{id}_v$, and $\texttt{rs}_v$ from each $v \in N(u)$
3. Compute from this information: $\lambda^2 := (N^2(v), \texttt{id}_v^2, \texttt{rs}_v^2)$ and
$$E_v := \{(v,u) | u \in N(v)\} \cup \{(u,w) | u \in N(v)\}$$
4. $detect_u \leftarrow \texttt{DETECT}(\lambda^2, E_v) \vee detect_u$
5. **if** $detect_u \wedge \forall v \in N(u) : (detect_v \wedge (\{N(v) \cup v\} = \{N(u) \cup u\}))$ **then**
6. $\quad N(u) \leftarrow \texttt{REPAIR}(N(u) \cup u)$
7. $\quad detect_u \leftarrow false$
8. **else if** $detect_u \vee (\bigvee_{v \in N(u)} detect_v)$ **then**
9. $\quad N(u) \leftarrow N(u) \cup \{\bigcup_{v \in N(u)} N(v)\}$ //transitive closure
10. $\quad detect_u \leftarrow true$
11. **fi**

od

implication of this definition is that if the initial state of the system is network G, then some node v in G is $D_\lambda(G)$ hops from the closest detector and thus the TCF algorithm initiated by detectors requires $D_\lambda(G)$ rounds to reach v. The *detector diameter* $D(n)$ of a family ON of overlay networks is the maximum of $D_\lambda(G)$ over all $\lambda = (V, id, rs)$ with $|V| = n$ and all weakly connected, directed networks $G = (V, E)$. It is worth noting that if random strings are indeed used to define the family ON of overlay networks, then λ, D_λ, and $D(n)$ are all random variables.

We are able to show the following upper bound on the self-stabilization time of TCF.

Theorem 1. *The Transitive Closure Framework presented in Program 1 is a self-stabilizing algorithm for constructing any locally-checkable family of overlay networks in at most* $D(n) + \log(n) + 1$ *rounds.*

The proof of this theorem is omitted from the paper due to space constraints and will appear in the full version of the paper. The intuition behind the upper bound is that it takes $D(n)$ rounds (in the worst case) for all nodes to realize that the network is faulty. Subsequently, all nodes are participating in the transitive closure process. This process reduces the diameter of the network by a factor of 2 in each round and as a result the network becomes a clique in an additional $\log n$ rounds.

2.1 A Lower Bound

This section is devoted to the proof of the following lower bound on the stabilization time for constructing locally checkable overlay networks.

Theorem 2. *Let ON denote any family of locally checkable overlay networks and $Diam(ON(n))$ denote the maximum diameter of any n-node member of ON. Any silent self-stabilizing algorithm for constructing ON takes $\Omega(Diam(ON(n)))$ time, in the worst case.*

Proof. Let $\lambda = (V, \texttt{id}, \texttt{rs})$ with $|V| = n$. Let $G = ON(\lambda)$ and suppose that $d = \text{Diam}(G)$. There exists a shortest path consisting of distinct nodes $p_0, p_1, \cdots, p_d$ in the network G. Let $V' = V \setminus \{p_0\}$ and $\texttt{id}'$ and $\texttt{rs}'$ be restrictions of $\texttt{id}$ and $\texttt{rs}$ respectively to V'. Let $\lambda' = (V', \texttt{id}', \texttt{rs}')$ and $G' = ON(\lambda')$. There are two cases concerning the distance between nodes p_1 and p_d in G'.

Case 1: $dist_{G'}(p_1, p_d) > \frac{d}{2}$. If the distance between nodes p_1 and p_d remains at least $\frac{d}{2}$ in G', then we can insert node p_0 as a neighbor to node p_d. The network is now faulty, as p_0 must be a neighbor of p_1 (and vice-versa) in the ideal configuration. Furthermore, only p_d and its immediate neighbors have knowledge of node p_0, and these nodes are at least $\frac{d}{2}$ away from node p_1, a node that needs to change its local state. Therefore, the self-stabilization time from such a state is at least $\frac{d}{2}$.

Case 2: $dist_{G'}(p_1, p_d) < \frac{d}{2}$. Let nodes p_1 and p_d be closer than $\frac{d}{2}$ in G'. A node within $\frac{d}{2}$ of p_d must have then changed its neighborhood from G to G', or else the set of nodes within $\frac{d}{2}$ of p_d will not have changed, and p_1 remains at least $\frac{d}{2}$ from p_d. Notice, too, that all nodes within $\frac{d}{2}$ of p_d in G are also at least $\frac{d}{2}$ from p_1 in G. Therefore, if node p_0 is removed from G and the network is *not* reconfigured, only node p_1 and its immediate neighbors are detectors, and a faulty node exists at least $\frac{d}{2}$ away from p_1. Thus, removing p_0 from G results in a network from which stabilization takes at least $d/2$ rounds.

2.2 Bounding the Detector Diameter

Theorem 1 yields an $O(D(n) + \log n)$ upper bound on the self-stabilization time of TCF, whereas Theorem 2 yields an $\Omega(\text{Diam}(ON(n)))$ lower bound on the running time of any silent self-stabilizing algorithm. How close are these bounds to each other? In the following we show that for a wide variety of overlay networks $D(n)$ and $\text{Diam}(ON(n))$ are asymptotically identical implying that the upper bound is only an $O(\log n)$ additive factor larger than the lower bound. Another consequence of this is that for overlay networks for which $\text{Diam}(ON(n)) = \Omega(\log n)$, the upper and lower bounds are identical. We start by identifying families of overlay networks for which $D(n) = O(\text{Diam}(ON(n)))$ by stating an "axiom" they need to satisfy.

Axiom 1. ***[Subgraph Monotonicity]*** *Let $\lambda = (V, \texttt{id}, \texttt{rs})$ and $G = (V, E)$ be an arbitrary directed graph on V. For $u \in V$ and nonnegative integer k, let $B_k(u)$ be the set of nodes in G that are at most k hops from u in G. Let $\lambda_k(u) = (B_k(u), \texttt{id}_k(u), \texttt{rs}_k(u))$, where $\texttt{id}_k(u)$ and $\texttt{rs}_k(u)$ are the respective restrictions of $\texttt{id}$ and $\texttt{rs}$ to $B_k(u)$. If none of the nodes in $B_k(u)$ are detectors*

in G, then $G[B_k(u)]$ (i.e., the subgraph of G induced by $B_k(u)$) is identical to $ON(\lambda_k(u))$.

Notice that many overlay networks satisfy subgraph monotonicity, including Skip+ and Linear networks. In fact, subgraph monotonicity may be an inherent property of locally checkable networks, although more investigation is required to support this possibility.

Theorem 3. *Let $\lambda = (V, \texttt{id}, \texttt{rs})$ and $G = (V, E)$ be an arbitrary directed graph on V. Let ON be a family of overlay networks satisfying Subgraph Monotonicity. Then $D_\lambda(G) \leq Diam(ON(n)) + 1$.*

This result is surprising in that $\text{Diam}(n)$ may grow quite large compared to the size of $\text{Diam}(ON(\lambda))$. However, increasing the diameter of a faulty configuration simply adds more detectors, meaning worst-case convergence time does not grow with $\text{Diam}(n)$.

We can use Theorem 3 to find the stabilization bounds on certain overlay networks. Consider, for instance, the Linear graph, which was discussed in Sect. 2. Prior work in linearization has attempted to achieve a polylogarithmic convergence time [6]. However, one can easily prove that the Linear graph satisfies subgraph monotonicity. Therefore, Linear's convergence time is linear, and a polylogarithmic convergence time is impossible.

3 TCF for SKIP+ Graphs

To provide an example of how the Transitive Closure Framework can be used to create a specific topology, we consider the Skip+ graph [4]. As mentioned earlier, the Skip+ graph is a locally-checkable extension of the Skip graph. In the following section, we describe the Skip+ graph, and show how our TCF instantiates a self-stabilizing algorithm that creates a Skip+ graph within $O(\log n)$ of optimal time.

3.1 The DETECT Predicate and REPAIR Subroutine

For Skip+ graphs, both DETECT and REPAIR follow trivially from the definition of a Skip+ graph – that is, each node simply computes its range in the ideal Skip+ graph using its 2-neighborhood, and either creates links within this range (with the REPAIR subroutine) or compares this range with the current 2-neighborhood (with the DETECT predicate).

3.2 Analysis of the Transitive Closure Framework

To evaluate the performance of the TCF with regards to Skip+ graphs, we provide the detector diameter $D(n)$. The proof verifying this result will appear in a future full version of the paper.

Lemma 1. *The detector diameter for the family of* Skip+ *graphs is* $D(n) = L + 1$*, where* $L = |\boldsymbol{rs}|$*.*

The Transitive Closure Framework allows us to provide the following theorem, which is easily derived using Theorem 1.

Theorem 4. *The Transitive Closure Framework produces a self-stabilizing overlay network algorithm for* Skip+ *graphs that converges in* $L + \log(n)$ *rounds (where* L *is the length of the random sequence).*

First note that the Transitive Closure Framework can produce a legal Skip+ graph in $O(\log n)$ rounds (when $L \in O(\log n)$). This is faster than the original self-stabilizing Skip+ graph algorithm, which converged in $O(\log^2 n)$ rounds. Obviously, the Transitive Closure Framework manages to lower run-time complexity by trading space – specifically, it causes a $\Theta(n)$ increase in node degree. However, this is equivalent to the worst-case performance of some nodes when using the self-stabilizing Skip+ graph construction in [4].

We can also use our above results to state that in fact our construction is within $O(\log n)$ of optimal (when $L \in O(\log n)$, the TCF algorithm is optimal).

Corollary 1. *By Theorems 2 and 4, the Transitive Closure Framework for* Skip+ *graphs runs in* $O(\log n)$ *time, which is optimal.*

4 The Local Repair Framework

The Transitive Closure Framework acts like a sledgehammer, initiating a full network rebuild even if the configuration is only minorly perturbed. However, there may be faulty configurations that are repairable efficiently, requiring action by only a small number of nodes (compared to n with TCF), and requiring only a limited amount of space (compared to $\Omega(n)$ for TCF).

Our local repair procedure uses a two-part approach: locally identifying those configurations that are locally repairable, and then executing the actions to repair these configurations. For a given λ, we identify two components for our framework: a program $Repair_\lambda^{ON}$ and a set of (potentially) locally-repairable network configurations $Repairable_\lambda^{ON}$. $Repairable_\lambda$ is defined in terms of a local predicate, so that locally repairable configurations may be detected as such. When ON is understood from the context, we drop ON and simply write $Repair_\lambda$ and $Repairable_\lambda$.

Definition 3. *Let* $CanRepair_u$ *be a predicate evaluated locally at each node* u*. Let* $Repairable_\lambda = \{G_\lambda : G_\lambda$ *is weakly connected and* $\forall v \in V : CanRepair_v\}$*.*

Definition 4. *Let* $Repair_\lambda$ *be a program executed by all nodes in* V*, and let* $Repair_\lambda^k[S]$ *represent the network resulting from executing* $Repair_\lambda$ *program* k *times, starting from configuration* S*.* $Repair_\lambda$ *satisfies the following properties:*

- *Convergence :* $\forall S \in Repairable_\lambda : Repair_\lambda^k[S] \notin Repairable_\lambda$*, for some finite* k

– *Connectivity : If* $S \in Repairable_\lambda$ *is weakly connected, then* $Repair_\lambda[S]$ *is also weakly connected*

Notice that $Repair_\lambda$ is guaranteed to converge to a configuration *not* in $Repairable_\lambda$. This configuration may be correct or faulty. The maximum number of rounds required for $Repair_\lambda$ to transform any n node network configuration in $G \in Repairable_\lambda$ to $ON(\lambda)$ is called the *repair time* $RT(n)$ of $Repair_\lambda$, while the maximum number of rounds required to transform any n node network configuration G to a faulty configuration $G' \notin Repairable_\lambda$ is called the *suppresion time* $ST(n)$ of $Repair_\lambda$, as it "suppresses" execution of the Transitive Closure process.

Notice that when $Repairable_\lambda = \emptyset$ ($CanRepair_u = false$), our LRF reduces to the simple TCF, and $RT(n)$ and $ST(n)$ are 0 (as no local repairs are recognized). Similarly, when $Repairable_\lambda = \mathcal{G}_\lambda$ ($CanRepair_u = true$), all network configurations are stabilized using $Repair_\lambda$ (this is the approach taken for prior self-stabilizing overlay networks). In our work, we focus on cases where $Repairable_\lambda$ is non-empty and does not include all configurations.

4.1 The Local Repair Program

The Local Repair Framework (LRF) is shown in Program 2. Each node u evaluates $CanRepair_u$ to see if its state is in $Repairable_\lambda$, and if so, initiates the local repair program $Repair_\lambda$. If the set is faulty and *not* in the $Repairable_\lambda$ set, then the Transitive Closure process is initiated as before.

Program 2. Local Repair Framework for Process u

Variables: neighborhood $N(u)$, predicate $CanRepair_u$
in each round do
1. **if** $CanRepair_u$ **then**
2. $\quad Repair_\lambda^{ON}$;
3. **else if** $N^2(u) \neq ON(N^2(u))$ **then**
4. $\quad$ Begin executing the TCF;
5. **fi**

od

Proving the correctness of LRF follows easily from the definitions and program given above. Due to space limitations, proofs of the following lemmas and theorem will appear in a future full version.

Theorem 5. *Program 2 is a self-stabilizing overlay network construction algorithm that can recover from any configuration in at most* $ST(n)+D(n)+\log(n)+1$ *rounds. Furthermore, a subset of configurations in* $Repairable_\lambda$ *will reach a correct network in* $RT(n)$ *rounds.*

5 Example: JOIN in SKIP+ Graphs

As overlay network membership is expected to be dynamic, accommodating nodes being added to the system is a commonly-addressed concern in overlay network algorithms. As with prior research, we assume that a node begins the JOIN process by connecting to a single node that is already a member of the correct network. The goal of JOIN algorithms are to integrate the node into the correct network within some efficient amount of time. In this section, we instantiate our LRF by adding node JOINs to SKIP+ graphs.

We begin by defining the following predicates that are evaluated locally on each node u, which we shall use to define $Repairable_\lambda$, $CanRepair_u$, and $Repair_\lambda^{\text{SKIP+}}$ for SKIP+ graphs. For ease of notation, let $pre(s1, s2)$ return the prefix match between strings $s1$ and $s2$, and let $|pre(s1, s2)|$ be the length of the matching prefix.

1. $AllConnected_u := |N(u)| > 1 \implies \forall v \in N(u) : \exists w \neq v \in N(u) s.t. v \in N(w) \wedge w \in N(v)$
2. $LongerMatchExists_u := \exists w \in (N^2(u) \setminus N(u)) s.t. \forall v \in N(u) : |pre(u.rs, w.rs)| > |pre(u.rs, v.rs)|$
3. $InitiatingJoin_u := LongerMatchExists_u \wedge AllConnected_u \wedge [\forall v \in N(u) : u \notin N(v) \wedge ON(N(v)) = N(v) \wedge (\nexists w \in N(u) s.t. |pre(u.rs, v.rs)| = |pre(u.rs, w.rs)|)]$
4. $CreatingJoinLinks_u := AllConnected_u \wedge \neg LongerMatchExists_u \wedge [\forall v \in N(u) : u \notin N(v)]$
5. $JoinCompleted_u := [N(u) = ON(N^2(u))] \wedge [\forall v \in N(u) : u \in N(v) \wedge ON(N^2(u)) \in N(v)]$
6. $JoinDetected_u := [\exists v \in N(u) : (N^2(u) \setminus \{v\}) = ON(N^2(u) \setminus v) \wedge (v \in ON(N(u))) \wedge (\forall x \in \{N(u) \setminus ON(N^2(u))\} : x \in N(v) \wedge v \in N(x))]$

The next step in our framework is to define the $CanRepair_u$, which will define the set $Repairable_\lambda$.

Definition 5. *For* SKIP+ *graphs, let* $CanRepair_u := [(N^2(x) = ON(N^2(x))) \vee InitiatingJoin_x \vee CreatingJoinLinks_x \vee JoinCompleted_x \vee JoinDetected_x]$.

We define the $Repair_\lambda$ program for JOINs in the SKIP+ graph below. The high-level idea of our joining algorithm is simple. A joining node u traverses the network searching for the node with the longest prefix match with u's random sequence ($InitiatingJoin_u$). Once this node is found, node u adds neighbors for each level ($CreatingJoinLinks_u$).Once all neighbors have been added, node u deletes the edges that it used to find the longest prefix match, and makes the remaining correct edges correct. Nodes in the network will then detect the presence of a joined node, and make the appropriate changes ($JoinDetected_u$).

Theorem 6. *The Local Repair Framework given in Program 2, instantiated with the* $Repair_\lambda^{\text{SKIP+}}$ *program from Program 3 and predicate* $CanRepair_u$ *from Definition 5, is a self-stabilizing algorithm for* SKIP+ *graphs with convergence time for* any *configuration at most* $3 \cdot L + \log(n) + 3$ *rounds.* JOINs *occur in at most* $2 \cdot L + 2$ *rounds, and require only* L *extra neighbors for the joining node.*

Program 3. $Repair_{\lambda}^{\text{SKIP+}}$ program for node u

Variables: neighborhood $N(u)$
in each round do
1. **if** $InitiatingJoin_u$ **then**
   `// search for best random sequence match`
2. $N(u) := N(u) + (x : x \in \{N^2(u) \setminus N(u)\} \wedge$
   $(\forall y \neq u \in N(u) : |pre(u.rs, x.rs)| > |pre(u.rs, y.rs)|))$;
3. **else if** $CreatingJoinLinks_u$ **then**
4. **if** $ON(N^2(u)) \in N(u)$ **then**
   `// delete edges used to find longest matching random sequence`
5. $N(u) := N(u) \setminus \{x \in N(u) : x \notin ON(N^2(u))\}$;
6. Make all remaining edges in $N(u)$ bidirectional;
7. **else**
   `// add links level-by-level`
8. $N(u) := N(u) \cup \{x : x \in ON(N^2(u)) \wedge$
   $|pre(u.rs, x.rs)| = max(|pre(u.rs, t.rs|), \forall t \in N^2(u) \setminus N(u)\}$;
9. **fi**
10. **else if** $JoinDetected_u$ **then**
11. $N(u) := ON(N^2(u))$; `// incorporate the joined node`
12. **fi**

od

References

1. Aspnes, J., Shah, G.: Skip graphs. In: SODA 2003: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 384–393. Society for Industrial and Applied Mathematics, Philadelphia (2003)
2. Aspnes, J., Wu, Y.: O($\log n$)-time overlay network construction from graphs with out-degree 1. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 286–300. Springer, Heidelberg (2007)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
4. Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: PODC 2009: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 131–140. ACM, New York (2009)
5. Kniesburges, S., Scheideler, C., Koutsopoulos, A.: Re-chord: A self-stabilizing chord overlay network. In: SPAA 2011: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures. ACM, New York (2011)
6. Onus, M., Richa, A.W., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: ALENEX. SIAM, Philadelphia (2007)
7. Peleg, D.: Distributed computing: a locality-sensitive approach. Society for Industrial and Applied Mathematics, Philadelphia (2000)

Active Stabilization*

Borzoo Bonakdarpour[1] and Sandeep S. Kulkarni[2]

[1] School of Computer Science
University of Waterloo
Waterloo N2L 3G1, Canada
borzoo@cs.uwaterloo.ca
[2] Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
sandeep@cse.msu.edu

Abstract. We propose the notion of *active stabilization* for computing systems. Unlike typical stabilizing programs (called passive stabilizing in this paper) that require that the faults are absent for a long enough time for the system to recover to legitimate states, active stabilizing programs ensure recovery in spite of constant perturbation during the recovery process by an adversary. We identify the relation between active and passive stabilization in terms of their behavior and by comparing their cost of verification. We propose a method for designing active stabilizing programs by a collection of passive stabilizing programs. Finally, we compare active stabilization with fault-contained stabilization and stabilization in the presence of Byzantine faults.

Keywords: Self-stabilization, reactive systems, adversary, formal methods.

1 Introduction

A *self-stabilizing* system [6] ensures that it will recover to a legitimate state even if it starts executing from an arbitrary state. For this reason, self-stabilization is often utilized to provide recovery from unexpected transient errors. A typical self-stabilizing protocol in the literature considers the case where *faults* perturb the system to an arbitrary state. Subsequently, the goal of the protocol is to ensure that the system will recover to a legitimate state with the assumption that no additional faults will occur. Moreover, by the nature of self-stabilization, even if faults occur during recovery, the system will still recover to a legitimate state as long as faults do not occur forever (or they stop for a long enough time). We call such self-stabilization as *passive* stabilization since faults play a dormant role during the recovery process.

* This work is partially sponsored by Canada NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, and ISOP IS09-06-037 grants, and, by USA AFOSR FA9550-10-1-0178 and NSF CNS 0914913 grants.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 77–91, 2011.

In this paper, we introduce the concept of *active stabilization*. To illustrate the motivation for active stabilization, we begin with the problem of pursuer-evader games [4]. The intuitive description of one instance of this problem in the context of sensor networks is as follows: The system consists of a set of computing nodes with sensors (called just sensors from in the subsequent discussion). Additionally, the system contains one (or more) pursuers and one (or more) evaders. The sensors' task is to organize themselves in a structure that will facilitate the capture of the evader. For example, one approach to achieve this is to have the sensors form a tree among themselves that is rooted at the location of the evader. The goal of the pursuer is to utilize this structure to capture the evader.

In such a system, there can be several faults. For example, the state of sensors could be corrupted due to false positive and/or false negative readings. Moreover, communication errors, errors in initialization etc. may perturb the sensor network to an arbitrary state. It is anticipated that such faults are rare and, hence, we can utilize passive self-stabilization for dealing with such faults; i.e., we can assume that faults stop for a long enough time for the sensor network to stabilize to a legitimate state. However, the network could also be perturbed by the evader itself. In particular, if the goal of the sensor network is to have a tree rooted at the evader, then the evader movement is tantamount to perturbation of the network. Moreover, it may be unreasonable to assume that these faults eventually stop or that they stop for a long enough time since the evader is actively trying to perturb the system so that it does not stabilize.

We can view three different contributing factors in such a scenario: (1) the system, (2) the faults, and (3) the *adversary*. In particular, the system actions are responsible for ensuring recovery to legitimate states. The faults are events that perturb the system randomly and rarely. It is anticipated that the faults could perturb the system to an arbitrary state, thereby requiring self-stabilization. However, because these events are rare, one can assume that they stop for a sufficiently long enough time to allow the system to recover. The adversary is *actively* attempting to prevent self-stabilization. However, unlike faults, the adversary may not be able to perturb the system to an arbitrary state. For example, in the above scenario, the evader would be able to move within the vicinity of its original location; i.e., it would not be able to move to a random location from its initial location. Also, unlike faults, the adversary actions may never stop. In particular, it would be unreasonable to assume that the adversary actions stop for a long enough time for the system to stabilize.

Although it is unreasonable to assume that adversary actions would stop for a long enough time, it is necessary to assume some fairness for the system actions. In particular, in the pursuer-evader example, it is anticipated that the evader movement is limited by laws of physics and, hence, the system can take a certain number (≥ 1) of steps between two steps of the evader.

The main contributions of the paper are as follows:

- We formally define different variations of active stabilization and relate it to passive stabilization. In particular, we consider the cases where the adversary (1) cannot lead the system to illegitimate states (called active stabilization),

(2) can perturb the program outside the legitimate states (called *fragile* active stabilization), and (3) can perturb the program outside the legitimate states from where the program can recover before the adversary takes another step (called *contained* active stabilization)
- We study the relation between different types of active stabilization.
- We compare the cost of automated verification of active and passive stabilization.
- The problem in designing an active stabilizing program lies in the fact that an adversary can disrupt the progress made by the program towards recovering to the invariant. Thus, we propose an approach for designing active stabilizing programs based on the convergence stair [10].
- Finally, we argue that active stabilization is a powerful and expressive concept by presenting comparison to fault-contained stabilization [8] and Byzantine self-stabilization [14]. In particular, we show that if a program is contained active stabilizing, then it is fault-contained stabilizing. We also show that a special type of active stabilization in the presence of Byzantine processes is Byzantine self-stabilization.

Organization of the paper. In Section 2, we introduce the notion of active stabilization. We compare different types of passive and active stabilization in Section 3. The complexity of automated formal methods for active stabilization is analyzed in Section 4. Section 5 discusses design methodology for active stabilizing programs. We compare active stabilization with fault-contained stabilization and Byzantine self-stabilization in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 The Concept of Active Stabilization

A traditional modeling of programs in the literature on self-stabilization includes a finite set of variables with (finite or infinite) domain. Additionally, it includes guarded commands [7] that update those program variables. Since these internals of the program are not relevant in our definition of active stabilization, in our work, we define program p in terms of its state space S_p and its transitions $\delta_p \subseteq S_p \times S_p$. Intuitively, the state space can be obtained by assigning each variable in p a value from its domain.

Definition 1 (Program). *A* program p *is of the form* $\langle S_p, \delta_p \rangle$ *where* S_p *is the state space of program* p *and* $\delta_p \subseteq S_p \times S_p$.

Assumption 1 *For simplicity of definitions, we assume that program* p *has at least one outgoing transition from every state in* S_p*. If such a transition does not exist for some state, say* s*, then we consider the program where transition* (s, s) *is added. While this assumption simplifies subsequent definitions since we do not need to consider* terminating *behavior of a program explicitly, it is not restrictive in any way.*

Definition 2 (State Predicate). *A* state predicate *of* p *is any subset of* S_p.

Definition 3 (Closure). *A state predicate* S *of* $p = \langle S_p, \delta_p \rangle$ *is* closed *in* p *iff* $\forall s_0, s_1 \in S_p :: (s_0 \in S \wedge (s_0, s_1) \in \delta_p) \Rightarrow (s_1 \in S)$.

Definition 4 (Faults). *We define* faults *for program* $p = \langle S_p, \delta_p \rangle$ *to be* $S_p \times S_p$*; i.e., the faults can perturb the program to any arbitrary state.*

The *adversary* for program, say *adv*, is defined in terms of its transitions, say $a_p \subseteq S_p \times S_p$. Note that, based on the discussion in the introduction, this allows us to model the limited set of actions the adversary may be allowed to execute.

Definition 5 (Adversary). *We define an* adversary *for program* $p = \langle S_p, \delta_p \rangle$ *to be a subset of* $S_p \times S_p$.

Next, we define a computation of the program, say p, in the presence of adversary, say *adv*.

Definition 6 ($\langle p, adv, k \rangle$-computation). *Let* p *be a program with state space* S_p *and transitions* δ_p*. Let adv be an adversary for program* p*. And, let* k *be an integer greater than 1. We say that a sequence* $\langle s_0, s_1, s_2, ... \rangle$ *is a* $\langle p, adv, k \rangle$-computation *iff*

- $\forall j \geq 0 :: s_j \in S_p$*, and*
- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p \cup adv$*, and*
- $\forall j \geq 0 :: ((s_j, s_{j+1}) \notin \delta_p) \quad \Rightarrow \quad (\forall l \mid j < l < j + k :: (s_l, s_{l+1}) \in \delta_p)$

Observe that a $\langle p, adv, k \rangle$-computation involves only the transitions of program p or its adversary *adv*. Moreover, the adversary is required to execute with fairness to the program; i.e., the program can take at least $k - 1$ steps between two adversary steps. This ensures that the adversary cannot simply block the program from executing, thereby make it impossible to provide recovery. However, the adversary is not required to execute and the program can execute forever.

Remark 1 (Fairness among program transitions 1). Since the focus of this paper is on the defining active stabilization based on the interaction between the program and the adversary, we omit the issue of fairness among program transitions themselves. Specifically, in some instances, we can consider the program to consist of multiple processes and require that each process executes with some fairness. In this instance, the above definition can be modified to add an additional constraint that identifies fairness conditions. For reasons of space, this issue is outside the scope of this paper.

Remark 2 (Round-based computations). The definition of $\langle p, adv, k \rangle$-computation is based on the number of *steps* that a program takes between two adversary steps. In scenarios where a program consists of multiple processes, a *round* [15] based notion is sometimes used. Intuitively, in one round, every process is given at least one chance to execute. (However, the process may not actually be able to execute a transition if it was given that chance when none of its transitions could be executed.) The definition of active stabilization can also be extended to handle such a case by using *rounds* instead of *steps*. This issue is also outside the scope of this paper.

Definition 7 (Active stabilization). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that program p is* strong k-active stabilizing *with adversary adv for* invariant S *iff*

- S *is closed in* p
- S *is closed in adv*
- *For any sequence* σ $(=\langle s_0, s_1, s_2, ...\rangle$ *) if* σ *is a* $\langle p, adv, k\rangle$*-computation then there exists* l *such that* $s_l \in S$.

The definition of active stabilization requires that the invariant S be closed in the execution by the adversary; i.e., when the invariant S is reached, the adversary does not perturb the program outside S. In other words, only a fault can do so. This requirement can be difficult to satisfy in many programs. For such programs, we introduce the notion of *fragile active stabilization* where the adversary can perturb the program outside the invariant.

Definition 8 (Fragile Active Stabilization). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that program p is* fragile k-active stabilizing *with adversary adv for invariant S iff*

- S *is closed in* p
- *For any sequence* σ $(=\langle s_0, s_1, s_2, ...\rangle$ *) if* σ *is a* $\langle p, adv, k\rangle$*-computation then there exists* l *such that* $s_l \in S$.

Observe that if p is fragile k-active stabilizing with adversary *adv* for invariant S, then starting from an arbitrary state, p will reach a state in S even if adversary *adv* tries to disrupt it. Moreover, after the program reaches S, the adversary can still perturb it outside S. However, in subsequent computation, p is still guaranteed to reach a state in S again. Thus, a fragile active stabilizing program will reach the invariant infinitely often.

One issue with fragile active stabilization is that after the adversary perturbs the program from a state in S to a state outside S, there is no bound on how long it will take to return to S. Our notion of *contained active stabilizing programs* addresses this issue by requiring the program to recover to S quickly; i.e., before the adversary can perturb it again.

Definition 9 (Contained Active Stabilization). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that program p is* contained k-active stabilizing *with adversary adv for invariant S iff*

- S *is closed in* p
- *For any sequence* σ $(=\langle s_0, s_1, s_2, ...\rangle$ *) if* σ *is a* $\langle p, adv, k\rangle$*-computation then there exists* l *such that* $s_l \in S$.
- *For any finite sequence* α $(=\langle s_0, s_1, s_2, ...s_k\rangle$ *) if* $s_0 \in S$, $(s_0, s_1) \in adv$ *and* $(\forall j : 0 < j < k : (s_j, s_{j+1}) \in \delta_p$ *then* $s_k \in S$.

Finally, we also define the traditional notion of stabilization. Towards this end, we define pure-computations of p; i.e., computations where only p is allowed to execute. Then, we define the standard definition of stabilization, which in this paper we will call as *passive* stabilization.

Definition 10 (pure-computation). *Let p be a program with state space S_p and transitions δ_p. We say that a sequence $\langle s_0, s_1, s_2, ...\rangle$ is a* pure-computation *iff*

- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p$

Remark 3 (Fairness among program transitions 2). Similar to Remark 1, the above definition can include fairness requirements.

Definition 11 (Passive stabilization). *Let p be a program with state space S_p and transitions δ_p. We say that program p is* (passive) stabilizing *for invariant S iff*

- *S is closed in p*
- *For any sequence σ $(=\langle s_0, s_1, s_2, ...\rangle$ $)$ if σ is a pure-computation then there exists l such that $s_l \in S$.*

3 Relation between Different Types of Active and Passive Stabilization

In this section, we study the conceptual relation between different types of stabilization presented in Section 2. In Subsection 3.1, we compare active stabilization with passive stabilization. Then, in Subsection 3.2, we compare different types of active stabilization.

3.1 Relation between Active and Passive Stabilization

In this section, we evaluate the relation between active stabilization and passive stabilization. In particular, we show that any active stabilizing program is also passive stabilizing. Moreover, we present necessary conditions under which passive stabilizing programs can be transformed into fragile and contained active stabilizing programs.

Theorem 1. *If there exists k and adv such that program p is k-active stabilizing with adversary adv for invariant S, then p is passive stabilizing for invariant S.*

Proof. Since p is k-active stabilizing with adversary adv for invariant S, every $\langle p, adv, k\rangle$-computation reaches a state in S. And, by definition, a pure-computation of p is also a $\langle p, adv, k\rangle$-computation. Thus, every pure-computation of p reaches a state in S (recall that a program can execute adversary transitions as well). Furthermore, by definition of active stabilization, S is closed in p. It follows that p is passive stabilizing for invariant S. □

Theorem 2. *If program $p = \langle S_p, \delta_p \rangle$ is passive stabilizing for invariant S and S_p is finite, then there exists k such that for any adversary adv, p is fragile k-active stabilizing with adversary adv for invariant S.*

Proof. First, by definition of passive stabilization, S is closed in p. Hence, to prove this theorem, we only need to prove the second constraint in the definition of fragile active stabilization. To this end, we let $k = |S_p|$. Consider any $\langle p, adv, |S_p| \rangle$-computation. This computation includes a subsequence of size $|S_p| - 1$, say α that only includes transitions of p. If α includes any state, say s such that $s \notin S$ and s occurs twice in α then there is a pure-computation of p that starts from s and never reaches S. Thus, α does not include any state outside S more than once. Hence, by the pigeon hole principle, α contains at least one state in S. □

Note that the above theorem does not hold if the state space of p is infinite. We can illustrate this using a the following simple example. Let the state space of p be the set $S_p = \mathbb{Z}_{\geq 0}$ of non-negative integers and the transitions of p be $\delta_p = \{(x+1, x) \mid x \in \mathbb{Z}_{\geq 0} \cup \{(0,0)\}$. Also, let the invariant of p be $S = \{0\}$. Clearly, p is passive stabilizing for invariant S. However, for adversary $N \times N$, the above theorem is not valid.

Also, note that the above theorem will be incorrect if we remove 'fragile' from the statement of the theorem. This is due to the fact that S may not be closed in the transitions of an arbitrary adversary.

Corollary 1. *If program $p = \langle S_p, \delta_p \rangle$ is passive stabilizing for invariant S and S_p is finite, then there exists k such that for any adversary adv, p is contained k-active stabilizing with adversary adv for invariant S.*

Finally, if a program is active stabilizing with some adversary, then the program is also active stabilizing with a *slower* adversary. Thus, we have the following theorem.

Theorem 3. *If program p is k-active stabilizing with adversary adv for invariant S and $l \geq k$, then p is l-active stabilizing with adversary adv for invariant S.*

3.2 Relation between Active, Fragile Active, and Contained Active Stabilization

Since ensuring the closure of invariant in the transitions of an adversary can be unrealistic, we introduced the notion of fragile and contained active stabilization. In this section, we show that the definition of contained active stabilization and active stabilization are exchangeable; i.e., given a program p that is contained k-active stabilizing, we can find a corresponding program p' that is k-active stabilizing and vice versa. Thus, these results show that instead of showing a program to be active stabilizing, one can show that the program is contained active stabilizing.

Definition 12 ($R^j_{(p,adv)}(S)$)**.** *Let S be a state predicate of program p and adv be an adversary for program p. We define $R^j_{(p,adv)}(S)$, where $j \geq 0$, as follows:*

- *if $j = 0$, then*
  $R^j_{(p,adv)}(S) = S \cup \{s_1 \mid \exists s_0 \in S : (s_0, s_1) \in adv\}$.
- *if $j > 0$, then*
  $R^j_{(p,adv)}(S) = \{s_1 \mid \exists s_0 \in R^{j-1}_{(p,adv)}(S) : (s_0, s_1) \in \delta_p\}$.

Definition 13. *Let p be a program with state space S_p and transitions δ_p. We define $count_p$ to be the program where:*

- *State space $S_{count_p} = S_p \times \mathbb{Z}_{\geq 0}$, and*
- *Transitions $\delta_{count_p} = \{(\langle s_0, j\rangle, \langle s_1, j+1\rangle) \mid (s_0, s_1) \in \delta_p\}$.*

Definition 14. *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. We define $count_{adv_k}$ to be the adversary for $count_p$ where:*

- *Transitions $= \{(\langle s_0, j\rangle, \langle s_1, 0\rangle) \mid ((s_0, s_1) \in adv) \ \wedge \ (j \geq k)\}$.*

Theorem 4. *If program p is contained k-active stabilizing with adversary adv for invariant S, then $count_p$ is k-active stabilizing with adversary $count_{adv_k}$ for invariant S' where:*

$$S' = \bigcup_{l=0}^{\infty} \{\langle s, l\rangle \mid s \in R^l_{(p,adv)}(S)\}$$

Proof. To prove the above theorem, we need to prove the three conditions in the definition of active stabilization:

1. S' is closed in $count_p$.
   Towards this end, we need to show that if $count_p$ executes in any state in S', then the resulting state would also be in S'. Let $\langle s_0, l\rangle$ be a state in S'. Hence, s_0 is included in $R^l_{(p,adv)}(S)$. By Definition 13, transition of $count_p$ is of the form $(\langle s_0, l\rangle, \langle s_1, l+1\rangle)$. Furthermore, if $(\langle s_0, l\rangle, \langle s_1, l+1\rangle)$ is a transition of $count_p$, then (s_0, s_1) is a transition of p as well. Hence, by the Definition 12, s_1 is in the set $R^{l+1}_{(p,adv)}(S)$. Finally, from Definition of S', $\langle s_1, l+1\rangle$ is in the set S'. Thus, S' is closed in $count_p$.
2. S' is closed in $count_{adv_k}$.
   Let $\langle s, l\rangle$ be a state in S'. If $count_{adv_k}$ can execute in state $\langle s, l\rangle$, then $l \geq k$. Hence, by Definition 12, there exists a sequence, $\langle s_0, s_1, ..., s_l\rangle$ such that (1) the first state in the sequence is in S (i.e., $s_0 \in S$), (2) the first transition in the sequence is executed by the adversary (i.e., $(s_0, s_1) \in adv$), and (3) subsequent transitions are executed by program (i.e., $\forall x | 0 < x < l :: (s_x, s_{x+1}) \in \delta_p$). Furthermore, since $l \geq k$ and the fact that p is contained

k-active stabilizing with adv for S, if $count_{adv_k}$ can execute in $\langle s, l\rangle$, then $s \in S$. Moreover, by Definitions 12 and 14, the resulting state is also in S'.

3. For any sequence σ $(=\langle s_0, s_1, s_2, ...\rangle)$, if σ is a $\langle p, adv, k\rangle$-computation, then there exists l such that $s_l \in S$. This follows trivially from the definition of contained k-active stabilization. □

Finally, the following theorem trivially holds from the definition of active and contained active stabilization.

Theorem 5. *If p is k-active stabilizing with adversary adv for invariant S Then p is contained k-active stabilizing with adversary adv for invariant S.*

4 Comparing the Cost of Automated Verification for Active and Passive Stabilization

The problem of verifying stabilizing programs involves two parts: The first part relates to proving that in legitimate states (i.e., invariant), the program satisfies the specification at hand. And, the other part relates to proving that starting from an arbitrary state, the program recovers to legitimate states. Since the goal of this section is to compare the cost of verification for passive stabilization and active stabilization, we only focus on the second part. In other words, we compare the complexity of verification of *convergence* for passive and for active stabilization. We introduce instance of verification for passive and active stabilization. Then, we present the corresponding complexity results.

Instance. A program $p = \langle S_p, \delta_p\rangle$ and a state predicate S of p.
Verifying passive stabilization decision problem (VPS). Is p passive stabilizing for invariant S?

Theorem 6. *VPS can be solved in polynomial-time in $|S_p|$.*

Proof. This is a well-known result that can be proved with the following simple algorithm:

1. Closure property can be trivially verified by considering each transition in δ_p.
2. For convergence, if δ_p included any states outside S where there are no outgoing transitions, the answer to the decision problem is false. Assuming that this is not the case, we begin with program q which has the state space $S_q = S_p - S$ and transitions $\delta_q = \delta_p - \{(s_0, s_1) \mid s_0 \in S \vee s_1 \in S\}$. Since we have removed some transitions, q may contain a deadlock state. If so, we remove that state from S_q and the corresponding transitions that enter and exit that state. Upon termination, if S_q is empty, p is passive stabilizing for invariant S. If not, there is a cyclic-computation that does not include any state in S. In other words, p is not passive stabilizing for invariant S. □

Next, we present the results for verification of active stabilization.

Instance. A program $p = \langle S_p, \delta_p \rangle$, an adversary adv for p, a state predicate S of p, and an integer $k \geq 2$.
Verifying k-Active Stabilization Decision Problem (VkAS). Is p k-active stabilizing with adversary adv for invariant S?

Theorem 7. *VkAS can be solved in polynomial-time in $|S_p|$.*

Proof. First, as stated earlier, closure proofs can be performed in polynomial time in $|S_p|$. The remaining proof is predicated under the assumption that the closure properties are satisfied. In particular, to prove convergence, we map the problem of verifying active stabilization to the problem of verifying passive stabilization. Specifically, we construct program p_1 as follows.

$$p_1 = \{(s_0, s_1) |\ \exists l : l \geq k-1 : reach(s_0, s_1, l) \vee (\exists s_2 :: reach(s_0, s_2, l) \wedge (s_2, s_1) \in adv)\}, \text{ where}$$

$reach(s_0, s_1, l)$ denotes that s_1 can be reached from s_0 by execution of exactly l transitions of p.

Intuitively, program p_1 executes $k-1$ or more transitions of program p and then (optionally) one transition of the adversary. Next, we show that p is k-active stabilizing with adversary adv for invariant S iff p_1 is passive stabilizing for invariant S

1. $\Rightarrow$ Let σ be a pure-computation of p_1. We construct a corresponding $\langle p, adv, k \rangle$-computation as follows: For each transition (s_0, s_1) in σ, we replace it by a sequence (that begins in s_0 and ends in s_1) of $k-1$ or more transitions of p followed by an optional transition of adv. By construction of p_1, this is always feasible. Let the resulting sequence be σ_1. Since σ_1 is a $\langle p, adv, k \rangle$-computation, it contains a suffix where all states are in S. Hence, σ also contains a state in S.
2. $\Leftarrow$ Let σ be a $\langle p, adv, k \rangle$-computation of p. We construct a corresponding pure-computation of p as follows: If σ contains a transition by the adversary in the first k transitions, we consider the suffix that begins in the state after the transition of the adversary. Hence, without loss of generality we can assume that the first $k-1$ transitions in σ are transitions of p. Now, we obtain a pure-computation of p_1 as follows: The initial state in σ_1 is the same as that in σ. Let this state be s_0. Now, to obtain the next state in σ_1, we identify the first occurrence of the transition of the adversary in σ. If such a transition, say (s_a, s_b), exists then the successor state is s_b. If such a transition does not exist then the successor state is the one obtained by executing $k-1$ transitions of p. It follows that σ_1 is a pure-computation of p and, hence, includes a suffix that is entirely within S. Hence, σ also contains a state in S. □

5 Methodology for Designing Active Stabilizing Programs

In this section, we identify an approach for designing a program to be self-stabilizing. This approach is based on the convergence stair [10] approach for designing self-stabilizing programs. In particular, the problem in designing an active stabilizing program lies in the fact that an adversary can disrupt the progress made by the program towards recovering to the invariant. However, if we can prove that the program manages these disruptions in a suitable fashion, it can be proved that the program is active stabilizing to the adversary. Specifically, we prove the following theorem.

Theorem 8. *Let $p = \langle S_p, \delta_p \rangle$ be a program, adv be an adversary for p, and $S_0, S_1, \ldots, S_n$ be a sequence of state predicates of p. If*

- $S_0 = S_P$
- $\forall j : 0 \leq j < n : (S_{j+1} \Rightarrow S_j)$,
- $\forall j : 0 \leq j \leq n : S_j$ *is closed in p,*
- $\forall j : 0 \leq j \leq n : S_j$ *is closed in adv,*
- *For any finite sequence $\alpha = \langle s_1, s_2, ..., s_k \rangle$, if $s_1 \in S_j$ and $\forall l : 0 < l < k : (s_l, s_{l+1}) \in \delta_p$ then $s_k \in S_{j+1}$.*

Then

- *p is k-active stabilizing with adversary adv for S_n.*

Proof. The closure requirements for active stabilization are trivially satisfied. Regarding the last requirement in Definition 7, consider any $\langle p, adv, k \rangle$-computation, say σ. Based on the last constraint in this theorem and definition of $\langle p, adv, k \rangle$-computation, there exists a state, say s_a in σ such that $s_a \in S_1$. Since S_1 is closed in p and adv, all states in the suffix of σ that starts from s_a are in S_1. Again, by the same argument, σ contains a state, say s_b, in S_2, and so on. Thus, σ contains a state in S_n. □

The above theorem suggests the following approach to design active stabilizing programs. First, we identify a sequence of stair predicates $S_1, S_2, \ldots, S_{n-1}$ such that each of these predicates is closed in *adv*. Then, we ensure that the recovery from any of these predicates to the next state predicate is achieved before the adversary can perturb the program. Observe that in this fashion, the adversary can in fact execute several times before the program reaches the invariant. However, intuitively, the disruption by the adversary is less than the progress made by the program.

Moreover, if we focus on last condition in Theorem 8, it only focuses on pure-computations where the adversary is not allowed to execute. Thus, design of each stair is equivalent to the design of passive stabilization where the *convergence steps* are bounded. Thus, each stair of the active stabilizing program can potentially be constructed out of a collection of passive stabilizing programs.

6 Relation between Active Stabilization and Other Stabilization Techniques

In this section, we compare active stabilization with other stabilization techniques, namely *fault-contained stabilization* and *Byzantine self-stabilization* in Subsections 6.1 and 6.2, respectively.

6.1 Fault-Contained Stabilization

The problem of fault-containment stabilization has been studied (e.g., [8, 9, 16]) in the literature to deal with two problems with stabilizing programs. The first problem is that stabilizing programs do not typically differentiate between an arbitrary global state and a state that is "almost legitimate" [8]. Hence, the goal of these algorithms is that if the state is "almost legitimate" then it reaches a legitimate state within a small number of steps. However, the recovery from an arbitrary state may take longer. The second problem is that after the program reaches a legitimate state, there is a high probability that transient faults will only perturb it to a state that is "almost legitimate" as opposed to an arbitrary state.

With this intuition, in [8, 9, 16], the authors introduce a notion of limited fault class, lf_p, for program p. lf_p is a subset of $S_p \times S_p$. Moreover, if the program is perturbed by lf_p in a legitimate state, a quick recovery is provided if no additional faults occur. Moreover, if multiple faults from lf_p occur or if faults outside lf_p occur then stabilization is still provided. Thus, fault-contained stabilization can be defined as follows:

Definition 15 (Fault-contained Stabilization). *Let $p = \langle S_p, \delta_p \rangle$, S be a state predicate of p, and lf_p be a subset of $S_p \times S_p$. And, let $w \geq 1$ be an integer. p is* fault-contained stabilizing *for lf_p with w steps for invariant S iff*

- *p is passive stabilizing for invariant S, and*
- *For every sequence $\sigma = \langle s_0, s_1, \ldots \rangle$, if $s_0 \in S$, $(s_0, s_1) \in lf_p$ and $\forall j > 1 :$ $(s_j, s_{j+1}) \in \delta_p$ then $s_w \in S$.*

Now, we can show that if p is contained k-active stabilizing then it provides fault-contained stabilization. Note that the converse of this theorem is not correct since a fault-contained stabilizing program may not recover to legitimate states if it is continuously perturbed by faults.

Theorem 9. *If p is contained k-active stabilizing with adversary adv for S, then p is fault-contained stabilizing for adv with k steps for S.*

Proof. Follows trivially from Definitions 9 and 15. □

Corollary 2. *If p is k-active stabilizing with adversary adv for S, then p is fault-contained stabilizing for adv with 1 steps for S.*

6.2 Byzantine Self-Stabilization

While all the formalization in Byzantine self-stabilization work cannot be presented here, we give a brief approach considered in these papers and its relation to active stabilization. In [14], the program is viewed in terms of a set of, say n, processes. Thus, the state of the program is of the form $\langle v_1, v_2, ..., v_n \rangle$, where v_j denotes the state of process j. (If channels are used corresponding entry is added for channel contents as well.) Thus, the state space of the program, (S_p in Definition 1) is obtained by considering all possible tuples of $\langle v_1, v_2, ..., v_n \rangle$, where the domain of v_j depends on the application at hand.

Moreover, some processes can be Byzantine. If process j is Byzantine, it can change the value of v_j arbitrarily. Thus, transitions of p (cf. Definition 1), δ_p is of the form: $\delta_p = \delta_{p_g} \cup \delta_{p_b}$, where δ_{p_b} denotes the transitions that correspond to the transitions executed by Byzantine process(es) and δ_{p_g} denotes the transitions executed by non-Byzantine processes. Furthermore, δ_{p_g} and δ_{p_b} are disjoint. It is also assumed that Byzantine processes do not prevent non-Byzantine processes from executing. However, Byzantine processes can disrupt the recovery process. Thus, the definition of stabilization in the presence of Byzantine faults is adapted as follows.

Definition 16. *A program $p = \langle S_p, \delta_p \rangle$, where $\delta_p = \delta_{p_g} \cup \delta_{p_b}$, is said to be stabilizing in the presence of Byzantine faults for invariant S iff*

- *S is closed in p, and*
- *for any state sequence, say σ, of the form $\langle s_0, s_1, s_2, ... \rangle$, if*
  - *$\forall j \geq 0 : (s_j, s_{j+1}) \in \delta_p$*
  - *Number of transitions of δ_{p_g} is σ is infinite.*
- *then*
  - *there exists l such that $s_l \in S$.*

Theorem 10. *If program $p = \langle S_p, \delta_p \rangle$, where $\delta_p = \delta_{p_g} \cup \delta_{p_b}$ is stabilizing in the presence of Byzantine faults for invariant S, then $\langle S_p, \delta_{p_g} \rangle$ is 2-active stabilizing with adversary δ_{p_b} for S.*

Proof. Closure proofs are trivially satisfied from Definition 16. In a $\langle \langle S_p, \delta_{p_g} \rangle, \delta_{p_b}, 2 \rangle$-computation, there is at least one transition of δ_{p_g} between any two transitions of δ_{p_b}. Hence, the number of occurrences of transitions in δ_{p_g} is infinite. Hence, in any $\langle \langle S_p, \delta_{p_g} \rangle, \delta_{p_b}, 2 \rangle$-computation, a state in S is reached. □

To prove the converse of the above theorem, we recall the standard definition of transitive closure.

Definition 17 (Transitive Closure). *A set of transitions δ_{p_b} is transitive closed iff* $\forall a, b, c :: ((a, b) \in \delta_{p_b} \wedge (b, c) \in \delta_{p_b}) \quad \Rightarrow \quad ((a, c) \in \delta_{p_b})$

Theorem 11. *Let p be a program whose transitions are partitioned in terms of δ_{p_g} and δ_{p_b}, where δ_{p_b} are the transitions executed by Byzantine processes. If $\langle S_p, \delta_{p_g} \rangle$ is 2-active stabilizing with adversary δ_{p_b} for S, and δ_{p_b} is transitive-closed, then p is stabilizing in the presence of Byzantine faults for S.*

Proof. The closure proof is trivially satisfied. Consider any computation, say σ, of p where transitions of δ_{p_g} execute infinitely often. Consider a compacted version of σ, say c_σ that is obtained as follows: If σ contains two consecutive transitions, say (s_j, s_{j+1}) and (s_{j+1}, s_{j+2}), in δ_{p_b} then we replace them by (s_j, s_{j+2}). And, repeat this process until there are no two successive transitions in δ_{p_b}. Since δ_{p_b} is transitive closed, c_σ is a $\langle\langle S_p, \delta_{p_g}\rangle, \delta_{p_b}, 2\rangle$-computation. Hence, it includes a state in S. Thus, σ includes a state in S. □

7 Related Work

There are several variations of stabilization (denoted by passive stabilization in this paper) that are considered in the literature. These include fault-containment stabilization, byzantine stabilization, FTSS, multitolerant stabilization, weak stabilization, probabilistic stabilization, and nonmasking fault-tolerance.

Fault-containment stabilization refers to stabilizing program that ensure that if only one (respectively, small number of) fault occurs then quick recovery is provided to the invariant. Examples of such programs include [8, 16]. Byzantine stabilization refers to stabilizing programs that tolerate the scenario where a subset of processes is Byzantine. Examples of such programs include [13, 14]. FTSS refers to stabilizing programs that tolerate permanent crash faults. Examples of such programs include [3]. Multitolerant stabilizing systems ensure that in addition to stabilization property, the program ensures that the safety property is never violated when only a limited class of faults occur. Examples of such systems include [12]. As discussed in the last two sections, fault-containment stabilization and Byzantine stabilization are closely related to Active stabilization.

Weak stabilization [5, 11], as the name suggests, is a weaker version of stabilization. In weak stabilizing programs, from every state, there is a path to reach a state in the invariant. However, the program may contain loops that are outside legitimate states. In [11], it is shown that under certain fairness condition, a weak stabilizing program is also a stabilizing program. In probabilistic stabilization, the program recovers to legitimate states with high probability. Finally, nonmasking fault-tolerance [1, 2] refers to programs where the program recovers from states reached in the presence of a limited class of faults. However, this limited set of states may not cover the set of all states.

8 Conclusion

In this paper, we proposed the concept of *active stabilization*, where program's state can be perturbed by *faults* to any arbitrary state and recovery is accomplished in the presence of constant perturbation by an *adversary*. We introduced different types of active stabilizing programs depending upon the behavior of the adversary and the ability of program to recover. We evaluated the cost of verification for passive and active stabilization. We also argued that active stabilization is a highly expressive concept by presenting comparison to fault-contained stabilization and Byzantine self-stabilization.

For future work, we are considering several research directions. We are currently working on developing efficient techniques for verification of active stabilization as well as results about composition of active stabilizing programs.

References

1. Arora, A.: Efficient reconfiguration of trees: A case study in methodical design of nonmasking fault-tolerant programs. In: Science of Computer Programming (1996)
2. Arora, A., Gouda, M.G., Varghese, G.: Constraint satisfaction as a basis for designing nonmasking fault-tolerance. In: ICDCS, pp. 424–431 (1994)
3. Beauquier, J., Kekkonen-Moneta, S.: On ftss-solvable distributed problems. In: WSS, pp. 64–79 (1997)
4. Demirbas, M., Arora, A., Gouda, M.: A pursuer-evader game for sensor networks. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 1–16. Springer, Heidelberg (2003)
5. Devismes, S., Tixeuil, S., Yamashita, M.: Weak vs. self vs. probabilistic stabilization. In: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems, ICDCS 2008, pp. 681–688. IEEE Computer Society, Washington, DC, USA (2008)
6. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11) (1974)
7. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1990)
8. Ghosh, S., Gupta, A.: An exercise in fault-containment: Self-stabilizing leader election. Information Processing Letters (1996)
9. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.: Fault-containing self-stabilizing algorithms. In: Principles of Distributed Computing (PODC), pp. 45–54 (1996)
10. Gouda, M.G., Multari, N.: Stabilizing communication protocols. IEEE Transactions on Computers 40(4), 448–458 (1991)
11. 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)
12. Kulkarni, S.S., Arora, A.: Multitolerance in distributed reset. Chicago J. Theor. Comput. Sci (1998)
13. 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)
14. Nesterenko, M., Arora, A.: Local tolerance to unbounded byzantine faults. In: IEEE Symposium on Reliable Distributed Systems (SRDS), pp. 22–31 (2002)
15. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems 8(4), 424–440 (1997)
16. Zhang, H., Arora, A.: Guaranteed fault containment and local stabilization in routing. In: Computer Networks. Elsevier, Amsterdam (2006)

Robot Networks with Homonyms: The Case of Patterns Formation*

Zohir Bouzid and Anissa Lamani

[1] University Pierre et Marie Curie - Paris 6, LIP6-CNRS 7606, France
zohir.bouzid@lip6.fr
[2] University of Picardie Jules Verne, MIS, Amiens, France
anissa.lamani@u-picardie.fr

Abstract. In this paper, we consider the problem of formation of a series of geometric patterns by a network of oblivious mobile robots that communicate only through vision. So far, the problem has been studied in models where robots are either assumed to have distinct identifiers or to be completely anonymous. To generalize these results and to better understand how anonymity affects the computational power of robots, we study the problem in a new model in which n robots may share up to $1 \leq h \leq n$ different identifiers. We present necessary and sufficient conditions, relating symmetricity and homonymy, that makes the problem solvable. We also show that in the case where $h = n$, making the identifiers of robots invisible does not limit their computational power. This contradicts a recent result of Das et al. To present our algorithms, we use a function that computes the Weber point for many regular and symmetric configurations. This function is interesting in its own right, since the problem of finding Weber points has been solved up to now for only few other patterns.

1 Introduction

Robot networks [9] is an area of distributed computing in which the object of the study is the positional (or spacial) communication paradigm [1]: robots are devoid of any means of *direct* communication; instead, they communicate *indirectly* through their movements and the observation of the current positions of their peers. In most of the studies, robots are oblivious, *i.e.* without any memory about their past observations, computations and movements. Hence, as with communication, memory is *indirect* in some sense, it is collective and spacial. Since any algorithm must use a kind of memory, resolving problems in the context of robot networks is the art of making them remember without memory [7]. But this indirect memory has its limits, and one of the main goals of research in this field is to precisely characterize the limiting power of obliviousness.

The problem of *formation of series of patterns* (or patterns formation) is perhaps the best abstraction that captures the need of robots to have a form of

* This work is supported in part by the DIGITEO project PACTOLE and ANR SPADES.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 92–107, 2011.

memory even when the model does not provide it directly. In this problem, introduced by Das et al. in [5], robots are required to form periodically a sequence of geometric patterns $S = \langle P_1, P_2, \ldots, P_m \rangle$. At each instant, robots should be able to know, just by observing the environment, which patterns they are about to form. In [5], the authors study the problem in both anonymous and eponymous systems. In particular, they show that the only formable series when robots are anonymous are those in which all the patterns of the series have the same symmetricity. In contrast, if robots are endowed with identifiers, nearly all possible series are formable.

The gap between the two worlds, one in which almost nothing is possible and another one where everything is possible, illustrates how much anonymity can be a limiting factor when we are interested to computability. This brings us to ask the following question: is there a possible model where robots are neither eponymous nor completely anonymous ? and if so, what are the possible series we are able to form ? In a recent paper [6], Delporte et al. use a model of "partial anonymity" which they call *homonymy* and they apply it to study Byzantine agreement. In this model, the number of distinct identifiers in the system is given by a parameter h which may take any value between 1 and n. In the current paper, we inject this notion of homonymy to the Suzuki-Yamashita model [9,5] which creates a new model that we call *robot networks with homonyms*.

Studying the problem of patterns formation in this context allows us to get a better insight on how the combined effect of anonymity and obliviousness affects the computational power of robots. We consider series of patterns where all robots are located in *distinct* positions and we assume that identifiers are *invisible*. Our main result is to prove that for a series $S = \langle P_1, P_2, \ldots, P_t \rangle$ to be formable, it is necessary and sufficient that the number of labels h to be strictly greater than $\frac{n}{\text{SNCD}(sym(P_i), sym(P_{i+1}))}$ where P_i, P_{i+1} are any two successive patterns of S, $sym(P)$ denotes the symmetricity of P and $\text{SNCD}(x, y)$ is equal to the smallest divisor of x that does not divide y if any, $n + 1$ otherwise.

To present our algorithms, we use a function that computes the Weber point [10] for many regular configurations, *i.e.* all those in which there is a sort of rotational symmetry around a point. Given a point set P, the Weber point c minimizes $\sum_{r \in P} distance(x, r)$ over all points x in the plane. Our result may be interesting in its own right, since the problem of finding Weber points has been solved up to now for only few other patterns (e.g. regular polygon [2], a line [4]).

Finally, we consider the case where $(n = h = 3)$ and where multiplicity points (in which many robots are located) are allowed. In this setting, we prove that robots are able to form any series of patterns, contradicting a result of [5]. This has an interesting consequence: it means that making the identifiers of robots invisible does not limit their computational power in the considered model, *i.e.* they can form the same series of patterns as robots with visible identifiers. Due to space limitations, the description of this last result is provided in the technical report [3].

Roadmap. The paper is made up of six sections. In Section 2 we describe the computation model we consider in this paper. Section 3 defines the important

notions of symmetricity, regularity and their relations with Weber points. In Section 4 our algorithms for the detection of Weber points are presented. Then, we prove in Sections 5 and 6 the necessary and sufficient conditions that makes series of geometric patterns formable in our model. Finally, we conclude the paper in Section 7. Many proofs are omitted to respect the space limitations. Please see our technical report [3] for more details.

2 Model

Our model is based on a variation of the ATOM model [9,11] used in [5], to which we introduce the notion of homonymy [6]. The system is made up of n mobile robots $r_1, \ldots, r_n$ that communicate only through vision. That is, robots are devoid of any mean of direct communication, the only way of them to communicate is by observing the positions of their peers (a "read") and by moving in the plane (a "write"). Each robot is assigned an identifier (or label) taken from a set of h distinct identifiers $\{1, \ldots, h\}$. The parameter $1 \leq h \leq n$ is called the *homonymy* of the system. The identifier of robot r is denoted by $label(r)$. When two robots have the same label we say that they are *homonymous*. Robots are *oblivious* in the sense that they do not have any memory of their past observations, computations and movements. Hence, their actions are based entirely on their currently observed configuration and their label. Each robot is viewed as a *point* in a plane, thus multiple robots may lie in the same location forming a *multiplicity* point.

We assume also that robots do not obstruct the vision of each others. Robots are *disoriented*, that is, each robot has its own local coordinate system with its own origin, axis, and unit of length which may change at each new activation. However, as in [5], we assume that robots share the same notion of clockwise direction, we say that they have the same *chirality*.

The execution unfolds in *atomic* cycles of three phases Look, Compute and Move. During the Look phase, an activated robot take a snapshot of the environment using its visual sensors. Then it calculates a destination in the Compute phase. The chosen destination is based solely on the label of the robot and the snapshot it obtained in the preceding phase. Finally, In the Move phase, the robot *jumps* to its destination. A subset of robots are chosen for execution (activated) at each cycle by a fictional external entity called a *scheduler*. We require the scheduler to be fair *i.e.* each robot is activated infinitely often. Robots that are activated at the same cycle execute their actions synchronously and atomically.

Notations. Given two distinct points in the plane x and y, $|x, y|$ denotes the Euclidean distance between x and y, $[x, y]$ the line segment between them and (x, y) is the line that passes through both of them. Given a third point c, $\sphericalangle(x, c, y, \circlearrowright)$ denotes the clockwise angle at c from x to y. $\sphericalangle(x, c, y, \circlearrowleft)$ is defined similarly. When the information about orientation can be understood from context, we may omit the parameters $\circlearrowright$ and $\circlearrowleft$. Note that $\sphericalangle(x, c, y, \circlearrowright)$ denotes both the angle and its size, the difference between the two notions should be clear from context. Given a configuration C, its smallest enclosing circle, denoted $\mathrm{SEC}(C)$,

is the circle of minimum diameter such that every point in C is either on or in the interior of this circle. Given a circle CIR, $rad(CIR)$ and $diam(CIR)$ denote its radius and diameter respectively. A point p belong to CIR, written $p \in CIR$ if it on or inside CIR. If S is a set, we denote its cardinality by $|S|$.

Problem Definition. [5] A configuration C of n robots is a set of n elements $\{(p_1, l_1), \ldots (p_n, l_n)\}$ where p_i is the position of robot r_i and l_i is its label. The set of positions $L(C)$ is the set of points occupied by at least one robot in C. A pattern P is represented by a set of n distinct points $\{p_1, \ldots, p_n\}$. Two patterns are said isomorphic if they can be obtained from each others by way of translation, rotation and uniform scaling. $size(P)$ is the cardinality of P. We say that a system of robots has formed the pattern P if the set of of points of the current configuration $L(C)$ is isomorphic to P. **The Problem of Formation of Series of Patterns** is defined as follows. As input, robots are given a periodic series of patterns $\langle P_1, P_2, \ldots, P_m \rangle^{\infty}$. It is required that for each time $\tau : \forall P_i \in S : \exists \tau_i :$ robots form P_i at time τ_i.

3 Symmetricity, Regularity and Weber Points

In this section we formally define two metrics that quantify how much configurations of *distinct* robots may be symmetric: the strongest one, called symmetricity and the weaker one, called regularity. We then explain how these two notions relate to Weber points. We start by defining a polar coordinate system which we use to state our definitions and algorithms.

3.1 Polar Coordinate System

In the remaining of the paper, we express the locations of robots using a local *polar* coordinate system based on the SEC of the current configuration and the position of the local robot [8]. The *center* c of the coordinate system is common to all robots and it coincides with the center of the SEC. In contrast, the *unit of measure* is local to each robot r and its definition depends on whether the robot is located on c or not. In the first case, the point $(1, 0)$ of the local coordinate system of r is any point in the plane that is not occupied by a robot. In the second case, the point $(1, 0)$ is the current location of r. The positive common *clockwise orientation* is provided by the underlying model. Note that (d, θ) denotes the point located at distance d from c and angle θ.

3.2 Symmetricity

Definition 1 (View). *The* view *of robot r, denoted $\mathcal{V}(r)$, is the current configuration expressed using the local polar coordinate system of r.*

Notice that since robots are located in distinct positions, our definition of the view implies that if a robot is located at the center c, then its view is unique, *i.e.* it cannot be equal to the view of another robot. Now we define the following equivalence relation on robots based on their views.

Definition 2 ($\backsim$). *Given a configuration P, given any two robots $r, r' \in P$:*

$$(r \backsim r') \Leftrightarrow (\mathcal{V}(r) = \mathcal{V}(r'))$$

The equivalence class of r is denoted by $[r]$.

Property 1. [5] Let $r \in P$. If $|[r]| > 2$, $[r]$ is a set of robots located at the vertices of a convex regular polygon with $|[r]|$ sides whose center c is the center of $SEC(P)$.

Definition 3 (Symmetricity). *The* symmetricity *of a configuration P, denoted $sym(P)$, is the cardinality of the smallest equivalence class defined by $\backsim$ on P. That is, $sym(P) = min\{|[r]| \mid r \in P\}$. If $sym(P) = m$, we say that P is m-symmetric.*

Note that despite the fact that our definition of symmetricity is different from the one used in [5], it is still equivalent to it when configurations does not contain multiplicity points, the case in which we are interested in the current paper. The next lemma follows from our definition of symmetricity and the fact that the view of a robot located at c is unique.

Lemma 1. *If a configuration P of distinct robots is m-symmetric with $m > 1$, then no robot of P is located in c.*

In the following lemma we prove that if no robot is located at the center of $SEC(P)$, then all the equivalence classes have the same cardinality.

Lemma 2. *Let P be a configuration of n distinct robots, and let c be the center $SEC(P)$. If no robot in P is located at c, it holds that:*

$$\forall r, r' \in P : |[r]| = |[r']| = m$$

Lemma 3. *Let P be a m-symmetric configuration of n distinct points with center (of symmetricity) c and with $m > 1$. There exists a partition of P into $x = n/m$ subsets $S_1, \ldots, S_x$ such that each of them is a convex regular polygon of m sides with center c.*

The following lemma will be used by our algorithm for series formation in Section 6 to break the symmetricity of configurations.

Lemma 4. *Let P be a m-symmetric configuration of n distinct points with $m > 1$. If $h > n/m$, there exists two robots r and r' such that $\mathcal{V}(r) = \mathcal{V}(r')$ and $label(r) \neq label(r')$.*

3.3 Regularity

In this section we formally define a weaker form of symmetry that we call regularity. In the next section we state its precise relation with symmetricity. We start by giving some useful definitions:

Definition 4 (Successor). *Given a set P of distinct points in the plane, and $c \notin P$ a fixed point. Given some polar ordering of points in P around c. Let us denote by $S(r, c, \circlearrowleft)$ the the clockwise successor of r according to the clockwise polar ordering of points P around c. The anticlockwise successor of r, denoted $S(r, c, \circlearrowright)$, is defined similarly.*

Formally [8], $r' = S(r, c, \circlearrowleft)$ is a point of P distinct from r such that:

- *If r is not the only point of P that lies in $[c, r]$, we take r' to be the point in $P \cup [c, r]$ that minimizes $|r, r'|$.*
- *Otherwise, we take r' such that no other point of P is inside $\sphericalangle(r, c, r', \circlearrowleft)$. If there are many such points, we choose the one that is further from c.*

When the center c and the clockwise orientation are clear from context or meaningless, we simply write $S(r, c)$, $S(r, \circlearrowright)$ or $S(q)$ to refer to the successor of r.

Definition 5 (k-th Successor). *The k-th clockwise successor of r around c, denoted $S^k(r, c, \circlearrowleft)$ (or simply $S^k(q)$) , is defined recursively as follows:*

- *$S^0(r) = r$ and $S^1(r) = S(r)$.*
- *If $k > 1$, $S^k(r) = S(S^{k-1}(r))$.*

Definition 6 (String of angles). *Let P be a set of distinct points in the plane, and $c \notin P$ a fixed point. The (clockwise) string of angles of center c in r, denoted by $SA(r, c, \circlearrowleft)$ is the string $\alpha_1 \dots \alpha_n$ such that $\alpha_i = \sphericalangle(S^{i-1}(r), c, S^i(r), \circlearrowleft)$. The anticlockwise string of angles, $SA(r, c, \circlearrowright)$, is defined in a symmetric way. Again, when the information about the center of the string c and/or its clockwise orientation is clear from context, we simply write $SA(r, c)$, $SA(r, \circlearrowright)$ or $SA(r)$.*

Definition 7 (Periodicity of a string). *A string x is k-periodic if there exists a string w and an integer $1 \leq k \leq n/2$ such that $x = w^k$. The greatest k for which x is k-periodic is called* the periodicity *of x and is denoted by $per(x)$.*

The following property states that the periodicity of the string of angles does not depend on the process in which it is started nor on the clockwise or anticlockwise orientation.

Lemma 5. *Let P be a set of distinct points in the plane, and let $c \notin P$ be a point in the plane. The following property holds:*

$$\exists m : \forall r \in P : \forall d \in \{\circlearrowright, \circlearrowleft\} : per(SA(r, c, d)) = m$$

Theorem 1. *Given is a set of points P, a center $c \notin P$. There exists an algorithm with running time $O(n \log n)$ that computes $SA(c)$.*

The lemma means that when it comes to periodicity, the important information about a string of angles is only its center c. Hence, in the following we may refer to it by writing $SA(c)$.

Definition 8 (Regularity). *Let P be a set of n distinct points in the plane. P is m-regular (or regular) if there exists a point $c \notin P$ such that $per(SA(c)) = m > 1$. In this case, the* regularity *of P, denoted $reg(P)$, is equal to $per(SA(c))$. Otherwise, it is equal to 1. The point c is called the* center *of regularity.*

Theorem 2. *Given is a set of points P, a center $c \notin P$. There exists an algorithm with running time $O(n \log n)$ that detects if c is a center of regularity for P.*

3.4 Weber Points

In this section we state some relations between symmetricity and regularity. Then we prove that their centers are necessarily Weber points, hence unique when the configuration is not linear. The following lemma is trivial, its proof is left to the reader. It states the fact that if a configuration is m-symmetric with $m > 1$, it is necessarily m-regular.

Lemma 6. *Let P be any configuration with $sym(P) > 1$. It holds that $reg(P) = sym(P)$ and the center of regularity of P coincides with its center of symmetricity.*

Note that the claim holds only if $sym(P) > 1$. In general, $reg(P) \geq sym(P)$. The next lemma shows in what way the regularity of a configuration can be strengthened to become a symmetricity.

Lemma 7. *Let P be any configuration with $reg(P) = m > 1$, and let c be its center of regularity. There exists a configuration P' that can be obtained from P by making robots move along their radius such that $sym(P') = m$. Moreover, c is the center of symmetricity of P'.*

The following lemma states that the center of symmetricity of any configuration P if any is also its Weber point. The same claim was made in [2] in the cases when $sym(P) = n$ (equiangular) and $sym(P) = n/2$ (biangular). Our proof uses the same reasoning as theirs.

Lemma 8. *Let P be a configuration such that $sym(P) > 1$ and let c be its center of symmetricity. It holds that c is the Weber point for P.*

Lemma 9. *Let P be a configuration such that $reg(P) = m > 1$ and let c be its center of regularity. It holds that c is the Weber point for P.*

The next corollary follows directly from Lemma 9 and the fact that Weber points are unique.

Corollary 1 (Unicity of c). *The center of regularity is unique if it exists.*

4 Detection of Regular Configurations

In this section we show how to identify geometric configurations that are regular. We present two algorithms that detects whether a configuration P of n points

given in input is m-regular for some $m > 1$, and if so, they output its center of regularity. The first one detects the regularity only if m is even, it is very simple and runs in $O(n \log n)$ time. The second one can detect any regular configuration, provided that $m \geq 3$. It is a little more involved and runs in $O(n^4 \log n)$ time. We assume in this section that P is not a configuration in which all the points are collinear.

4.1 Preliminaries

In this section, we state some technical lemmas that help us in the presentation and the proofs.

Lemma 10. *Let P be a regular configuration of n distinct points and let c be its center of regularity. Let $m = reg(P)$. The following property holds:*

$$\forall r \in P : (\sphericalangle(r, c, S^{n/m}(r, \circlearrowright), \circlearrowright) = 2\Pi/m) \wedge (\sphericalangle(r, c, S^{n/m}(r, \circlearrowleft), \circlearrowleft) = 2\Pi/m)$$

The following lemma proves that when a configuration P is m-regular with m even, then for each point in $r \in P$ there exists a corresponding point $(S^{n/2}(r))$ that lies on the line that passes through r and c with c being the center of regularity of P.

Lemma 11. *Let P be a regular configuration of n distinct points and let c be its center of regularity. The following property holds:*

$$(reg(P) \text{ is even}) \Rightarrow (\forall r \in P : \sphericalangle(r, c, S^{n/2}(r)) = \Pi)$$

4.2 Detection of Even Regularity

Theorem 3. *Given P a configuration of n distinct points with n even. There exists an algorithm (running in $O(n \log n)$ steps) that detects if P is m-regular with m even, and if so, it outputs m and the center of regularity.*

4.3 Detection of Odd Regularity

Definition 9 (α-Circle). *Given two distinct points x and y and an angle $0 < \alpha < \Pi$, we say that the circle C_{xy} is a α-circle for x and y if $x, y \in C_{xy}$ and there* exists *a point $p \in C_{xy}$ such that $\sphericalangle(x, p, y) = \alpha$.*

In the following we present three known properties [2] about α-circles.

Property 2. it holds that $\sphericalangle(x, p', y) = \alpha$ for every point $p' \in C_{pq}$ on the same arc as p.

Property 3. If $\alpha = \Pi/2$, the α-circle is *unique* and is called the Thales circle.

Property 4. If $\alpha \neq \Pi/2$, there are *exactly* two α-circles. We denote them in the following by C_{xy} and C'_{xy}. $\alpha = \Pi/2$ can be seen as a special case in which $C_{xy} = C'_{xy}$.

Definition 10 ($C_{xy} \cap C_{yz}$)**.** *Given C_{xy} and C_{yz}, we define their intersection, denoted $C_{xy} \cap C_{yz}$, as the point p such that $(p \neq y) \wedge (p \in C_{xy}) \wedge (p \in C_{yz})$. If p does not exists we write $C_{xy} \cap C_{yz} = \emptyset$.*

Lemma 12. *Let $m \geq 3$. Let P be an m-regular configuration of n distinct points with center $c \notin P$. Let x be any point in P, and let us denote by y and z the points $S^{n/m}(x, c, \circlearrowleft)$ and $S^{n/m}(x, c, \circlearrowright)$ respectively. Let $C_{xy}, C'_{xy}, C_{xz}, C'_{xz}$ be the $2\Pi/m$-circles for the pairs of points (x,y) and (x,z). It holds that $c \in (C_{xy} \cap C_{xz}) \cup (C'_{xy} \cap C_{xz}) \cup (C_{xy} \cap C'_{xz}) \cup (C'_{xy} \cap C'_{xz})$.*

Lemma 13. *Given $3 \leq m \leq n$, given P a configuration of n distinct points. Let x be any point in P The following property holds:*

$$(P \text{ is } m\text{-regular with center } c)$$

$$\Leftrightarrow$$

$$(\exists y, z \in P : (x \neq y \neq z) \wedge (c \in (C_{xy} \cap C_{xz}) \cup (C'_{xy} \cap C_{xz}) \cup (C_{xy} \cap C'_{xz}) \cup (C'_{xy} \cap C'_{xz})))$$

Where $C_{xy}, C_{xz}, C'_{xy}, C'_{xz}$ are $2\Pi/m$-circles of the corresponding points.

Theorem 4. *Given $3 \leq m \leq n$, given P a configuration of n distinct points. There exists an algorithm (running in $O(n^3 \log n)$) that detects if P is m-regular, and if so, it outputs the center of regularity.*

Proof. The algorithm is the following. We fix any robot $x \in P$. Then, for every $y \in P \backslash \{x\}$, for every $z \in P \backslash \{x, y\}$, for every $c \in (C_{xy} \cap C_{xz}) \cup (C'_{xy} \cap C_{xz}) \cup (C_{xy} \cap C'_{xz}) \cup (C'_{xy} \cap C'_{xz})$, we test if c is a center of regularity (Theorem 2, $O(n \log n)$ time). Lemma 13 guarantees that if P is m-regular, the test will be conclusive for at least one pair (y, z) of robots. The whole algorithm executes in $O(n^3 \log n)$: we browse all the possible pairs (y, z), and for each pair we generate up to four candidates for the center of regularity, hence we have $O(n^2)$ candidates. Then, $O(n \log n)$ time is needed to test each candidate. Note that our algorithm follows the same patterns as those presented in [2]: generating a restricted set of candidates (points) and testing whether each of them is a center of regularity. □

Theorem 5. *Given P a configuration of n distinct points. There exists an algorithm (running in $O(n^4 \log n)$ steps) that detects if P is m-regular with $m \geq 3$, and if so, it outputs m and the center of regularity.*

Proof. It suffices to generates all the divisors m of n that are greater than 2. Then, for each m, we test if P is m-regular using the algorithm of Theorem 4). When the test is conclusive, this algorithm return the center of regularity c, so we can output Regularity m, Center c. If test was inconclusive for every generated m, we output Not Regular. □

Theorem 6. *Given P a configuration of n distinct points. There exists an algorithm (running in $O(n^4 \log n)$ steps) that detects if P is m-regular with $m \geq 2$, and if so, it outputs m and the center of regularity.*

Proof. We combine the algorithms of Theorems 3 and 5. First, we test if P is m-regular for some even m using the algorithm of Theorem 3 ($O(n \log n)$). If so, we output m and the center of regularity c which are provided by the called algorithm. Otherwise, we test odd regularity using the algorithm of Theorem 5 ($O(n^4 \log n)$) but by restricting the analysis to only the *odd* divisors of n (the even divisors were already tested). □

5 Formation of a Series of Geometric Patterns: Lower Bound

In this section we prove a necessary condition that geometric series have to satisfy in order to be formable. The condition relates three parameters: the number of robots in the system n, its homonymy h and the symmetricity of the patterns to form. It is stated in Theorem 7.

Property 5. [5] For any configuration P of n distinct robots, $sym(P)$ divides n.

Lemma 14. *Let P be a configuration of n distinct robots with symmetricity s,* i.e. *$sym(P) = s$. For any divisor d of s, if $h \leq \frac{n}{d}$, then for any pattern formation algorithm, there exists an execution where all subsequent configurations P' satisfy $sym(P') = k \cdot d$, $k > 1$.*

Proof. The lemma holds trivially if $s = 1$, hence we assume in the following that $s > 1$. According to Lemma 3, there exists a partition of P into $x = n/s$ subsets $S_1, \ldots, S_x$ such that the s robots in each S_i occupy the vertices of a regular convex polygon of s sides whose center is c.

Now, partition each set S_i into $\frac{s}{d}$ subsets $T_{i1}, \ldots, T_{(i\frac{s}{d})}$ with $|T_{ij}| = d$ for each $i \in \{1, \ldots, x\}, j \in \{1, \ldots, \frac{s}{d}\}$. Each subset T_{ij} is chosen in such a way that the d robots belonging to it are located in the vertices of a regular convex polygon of d sides with center c. This choice is possible because d is a divisor of $s = |S_i|$. For example, let $r_1, \ldots, r_s$ be the robots of S_i ordered according to some polar ordering around c. T_{i1} is the set of robots $\{r_1, r_{(\frac{s}{d}+1)}, r_{(\frac{2s}{d}+1)}, \ldots, r_{(\frac{(d-1)s}{d}+1)}\}$. Clearly, this subset defines a regular polygon of d sides with center c.

There are total of $\frac{s \cdot x}{d} = \frac{n}{d}$ subsets T_{ij}. So we have also a total of $\frac{n}{d}$ concentric regular polygons of d sides. What is important to notice now is that the robots in each T_{ij} have the same view.

Since $h \leq \frac{n}{d}$, there exists a set of labels $|\mathcal{L}| = h$, and a labeling of robots in P such that (1) The same label is assigned to the robots that belong to the same subset T_{ij}. (2) For each label $l \in \mathcal{L}$, there exists a robot $r_i \in P$ such that l is the label of r_i.

Since we assume that algorithms are deterministic, the actions taken by robots at each activation depend solely on they observed view and their identity (label). Hence, two robots having the same view and the same label will take the same actions if they are activated simultaneously. Therefore, the adversary can

guarantee that the network will always have a symmetricity $\geq d$ by activating each time the robots that belong to the same T_{ij} together. This way, we are guaranteed to have all the subsequent configurations that consists of a set of $\frac{n}{d}$ concentric regular polygons of d sides. That is, all subsequent configuration have a symmetricity that is a multiple of d. This proves the lemma. □

The following lemma states a necessary condition for a geometric figure P_j to be formable starting from P_i.

Lemma 15. *If the current configuration P_i has symmetricity $sym(P_i) = s$, the configuration P_j with symmetricity $sym(P_j) = s'$ is formable only if (1) $size(P_j) = size(P_i)$ and (2) $h > \frac{n}{\text{SNCD}(s,s')}$ where* SNCD *(read smallest non common divisor) is equal to the smallest x that divides s but not s' if any, $n+1$ otherwise.*

Proof. Assume towards contradiction that (1) $h \leq \frac{n}{\text{SNCD}(s,s')}$ and (2) P_j is formable. Note that since $h \geq 1$, (1) implies that $\text{SNCD}(s, s') \neq n+1$. Otherwise we would have $h \leq 0$, contradiction. By definition, $\text{SNCD}(s, s') \neq n+1$ implies that $\text{SNCD}(s, s') = t$ divides s but not s'.

According to Lemma 14, for any algorithm, there exists an execution starting from P_i where all subsequent configurations P' satisfy $sym(P') = k{\cdot}t$, $k > 1$. But $sym(P_j) = s'$ is not multiple of t, hence P_j is never reached in this execution. This means that P_j is not formable starting from P_i, which contradicts (2). Hence, the lemma is proved. □

Now, we are ready to state the necessary condition for formation of geometric series. It relates the symmetricity of its constituent patterns and the homonymy of the system.

Theorem 7. *A cyclic series of distinct patterns $\langle P_1, P_2, ..., P_m\rangle^\infty$ each of size n is formable only if*

$$\forall i \in \{1, \ldots, m\} : h > \frac{n}{\text{SNCD}(sym(P_i), sym(P_{(i \bmod m)+1}))}$$

Proof. Follows from Lemma 15. □

6 Formation of Series of Patterns: Upper Bound

In this section, we present an algorithm that allows robot to form a series of patterns, provided that some conditions about homonymy and symmetricity are satisfied. The result is stated in the following theorem:

Theorem 8. *A cyclic series of distinct patterns $\langle P_1, P_2, ..., P_m\rangle^\infty$ each of size n is formable if and only if*

$$\forall i \in \{1, \ldots, m\} : h > \frac{n}{\text{SNCD}(sym(P_i), sym(P_{(i \bmod m)+1}))}$$

The only if part was proved in Section 5. The remaining of the current section is devoted to the proof of the if part of the theorem.

6.1 Intermediate Configurations

During the formation of a pattern P_i (starting from P_{i-1}), the network may go through several intermediate configurations. We define in the following four classes of intermediate patterns $\mathcal{A}, \mathcal{B}, \mathcal{C}$ and $\mathcal{D}$. Each one of them encapsulate some information that allows robots to unambiguously determine which pattern the network is about to form. This information is provided by a function, STRETCH, which we define separately for each intermediate pattern.

Definition 11 (Configuration of type $\mathcal{A}$). *A configuration P of n points is of type $\mathcal{A}$ (called BCC in [5]) if the two following conditions are satisfied:*

1. *there exists a point $x \in P$ such that the diameter of $SEC_1 = SEC(P)$ is a least ten times the diameter of $SEC_2 = SEC(P \setminus \{x\})$.*
2. *SEC_1 and SEC_2 intersect at exactly one point called the* base-point *(BP).*

The point x is called the pivot *whereas the point on SEC_2 directly opposite BP is called the* frontier point *(FP).*

STRETCH(P) *is equal to* $\lfloor \frac{rad(SEC_1)}{(h+1) \cdot rad(SEC_2)} \rfloor$.

Definition 12 (Configuration of type $\mathcal{B}(m)$). *A configuration P of n distinct robots is of type $\mathcal{B}(m)$ (adapted from $SCC[m]$ in [5]) with $1 < m < n$, if the following conditions are satisfied:*

1. *$SEC_1 = SEC(P)$ has exactly m points on its circumference which form a regular convex polygon with m sides.*
2. *Let SEC_2 be the SEC of the robots that are not on the SEC_1,* i.e. *$SEC_2 = SEC(P \setminus \{r \in P \mid r \text{ is on } SEC_1\})$. SEC_1 and SEC_2 are concentric such that $rad(SEC_1) > 10 \cdot rad(SEC_2)$.*

STRETCH(P) *is equal to* $\frac{1}{2} \cdot \lfloor \frac{rad(SEC_1)}{(h+1) \cdot rad(SEC_2)} \rfloor$. *It can be easily checked that a given configuration cannot be of both types $\mathcal{A}$ and $\mathcal{B}$ (their respective SEC_2 do not intersect).*

Definition 13 (Configuration of type $\mathcal{C}(m)$). *A configuration P of n distinct robots is of type $\mathcal{C}(m)$ with $1 < m < n$ if 1) it is not of type $\mathcal{B}$ and 2) it is m-symmetric (ref. Definition 3).*

STRETCH(P) *is computed as follows. Since P is m-symmetric with $m > 1$, there exists a partition of P into $x = n/m$ subsets $S_1, \ldots, S_x$ such that each of them is a convex regular polygon of m sides with center c (Lemma 3). This means that each S_i defines a circle with center c. Assume w.l.o.g that $\forall i \in \{1 \ldots x-1\}$: $rad(S_i) \leq rad(S_{i+1})$.* STRETCH$(P) = Max\{\lfloor \frac{rad(S_{i+1})}{(h+1) \cdot rad(S_i)} \rfloor \mid i \in \{1 \ldots x-1\}\}$.

Clearly, a configuration of type $\mathcal{C}$ cannot be of type $\mathcal{A}$ since the former is symmetric while the latter is not.

Definition 14 (Configuration of type $\mathcal{D}(m)$). *A configuration P of n distinct robots is of type $\mathcal{D}(m)$ with $1 < m < n$ if (1) Points are not all on the*

same line, (2) P is not of type $\mathcal{B}$ and (3) P is m-regular but not symmetric ($Sym(P) = 1$).

STRETCH(P) *is computed as follows. Let c be the center of regularity. c can be computed in polynomial time using the algorithm of Section 4 (Theorem 6). Then for each $r_i \in P$, we compute $|c, r_i|$, its distance from c. Assume w.l.o.g. that $\forall i \in \{1 \dots x-1\} : |c, r_i| \leq |c, r_{i+1}|$.* STRETCH$(P) = Max\{\lfloor \frac{|c,r_{i+1}|}{(h+1)\cdot|c,r_i|} \rfloor \mid i \in \{1 \dots x-1\}\}$. *Note that the stretch of configurations of type $\mathcal{C}(m)$ is a particular case of that of type $\mathcal{D}(m)$, but since the former configurations are symmetric, we can compute their stretch without resorting to the computation of the center of regularity.*

Decoding the stretch. Let F be a one-to-one function [5] that maps each pattern P_i to a real number $t_i = F(P_i)$. If there is a pattern P_i that is of type $\mathcal{A}, \mathcal{B}, \mathcal{C}$ or $\mathcal{D}$, we exclude the value STRETCH(P_i) from the domain of F. To simplify the proofs, we assume that $F(P_i) > 10$ for any P_i. When robots are about to form the pattern P_i, they use intermediate configurations with stretch t_i. By computing the stretch, robots can unambiguously identify which configuration they are about to form ($F^{-1}(t_i)$).

6.2 Transitions between Configurations

In this section we describe some algorithms that describe some transformations between patterns.

Lemma 16. *Starting from any configuration of type $\mathcal{D}$ with stretch t, there exists an algorithm that builds a configuration of type $\mathcal{A}$ with the same stretch.*

Lemma 17. *Given a sequence of patterns $\langle P_1, P_2, ..., P_m \rangle^\infty$. Starting from any configuration of type P_i with stretch $sym(P_i) = 1$, there exists an algorithm that builds a configuration of type $\mathcal{A}$ with stretch $F(P_{i+1})$.*

The following two lemmas are from [5].

Lemma 18. *Starting from any configuration of type A, it is possible to form any single pattern.*

Lemma 19. *Starting from any configuration of type $B(m)$, it is possible to form any single pattern P such that $sym(P) = k \cdot m, k > 1$.*

Lemma 20. *Consider a robot network of n robots in configuration P_i. Let $sym(P_i) = m > 1$ and $sym(P_{i+1}) = m'$. If $h > \frac{n}{\text{SNCD}(m,m')}$, there exists an algorithm that brings the network to a configuration Q such that either Q is of type $\mathcal{B}(x), x <$ SNCD(m, m') or of type $\mathcal{A}$ both with stretch $F(P_{i+1})$.*

The remaining of this section is devoted to the proof of this lemma.

The transformation algorithm is given in Figure 1. It is executed by robots during their Compute phases until one of the two desired configurations is obtained. Its description is based on the polar coordinate system of Section 3.1.

Function:
$targetSym(k)$: The target symmetricity when robots try to form P_{k+1}
It is equal to the greatest divisor of $sym(P_{k+1})$ that is $<$ SNCD$(sym(P_k), sym(P_{k+1}))$

Actions:
(1) $P \leftarrow$ Observed Configuration
(2) $SEC \leftarrow$ Smallest Enclosing Circle(P)
(3) $c \leftarrow$ Center(SEC)
(4) $sym \leftarrow sym(P)$
(5) **If** $(P = P_i)$ **then** $t \leftarrow F(P_{i+1})$
(6) **else** $t \leftarrow stretch(P)$ **endif**
(7) $id \leftarrow$ My Identifier
(8) $rad \leftarrow$ Radius(SEC)
(9) **if** $(sym > 1) \wedge ((sym > targetSym(k)) \vee (P$ is not of type $\mathcal{B}(sym)))$
(10) $d \leftarrow Min(|r_i, c|; r_i \in P)$
(11) $S \leftarrow \{r_i \in P \mid |r_i, c| = d\}$
(12) $minView \leftarrow Min(V(r_i); r_i \in S)$
(13) $Elected \leftarrow \{r_i \in S \mid V(r_i) = minView\}$
(14) **if** $(r_i \in Elected)$
(15) return $((t \cdot (h + 1) + id - 1) \cdot rad, 0)$
(16) **else**
(17) return My Position
(18) **endif**
(19) **endif**

Fig. 1. Symmetry Breaking

The principal idea is to use identifiers of robots in order to break the symmetry of configurations. It does so by making robots choose their destination according to their identity. This way, if two robots with similar views but different identities are activated simultaneously, their views at the end of the cycle will be different and the symmetricity decreases.

Let us observe the following four properties about the algorithm:

1. Let c be the center of symmetricity of the initial configuration P_i. Note that c is therefore a Weber point (Lemma 8). Since the algorithm makes robots move only through their radius with c (line 15), the Weber point remains invariant during all the execution. This implies that any successive regular/symmetric configuration will have necessarily c as its center of regularity/symmetricity.
2. Again, since robots move only through their radius with c, regularity remains invariant during all the execution. It is thus equal to $reg(P_i)$. But since $sym(P_i) = m > 1$, it holds according to Lemma 6 that $reg(P_i) = sym(P_i)$. Hence, all the successive configurations will be m-regular, including the final one. This means that if a configuration P' with $sym(P')$ is reached, it is of type $\mathcal{D}(m)$.
3. At each cycle, the algorithm chooses a set *Elected* of robots having the same view (equivalence class) (lines 10-13).

   Since the algorithm is executed only if the current configuration is m-symmetric $(m > 1)$, it holds according to Lemma 2 that $|Elected| = m$. The positions chosen by robots in *Elected* are located outside the current

SEC. Hence, moving to these positions cannot increase the symmetricity of the configuration. It follows that the symmetricity of the configurations can either *decrease* or remain the same during the execution.

4. The actions of robots maintain the same stretch during the whole execution, and it is equal to $F(P_{i+1})$ (line 5).

We prove the following claim about the algorithm:

Lemma 21. *Given the conditions of Lemma 20, if robots are executing algorithm 7, then there exists a time at which they reach a configuration P' with $sym(P') < \text{SNCD}(m, m')$.*

Property 6. Let m' be the smallest symmetricity of all the configurations reached by the execution of the algorithm. According to Lemma 21 $m' < \text{SNCD}(m, m')$. Let T be the first reached configuration for which $sym(T) = m'$. Since all the configurations reached after T if any have a symmetricity equal to m' this means that at each cycle after T is reached, there are m' robots that are elected to move.

Lemma 22. *If $sym(T) = m' > 1$, then either T is of type $\mathcal{B}(m')$ or a configuration of type $\mathcal{B}(m')$ can be obtained from T after one cycle.*

Lemma 23. *If $m' = 1$ and all the points in T are collinear, then either T is of type $\mathcal{A}$ or a configuration of type $\mathcal{A}$ can be reached from T be a movement of a single robot.*

Lemma 24. *If $m' = 1$ and the points in T are* not *all collinear, then either T is of type $\mathcal{A}$ or $\mathcal{D}(m)$*

Proof of Lemma 20. Follows from Lemma 21, Property 6 and Lemmas 23, 24 and 22.

Theorem 9. *Consider a robot network of n robots in configuration P_i. Let $sym(P_i) = m > 1$ and $sym(P_{i+1}) = m'$. If $h > \frac{n}{\text{SNCD}(m,m')}$, there exists an algorithm that forms P_{i+1}.*

Proof. Follows from Lemmas 18, 19 and 20. □

7 Conclusion

In this paper, we considered the problem of formation of series of geometric patterns. We studied the combined effect of obliviousness and anonymity on the computational power of mobile robots with respect to this problem. To this end, we introduced a new model, robots networks with homonyms that encompasses and generalizes the two previously considered models in the literature in which either all robots have distinct identifiers or they are anonymous. Our results suggest that this new model may be a useful tool to get a better insight on how anonymity interacts with others characteristics of the model to limit its power.

Acknowledgement. The authors would like to thank Shantanu Das for his generous help and the anonymous referees for their comments and suggestions to improve the quality of the paper.

References

1. Agmon, N., Peleg, D.: Fault-tolerant gathering algorithms for autonomous mobile robots. SODA 11(14), 1070–1078 (2004)
2. Anderegg, L., Cieliebak, M., Prencipe, G.: Efficient algorithms for detecting regular point configurations. In: Coppo, M., Lodi, E., Pinna, G.M. (eds.) ICTCS 2005. LNCS, vol. 3701, pp. 23–35. Springer, Heidelberg (2005)
3. Bouzid, Z., Lamani, A.: Robot networks with homonyms: The case of patterns formation. hal.inria.fr (2011)
4. Chandrasekaran, R., Tamir, A.: Algebraic optimization: the fermat-weber location problem. Mathematical Programming 46(1), 219–224 (1990)
5. Das, S., Flocchini, P., Santoro, N., Yamashita, M.: On the computational power of oblivious robots: forming a series of geometric patterns. In: PODC, pp. 267–276. ACM, New York (2010)
6. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kermarrec, A.M., Ruppert, E., Tran-The, H.: Byzantine agreement with homonyms. In: PODC (2011)
7. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Remembering without memory: Tree exploration by asynchronous oblivious robots. Theoretical Computer Science 411(14-15), 1583–1598 (2010)
8. Katreniak, B.: Biangular circle formation by asynchronous mobile robots. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 185–199. Springer, Heidelberg (2005)
9. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM Journal of Computing 28(4), 1347–1363 (1999)
10. Weiszfeld, E.: Sur le point pour lequel la somme des distances de n points donnes est minimum, t6hoku math. J 43, 355–386 (1937)
11. Yamashita, M., Kameda, T.: Leader election problem on networks in which processor identity numbers are not distinct. IEEE Trans. Parallel Distrib. Syst. 10(9), 878–887 (1999)

A Non-topological Proof for the Impossibility of k-Set Agreement

Hagit Attiya[1] and Armando Castañeda[2]

[1] Department of Computer Science, Technion, Haifa 32000, Israel
hagit@cs.technion.ac.il
[2] IRISA-INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
armando.castaneda@inria.fr

Abstract. In the *k-set agreement* task each process proposes a value, and it is required that each correct process has to decide a value which was proposed and at most k distinct values must be decided. Using topological arguments it has been proved that k-set agreement is unsolvable in the asynchronous *wait-free* read/write shared memory model, when $k < n$, the number of processes.

This paper presents a simple, non-topological impossibility proof of k-set agreement. The proof depends on two simple properties of the *immediate snapshot executions*, a subset of all possible executions, and on the well known graph theory result stating that every graph has an even number of vertices with odd degree (the *handshaking lemma*).

Keywords: Set agreement, Shared memory, Wait-freedom.

1 Introduction

In a breakthrough result, Fischer, Lynch and Paterson proved [7] that it is impossible to solve consensus in the asynchronous message passing system in which at most one process, which is unknown in advance, can crash. Herlihy [11] and Loui and Abu-Amara [17] extended this impossibility result to the asynchronous wait-free read/write shared memory model, where *wait-free* means that in each execution all processes but one can fail by crashing. Recall that in the *consensus* task, each process proposes a value, and it is required that every correct process decides on a value proposed by some process and no two correct processes decide distinct values.

Later, in order to study the border between solvable and unsolvable tasks in presence of asynchrony and failures, Chaudhuri [6] introduced a natural generalization of consensus, called *k-set agreement*; in this task, each process proposes a value and it is required that each correct process decides on a value proposed by a process and at most $k \geq 1$ distinct values are decided. For $k = 1$, k-set agreement is exactly consensus, and for $k = n$, the number of processes in the system, k-set agreement is trivial, since every process can decide on its proposal. The paper shows that k-set agreement can be solved by a t-resilient asynchronous algorithm, when $t < k$. An algorithm is *t-resilient*, $1 \leq t \leq n-1$, if it solves the

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 108–119, 2011.

problem even in executions where up to t processes crash. This means that if the number of failures is strictly smaller than the number of possible decision values, then k-set agreement is solvable. Chaudhuri [6] also conjectured that k-set agreement is unsolvable if $t \geq k$. For the case $k = 1$, this conjecture matches the impossibility of solving consensus, namely, 1-set agreement, with a single crash failure [11,17]. Notice that for the the wait-free case, i.e., $t = n-1$, this conjecture says that only the trivial n-set agreement task has a wait-free solution.

Chaudhuri's conjecture was proved by Borowsky and Gafni [3], Herlihy and Shavit [15] and Saks and Zaharoglou [18]. Indeed, these three papers discovered a strong relationship between distributed computing and topology and used this topological approach for proving the conjecture of Chaudhuri. Later, Attiya and Rajsbaum [1] and Herlihy and Rajsbaum [13] presented two other topological impossibility proofs of k-set agreement. Although these proofs are not extremely complicated, they use concepts and results that are not widely known by the distributed computing community; the kind of arguments they use vary from combinatorial and algebraic to continuous arguments.

This paper presents a simple, non-topological impossibility proof for k-set agreement. This proof does not demand from the reader any previous knowledge of topology at all. Very roughly, the proof considers the *immediate snapshot (IS) executions* [1,3,4,18] of an algorithm, a subset of all possible executions, and constructs a graph, whose vertices are the IS executions, and whose edges connect two IS executions (vertices) only if they satisfy certain indistinguishability conditions. Then, using the well known *handshaking lemma*, stating that every graph has an even number of vertices with odd degree, the proof concludes that there must be at least one execution in which at least $k + 1$ values are decided.

We believe that it is valuable to have several proofs of this important result, since they provide different perspectives on it. In particular, the proof we present in this paper gives an operational insight into the impossibility of set agreement. Such a perspective cannot be easily obtained from the known topological impossibility proofs for k-set agreement.

We stress that this paper does not argue that the use of topology for proving the impossibility of k-set agreement is "artificial", namely, that topology has been used before as a "trick" for proving the result. The reader has not to understand this paper in that way. The fundamental reason why k-set agreement is not solvable is topological, and more precisely, it has to do with Sperner's lemma [10, p. 36]. We strongly encourage the reader to see the connections between the proof we present here and Sperner's lemma. Moreover, we believe that the topological approach to distributed computing must be studied and extended, since it has been extensively and successfully used in the past for proving a variety of results (see, for example, [5,8,9,12,14,16]), and it is unlikely that all these results have non-topological proofs.

The paper is organized as follows. Section 2 describes the asynchronous wait-free read/write shared memory model. Section 3 presents the subset of IS executions, which is used for proving the impossibility of k-set agreement in Section 4. Finally, Section 5 provides an operational perspective of this proof.

2 Model of Computation

This section describes the standard asynchronous wait-free read/write shared memory model, considered in this paper, following [1,2].

System and executions. A *system* consists of n asynchronous sequential processes $p_1, \ldots, p_n$. Each process is a deterministic state machine with a (possibly infinite) set of *local states* S and two subsets of S called *initial states* and *output states*, respectively. The processes communicate by using a shared memory with a finite number of *single-writer multi-reader atomic registers*. No assumption is made regarding the size of the registers, thus we can assume that process p_i has a single register r_i to which it can write its entire state. Process p_i has two atomic operations available to it: $write_i(v)$ that writes the value v into r_i, and $read_i(j)$ that returns the current value in r_j. A *step* is performed by a single process p_i, which executes one of its two available operations, $read_i$ or $write_i$, performs some local computation and then changes its local state.

A *configuration* of the system consists of the local states of the processes and the content of the registers. An *initial configuration* is a configuration in which all local states are initial states and all registers are set to a distinguished value $\perp$. An *output configuration* is a configuration in which all local states are output states.

An *execution* of the system is a, possibly infinite, alternating sequence of configurations and steps $\alpha = C_0, s_0, C_1, s_1, C_2, \ldots$, where C_0 is an initial configuration and $C_{\ell+1}$ is the result of applying the step s_ℓ to C_ℓ, for $\ell \geq 0$. The *view* of a process p_i in α, denoted $\alpha|p_i$, is the sequence of p_i's local states in configurations $C_0, C_1, \ldots$. The *participating set* of an execution α, denoted $\text{ps}(\alpha)$, is the set of processes that take at least one step in the execution.

Two executions α and α' are *indistinguishable* for a set of processes P, denoted $\alpha \overset{P}{\sim} \alpha'$, if all processes in P have the same view in both executions, namely, $\forall p_i \in P, \alpha|p_i = \alpha'|p_i$. In the following section we are interested in pairs of executions α and α' that are distinguishable to exactly one process, that is, there is a process p_i such that $\alpha|p_i \neq \alpha'|p_i$ and $\alpha \overset{P}{\sim} \alpha'$, where $P = \{p_1, \ldots, p_n\} \setminus \{p_i\}$. In this case, we write $\alpha \overset{\neg p_i}{\sim} \alpha'$.

Algorithms. The state machine of a process p_i models a *local algorithm* A_i that determines p_i's next step. An *algorithm* A is a collection of local algorithms $A_1, \ldots, A_n$.

Each process has two distinguished components, *input* and *output*, that allow the system to model decision tasks. The input component never changes and cannot contain the distinguished value $\perp$. The output component contains initially $\perp$, and once a process reaches a local state in which a non-$\perp$ value is written in it, the output component never changes. When that happens, we say that the process *decides*. The output states are the states with non-$\perp$ output values. If a process decides v in an execution α, we say v is *decided* in α.

A view of a process p_i in a finite execution α is *final*, if p_i decides in α. A final view of p_i in α is *minimal* if none of its prefixes is final. In other words,

the minimal final view of p_i in α is the prefix of the view of p_i up to the first configuration in which p_i decides. A finite execution α is *minimal final* if the view of each process in $\text{ps}(\alpha)$ is a minimal final; in particular, each participating process decides in α. For a minimal final execution α, $\text{dec}(\alpha)$ is the set of all the values that are decided in α, and for a process $p_i \in \text{ps}(\alpha)$, $\text{dec}(\alpha, \neg p_i)$ is the set $\text{dec}(\alpha) \setminus \{v\}$, where v is the value that p_i decides in α. Note that if two distinct processes p_i and p_j decide on the same value v, then $\text{dec}(\alpha, \neg p_i) = \text{dec}(\alpha, \neg p_j)$.

An algorithm A is *wait-free* if in each execution of A, every process executes a finite number of steps or decides. Therefore, a process must decide if it executes an infinite number of steps. We only consider wait-free algorithms.

An algorithm is *full-information* if a process writes its entire local state in every write operation. We say that an algorithm is in *standard-form* if processes proceed in a sequence of asynchronous rounds. In a round, every process first executes a write operation and then asynchronously reads all registers. Note that if there is a wait-free algorithm solving some task, then there is a, possibly inefficient, full-information and standard-form wait-free algorithm solving this task. Since efficiency is not an issue in this paper, we only consider full-information and standard-form algorithms.

k-set agreement. In *k-set agreement* [6], $1 \leq k \leq n$, each process p_i proposes a value has to decide on a value, such that the following properties hold.

Termination: Each process executes a finite number of steps or decides.
Validity: A decided value is a proposed value.
k-Agreement: At most k different values are decided.

We now define a trivial task $\mathcal{T}$ that will be used in the impossibility proof of k-set agreement. Indeed, in Section 4 we shall see that every wait-free algorithm that solves $\mathcal{T}$ possesses a property which implies that there is no algorithm that solves k-set agreement for $k < n$.

In the task $\mathcal{T}$ each process p_i proposes a value and each process has to decide a value such that the termination and validity properties of k-set agreement are satisfied. Obviously, $\mathcal{T}$ is wait-free solvable: Each process can just decide it proposal, or each process can decide the smallest proposed value it sees in the shared memory. The following lemma is immediate from the definition of k-set agreement and $\mathcal{T}$.

Lemma 1. *Any wait-free algorithm that solves k-set agreement for some k, $1 \leq k \leq n$, also solves $\mathcal{T}$.*

3 Immediate Snapshot Executions

Consider a full-information and standard-form wait-free algorithm. The immediate snapshot executions of this algorithm form a subset of all its possible executions. The executions in this set have a structure that makes them easier to analyze.

An *immediate snapshot* (IS) execution [1,3,4,18] is modeled by a sequence of non-empty sets of processes $\alpha = s_1, \ldots, s_l, \ldots$. Processes in s_l first perform, one by one, a write operation and then read all registers. Intuitively, the processes execute a concurrent write followed by a concurrent atomic snapshot of the shared memory. If $p_i \in s_l$, then we say that p_i is *active* in the l-th set of α.

Since we only consider wait-free algorithms, we can restrict our attention to minimal final IS executions, namely, each process executes computation steps until it decides. Indeed, each process decides in the last round in which it is active. We sometimes write α as a concatenation of sequences of sets, i.e., $\alpha = \alpha_1\ \alpha_2$, where $\alpha_1 = s_1\ s_2\ \ldots\ s_\ell$ and $\alpha_2 = s_{\ell+1}\ s_{\ell+2}\ \ldots\ s_t$.

For example, $\alpha = \{p, q\}\ \{r\}$ denotes an IS execution made of 2 sets. Processes p and q are active in the first set and r is active in the second one. Observe that p and q see each other, but do not see r because it executes steps of computation after p and q read the whole memory.

Although IS execution are well-structured they still have uncertainty, as the following example shows. In addition to the execution α described in the previous paragraph, consider the IS execution $\alpha' = \{p\}\ \{q\}\ \{r\}$. Note that p only sees itself in α' while sees itself and q in α. The reader can verify that the views of q and r are the same in α and α'. Therefore, $\alpha \overset{\neg p}{\sim} \alpha'$.

For a sequence $\alpha = s_1\ s_2\ \ldots\ s_t$ of sets of processes and a process p_i, we write $p_i \notin \alpha$ if $p_i \notin s_\ell$ for every ℓ, $1 \leq \ell \leq t$, . Alternatively, we say that p_i *does not appear* in α. Also, for p_i and $r \geq 1$, we let $\{p_i\}^r$ denote the sequence containing r times the set $\{p_i\}$.

Formally, we say that a process p_i is *unseen* in an IS execution $\alpha = s_1\ s_2\ \ldots\ s_t$ if and only if there exists ℓ, $1 \leq \ell \leq t-1$, such that $p_i \notin s_x$, $1 \leq x \leq \ell$ and $s_y = \{p_i\}$, $\ell < y \leq t$. Therefore, if p_i is unseen in α, then $\alpha = \alpha'\,\{p_i\}^r$, for some $r \geq 1$ and α', such that $p_i \notin \alpha'$. Intuitively, p_i is unseen in α since every step of p_i occurs after all other processes decided. A process p_i is *seen* in an IS execution α if it is not unseen in α.

Lemmas 2 and 3 below are from [1]. They capture two properties about the uncertainty in minimal final IS executions. They will be used in the impossibility proof of k-set agreement presented in the following section. Lemma 2 is Lemma 3.3 in [1] and Lemma 3 is a restatement of Lemma 3.4 in [1].

Lemma 2. *If a process p_i is unseen in a minimal final IS execution α, then there is no minimal final IS execution α' such that $\alpha \overset{\neg p_i}{\sim} \alpha'$.*

Lemma 3. *Let α be a minimal final IS execution in which a process p_i is seen. Then there is a unique minimal final IS execution α' such that $\alpha \overset{\neg p_i}{\sim} \alpha'$.*

Let A be a wait-free algorithm. For any given collection C of inputs to the processes, let S be the set containing all minimal final IS executions of A in which the processes start with inputs C. For a subset P of processes, let $PS_P(S)$ be the set containing all executions in S with participating set P. If there is no ambiguity, we write PS_P instead of $PS_P(S)$.

For a proper subset of processes P and a process $p \notin P$, the next lemma shows that there is a one-to-one correspondence between the executions in PS_P and the executions in $PS_{P\cup\{p\}}$ in which p is unseen.

Lemma 4. *For every proper subset of processes P and a process $p_i \notin P$, the following properties hold:*

1. *For every execution $\alpha \in PS_P$ there is a unique execution $\alpha' \in PS_{P\cup\{p_i\}}$ such that p_i is unseen in α' and α is equal to the maximal prefix of α' in which p_i does not appear.*
2. *For every execution $\alpha \in PS_{P\cup\{p_i\}}$ such that p_i is unseen in α, PS_P contains the maximal prefix α' of α in which p_i does not appear.*

Proof. First let us consider an execution $\alpha \in PS_P$. Since A is asynchronous wait-free, α can be extended to a minimal final execution α' with $\mathrm{ps}(\alpha') = P \cup \{p_i\}$, namely, $\alpha' = \alpha \{p_i\}^r$, for some $r \geq 1$. Thus $\alpha' \in PS_{P\cup\{p_i\}}$. Note that p_i is unseen in α' because by hypothesis $p_i \notin \alpha$. In addition, α is the maximal prefix of α' in which p_i does not appear. Moreover, α' is unique because A is deterministic.

Now let us consider an execution $\alpha \in PS_{P\cup\{p_i\}}$ such that p_i is unseen in α. We have that $\alpha = \alpha' \{p_i\}^r$, for some $r \geq 1$ and $p_i \notin \alpha'$. Note that α' is the maximal prefix of α in which p_i does not appear. Moreover, $\alpha' \in PS_P$ because, by hypothesis, each process that appears in α' executes steps of computation until it decides, namely, it is a minimal final IS execution with $\mathrm{ps}(\alpha') = P$. □

4 Impossibility of k-Set Agreement

This section is devoted to proving the impossibility of the k-set agreement task in the asynchronous wait-free read/write shared memory model (Theorem 1). The core of the proof is Lemma 6, roughly showing that every wait-free algorithm solving the task $\mathcal{T}$ (defined in Section 2) maintains an invariant concerning the number of its executions in which ℓ distinct processes decide ℓ distinct values. Indeed this lemma can be regarded as the operational counterpart of the Sperner's lemma [10, p. 36]. Then, using Lemma 1 and the invariant in Lemma 6, we conclude that k-set agreement is not wait-free solvable.

Let A be a wait-free algorithm that solves $\mathcal{T}$. Consider n distinct input values $v_1, \ldots, v_n$ and let S be the set containing all minimal final IS executions of A in which process p_i has input v_i, $1 \leq i \leq n$.

Recall that PS_P is the set of all executions in S with participating set P. The next simple lemma directly follows from the validity property of $\mathcal{T}$, and it is used in the proof of Lemma 6 below.

Lemma 5. *For every subset of processes P and for every execution $\alpha \in PS_P$, $\mathrm{dec}(\alpha)$ contains only the inputs of processes in P.*

When $P = \{p_1, \ldots, p_t\}$, $1 \leq t \leq n$, we write PS_t instead of PS_P.

Lemma 6. *For all $t, 1 \leq t \leq n$, PS_t contains exactly an odd number of executions $\alpha_1, \ldots, \alpha_{2q+1}$, $q \geq 0$, such that $\mathrm{dec}(\alpha_j) = \{v_1, \ldots, v_t\}$, $1 \leq j \leq 2q+1$.*

Intuitively, the proof of Lemma 6 proceeds by constructing a graph G using the executions in PS_t as vertices and putting an edge between two executions $\alpha, \alpha' \in PS_t$ if and only if at most one process distinguishes between them and at least $t-1$ distinct values are decided in α and α'. The graph G also contains an "imaginary" vertex v^*, which is defined in such a way that it has odd degree. This guarantees that G contains at least one vertex with odd degree. All other vertices of G with odd degree, if they exist, correspond to the executions of PS_t in which the values $v_1, \ldots, v_t$ are decided. Let M be the set containing all vertices of G with odd degree except v^*. Using the well known handshaking lemma, stating that every graph has an even number of vertices with odd degree,[1] the proof finally concludes $|M \cup \{v^*\}|$ is even, hence $|M|$ is odd, which proves the lemma.

Proof (of Lemma 6). We proceed by induction on t. For $t = 1$, PS_1 only contains p_1's solo execution, that is, PS_1 only contains $\alpha = \{p_1\}^r$, for some $r \geq 1$. Obviously, p_1 must decide v_1 in α, which proves the base of the induction. Now suppose that the lemma holds for $t, 1 \leq t \leq n-1$. We prove it holds for $t+1$.

We define a graph $G = (V, E)$ whose vertices are the executions in PS_{t+1}, plus an additional vertex v^*, that is, $V = PS_{t+1} \cup \{v^*\}$. The edges of G are defined as follows (recall that v_i is the input of process p_i).

- For every pair of executions $\alpha, \alpha' \in PS_{t+1}$, $(\alpha, \alpha') \in E$ if and only if there is a process p_i such that $\alpha \overset{\neg p_i}{\sim} \alpha'$ and $\mathrm{dec}(\alpha, \neg p_i) = \{v_1, \ldots, v_t\}$; hence, $\mathrm{dec}(\alpha', \neg p_i) = \{v_1, \ldots, v_t\}$.
- For every execution $\alpha \in PS_{t+1}$, $(v^*, \alpha) \in E$ if and only if p_{t+1} is unseen in α and $\mathrm{dec}(\alpha, \neg p_{t+1}) = \{v_1, \ldots, v_t\}$.

Fig. 1 depicts an example of the graph G constructed in the inductive step $t = 2$ for a one-round algorithm for three processes p, q and r. In the algorithm, a process decides its proposal if it sees less than 2 processes. Otherwise, if it sees that its proposal is not the largest one, then decides the smallest proposal, and it decides the second smallest proposal in any other case. In Fig. 1 the inputs for p, q and r are 1, 2 and 3, respectively, and process r corresponds to p_{t+1}. In each execution, above each process it appears the value that the process decides in that execution. Observe that there is an edge between executions $\alpha = \{p\}\,\{q, r\}$ and $\alpha' = \{p, q, r\}$ because, first, $\alpha \overset{\neg p}{\sim} \alpha'$ (p sees no other process than itself in α while it sees q and r in α'), and second, q and r decide 1 and 2, respectively, in α. Also there is an edge between v^* and $\alpha = \{p, q\}\,\{r\}$ because r is unseen in α and p and q decide 1 and 2. Finally, note that there is no edge between $\alpha = \{p, q, r\}$ and $\alpha' = \{r\}\,\{p, q\}$ because although $\alpha \overset{\neg r}{\sim} \alpha'$, both p and q decide 1 in α.

For the rest of the proof, let $\deg(v)$ denote the degree of a vertex v of G. Below, we prove the following properties about the degrees of the vertices in G.

1. $\deg(v^*)$ is odd.

[1] This result is a consequence of Euler's observation that for every graph $G = (V, E)$, $\sum_{v \in V} \deg(v) = 2\,|E|$.

2. For all $\alpha \in PS_{t+1} = V \setminus \{v^*\}$:
   (a) If $\mathrm{dec}(\alpha) = \{v_1, \ldots, v_{t+1}\}$, then $\deg(\alpha) = 1$.
   (b) If $\mathrm{dec}(\alpha) = \{v_1, \ldots, v_t\}$, then $\deg(\alpha) = 2$.
   (c) Otherwise, $\deg(\alpha) = 0$.

These properties, can be use to derive the induction step, together with the next well-known result of graph theory.

Handshaking Lemma. *Every graph has an even number of vertices with odd degree.*

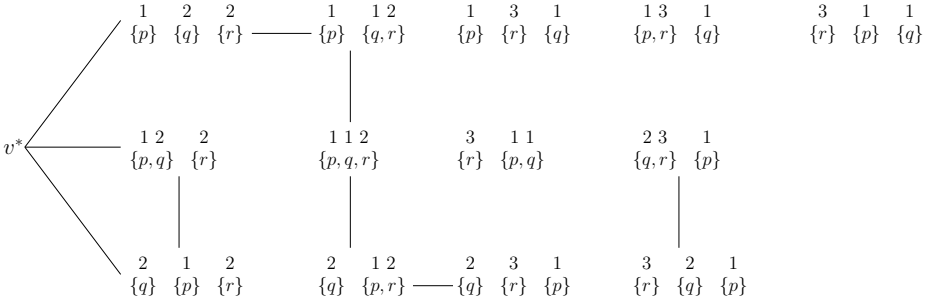


Fig. 1. Graph associated with a 3-process 1-round algorithm

The vertices of G with odd degree are exactly $M \cup \{v^*\}$, where M is the set that contains every execution $\alpha \in PS_{t+1}$ such that $\mathrm{dec}(\alpha) = \{v_1, \ldots, v_{t+1}\}$. Thus, by the Handshaking Lemma, $|M \cup \{v^*\}|$ is even, hence $|M|$ is odd. We now prove the properties of the degrees.

$\deg(v^*)$ is odd. Consider an execution $\alpha \in PS_t$. By Lemma 4(1), there is a unique execution $\alpha' \in PS_{t+1}$ such that p_{t+1} is unseen in α' and α is equal to the maximal prefix of α' in which p_{t+1} does not appear. Conversely, by Lemma 4(2), for every execution $\alpha \in PS_{t+1}$ such that p_{t+1} is unseen in α, PS_t contains the maximal prefix α' of α in which p_{t+1} does not appear.

By the induction hypothesis, we have that PS_t contains exactly an odd number of executions $\alpha_1, \ldots, \alpha_{2q+1}$, $q \geq 0$, such that $\mathrm{dec}(\alpha_j) = \{v_1, \ldots, v_t\}$, $1 \leq j \leq 2q+1$. Thus for every α_j, $1 \leq j \leq 2q+1$, there is a unique execution $\alpha'_j \in PS_{t+1}$ such that p_{t+1} is unseen in α'_j and α_j is a prefix of α'_j. Hence, $(v^*, \alpha'_j) \in E$, since $\mathrm{dec}(\alpha'_j, \neg p_{t+1}) = \{v_1, \ldots, v_t\}$ because $\mathrm{dec}(\alpha_j) = \{v_1, \ldots, v_t\}$. Moreover, these are all the edges adjacent to v^* because, as explained above, there is a one-to-one correspondence between the executions of PS_t and the executions of PS_{t+1} in which p_{t+1} is unseen. Therefore, $\deg(v^*)$ is odd.

For every execution $\alpha \in PS_{t+1}$, if $\mathrm{dec}(\alpha) = \{v_1, \ldots, v_{t+1}\}$, then $\deg(\alpha) = 1$
Since $|\mathrm{dec}(\alpha)| = |\{v_1, \ldots, v_{t+1}\}| = t+1$ and $|\mathrm{ps}(\alpha)| = |\{p_1, \ldots, p_{t+1}\}| = t+1$, it follows that for every $v \in \{v_1, \ldots, v_{t+1}\}$, there is exactly one process in α that

decides v. Let p denote the process that decides v_{t+1} in α. Thus, $\mathrm{dec}(\alpha, \neg p) = \{v_1, \ldots, v_t\}$. We identify two subcases: p is unseen in α or p is seen in α.

If p is unseen in α, then, by Lemma 4(2), PS_P contains the maximal prefix α' of α in which p does not appear, where $P = \{p_1, \ldots, p_{t+1}\} \setminus \{p\}$. Observe that $\mathrm{dec}(\alpha') = \{v_1, \ldots, v_t\}$. Lemma 5 and the assumption that each p_i has a distinct input v_i, imply that $P = \{p_1, \ldots, p_{t+1}\} \setminus \{p\} = \{p_1, \ldots, p_t\}$, hence $p = p_{t+1}$. Therefore, p_{t+1} is unseen in α and $\mathrm{dec}(\alpha, \neg p_{t+1}) = \{v_1, \ldots, v_t\}$, because as explained above $\mathrm{dec}(\alpha, \neg p) = \{v_1, \ldots, v_t\}$. By definition of G, $(v^*, \alpha) \in E$.

We claim that (v^*, α) is the only edge that is adjacent to α. First, there is no execution $\alpha' \in PS_{t+1}$ such that $\alpha \overset{\neg q}{\sim} \alpha'$ with $q = p_{t+1}$, by Lemma 2. Second, for every process $q \in \{p_1, \ldots, p_t\}$, $\mathrm{dec}(\alpha, \neg q) \neq \{v_1, \ldots, v_t\}$ because for each $v \in \{v_1, \ldots, v_{t+1}\}$ there is exactly one process in α that decides v, and q decides on a value in $\{v_1, \ldots, v_t\}$. These two observations imply that there is no $\alpha' \in PS_{t+1}$ such that $(\alpha, \alpha') \in E$, and hence $\deg(\alpha) = 1$.

For the second subcase, namely, p is seen in α, there is only one execution $\alpha' \in PS_{t+1}$ such that $\alpha \overset{\neg q}{\sim} \alpha'$, by Lemma 3. Then $(\alpha, \alpha') \in E$, because $\mathrm{dec}(\alpha, \neg p) = \{v_1, \ldots, v_t\}$.

The edge (α, α') is the only edge that is adjacent to α: We have that for every $q \in \{p_1, \ldots, p_t\}$, $\mathrm{dec}(\alpha, \neg q) \neq \{v_1, \ldots, v_t\}$ because for each $v \in \{v_1, \ldots, v_{t+1}\}$ there is exactly one process in α that decides v, and q decides a value of $\{v_1, \ldots, v_t\}$. Therefore, there does not exist a $\alpha'' \in PS_{t+1}$ such that $\alpha'' \neq \alpha'$ and $(\alpha, \alpha'') \in E$. Hence we get $\deg(\alpha) = 1$.

For every $\alpha \in PS_{t+1}$, *if* $\mathrm{dec}(\alpha) = \{v_1, \ldots, v_t\}$, *then* $\deg(\alpha) = 2$. Since $|\mathrm{dec}(\alpha)| = |\{v_1, \ldots, v_t\}| = t$ and $|\mathrm{ps}(\alpha)| = |\{p_1, \ldots, p_{t+1}\}| = t + 1$, we get that there must be a value $\bar{v} \in \{v_1, \ldots, v_t\}$ such that there are two distinct processes $q_1, q_2 \in \{p_1, \ldots, p_{t+1}\}$ that decide $\bar{v}$ in α. Therefore, for each $i \in \{1, 2\}$, $\mathrm{dec}(\alpha, \neg q_i) = \{v_1, \ldots, v_t\}$. Also observe that $\bar{v}$ is the unique value of $\{v_1, \ldots, v_t\}$ that has that property.

We identify three subcases: q_1 is unseen and q_2 is seen in α, q_1 is seen and q_2 is unseen in α, and both q_1 and q_2 are seen in α. Note that it is impossible that both q_1 and q_2 are unseen.

Consider first the subcase q_1 is unseen and q_2 is seen in α. The argument is similar to the one in the previous case. If q_1 is unseen in α, then, by Lemma 4(2), PS_P contains the maximal prefix α' of α in which q_1 does not appear, where $P = \{p_1, \ldots, p_{t+1}\} \setminus \{q_1\}$. Observe that $\mathrm{dec}(\alpha') = \{v_1, \ldots, v_t\}$. Thus, by Lemma 5 and the assumption that each p_i has a distinct input v_i, we get $P = \{p_1, \ldots, p_{t+1}\} \setminus \{q_1\} = \{p_1, \ldots, p_t\}$, and hence $q_1 = p_{t+1}$. Therefore, p_{t+1} is unseen in α and $\mathrm{dec}(\alpha, \neg p_{t+1}) = \{v_1, \ldots, v_t\}$, because $\mathrm{dec}(\alpha, \neg q_1) = \{v_1, \ldots, v_t\}$. Hence $(v^*, \alpha) \in E$.

For q_2, by Lemma 3, there is an execution $\alpha' \in PS_{t+1}$ such that $\alpha \overset{\neg q_2}{\sim} \alpha'$. Then $(\alpha, \alpha') \in E$, because $\mathrm{dec}(\alpha, \neg q_2) = \{v_1, \ldots, v_t\}$.

We claim that (v^*, α) and (α, α') are the unique edges adjacent to α. First, for q_1, there is no execution $\alpha'' \in PS_{t+1}$ such that $\alpha \overset{\neg q_1}{\sim} \alpha''$, by Lemma 2. Second, for q_2, there is no $\alpha'' \neq \alpha'$ such that $\alpha \overset{\neg q_2}{\sim} \alpha''$, by Lemma 3. And third, for every

process $q \in \{p_1, \ldots, p_{t+1}\}$ distinct from q_1 and q_2, $\text{dec}(\alpha, \neg q) \neq \{v_1, \ldots, v_t\}$ because q decides on a value in $\{v_1, \ldots, v_t\}$ and, as mentioned above, there is no other process in $\{p_1, \ldots, p_{t+1}\}$ that decides the same value as q. Therefore, $\deg(\alpha) = 2$.

The subcase in which q_1 is seen and q_2 is unseen in α is symmetric to the previous one.

For the third subcase, that is, q_1 and q_2 are seen in α, for each $i \in \{1, 2\}$ there is an execution $\alpha'_i \in PS_{t+1}$ such that $\alpha \overset{\neg q_i}{\sim} \alpha'_i$, by Lemma 3. Then $(\alpha, \alpha'_i) \in E$, because $\text{dec}(\alpha, \neg q_i) = \{v_1, \ldots, v_t\}$. As in the first subcase, it can be easily proved that there is no extra edge that is adjacent to α. Hence $\deg(\alpha) = 2$.

Otherwise, $\deg(\alpha) = 0$. We have two subcases: $|\text{dec}(\alpha)| < t$, or $|\text{dec}(\alpha)| = t$ and $\text{dec}(\alpha) \neq \{v_1, \ldots, v_t\}$. In both subcases, for every process $p \in \{p_1, \ldots, p_{t+1}\}$, $\text{dec}(\alpha, \neg p) \neq \{v_1, \ldots, v_t\}$, hence it follows from the definition of G that $\deg(\alpha) = 0$. □

Theorem 1. *There is no wait-free algorithm that solves k-set agreement for $k < n$.*

Proof. Suppose that there is a wait-free algorithm A that solves k-set agreement, for some k, $1 \leq k \leq n-1$. By Lemma 1, A solves $\mathcal{T}$. Consider n distinct input values $v_1, \ldots, v_n$ and let S be the set containing all minimal final IS executions of A in which process p_i has input v_i, $1 \leq i \leq n$. Consider the set $PS_{k+1}(S)$, namely, the set containing all executions in S with participating set $p_1, \ldots, p_{k+1}$. By Lemma 6, $PS_{k+1}(S)$ contains an execution α such that $|\text{dec}(\alpha)| = k+1$. But this contradicts the fact that A solves k-set agreement, since the k-agreement property means that in all executions $\alpha \in PS_{k+1}(S)$, $|\text{dec}(\alpha)| \leq k$. □

5 An Operational Perspective

From an operational perspective we can think of the proof of Lemma 6, and hence the impossibility proof of k-set agreement, in the following way.

The induction hypothesis of the proof states that there is an odd number of minimal final IS executions $\alpha_1, \ldots, \alpha_{2q+1}$, $q \geq 0$, such that for each $\alpha \in S = \{\alpha_1, \ldots, \alpha_{2q+1}\}$, $\text{ps}(\alpha) = \{p_1, \ldots, p_k\}$ and $\text{dec}(\alpha) = D = \{v_1, \ldots, v_k\}$, where v_i is the input of p_i, $1 \leq i \leq k$. Namely, k processes decide k distinct values in α. As explained in the proof, each $\alpha \in E$ can be extended to a minimal final IS execution γ_1 such that $\text{ps}(\gamma_1) = \{p_1, \ldots, p_{k+1}\}$ and p_{k+1} is unseen in γ_1. Thus, α is the maximal prefix of γ_1 in which p_{k+1} does not appear, hence p_{k+1} decides after all processes $p_1, \ldots, p_k$ decide in γ_1. Moreover, $\text{dec}(\gamma_1, \neg p_{k+1}) = D$ because $\text{dec}(\alpha) = D$. Let us fix α and γ_1.

Then, taking advantage of the uncertainty related to IS execution, what the proof essentially does is to produce a path P in the graph G that starts in γ_1. If $|\text{dec}(\gamma_1)| = k+1$, then P only contains γ_1. Otherwise, $|\text{dec}(\gamma_1)| = |D| = k$ and the proof looks for an execution γ_2 such that there is a process p such that $\gamma_1 \overset{\neg p}{\sim} \gamma_2$, and hence $\text{dec}(\gamma_2, \neg p) = D$. If there is no such execution γ_2 then P

only contains γ_1, otherwise γ_2 is appended to P and the procedure continues by considering γ_2 instead of γ_1.

In the end we get a path $P = \gamma_1, \gamma_2, \ldots, \gamma_s$, such that $s \geq 1$ and for each i, $1 \leq i \leq s-1$, there is a process p such that $\gamma_i \overset{\neg p}{\sim} \gamma_{i+1}$ and $\mathrm{dec}(\gamma_i, \neg p) = D$. What is important to notice is that the last execution (vertex) γ_s of P holds either $|\mathrm{dec}(\gamma_s)| = k+1$, or $|\mathrm{dec}(\gamma_s)| = |D| = k$ and p_{k+1} is unseen in γ_s. In the second case we have that the maximal prefix α' of γ_s in which p_{k+1} does not appear, belongs to S. Therefore, in some sense, the path P "matches" α (the maximal prefix of γ_1 in which p_{k+1} does not appear) to α'. For example, in the graph in Fig. 1, the sequence of IS executions $\{p\ q\}\,\{r\}$, $\{q\}\,\{p\}\,\{r\}$ matches $\{p\ q\}$ to $\{q\}\,\{p\}$. However, only an even number of execution of S can be matched in this way, and since $|S|$ is odd, we get that there must be a path that matches an $\alpha \in S$ to an execution in which $k+1$ values are decided. In Fig. 1, the path that starts at $\{p\}\,\{q\}\,\{r\}$ and ends at $\{q\}\,\{r\}\,\{p\}$, matches $\{p\}\,\{q\}$ to $\{q\}\,\{r\}\,\{p\}$.

Finally, an execution γ with $|\mathrm{dec}(\gamma)| = k+1$, that is not matched to an execution in S, is matched to an execution γ' with $|\mathrm{dec}(\gamma')| = k+1$. In Fig. 1, $\{q\ r\}\,\{p\}$ is matched to $\{r\}\,\{q\}\,\{p\}$.

Acknowledgments. The second author would like to thank Michel Raynal and Julien Stainer for useful discussions and comments on earlier versions of this paper.

References

1. Attiya, H., Rajsbaum, S.: The Combinatorial Structure of Wait-Free Solvable Tasks. SIAM Journal on Computing 31(4), 1286–1313 (2002)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. McGraw-Hill, New York (1998)
3. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on Theory of Computing (STOC 1993), pp. 91–100. ACM Press, New York (1993)
4. 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)
5. Castañeda, 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)
6. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105(1), 132–158 (1993)
7. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
8. Gafni, E., Koutsoupias, E.: Three-Processor Tasks Are Undecidable. SIAM Journal on Computing 28(3), 970–983 (1999)
9. Gafni, E., Rajsbaum, S., Herlihy, M.P.: Subconsensus Tasks: Renaming is Weaker Than Set Agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 329–338. Springer, Heidelberg (2006)
10. Henle, M.: A Combinatorial Introduction to Topology. Dover, New York (1994)

11. Herlihy, M.: Wait-free synchronization. Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
12. Herlihy, M.P., Rajsbaum, S.: Set Consensus Using Arbitrary Objects (Preliminary Version). In: Proc. 13th Annual ACM Symposium on Principles on Distributed Computing (PODC 1994), pp. 324–333. ACM Press, New York (1994)
13. Herlihy, M.P., Rajsbaum, S.: Algebraic Spans. Mathematical Structures in Computer Science 10(4), 549–573 (2000)
14. Herlihy, M.P., Rajsbaum, S.: A Classification of Wait-free Loop Agreement Tasks. Theoretical Computer Science 291(1), 55–77 (2003)
15. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
16. Hoest, G., Shavit, N.: Toward a Topological Characterization of Asynchronous Complexity. SIAM Journal on Computing 36(2), 457–497 (2006)
17. Loui, M., Abu-Amara, 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)
18. 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)

Formal Verification of Consensus Algorithms Tolerating Malicious Faults

Bernadette Charron-Bost[1], Henri Debrat[2], and Stephan Merz[2]

[1] CNRS & LIX, Palaiseau, France
charron@lix.polytechnique.fr
[2] INRIA Nancy & LORIA, Nancy, France
{Henri.Debrat,Stephan.Merz}@loria.fr

Abstract. Consensus is the paradigmatic problem in fault-tolerant distributed computing: it requires network nodes that communicate by message passing to agree on common value even in the presence of (benign or malicious) faults. Several algorithms for solving Consensus exist, but few of them have been rigorously verified, much less so formally. The Heard-Of model proposes a simple, unifying framework for defining distributed algorithms in the presence of communication faults. Algorithms proceed in communication-closed rounds, and assumptions on the faults tolerated by the algorithm are stated abstractly in the form of communication predicates. Extending previous work on the case of benign faults, our approach relies on the fact that properties such as Consensus can be verified over a coarse-grained, round-based representation of executions. We have encoded the Heard-Of model in the interactive proof assistant Isabelle/HOL and have used this encoding to formally verify three Consensus algorithms based on synchronous and asynchronous assumptions. Our proofs give some new insights into the correctness of the algorithms, in particular with respect to transient faults.

1 Introduction

Fault-tolerant distributed computing is the art of making separate computing nodes cooperate for achieving a common objective, even in the presence of faults. In particular, the Consensus problem assumes that every node initially proposes some value, and requires that nodes eventually choose a common value among the proposed ones. Fault-tolerant distributed algorithms are often subtle, both in their operational design and in the assumptions they make, including the underlying model of communication and the kinds and numbers of faults they tolerate. *Benign* faults prevent processes from receiving expected messages, but do not affect the contents of messages received; these may be caused e.g. by process crashes or link breaks. *Malicious* faults are more severe as they are aimed to model any type of (process and link) malfunctioning, including corrupted process states and corrupted messages. It is well known [8] that Consensus is unsolvable in a fully asynchronous model of communication if at least one node may fail by crashing, but that it can be solved in partially synchronous models [6] even in the presence of malicious faults.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 120–134, 2011.

Given the subtle differences between communication and fault models, it is all too easy to make erroneous claims about what algorithms actually achieve, and formal statements and proofs about algorithms appear crucial for comparing them. Surprisingly, few rigorous correctness proofs exist in the literature, and even fewer of them have been fully checked with the help of formal verification methods and tools [18,13]. We believe that this lack of formal analysis is largely due to the absence of widely accepted frameworks in which models of computation and faults can be expressed and compared.

Charron-Bost and Schiper [5] proposed the Heard-Of (HO) model as a simple, unifying framework for defining distributed algorithms that operate in the presence of benign communication faults. In this model, computations are structured in rounds: during every round, each process first sends messages, then receives messages from other processes, and finally makes a local state transition. Rounds are communication-closed layers in the sense that processes receive messages solely sent at the round they currently execute. The round-based structure of the HO model greatly facilitates the design and understanding of distributed algorithms: an algorithm is simply specified by defining a message sending function and a next-state function, for every process and round. This operational description is then complemented by imposing communication predicates on executions, which restrict the kinds of faults that the algorithm tolerates. It has been shown [5] that common communication and fault models can be represented within the HO model. In previous work [3,4] we have proved that important correctness properties such as Consensus can be verified over a "coarse-grained" model of executions in which rounds are executed atomically, and have applied this result for formally verifying algorithms in the HO model, using model checking and interactive theorem proving. Independently, Tsuchiya and Schiper [19,20] also proposed the use of symbolic model checking for verifying HO algorithms.

In the meantime, the HO model has been augmented for supporting malicious communication faults [2], and it is this extension that we refer to when we speak of the HO model in the following. Extending [4], we show in this paper that the reduction theorem that allows us to consider only round-based executions remains valid in the presence of malicious faults. We present an encoding of the HO model in the interactive theorem prover Isabelle/HOL [16]. We have used this encoding to formally prove the correctness of three algorithms for achieving Consensus in the presence of malicious faults: the $\mathcal{U}_{T,E,\alpha}$ and $\mathcal{A}_{T,E,\alpha}$ algorithms from [2], and the well-known *Exponential Information Gathering* ($EIGByz_f$) algorithm [1,15]. The overall approach to verification is quite similar for all three algorithms. In particular, the $EIGByz_f$ algorithm could be transposed "as is" in the HO model, although it was originally introduced in a traditional model that caters for faults of processes. We show that $EIGByz_f$ tolerates certain transient faults, which could not be expressed by the original model. The precise, yet abstract representation of hypotheses on allowed faults in the HO model lets us not only express such faults but also analyze precisely to which property each hypothesis contributes. The full Isabelle theories are available on the Web[1].

[1] http://www.loria.fr/~debrat/

The paper is structured as follows: Section 2 reviews the HO model, formally defines fine-grained and coarse-grained executions, states the fundamental reduction theorem, and formally defines the Consensus problem. Our encoding of the HO model in Isabelle is described in Section 3. Its application to the verification of the three algorithms we consider appears in Section 4. Section 5 discusses related work and concludes the article.

2 The Heard-of Model for Distributed Algorithms

Computations in the HO model are composed of rounds, in which processes exchange messages, take a step, and then proceed to the next round. In the parlance of Elrad and Francez [7], each round is a communication-closed layer in the sense that any message sent in a round can be received only in that round. Communication faults are abstractly represented in the HO model by means of *heard-of sets* (HO) that indicate which links are alive, thus capturing message omissions, and *safe heard-of sets* (SHO) that indicate which links are safe, thus capturing message corruption.

2.1 A Round-Based Computational Model

We suppose that we have a finite, nonempty set Π of process identifiers (or simply *processes*) and a set of messages M. By including a designated *empty message* in M that processes use to indicate absence of useful information, we may assume that each process sends a message to every process in Π, in each round. We denote the cardinality of Π by $N > 0$, let $\bot \notin M$ be a placeholder indicating that no message has been received, and write $M_\bot = M \cup \{\bot\}$. To each p in Π, we associate a *process specification* $Proc_p = (States_p, Init_p, S_p, T_p)$ whose components are the following:

- $States_p$ is a set of p's *states*, and $Init_p \subseteq States_p$ is a nonempty subset of *initial states* of process p,
- for each integer $r \in \mathbb{N}$, a *message-sending function* $S_p^r : States_p \times \Pi \to M$;
- for each integer $r \in \mathbb{N}$, a *next-state function* $T_p^r : States_p \times M_\bot^\Pi \to States_p$.

The next-state function T_p^r takes as its arguments the current state of process p and a mapping from (sender) processes to messages or $\bot$, and returns the next state of p.[2] In particular, the HO model is built on the assumption of *point-to-point* communications, and the next-state function definition is such that in its second argument, received messages are indexed by Π. The collection of process specifications $Proc_p$ is called an *algorithm* on Π.

As an example of an HO algorithm, a specification of the $\mathcal{U}_{T,E,\alpha}$ algorithm introduced in [2], appears as Algorithm 1. We consider a nonempty set V, with

[2] For notational simplicity, we assume here that T_p^r is a function, but all results carry over to the case of next-state relations (i.e., non-deterministic processes) [3], which are accommodated by our Isabelle representation.

Algorithm 1. The $\mathcal{U}_{T,E,\alpha}$ algorithm

1: **Initialization:**
2: $x_p \in V$; initially $x_p = v_p$ { v_p *is the initial value of* p }
3: $vote_p \in V \cup \{?\}$; initially $vote_p = ?$
4: $decide_p \in V \cup \{null\}$; initially $decide_p = null$

5: **Round** $r = 2\phi$
6: S_p^r : send $\langle\, x_p \,\rangle$ to all processes
7: T_p^r : **if** received $> T$ values equal to v with $v \in V$ **then** $vote_p := v$

8: **Round** $r = 2\phi + 1$
9: S_p^r : send $\langle\, vote_p \,\rangle$ to all processes
10: T_p^r : **if** received $> \alpha$ messages with value $v \in V$ **then** $x_p := v$ **else** $x_p := v_0$
11: **if** received $> E$ messages with value $v \in V$ **then** $decide_p := v$
12: $vote_p := ?$

a specific element $v_0 \in V$; the two values ? and *null* are assumed not to be in V. Each process p maintains three variables x_p, $vote_p$, and $decide_p$ initialized to some value in V, ?, and *null*, respectively. At each round r, every process p sends x_p or $vote_p$ to all, depending whether r is odd or even. Then, provided that p receives sufficiently many messages with the same value in V, p updates x_p, $vote_p$, or $decide_p$. The algorithm thus involves three threshold values T, E, and α, which are basic parameters of how it executes.

2.2 Executing HO Algorithms

Each process of an HO algorithm executes an infinite sequence of rounds, which are numbered consecutively, starting with round 0. At the beginning of each round r, process p first emits messages to all processes, computed according to the message sending function S_p^r. It then waits for messages to arrive for round r before it executes a state transition according to the next-state function T_p^r, based on its current state and the messages it has just received, and starts a new round.

We define executions with respect to a given collection of initial states (one per process) and a given *receive history* $\mu : \Pi \times \mathbb{N} \times \Pi \to M_\perp$ that specifies, for each pair p, q of processes and each round $r \in \mathbb{N}$, the message $\mu(p, r, q)$ that p receives from q at round r. The initial states, and the receive history determine, for each $p \in \Pi$, the sequence of p's states. Then we define, for each $p \in \Pi$ and round $r \in \mathbb{N}$, the *heard-of set*

$$HO(p, r) = \{q \in \Pi : \mu(p, r, q) \neq \perp\},$$

and the *safe heard-of set*

$$SHO(p, r) = \{q \in \Pi : \mu(p, r, q) = S_q^r(s_q, p)\}$$

where s_q is q's state at the beginning of round r. Both sets specify the discrepancy between *what should be sent* and *what is actually received*. As for the benign case [5], we make no assumption on the reason why $\mu(p, r, q)$ may be different

from $S_q^r(s_q, p)$: it may be due to an incorrect sending by q, an incorrect reception by p, or due to the corruption by the link. Obviously, $SHO(p,r) \subseteq HO(p,r)$, and $HO(p,r) \setminus SHO(p,r)$ is the set of processes q whose messages for p in round r are corrupted.

Assumptions on the underlying system model and communication network, such as the degree of synchronism and the failure model, are formally expressed by *communication predicates* $\mathcal{P} \subseteq (\Pi \times \mathbb{N} \to 2^\Pi \times 2^\Pi)$, and the correctness of an algorithm is asserted relative to a certain communication predicate $\mathcal{P}$. Note that communication predicates may refer to the (S)HO sets at different rounds and can therefore express assumptions about transient faults. As discussed in [5], standard failure models with various degrees of synchronism can be represented in this way: the weaker the communication predicate is, the more freedom the system has to provide heard-of and safe heard-of sets, and the harder it will be to achieve coordination among processes in the corresponding failure model. As an example, the following communication predicate guarantees that no process receives more than α corrupted messages in any round, but that every process receives more than β correct messages at each round:

$$\mathcal{P}_{\alpha,\beta} :: \forall p \in \Pi, \forall r \in \mathbb{N} : |HO(p,r) \setminus SHO(p,r)| \leq \alpha \ \wedge \ |SHO(p,r)| > \beta$$

It is worth noting that, with $\alpha = 0$ and $\beta = -1$, only benign faults may occur, i.e., all received messages carry the expected content:

$$\mathcal{P}_{benign} :: \forall p \in \Pi, \forall r \in \mathbb{N} : HO(p,r) = SHO(p,r)$$

2.3 Two Models of Executions

We define two models of execution, whose relationship will be explored further: the *fine-grained model* and the *coarse-grained model*. Both are based on the notion of (global) *configuration* which is a tuple of process and channel states, one per component. The state of any component c in configuration σ is denoted $\sigma(c)$. An initial configuration is one in which the state of each process p is in $Init_p$, and the state of each channel is the empty set. The two models differ in the nature of the atomic steps which take a configuration to the next one.

Fine-grained executions. Each process p can execute three types of atomic actions that may change the state of p itself and the state of the channels incident on p: the sending of a message, the reception of a message, or an internal action. Only internal actions modify the process state, and process states at the end of round r do not depend on the order in which messages are received at round r. An *event* e consists of the execution of a single action by a process.

In the (classical) fine-grained model of execution, configuration σ' is a *successor configuration* of σ if there exists some event e that takes σ to σ'. By the definition of process specification, the pair (σ, σ') determines a unique event e, and we say that (σ, σ') *corresponds to* e.

A *fine-grained execution* of an algorithm is then defined to be an ω-sequence $\sigma_0\sigma_1\ldots$ of configurations where σ_0 is an initial configuration, σ_{i+1} is a successor

configuration of σ_i for all $i \in \mathbb{N}$, and for each $p \in \Pi$ there are infinitely many $i \in \mathbb{N}$ such that (σ_i, σ_{i+1}) corresponds to some event by p. The last condition specifies a condition of (local) progress for each process; since p can execute a local transition ending round r only if it has sent messages to all processes and has received messages from all $q \in HO(p, r)$, this condition implies the existence of sufficiently many transitions of type message sending and reception. Obviously, each fine-grained execution defines a unique receive history.

Coarse-grained executions. We now define an execution model of HO algorithms that is based on the much coarser abstraction where entire rounds are the unit of atomicity. A *coarse-grained execution* is an ω-sequence $\sigma_0\sigma_1 \ldots$ of configurations such that

- σ_0 is an initial configuration, and
- at every step, *all* processes make a transition according to their next-state function and messages they have received: there exists a receive history μ such that for all $p \in \Pi$ and all $r \in \mathbb{N}$,

$$\sigma_{r+1}(p) = T_p^r\big(\sigma_r(p), \mu_p^r\big) \quad \text{where} \quad \mu_p^r = \big(q \in \Pi \mapsto \mu(p, r, q)\big).$$

In words, the state $\sigma_{r+1}(p)$ is computed according to the next-state function T_p^r from the state $\sigma_r(p)$, and the messages that p receives at round r. A step of a coarse-grained execution thus encapsulates a move by each process. Channels are considered empty in each configuration σ_r of such a round-by-round execution since messages can be received only in the rounds for which they have been sent.

2.4 A Reduction Theorem

We now present a basic theorem, which asserts that in our model, the fine-grained and coarse-grained execution semantics are indistinguishable from the point of view of any process. Given a (fine-grained or coarse-grained) execution ρ and a process $q \in \Pi$, we define the *q-view* ρ^q of ρ for process q as the sequence of q's local states in ρ. More precisely, for a fine-grained or a coarse-grained execution $\rho = \rho_0\rho_1 \ldots$, the q-view is simply

$$\rho^q = \rho_0(q)\, \rho_1(q)\, \ldots$$

Any two executions ρ_1 and ρ_2 can be compared with respect to the views that they generate for the processes in Π. We say that two executions ρ_1 and ρ_2 are *q-equivalent* (for $q \in \Pi$) if $\rho_1^q \simeq \rho_2^q$ where $\simeq$ denotes *stuttering equivalence* [12], i.e. if their q-views agree up to finite repetitions of states. We call ρ_1 and ρ_2 *locally equivalent*, written $\rho_1 \approx \rho_2$, if they are q-equivalent for all $q \in \Pi$.

The following theorem asserts that fine-grained and coarse-grained executions generate the same set of local views.

Theorem 1. *For any fine-grained execution ξ of an HO algorithm, there exists a coarse-grained execution σ of the same algorithm for the same receive history such that $\sigma \approx \xi$, and vice-versa.*

The proof of this theorem given in [4] for benign faults extends to the more general context of malicious faults since it is based on some commutativity properties (among events) which do not depend on the fault model.

Theorem 1 can be used to verify linear-time properties of HO algorithms that are expressed in terms of local views of processes, and that are insensitive to specific interleavings. Formally, we say that a property P is *local* if for any (coarse- or fine-grained) executions ρ_1 and ρ_2 such that $\rho_1 \approx \rho_2$ we have $\rho_1 \models P$ iff $\rho_2 \models P$, i.e., ρ_1 satisfies P iff ρ_2 does. As an immediate consequence of Theorem 1, we obtain the following corollary:

Corollary 2. *If P is a local property, then $\sigma \models P$ holds for all coarse-grained executions σ of an algorithm if and only if $\xi \models P$ also holds for all fine-grained executions ξ of the same algorithm.*

Having to verify a given property just for all coarse-grained executions represents a significant reduction because coarse-grained executions afford a simpler representation of the system state (channels are all empty), and because fewer interleavings of events and fewer (types of) transitions must be considered.

We now indicate a sufficient syntactic criterion for determining when a formula of LTL-X, i.e., linear-time temporal logic without the next-time operator expresses a local property.[3] We assume that the set of state variables that appear in formulas is of the form $\mathcal{V} = \bigcup_{p \in \Pi} \mathcal{V}_p$ where $\mathcal{V}_p \cap \mathcal{V}_q = \emptyset$ for different processes $p \neq q$, and such that any state $s \in \Sigma_p$ of a process $p \in \Pi$ uniquely determines the values of the variables in $\mathcal{V}_p$.

We say that a formula φ is a *p-formula*, for $p \in \Pi$, if it contains only state variables from $\mathcal{V}_p$. It is easy to see that p-formulas are local properties, as are first-order combinations of p-formulas, for possibly different processes $p \in \Pi$. However, temporal combinations of p-formulas are in general not local because they can express the simultaneity of local states of different processes, or assert temporal relations between states of processes [3].

2.5 The Consensus Problem

In this paper, we concentrate on the well-known agreement problem, called *Consensus*, regarded as the fundamental problem that must be solved to implement a fault-tolerant system by replication. We assume that the state variables $\mathcal{V}_p$ include variables x_p and $decide_p$. The intuitive idea is that at the beginning of an execution the variable x_p holds the initial value of process p. Variable $decide_p$, initially *null*, represents the decision taken by process p in the sense that $decide_p$ is updated to the value $v \neq null$ when process p decides value v.

Consensus is specified as the conjunction of the following formulas of LTL-X, which are all local according to the criterion introduced in Section 2.4.

Integrity. Any decision value must be among the initial values.

$$\forall v : v \neq null \wedge \left(\bigvee_{p \in \Pi} \Diamond(decide_p = v) \right) \Rightarrow \bigvee_{q \in \Pi} x_q = v.$$

[3] LTL-X formulas are stuttering invariant [17].

Irrevocability. A process that has decided must never change its decision value.

$$\forall v : v \neq null \Rightarrow \Box\big(decide_p = v \Rightarrow \Box(decide_p = v)\big)$$

Agreement. The agreement property requires that if any two processes decide, they decide on the same value.

$$\begin{aligned} \forall v, w : \quad & v \neq null \wedge w \neq null \\ & \wedge \bigvee_{p,q \in \Pi} \big(\Diamond(decide_p = v) \wedge \Diamond(decide_q = w)\big) \\ \Rightarrow\ & v = w. \end{aligned}$$

Termination. The preceding properties are all safety properties; the sole liveness property requires that all processes eventually decide.

$$\Diamond(decide_p \neq null).$$

Contrary to classical approaches, the HO model does not flag processes as being faulty [5], and the above Consensus specification makes no exception: all processes must decide the same initial value of some process. Such a strong specification is not trivially unsolvable. Indeed, since there is no deviation from the next-state functions, processes may not take arbitrary steps, such as deciding arbitrary values. In the following, we formally prove that three HO algorithms solve the above strong Consensus specification under suitable communication predicates, thus demonstrating how the algorithms prevent every process from being contaminated by corrupted messages.

3 Representing the Heard-of Model in Isabelle

The uniform presentation of algorithms in the HO model by message-sending and next-state functions, and of system models by communication predicates, is attractive for formal verification, and the ability to verify these algorithms over coarse-grained executions significantly reduces the state space. Indeed, several algorithms solving Consensus under benign faults that were presented in [5] have been verified (for fixed-size instances) using model checking techniques [19,20,3]. Malicious faults, however, may introduce an infinite number of arbitrary values, making model checking prohibitive. We now describe our encoding of the HO model in the interactive proof assistant Isabelle/HOL [16], which allows us to verify arbitrary instances of algorithms.

3.1 Representing Algorithms and Communication Predicates

In the Isabelle model, the set Π of processes is represented by a type variable *'proc*. We will constrain *'proc* below so that it can only be instantiated by types with finitely many values. Similarly, the type variables *'pst* and *'msg* serve to represent the sets of local process states and messages, and corresponding concrete types will be defined for particular algorithms. Assignments of HO (or SHO) sets to processes are of type.

type_synonym *'proc HO = 'proc → 'proc set*,

i.e., functions from processes to sets of processes. The computational model is represented using Isabelle's *locale* mechanism: models of concrete algorithms are obtained as instances of the locale, whereas generic properties of the HO model can be proved within the locale and will be inherited by every instance.

locale *SHOAlgorithm* =
fixes
initState :: $[('proc :: finite), 'pst] \to bool$ **and**
sendMsg :: $[nat, 'proc, 'proc, 'pst] \to 'msg$ **and**
nextState :: $[nat, 'proc, 'pst, ('proc \rightharpoonup 'msg), 'pst] \to bool$ **and**
commPerRd :: $['proc\ HO, 'proc\ HO] \to bool$
commGlobal :: $[nat \to 'proc\ HO, nat \to 'proc\ HO] \to bool$

The interface of the Isabelle locale representing HO algorithms is shown above. It takes five parameters: *initState* represents a predicate (boolean function) such that *initState p s* is true iff *s* is an initial state of process *p*. (In Isabelle/HOL, function application is denoted by juxtaposition.) Similarly, the parameters *sendMsg* and *nextState* formally represent the message-sending and next-state functions S_p^r and T_p^r. For convenience, the communication predicate associated with the algorithm is split into a predicate *commPerRd*, which is evaluated at every round, and a predicate *commGlobal*, evaluated globally over ω-sequences of HO and SHO collections (cf. the definition of *SHORun* below).

3.2 Defining Coarse-Grained Executions

By Theorem 1, it is enough to verify Consensus algorithms over coarse-grained executions only, and we represent just these in Isabelle. As explained in Section 2.3, a coarse-grained execution is an ω-sequence of configurations, each of which is a function of type *'proc → 'pst*. Since channels are empty in every configuration of a coarse-grained execution, they need not be modeled.

In an initial configuration, every process is in an initial state:

definition *initConfig* **where** *initConfig cfg* ≡ ∀*p*. *initState p* (*cfg p*).

Configuration *cfg′* is a possible successor of configuration *cfg* at round *r* of an execution, given assignments *HO* and *SHO* of HO (resp., SHO) sets if for every process *p* there exists a vector μ of incoming messages compatible with *HO* and *SHO* such that the states of *p* before and after the transition and the message vector μ satisfy the *nextState* predicate.

definition *nextConfig* **where** *nextConfig r cfg HO SHO cfg′* ≡
∀*p*. ∃*m* ∈ *msgsVectors r p cfg HO SHO*. *nextState r p* (*cfg p*) *m* (*cfg′ p*)

where the set of possible message vectors is defined as

definition *msgsVectors* **where** *msgsVectors r p cfg HO SHO* ≡
{*m*. (∀*q*. *q* ∈ *SHO p* ⟷ *m q* = *Some* (*sendMsg r q p* (*cfg q*))) ∧
(∀*q*. *q* ∈ *HO p* ⟷ *m q* ≠ *None*)}

In words, vector m is compatible with *HO* and *SHO* if for all processes q in p's HO set, $m\ q \neq None$,[4] and moreover, for q in p's SHO set, $m\ q$ equals the message that q sent to p for the current round according to the *sendMsg* function. Because the value $m\ q$ is unconstrained for processes $q \in (HO\ p) \setminus (SHO\ p)$, any type-correct value may be received from these processes.

We now define a predicate characterizing executions of an HO algorithm, relative to collections *HOs* and *SHOs*, as infinite sequences of configurations $c_0\, c_1 \ldots$ where c_0 is an initial configuration, for all r, configuration c_{r+1} is a successor of c_r, and the Heard-Of collections satisfy the communication predicate.

definition *SHORun* **where** *SHORun rho HOs SHOs* $\equiv$
  initConfig (*rho* 0)
$\wedge\ \forall r.$ *nextConfig r* (*rho r*) (*HOs r*) (*SHOs r*) (*rho* (*Suc r*))
$\wedge\ \forall r.$ *commPerRd* (*HOs r*) (*SHOs r*)
$\wedge$ *commGlobal HOs SHOs*

4 Verifying Concrete Algorithms

We outline how different Consensus algorithms can be represented and verified as instances of the locale *SHOAlgorithm* introduced previously.

4.1 Modeling and Verifying Non-synchronous Algorithms in Isabelle

Biely et al. [2] introduce two non-synchronous Consensus algorithms tolerating malicious faults: the $\mathcal{U}_{T,E,\alpha}$ algorithm introduced in Section 2.1, and a one-round algorithm called $\mathcal{A}_{T,E,\alpha}$. We instantiate the generic Isabelle locale *SHOAlgorithm* for these algorithms and verify their correctness.

Figure 1 shows the representation of $\mathcal{U}_{T,E,\alpha}$ in Isabelle. We begin by declaring an anonymous type *Proc* of processes that is assumed to be finite. We then introduce the parameters T, E and α and indicate the assumed relations between them. Process states are represented as a record *pstate*, and messages are similarly represented as a data type *msg*. The definitions of the initial state predicate and the message-sending function are straightforward. Observe that the x field of initial states is left unconstrained, hence the initial value of processes may be any type-correct value. The definition of the next-state relation is split into two cases depending on the round number being even or odd.

The communication predicate for the $\mathcal{U}_{T,E,\alpha}$ algorithm, as specified in [2], is defined as the conjunction of the two following predicates:

definition *Ute_commPerRd* **where** *Ute_commPerRd HO SHO* $\equiv$
$\forall p.$ *card* $((HO\ p) \setminus (SHO\ p)) \leq alpha$
  $\wedge$ *card* $(SHO\ p) > N + 2 * alpha - E - 1$
  $\wedge$ *card* $(SHO\ p) > T$

[4] Isabelle's *None* corresponds to the pseudo-value $\perp \notin M$ introduced in Section 2.

```
typedecl Proc
axiomatization where procFinite : finite (UNIV :: Proc set)
abbreviation N ≡ card (UNIV :: Proc set) – cardinality of the set of processes
axiomatization T :: nat and E :: nat and α :: nat where
  E − α > N ÷ 2 and T − α > N ÷ 2 and E < N and T < N
consts defaultv :: 'val
record 'val pstate =
  x :: 'val
  vote :: 'val option
  decide :: 'val option
datatype 'val msg =
    Val 'val
  | Vote 'val option
definition step where step r ≡ r mod 2
definition initState where
initState p st ≡ vote st = None ∧ decide st = None
definition sendMsg where
sendMsg r ≡ if step r = 0 then Val(x st) else Vote(vote st)
definition next0 where
next0 r p st msgs st' ≡
  (∃v. card{q. msgs q = Some (Val v)} > T ∧ st' = st(|vote := Some v|))
∨ ((¬∃v. card{q. msgs q = Some (Val v)} > T) ∧ st' = st(|vote := None|))
definition next1 where
next1 r p st msgs st' ≡
  vote st' = None
∧ (∃v. card {q. msgs q = Some (Vote (Some v))} > α ∧ x st' = v) ∨
  ((¬∃v. card {q. msgs q = Some (Vote (Some v))} > α) ∧ x st' = defaultv)
∧ (∃v. card {q. msgs q = Some (Vote (Some v))} > E ∧ decide st' = Some v) ∨
  ((¬∃v. card {q. msgs q = Some (Vote (Some v))} > E) ∧ decide st' = decide st)
definition nextState where nextState r ≡ if step r = 0 then next0 r else next1 r
```

Fig. 1. Isabelle representation of the $\mathcal{U}_{T,E,\alpha}$ algorithm

```
definition phase where phase r ≡ r div 2
definition Ute_commGlobal where
Ute_commGlobal HOs SHOs ≡ ∀r. ∃φ > phase r. ∃r'. ∃π. ∀p.
    r' = 2 ∗ φ + 1
  ∧ π = HOs r' p ∧ π = SHOs r' p
  ∧ card (SHOs (r' + 1) p) > T ∧ card (SHOs (r' + 2) p) > E
```

The "round-by-round" predicate *Ute_commPerRd* is just the predicate $\mathcal{P}_{\alpha,\beta}$ introduced in Section 2.2 for $\beta = \max(N + 2\alpha - E - 1, T)$. It ensures the safety properties of the algorithm. The "global" predicate *Ute_commGlobal* is used to prove termination. It requires that there are infinitely many phases ϕ such that (1) the HO and SHO processes for all processes are identical in the second step

of phase ϕ and (2) the cardinality of the SHO sets for all processes exceeds T (resp., E) in the first (resp., second) step of the subsequent phase.

Finally, we declare $\mathcal{U}_{T,E,\alpha}$ to be an instance of the generic locale for SHO algorithms described in Section 3. This is achieved by the following Isabelle command, which instantiates the parameters of the locale *SHOAlgorithm* by the operators defined for the $\mathcal{U}_{T,E,\alpha}$ algorithm.

interpretation *SHOAlgorithm*
initState sendMsg nextStateUte_commPerRd Ute_commGlobal
by *unfold_locales*

We have used Isabelle to formally prove the correctness of $\mathcal{U}_{T,E,\alpha}$ (for an arbitrary number of processes). The proof is based on the informal proof given in [2], which we have split into a sequence of lemmas. Our main contribution is that we have been able to carry out a formal proof of a non-synchronous algorithm tolerating malicious faults with reasonable effort (the overall size of the verbose Isar proof script is under 900 lines, including comments). This would not have been possible without the high level of abstraction provided by the HO model. Based on the machine-checked proof, we can confidently assert the correctness of $\mathcal{U}_{T,E,\alpha}$.

Theorem 3. *The $\mathcal{U}_{T,E,\alpha}$ algorithm solves Consensus under the communication predicate specified by Ute_commPerRd and Ute_commGlobal.*

The $\mathcal{A}_{T,E,\alpha}$ algorithm, introduced together with $\mathcal{U}_{T,E,\alpha}$ in [2], is represented in Isabelle in an analogous way. It is a one-round HO algorithm in which a decision is taken immediately if a sufficient number of identical messages is received.

$\mathcal{U}_{T,E,\alpha}$ and $\mathcal{A}_{T,E,\alpha}$ differ in the algorithmic structure and rely on different communication predicates. In particular, $\mathcal{A}_{T,E,\alpha}$ has a simpler "round-by-round" predicate but a more elaborate "global" predicate:

definition *Ate_commPerRd* **where**
$Ate\_commPerRd\ HO\ SHO \equiv \forall p.\ card\ ((HO\ p) \setminus (SHO\ p)) \leq \alpha$

definition *Ate_commGlobal* **where**
$Ate\_commGlobal\ HOs\ SHOs \equiv$
$\quad \forall r\ p.\ \exists r' > r.\ card\ (HOs\ r'\ p) > T$
$\wedge\ \forall r\ p.\ \exists r' > r.\ card\ (SHOs\ r'\ p) > E$
$\wedge\ \forall r.\ \exists r' > r.\ \exists \pi_1\ \pi_2.\ card\ \pi_1 > E - \alpha\ \wedge\ card\ \pi_2 > T\ \wedge$
$\qquad \forall p \in \pi_1.\ HOs\ r'\ p = \pi_2 \wedge SHOs\ r'\ p = \pi_2$

These two predicates require that:

- the number of corrupted messages in each round is never greater than α,
- T and E thresholds are passed infinitely often, for every process,
- infinitely often, there exists a sufficiently big set π_1 of processes whose HO and SHO sets are all identical to some set π_2 that passes the T threshold.

We have again formally proved in Isabelle the correctness of the $\mathcal{A}_{T,E,\alpha}$ algorithm under this communication predicate. Despite the differences in the algorithms and the predicates, the effort required for carrying out the two proofs is quite comparable.

4.2 Verifying a Synchronous Algorithm

Our third case study is the well-known $EIGByz_f$ [1,15] algorithm, which decides after $f+1$ rounds and is designed for synchronous system models. Encoding $EIGByz_f$ in the HO model is straightforward. We have proved in Isabelle that the algorithm solves Consensus under the communication predicate defined by the round-by-round predicate $\mathcal{R}(r)$ and the global predicate $\mathcal{G}$, defined as

$$\mathcal{R}(r) :: \left| \bigcap_{p \in \Pi} SHO(p,r) \right| > \frac{N+f}{2} \qquad \mathcal{G} :: \left| \bigcap_{p \in \Pi, r \in \mathbb{N}} SHO(p,r) \right| \geq N - f.$$

$EIGByz_f$ was designed for synchronous systems with reliable links and at most f faulty processes. In such a system, every process receives the correct message from at least the non-faulty processes at every round, and therefore the predicate $\mathcal{G}$ is satisfied. The standard correctness proof for $EIGByz_f$ [1,15] assumes that $N > 3f$, and therefore $N - f > \frac{N+f}{2}$. Since moreover, for any $r \in \mathbb{N}$, we obviously have

$$\left(\bigcap_{p \in \Pi, r' \in \mathbb{N}} SHO(p,r') \right) \subseteq \left(\bigcap_{p \in \Pi} SHO(p,r) \right),$$

it follows that any execution of $EIGByz_f$ where $N > 3f$ also satisfies $\forall r : \mathcal{R}(r)$. The standard correctness hypotheses thus imply our communication predicates.

However, our proof shows that $EIGByz_f$ can indeed tolerate more transient faults than the standard bound can express. For example, consider the case where $N = 5$ and $f = 2$. Our predicates are satisfied in executions where two processes exhibit transient faults, but never fail simultaneously. Indeed, in such an execution, every process receives four correct messages at every round r, hence $\mathcal{R}(r)$ holds. Also, $\mathcal{G}$ is satisfied because there are three processes from which every process receives the correct messages at all rounds. By our correctness proof, it follows that $EIGByz_f$ then achieves Consensus, unlike what one could expect from the standard correctness predicate. This observation underlines the interest of expressing assumptions about transient faults, as in the HO model.

Finally, it is worth noting that, unlike $\mathcal{U}_{T,E,\alpha}$ and $\mathcal{A}_{T,E,\alpha}$, no assumption on the sets $HO(p,r) \setminus SHO(p,r)$ is ever required for the correctness of $EIGByz_f$: our predicates for $EIGByz_f$ are expressed in terms of the SHO sets only. In other words, the conditions ensuring the correctness of $EIGByz_f$ only specify how links must be both safe and live. However, contrasting with $\mathcal{U}_{T,E,\alpha}$ and $\mathcal{A}_{T,E,\alpha}$, which are correct under round-by-round conditions, $EIGByz_f$ requires a global predicate on the sequence of rounds (namely, $\mathcal{G}$). Such global predicates on just the safe heard-of sets actually correspond to what is classically called the "synchronous approach".

5 Related and Future Work

Despite the crucial need for rigorous correctness proofs, especially in the context of fault-tolerance, few distributed algorithms have been formally verified.

Moreover, formal verification of these algorithms mostly concerns benign faults (e.g., [19,20,3]) or even assumes that no fault may occur (e.g., [9,10]). We are aware of a few contributions addressing formal verification of distributed algorithms in the context of malicious faults, but all of them consider perfectly synchronous systems (e.g., [14,18]), with the exception of recent work by Lamport [13]. Lamport gives a formal safety proof of a variant on the Paxos algorithm that tolerates Byzantine faults (i.e., processes may deviate from their transition function). This algorithm, like most non-synchronous Consensus algorithms designed to tolerate malicious faults, assumes that processes can *authenticate* their communications, for example based on the use of *digital signatures*. A digital signature for process p is an extra information that p can add to any of its outgoing messages in order to prove that the message really originated at p, even if it has been relayed by several other processes. This informal description actually refers to the semantics of messages, and as pointed out by Lynch [15], no formal definition of malicious faults with authentication has ever been given. We therefore contend that relying on properties of authenticated messages represents a gap in the proof of an algorithm that uses them. Since neither $\mathcal{A}_{T,E,\alpha}$, $\mathcal{U}_{T,E,\alpha}$, nor *EIGByz* need authentication, we have been able to formally verify each of these Consensus algorithms in the context of malicious (communication) faults.

The Heard-Of model, in which we have carried out our work, lets us describe different algorithms, designed for different communication and fault models, in a uniform way. We verified three Consensus algorithms ($\mathcal{U}_{T,E,\alpha}$, $\mathcal{A}_{T,E,\alpha}$ and $EIGByz_f$) that tolerate malicious faults in our encoding of the HO model in the interactive proof assistant Isabelle/HOL, and we are confident that other algorithms can be verified with similar effort. Our proofs are at least an order of magnitude shorter than proofs for comparable algorithms under benign faults, such as the correctness proof for DiskPaxos [11] in Isabelle/HOL. This difference is essentially due to the higher level of abstraction gained through the use of the HO model, which allows us to consider only coarse-grained executions.

In future work, we would like to extend the framework to also cover malicious (Byzantine) transition faults. Although transition faults are indistinguishable from malicious communication faults to other processes in the network, the definition of Consensus has to be adapted, since no requirements can be placed on faulty processes. We are also interested in the representation of and formal reasoning about properties such as authentication, atomic broadcast or weak-interactive consistency in the HO model.

References

1. Bar-noy, A., Dolev, D., Dwork, C., Strong, H.R.: Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. In: Information and Computation, pp. 42–51 (1987)
2. Biely, M., Widder, J., Charron-Bost, B., Gaillard, A., Hutle, M., Schiper, A.: Tolerating corrupted communication. In: Proc. 26th Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, pp. 244–253. ACM, New York (2007)

3. Chaouch-Saad, M., Charron-Bost, B., Merz, S.: A reduction theorem for the verification of round-based distributed algorithms. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 93–106. Springer, Heidelberg (2009)
4. Charron-Bost, B., Merz, S.: Formal verification of a Consensus algorithm in the Heard-Of model. Int. J. Software and Informatics 3(2-3), 273–303 (2009)
5. Charron-Bost, B., Schiper, A.: The Heard-Of model: Computing in distributed systems with benign failures. In: Distributed Computing (2009)
6. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988)
7. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Science Comp. Prog. 2(3) (April 1982)
8. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
9. Georgiou, C., Lynch, N.A., Mavrommatis, P., Tauber, J.A.: Automated implementation of complex distributed algorithms specified in the IOA language. Intl. J. Software Tools for Technology Transfer 11(2), 153–171 (2009)
10. Hesselink, W.H.: The verified incremental design of a distributed spanning tree algorithm: Extended abstract. Formal Asp. Comput. 11(1), 45–55 (1999)
11. Jaskelioff, M., Merz, S.: Proving the correctness of Disk Paxos. Archive of Formal Proofs (2005), http://afp.sourceforge.net/entries/DiskPaxos.shtml
12. Lamport, L.: What good is temporal logic? In: Mason, R.E.A. (ed.) Information Processing 1983: Proceedings of the IFIP 9th World Congress, Paris. IFIP, pp. 657–668. North-Holland, Amsterdam (September 1983)
13. Lamport, L.: Byzantining Paxos by refinement. Technical report, Microsoft Research (December 2010)
14. Lamport, L., Merz, S.: Specifying and verifying fault-tolerant systems. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 41–76. Springer, Heidelberg (1994)
15. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
16. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. Peled, D., Wilke, T.: Stutter-invariant temporal properties are expressible without the next-time operator. Inf. Proc. Letters 63(5), 243–246 (1997)
18. Schmid, U., Weiss, B., Rushby, J.M.: Formally verified byzantine agreement in presence of link faults. In: 22nd Intl. Conf. Distributed Computing Systems (ICDCS 2002), Vienna, Austria, pp. 608–616. IEEE Comp. Society, Los Alamitos (2002)
19. Tsuchiya, T., Schiper, A.: Model checking of consensus algorithms. In: 26th IEEE Symp. Reliable Distributed Systems (SRDS 2007), Beijing, China, pp. 137–148. IEEE Comp. Society, Los Alamitos (2007)
20. Tsuchiya, T., Schiper, A.: Using bounded model checking to verify consensus algorithms. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 466–480. Springer, Heidelberg (2008)

The Computational Power of Simple Protocols for Self-awareness on Graphs*

Ioannis Chatzigiannakis[1,2], Othon Michail[1],
Stavros Nikolaou[1,2], and Paul G. Spirakis[1,2]

[1] Research Academic Computer Technology Institute (CTI), Patras, Greece
[2] Computer Engineering and Informatics Department (CEID), University of Patras
{ichatz,michailo,nikolaou,spirakis}@cti.gr

Abstract. We explore the capability of a network of extremely limited computational entities to decide properties about any of its subnetworks. We consider that the underlying network of the interacting entities (devices, agents, processes etc.) is modeled by a complete *interaction graph* and we devise simple graph protocols that can decide properties of some *input subgraph* provided by some preprocessing on the network. The agents are modeled as finite-state automata and run the same global graph protocol. Each protocol is a fixed size grammar, that is, its description is independent of the size (number of agents) of the network. This size is not known by the agents. We propose a simple model, the *Mediated Graph Protocol* (*MGP*) model, similar to the Population Protocol model of Angluin *et al.*, in which each network link is characterized by a *state* taken from a finite set. This state can be used and updated during each interaction between the corresponding agents. We provide some interesting properties of the MGP model among which is the ability to decide properties on stabilizing (initially changing for a finite number of steps) input graphs and we show that the MGP model has the ability to decide properties of disconnected input graphs. We show that the computational power within the connected components is fairly restricted. Finally, we give an exact characterization of the class **GMGP**, of graph languages decidable by the MGP model: it is equal to the class of graph languages decidable by a nondeterministic Turing Machine of linear space that receives its input graph by its adjacency matrix representation.

1 Introduction

Consider an application that allows users to make voice calls over the Internet by executing a software agent. The software agents are organized in a peer-to-peer overlay network. Suppose that in order to achieve certain quality of service levels, statistical data have shown that each agent must have at most k concurrent incoming voice-traffic flows. We assume that the software agents are quite limited: each agent has a constant number of bits of memory and two

* This work has been partially supported by the ICT Programme of the European Union under contract number ICT-2008-215270 (FRONTS).

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 135–147, 2011.

agents can communicate only when they are required to forward voice traffic. We also assume that agents have access to a global storage in which very limited information can be stored. In this setting, software agents have no control over their interactions: users come and go, and requests for voice calls are made by the users. We assume that the underlying pattern of interactions guarantees a fairness condition on the interactions: every pair of agents in the network is repeatedly allowed to exchange control information for their users to have voice calls.

Under these assumptions, there is a simple protocol ensuring that every agent eventually contains the correct answer. Each agent stores a counter $(0, 1, \ldots, k+1$, where k is a constant) signifying the number of active incoming traffic flows. The global storage service stores 1 bit for each possible pair of interacting agents. Initially, all agents have their counter set to 0 and each bit of the global storage is set to 1. When two agents interact, e.g., to forward voice traffic, if the bit corresponding to this interaction is 1, then the receiving agent increases its counter by one and the corresponding bit is set to 0. If the bit is 0, then nothing happens (the incoming flow has been already counted). If some counter reaches the value $k+1$, then an alert state is propagated to the population and eventually, all agents are informed of the existence of a potential bottleneck in the network and can take appropriate actions.

Now consider the question of whether the overlay network is fully connected or not. Is there a protocol to answer such questions without any assumptions about the size of the network? In this work, we focus on the following question: what properties of the underlying network can be computed by populations of computationally restricted, interacting entities? We are interested in the levels of knowledge that such systems can achieve regarding their own properties and characteristics, in other words, to what extent they can become *self-aware*. Such knowledge can be used to optimize the system's overall behavior w.r.t. resource usage, performance, etc., and to adapt to changing conditions concerning internal changes (e.g., a topology change) and context changes (e.g., a modification of user behavior).

2 Previous Work

In [3], Angluin *et al.* introduced the *Population Protocol* (*PP*) model, which captures the notion of computation by a population of extremely limited communicating agents. In this model, the system consists of a collection of agents, represented as finite-state machines. The agents exchange information via pairwise interactions, which they are unable to predict or control. Via these interactions, the system organizes its computation and provides complex behavior as a whole. In [3,4], the computational power of the model was studied and has been proved to be exactly the class of *semilinear predicates*, consisting of all predicates definable by first-order logical formulas of Presburger arithmetic (see, e.g., [10]). The capability of the model to decide graph properties of restricted interaction graphs was explored in [2].

In an attempt to enhance the basic model, an interesting variation was proposed in [7], called the *Mediated Population Protocol* (*MPP*) model, in which the population is also capable of storing constant size information for each pairwise interaction. This extension is fitting for modeling more complex systems where relations are formed between the interacting entities and the information generated concerning these relations is required in each interaction of the respective entities. For example, biological and artificial neural networks concern networks of interconnected simple processing elements that exhibit complex global behavior determined by the connections between them. These connections (synapses) can store parameters called "weights" that influence the outcome of the computations. In [6], it was proven that, in complete graphs, the MPP model is computationally equivalent to a Nondeterministic Turing Machine (NTM) of $\mathcal{O}(n^2)$ space that computes symmetric predicates.

In [8], the *Graph Decision Mediated Population Protocols* (*GDMPPs*) were introduced, which are essentially MPPs without input, that may run on any graph from a specified family, trying to decide some property of that graph. GDMPPs were proven unable to compute any nontrivial property of disconnected input graphs. For introductory texts to the area of population protocols the interested reader is referred to [5,12,1].

3 Our Results - Roadmap

In Section 4, we give a formal definition of the proposed model, (MGP), provide an example protocol illustrating the computation of the model and present some important definitions that are used throughout this work. We then (Section 5) present some fundamental properties of the new model. In particular, we extend (Sec. 5.1) the MGP model to allow the input graph to oscillate for a finite number of steps. This extension then allows us (Sec. 5.2) to compose protocols. In Sec. 6, we present a protocol (Sec. 6.1) that decides whether the input graph is connected and we then extend this idea (Sec. 6.2) and show that the new model is able to compute properties of disconnected input graphs, something that neither the PP nor the GDMPP model were capable of. By studying the computations in each connected component, we provide a first indication that in unrestricted (not necessarily complete) connected graphs the computational power dramatically drops. In Sec. 7, we give an exact characterization of the computational power of the MGP model, which is the class of all graph properties decidable by a NTM of linear space that takes as input the adjacency matrix of the input graph. Finally, in Sec. 8, we conclude and discuss some future research directions.

4 A Formal Model: Mediated Graph Protocols

A *Mediated Graph Protocol* (*MGP*) consists of a *finite set* of *agent states* Q, where $q_0, q_1 \in Q$ are the *initial agent states*, an *output function* $O : Q \to \{0, 1\}$ mapping agent states to binary outputs, a finite set of *edge states* S, where $s_0, s_1 \in S$ are the *initial edge states*, and a *transition function* $\delta : Q \times Q \times S \to$

$Q \times Q \times S$. If $\delta(a, b, s) = (a', b', s')$ we call $(a, b, s) \to (a', b', s')$ a *transition* and we define $\delta_1(a, b, s) = a'$, $\delta_2(a, b, s) = b'$, $\delta_3(a, b, s) = s'$.

An MGP runs on an *interaction graph* $G = (V, E)$, where V is a population and E is an irreflexive binary relation on V. Throughout this work, we assume that this graph is *complete*, so that an MGP may run on any $K_n = (V, E)$, where $|V| = n$ and $E = V^2 \backslash \{(u, u) \mid u \in V\}$.

We assume that the initial states of the agents and the edges of the network are specified by some function $\iota : V \cup E \to \{q_0, q_1, s_0, s_1\}$, which is not part of the protocol but models some preprocessing on the network. ι is called a *network initialization function* if $\iota(e) \in \{s_0, s_1\}$ for all $e \in E$, $\iota(u) = q_1$ if u is incident to at least one edge in s_1 according to E and $\iota(u) = q_0$ otherwise, for all $u \in V$. Given an interaction graph $K_n = (V, E)$ and a network initialization function ι, we may define the *subgraph of* K_n *specified by* ι as $G_\iota[K_n] = (V', E')$, where $V' = \{u \in V \mid \iota(u) = q_1\}$ and $E' = \{e \in E \mid \iota(e) = s_1\}$. $G_\iota[K_n]$ is the *input graph* to the protocol.

A (*network*) *configuration* is a mapping $C : V \cup E \to Q \cup S$ specifying the agent state of each agent in the population and the edge state of each edge in the interaction graph. Note first that a network initialization function ι specifies the initial configuration. Let C and C' be 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 $C'(z) = C(z), \forall z \in (V - \{u, v\}) \cup (E - \{e\})$, that is, C' is the result of the interaction of the pair (u, v) under configuration C and is the same as C except for the fact that the states of u, v, and (u, v) have been updated according to δ_1, δ_2, and δ_3, respectively. Note that each interaction (u, v) is an *ordered pair* that is each agent has a distinct role in the interaction, u that of the *initiator* and v that of the *responder*. We say that C can go to C' in one step, denoted $C \to C'$, 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 $C_i \to C_{i+1}$ for all i, $0 \leq i < t$, in which case we say that C' is *reachable* from C. The *transition graph* $T(\mathcal{A}, G)$ of an MGP $\mathcal{A}$ running on G is a directed graph whose nodes are all possible configurations and whose edges are all possible transitions on those nodes.

An *execution* is a finite or infinite sequence of configurations $C_0, C_1, C_2, \ldots$, where C_0 is an initial configuration and $C_i \to C_{i+1}$, for all $i \geq 0$. The interactions are chosen by an adversary who is not a part of the protocol and can make any scheduling assumption on the interaction pattern as long as it keeps the execution fair. *Fairness* is a restriction imposed on the adversary to prevent it from avoiding a possible step forever. There are various notions of fairness for the protocols we study. In this work, we use the notion of *strong global fairness*, according to which an infinite execution is *fair* if for every pair of configurations C and C' such that $C \to C'$, if C occurs infinitely often in the execution, then so does C' (cf. [9]). A *computation* is an infinite fair execution. The output of any agent u under configuration C is $O(C(u))$.

Due to the constant size descriptions of MGPs (Q and S are finite), the protocols we study are *uniform* (independent of the population size) and *anonymous* (agents can't store unique identifiers).

In this work, we are interested in determining properties of the subgraph that is specified by the network initialization function. To formalize this, let $\mathcal{H}$ be the *family* of all simple directed graphs with no isolated nodes. Note that $\mathcal{H}$ also includes disconnected graphs whose connected components have at least two nodes. A *graph language* is any $L \subseteq \mathcal{H}$.

Definition 1. *We say that an MGP $\mathcal{A}$* stably decides *a graph language L if, for any complete interaction graph $K_n = (V, E)$, any network initialization function ι, and any computation of $\mathcal{A}$ on K_n beginning from the initial configuration specified by ι, all agents eventually output 1 (accept) if $G_\iota[K_n] \in L$ and 0 (reject) otherwise. A graph language is said to be* stably decidable *by the MGP model (or MGP-*decidable*) if there is an MGP $\mathcal{A}$ that stably decides it.*

We call a protocol $\mathcal{A}$ a *stabilizing output graph* MGP if, in any computation of $\mathcal{A}$, all agents' outputs and edges' states eventually stop changing. We define **GMGP** to be the class of all stably decidable graph languages by the MGP model. We denote by **LGNSPACE** the class of all decidable graph languages by a NTM of linear space which receives the input graph by its adjacency matrix representation. We denote by **SEM** the class of *semilinear predicates*.

As a simple illustration, we formalize a version of the count-$(k{+}1)$-in-neighbors protocol that was outlined in the introduction (for $k = 2$). The set of agent states is $Q = \{q_0, q_1, q_2, q_3, q_4\}$ and the set of edge states is $S = \{s_0, s_1\}$. The output function O maps all states except q_4 to 0 and the state q_4 to 1. The transition function δ is defined as follows: if $i < 4$ and $j < 3$ then $\delta(q_i, q_j, s_1) = (q_i, q_{j+1}, s_0)$; if $i = 4$ or $j \geq 3$ then $\delta(q_i, q_j, s_1) = (q_4, q_4, s_0)$; if $i = 4$ or $j = 4$ then $\delta(q_i, q_j, s_0) = (q_4, q_4, s_0)$. All remaining transitions leave all three components unaffected.

Assume now that the agents are u_1, u_2, u_3, u_4. Since the interaction graph is complete, the edges are $(1,2), (1,3), (1,4), (2,1), (2,2), \ldots, (4,3)$. Let the initial configuration, as described by some network initialization function, be $((q_1, q_1, q_1, q_0), \{(1,2), (1,3), (2,3)\})$, where the tuple describes the state of each agent and the set contains the s_1 edges and is used for simplicity.

Consider now the following possible computation: $((q_1, q_1, q_1, q_0), \{(1,2), (1,3), (2,3)\}) \xrightarrow{(2,3)} ((q_1,\ q_1, q_2, q_0),\ \{(1,2), (1,3)\}) \xrightarrow{(4,3)} ((q_1, q_1, q_2, q_0),\ \{(1,2), (1,3)\}) \xrightarrow{(1,2)} ((q_1, q_2, q_2, q_0),\ \{(1,3)\}) \xrightarrow{(2,3)} ((q_1, q_2,\ q_2, q_0),\ \{(1,3)\}) \xrightarrow{(4,1)} ((q_1, q_2, q_2, q_0), \{(1,3)\}) \xrightarrow{(1,3)} ((q_1, q_2, q_3, q_0), \{\})$. In the last configuration, all agents output 0, and this configuration is *output stable* in the sense that, from that point on, no agent can change its output. So, in this case, the protocol rejects the input graph that was specified by the network initialization function and this is a correct decision because none of its nodes has more than 2 in-neighbors.

5 Properties of MGPs

We present some useful properties of the model that will help us unfold its computational potential. Let $L^{-1} = \{H \mid \exists G \in L$ such that H is the inverse of $G\}$ [1] be the inverse of a language L.

Theorem 1 (Closure). GMGP *is closed under union, intersection, complement, and inversion.*

5.1 MGPs with Stabilizing Input Graphs

In this section, we define the *stabilizing input graphs MGP* (*SIMGP*) model (similar to the PP model with stabilizing inputs [2]), in which the initial state of each agent and edge of the interaction graph (and thus the input graph) may change finitely many times before it stabilizes to a final value. Here, we consider the computational capabilities of the MGP model when the network initialization function is working in parallel (and not as a preprocessing on the network) with the execution of an MGP, as if another protocol eventually designates the input graph for an MGP. We are interested in stably deciding membership of the stabilized input graph in a graph language. Intuitively, one can think of the case where we are concerned about properties of a dynamic overlay network (which could be a result of a protocol running on a complete network infrastructure, e.g., a peer-to-peer network over the Internet) where the overlay can initially change but eventually stabilizes.

In a similar way to that of [2], let each agent/edge store its current initial state (which corresponds to the initial value given by ι) to a special component of its state. This state is available to the agent/edge at every computation step and may change arbitrarily (all the possible values, however, constantly belong to $\{s_0, s_1\}$ for the edges and to $\{q_0, q_1\}$ for the agents) between any two subsequent steps. The transition function is now of the form $\delta : ((Q \times \{q_0, q_1\}) \times (Q \times \{q_0, q_1\}) \times (S \times \{s_0, s_1\})) \rightarrow (Q \times Q \times S)$ and a configuration is a mapping $C : V \cup E \rightarrow (Q \times \{q_0, q_1\}) \cup (S \times \{s_0, s_1\})$ taking into account the current initial states of the agents and edges.

In the next theorem, we show that every MGP-decidable language is stably decidable even if the input graph is initially changing for a finite number of steps. To do so, we construct a protocol similar to the one presented in [6,11]. That protocol is also executed on complete interaction graphs. It constructs a correctly labeled spanning pseudo-path subgraph of the interaction graph and then exploits this construction to simulate a NTM on its input assignment. Informally, a pseudo-path graph is a straight line with arbitrary link directions. The agents become ordered according to this line and all the remaining edges of the complete interaction graph (those that are not part of the line) form the tape cells of the TM. These can be visited in an ordered fashion due to the ordering of the agents. Let L be any graph language:

[1] Let $G = (V, E)$ be a simple digraph and K the irreflexive subset of V^2. Then the *inverse* or *complement* of G is defined as $H = (V, K \backslash E)$.

Theorem 2. *L is stably SIMGP-decidable iff it is MGP-decidable.*

Proof. The straight direction holds trivially due to our focus on languages of stabilized input graphs. For the inverse, let $\mathcal{A}$ be an MGP that stably decides L. We can construct an SIMGP $\mathcal{B}$ which consists of protocol $\mathcal{A}$ and the protocol of [6] (see above) running in parallel so that the population can be organized into a pseudo-path graph. This construction ends in a finite number of interactions with the reinitialization of $\mathcal{A}$'s execution and can be used to perform further reinitializations whenever the input graph changes. Since the input graph stabilizes, $\mathcal{A}$ will eventually be executed correctly given the stabilized graph as input. □

5.2 Composition of MGPs

We will now present another interesting property of MGPs. According to this property, which we call *MGP-composition*, given two MGPs $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A}$ is a stabilizing output graph MGP, we can compose the two protocols to a new protocol $\mathcal{D}$. $\mathcal{D}$ will have the same output as $\mathcal{B}$ as if the latter was running on the stabilizing input graph defined by $\mathcal{A}$'s execution. $\mathcal{B}$'s input graph, provided by $\mathcal{A}$'s execution, is stabilizing since the edges' states eventually stabilize (by $\mathcal{A}$'s definition) and $\mathcal{A}$'s outputs 0 and 1 can be trivially mapped to initial agent states q_0 and q_1 respectively. The property is formalized below:

Theorem 3. *For any two MGPs $\mathcal{A}, \mathcal{B}$ where $\mathcal{A}$ is a stabilizing output graph MGP, there is an MGP $\mathcal{D}$ which is a composition of $\mathcal{A}$ and $\mathcal{B}$, has as input the input graph of $\mathcal{A}$ and as output the output of $\mathcal{B}$ running on the stabilized input graph provided by $\mathcal{A}$.*

Proof. From Theorem 2, we have that $\mathcal{B}$ can be replaced by a stabilizing input graphs protocol $\mathcal{B}'$ that runs on the stabilizing graph defined by $\mathcal{A}$ and works exactly like $\mathcal{B}$ running on the same graph. □

6 Disconnected Graphs

We now discuss the capability of our model to decide languages on disconnected graphs. Neither the PP model [2] nor the GDMPP model [8] are capable of supporting this feature. We here exploit the complete infrastructure to communicate information between the connected components of the disconnected input graph and make a decision according to the exchanged information. In Sec. 6.1, we present a simple protocol that can decide whether the input graph is a connected graph and, in Sec. 6.2, we generalize the idea to prove that any semilinear predicate on the multiset of decisions of any GDMPP running on the connected components (where the decision of each component is counted only once) that constitute the input graph is stably decidable.

6.1 Deciding Connectivity

In this section, we present an MGP CP that decides the language $L_C = \{G \mid G \text{ is a connected graph}\}$.

Protocol 1. *Connectivity Protocol (CP)*

1: $Q = \{q_0, q_1, t, t', l, l'\}$, $S = \{s_0, s_1\}$,
2: $O(q_0) = 0, O(q_1) = 0, O(t) = 1, O(t') = 0, O(l) = 1, O(l') = 0$,
3: δ:

a single leader is generated
$(q_1, q_1, s_1) \to (l, t, s_1)$

the single leader turns all nodes of the input graph it can reach to followers
$(l, t, s_1) \to (t, l, s_1), (t, l, s_1) \to (l, t, s_1), (l, q_1, s_1) \to (t, l, s_1), (q_1, l, s_1) \to (l, t, s_1)$

the single leader turns all nodes that do not belong to the input graph to followers of a single leader
$(l, q_0, s_0) \to (l, t, s_0)$

two single leaders meet in the same connected component of the input graph; one is turned to follower
$(l, l, s_1) \to (l, t, s_1)$

two non-adjacent single leaders meet in the same connected component of the input graph or in different connected components (in the case of disconnected input graph); they become non-unique leaders
$(l, l, s_0) \to (l', l', s_0)$

the non-unique leaders turn non-leaders into their followers
$(l', t, s_1) \to (t', l', s_1)$, $(t, l', s_1) \to (l', t', s_1)$, $(l', q_1, s_1) \to (t', l', s_1)$, $(q_1, l', s_1) \to (l', t', s_1)$, $(l', q_0, s_0) \to (l', t', s_0)$, $(l', t, s_0) \to (l', t', s_0)$

any two leaders meet in the same component; one single leader remains
$(l', l, s_1) \to (l, t, s_1), (l, l', s_1) \to (l, t, s_1), (l', l', s_1) \to (l, t, s_1)$

any two leaders meet in different components; both become non-unique
$(l', l, s_0) \to (l', l', s_0), (l, l', s_0) \to (l', l', s_0)$

a single leader restores all followers of multiple leaders to followers of single leaders
$(l, t', s_0) \to (l, t, s_0), (l, t', s_1) \to (t, l, s_1), (t', l, s_1) \to (l, t, s_1)$

Theorem 4. *Protocol CP stably computes L_C for any input graph $G_\iota[K_n]$ on the complete interaction graph K_n.*

Proof. As can be observed by the description of Protocol 1, each connected component elects a leader which eventually becomes unique for the component. Therefore, if there are more than one connected components, their leaders will interact via an s_0 edge and all agents will be informed that there are at least 2 components in the input graph and will output 0. If no such interaction takes place, all agents output 1. □

6.2 Computing Graph Languages on Disconnected Graphs

In this section, we are interested in languages that describe properties of disconnected input graphs, that is, graphs with > 1 connected components. We use the term "connected components" for weakly-connected components as well. We are not interested in components consisting of a single agent (input graphs with isolated nodes). To achieve this, we propose a construction which combines the functionality of four MGPs that allow information exchange between the connected components by exploiting the complete underlying infrastructure. In what follows, we give a description of these protocols.

First, we have the *spanning pseudo-path graph protocol* described in Section 5.1 that is required for the composition of protocols. We will call it *RP* (*Reinitialization Protocol* since it is mainly used to reinitialize the execution of the composed protocols).

Then, there is a *Leader Election MGP* (*LE*) that practically runs on the connected components of the input graph (leaving all q_0 agents of the population intact). $LE = \{Q_{LE}, S_{LE}, \delta\}$, where $Q_{LE} = \{q_0, q_1, l, f\}$, $S_{LE} = \{s_1\}$ and δ has the following transitions: $(q_1, q_1, s_1) \to (l, f, s_1)$ in which a leader is generated; $(l, q_1, s_1) \to (f, l, s_1)$ and $(q_1, l, s_1) \to (l, f, s_1)$ via which the leader turns non-leaders to followers; $(l, f, s_1) \to (f, l, s_1)$ and $(f, l, s_1) \to (l, f, s_1)$ which allow the leader state to move among the agents of a connected component; $(l, l, s_1) \to (l, f, s_1)$ which removes multiple leaders. Interactions between agents not defined by the previous transitions are ineffective (leave the states of both agents unchanged).

Lemma 1. *LE eventually elects a unique, constantly moving leader in each connected component of the input graph.*

Proof. The functionality is similar to the one of Protocol 1 of Sec. 6.1 without the interactions between leaders of different connected components. The remaining leader moves constantly due to the transitions $(l, f, s_1) \to (f, l, s_1)$ and $(f, l, s_1) \to (l, f, s_1)$. □

The third protocol is a parameter to the composition. It can be any MGP that works only within the connected components (effective interactions take place only between agents linked with s_1 edges). This protocol, that we call *BGP* (*Basic Graph Protocol*) hereafter, is practically a GDMPP [8] since it runs on any connected graph (instead of a complete one). *BGP* runs in parallel with *LE* within the connected components of the input graph and decides the same graph property within each component. This means that, *once all components of the input graph stabilize w.r.t. BGP's execution, all agents within each component will output* 1 *if the component satisfies the property and* 0 *otherwise.* Obviously, agents of different components may have different outputs.

The parallel execution of *LE* and *BGP* ends up with a unique moving leader in each connected component, all other agents are followers and every agent knows the decision of *BGP* for the component it belongs to. An agent that is not part of the input graph (q_0) is not affected and outputs by default 0. For each connected component of the input graph:

Definition 2. *We call an agent representative of the component if it is the unique leader and its output w.r.t. BGP has stabilized.*

In other words, once the number of leaders stops changing and BGP stabilizes, the unique leader of each connected component becomes a representative (the elected agent that bears the decision of the component w.r.t. the graph property that BGP decides). Note that regardless of the movement of the leader state within each component, once BGP stabilizes, the leader's output w.r.t. BGP remains the same no matter which agent is the leader.

The final protocol is also a parameter to the composition and is practically a population protocol (see [2]) running on the population of representatives. We call this protocol REP (*REpresentative Protocol*) and it runs in parallel with RP, LE and BGP. Since the interaction graph of MGPs is complete the representatives' population will be fully connected via the s_0 edges. The inputs of REP will be the outputs (decisions on the satisfiability of the graph property) of the agents w.r.t. BGP. We consider that effective interactions w.r.t. to REP can take place only between the representatives (via s_0 edges) of the population. In addition, we demand that whenever a representative moves to a neighboring agent within its component (since the leaders constantly move), it also copies its REP-state component to that agent. We consider that the output of REP is the output of the whole composition and we extend REP so that the representatives propagate their state when interacting with q_0 agents. In this way, all agents (the followers in each component due to representatives' movement and the q_0 agents due to the previous extension) will eventually have in their REP-state components the contents of the representatives' REP-state components.

The Composition of the Protocols: The composition is similar to the one described in Section 5.2. Firstly, RP constructs the spanning pseudo-path graph of the interaction graph reinitializing all other protocols during the process, then LE and BGP run in parallel to generate the representatives reinitializing REP, and finally, REP runs on the population of the representatives. We call the protocol resulting from the previous composition $GLADIS$ (*Graph LAnguages on DIsconnected graphS*). Since BGP and REP can be any GDMPP and PP, respectively, we denote the composition as $GLADIS(BGP, REP)$. The conclusion of this section is captured by the Theorem 5.

For all G, denote by $N_{G,L}$ the number of components of G that belong to a language L and by $N_{G,\overline{L}}$ the number of those that do not.

Theorem 5. *Let L be a GDMPP-decidable language. Let p be a semilinear predicate on $\mathbb{N}^2$. Then $L' = \{G \mid p(N_{G,L}, N_{G,\overline{L}}) = 1\}$ is MGP-decidable.*

Proof. $GLADIS(BGP, REP)$ takes an input graph given by some network initialization function and computes any semilinear predicate (due to REP) on the decisions (outputs of BGP) of the connected components (each component's decision is counted only once) of this input graph concerning any GDMPP-decidable graph property (since BGP is a essentially a GDMPP). □

The applications of Theorem 5 are various, depending on the *BGP* and *REP* we use. Given a GDMPP-decidable graph language, e.g., $L = \{G \mid G$ contains at least one 2-cycle $\}$ (the decidability of L was an important question left open by [8]; in fact, it turns out that L is decidable even by PPs), we can now answer questions about predicates on the number of components that satisfy L; questions like whether at least 25% of the components contain some 2-cycle. Whether the whole graph contains some 2-cycle can be simply decided by an OR population protocol on the representatives' population.

A fair question that arises is: what graph properties are stably decidable by the GDMPPs running on the components of the input graph? The exact computational power of the GDMPP model has not been characterized yet [8]. We approach the answer indirectly, by extending the notion of input graphs to vertex-labeled input graphs (whose labels are taken from a finite set X and are assigned by ι) and by considering computations on the multisets of the labels. These are, in fact, computations performed by any MPP (see [7]) on any connected graph given as input the multiset of labels. We call the corresponding computational class **MPU**. We provide the following exact characterization of **MPU**.

Theorem 6. MPU = SEM.

Proof. Let $p \in$ **MPU** be computable by an MPP $\mathcal{A}$. $\mathcal{A}$ still computes p if we restrict our attention on star graphs, which consist of 1 internal node of degree n and n external nodes of degree 1. The latter situation can be simulated by a PP, running on complete graphs, with a unique leader in its initial configuration since the leader can play the role of the internal node and also can safely (since external nodes have no effective interactions with each other) store the states of the edges on the external nodes. The latter, in turn, can be simulated by a PP (as we have proved recently) which is the composition of a leader election protocol, that elects a leader while leaving the inputs unaffected, and the stabilizing inputs implementation of a PP, whose transition function is the same as the simulated protocol, extended appropriately by ineffective transitions. □

The consequences of the above theorem are twofold. First of all, the GDMPP model on unrestricted connected graphs seems to be computationally weak; this is a first step towards answering a major question left open by [8]. Secondly, this characterization, if compared to the one for the MGP model that is provided in the following section, shows that the MGPs seem to be significantly more powerful than the GDMPPs.

7 An Exact Characterization: GMGP = LGNSPACE

In this section, we will develop an MGP that is capable of simulating a linear-space NTM on input $G_\iota[K_n] = (V', E')$. In this manner, we establish that any graph language $L \in$ **LGNSPACE** is stably decidable by the MGP model, or equivalently that **LGNSPACE** $\subseteq$ **GMGP**. By showing that the inverse is also possible, we conclude that the inclusion holds with equality.

Now, consider the following 3-component initial configuration: According to the first component, called the *label*, there is a unique spanning correctly labeled pseudo-path subgraph $L = (V, A)$ of a complete interaction graph K_n.[2] The second component stores the values of some network initialization function ι and is called the *membership indicator*. The third component is the (simulation) *tape* and has initially the value $\sqcup^k$ for some predetermined constant k.

Lemma 2. *There is an MPP that in any computation on K_n, beginning from such a configuration, stores the input graph $G_\iota[K_n]$ in the leftmost cells of the tape and halts in a finite number of steps having preserved the initial states of both the label and the membership indicator components.*

Proof. We use the protocol of [6,11] that constructs a spanning pseudo-path graph of K_n, L which allows the orderly visit of K_n's edges and the full use of the distributed memory of the population. To store the adjacency matrix, we visit the edges in an orderly fashion and store each time the distances of the two ends of the visited edge. The distance of an agent is the length of the unique pseudo-path from the fixed leader endpoint of L to that agent. Thus, the distance of both ends of a visited edge can be stored in two $\mathcal{O}(n)$ counters which are the indexes of the entry corresponding to the edge on the adjacency matrix. If the visited edge belongs to the input graph, then a 1 is stored in the $\mathcal{O}(n^2)$ distributed memory and 0 otherwise. □

Now the exact characterization follows easily:

Theorem 7. GMGP = LGNSPACE.

8 Conclusions - Future Research Directions

Many interesting issues arise by the findings of our work. The ability of the new model to use its complete network infrastructure enables us to compose protocols and decide graph properties of disconnected graphs. The additional memory provided by the extra edges of the complete interaction graph gives an important advantage to MGPs in comparison to GDMPPs. However, extra nodes do not seem to help: after all, in our model, the worst-case interaction graph of any input graph is itself made complete. Various questions arise from the above conclusions. How would the computability be affected if we had allowed more memory in each agent or each edge? Which interaction graph topologies allow the full use of the distributed memory? Do we truly require a complete interaction graph to decide graph languages in disconnected graphs or a connected infrastructure would suffice? How can we exploit the presence of extra nodes for increasing the computational power?

Acknowledgements. We would like to specifically thank Theofanis Raptis for his useful comments throughout the writing of this work.

[2] Note that, by definition of a correctly labeled pseudo-path subgraph, it holds that for all $e \in E - A$, e is inactive.

References

1. Àlvarez, C., Chatzigiannakis, I., Duch, A., Gabarró, J., Michail, O., Maria, S., Spirakis, P.G.: Computational models for networks of tiny artifacts: A survey. Computer Science Review, 5(1) (January 2011)
2. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 63–74. Springer, Heidelberg (2005)
3. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing, 235–253 (March 2006)
4. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)
5. Aspnes, J., Ruppert, E.: An introduction to population protocols. Bulletin of the EATCS 93, 98–117 (2007)
6. 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. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 270–281. Springer, Heidelberg (2010)
7. 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)
8. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Stably decidable graph languages by mediated population protocols. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 252–266. Springer, Heidelberg (2010)
9. Fischer, M., Jiang, H.: Self-stabilizing leader election in networks of finite-state anonymous agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 395–409. Springer, Heidelberg (2006)
10. Ginsburg, S., Spanier, E.H.: Semigroups, presburger formulas, and languages. Pacific Journal of Mathematics 16, 285–296 (1966)
11. Michail, O., Chatzigiannakis, I., Spirakis, P.G.: Mediated population protocols. Theor. Comput. Sci. 412, 2434–2450 (2011)
12. Michail, O., Chatzigiannakis, I., Spirakis, P.G.: New Models for Population Protocols. In: Lynch, N.A. (ed.) Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool (2011)

Self-stabilizing Labeling and Ranking in Ordered Trees*

Ajoy K. Datta[1], Stéphane Devismes[2], Lawrence L. Larmore[1], and Yvan Rivierre[2]

[1] School of Computer Science, University of Nevada Las Vegas, USA
firstname.lastname@unlv.edu,
http://www.egr.unlv.edu/~lastname
[2] VERIMAG UMR 5104, Université Joseph Fourier, France
firstname.lastname@imag.fr,
http://www-verimag.imag.fr/~lastname

Abstract. We propose two self-stabilizing algorithms for tree networks. The first one computes a special label, called *guide pair* of each process P in $O(h)$ rounds (h being the height of the tree) using $O(\delta_P \log n)$ space per process P, where δ_P is the degree of P and n the number of processes in the network. Guide pairs have numerous applications, including ordered traversal or navigation of the processes in the tree. Our second self-stabilizing algorithm, which uses the guide pairs computed by the first algorithm, solves the *ranking problem* in $O(n)$ rounds and has space complexity $O(b+\delta_P \log n)$ in each process P, where b is the number of bits needed to store a value. The first algorithm orders the tree processes according to their topological positions. The second algorithm orders (ranks) the processes according to the values stored in them.

Keywords: Self-stabilization, tree networks, tree labeling, ranking problem.

1 Introduction

Self-stabilization [4,5] is a versatile property, enabling an algorithm to withstand transient faults in a distributed system. A distributed algorithm is self-stabilizing if, after transient faults hit the system and place it in some arbitrary global state, the system recovers without external intervention in finite time.

An *ordered tree* $\mathcal{T}$ is a rooted tree, together with an order (called a left-to-right order) on the children of every node. In this paper, we give two self-stabilizing distributed algorithms for ordered trees. None of the two algorithms assumes knowledge of the size of the network n, or of a known upper bound of n, although, as it is usual in the literature, we assume that each process can store an integer in the range $1..n$, using $O(\log n)$ space. We choose the ordered tree topologies because results in such topologies can be easily extended to arbitrary rooted networks by composing our solutions with any existing self-stabilizing spanning tree construction algorithm (see [5] for the literature). However, the meaning of "traversing" or "ranking" processes in a general network is not clear.

* This work has been partially supported by the ANR project *ARESA*2.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 148–162, 2011.

Our first algorithm, GUIDE, computes a *guide pair* for each process P, which we write as $P.guide = (P.pre\_ind, P.post\_ind)$, where $P.pre\_ind$ and $P.post\_ind$ are the rank of P in the *preorder* and *reverse postorder* traversal, respectively, of the ordered tree. Figure 1 shows an example of ordered tree labeled with guide pairs. The guide pairs provide a labeling scheme that can be used for various applications [7]. In this work, we use these labels to navigate in the tree $\mathcal{T}$. We can define a partial ordering on the guide pairs as follows: We say $(i, j) \leq (k, \ell)$ if $i \leq k$ and $j \leq \ell$. Then, A process Q is a member of the subtree $\mathcal{T}_P$ rooted at P if and only if $P.guide \leq Q.guide$. The guide pairs can be used to implement routing between any two processes of the tree. If the two nodes satisfy the above partial ordering, then the routing path simply follows the list of ancestors/descendants. Otherwise, the routing must be established via the nearest common ancestor.

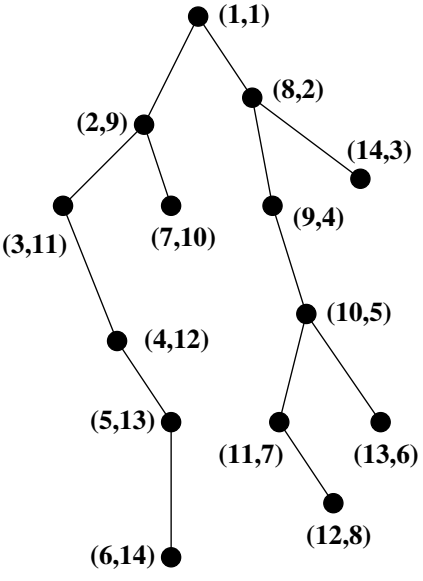


Fig. 1. Guide pairs

Our second algorithm, RANK, uses GUIDE, hence shows another application of guide pairs. The input of our second algorithm is a value $P.weight$, of some ordered type, for each process P. RANK computes the *rank* of each process, which is defined to be the index of that process if all processes were sorted by their weights.

1.1 Contributions

GUIDE has time complexity $O(h)$ rounds, where h is the height of $\mathcal{T}$. The time complexity of RANK is $O(n)$ rounds. The space complexity of GUIDE in each process P is $O(\delta_P \log n)$, where δ_P is the degree of P. RANK, which uses GUIDE as a subroutine, has space complexity $O(b + \delta_P \log n)$ in each process P, where b is the number of bits needed to store a value. GUIDE and RANK are self-stabilizing. GUIDE is *silent*, that is, it eventually reaches a terminal configuration where all actions of all processes are disabled. RANK correctly computes the rank of every process within $O(n)$ rounds. Unless the weights change, the ranking do not change once the system stabilizes. However, the algorithm repeatedly computes them to detect possible change of weights. If the weights do not change, the repeated computation of RANK will be transparent to the application that uses the output of RANK.

1.2 Related Work

The notion of guide pairs appeared first in [7], but that solution is not self-stabilizing. To the best of our knowledge, there exist no self-stabilizing algorithms for computing the guide pairs.

The only self-stabilizing solution to the ranking problem was given in [2]. This algorithm works in rooted trees. Like ours, that algorithm is not silent. Moreover, it assumes that each process has a unique identifier in the range $1..n$. The algorithm stabilizes in $\Omega(n^2)$ rounds using $O(\log n)$ space per process. The ranking problem is related to the *sorting problem*. There exist numerous self-stabilizing solutions to sorting in a tree, *e.g.*, [9,8,1]. However, all those previous problems are quite different than ours.

1.3 Roadmap

In the next section, we present the model we use throughout this paper. In Section 3, we present our self-stabilizing silent algorithm for computing guide pairs. In Section 4, we present our self-stabilizing algorithm for the ranking problem, which uses the guide pairs. Because of space limitations, the proofs have been omitted. See the technical report online (`http://www-verimag.imag.fr/~devismes/WWW/rapports/trRank.pdf`).

2 Preliminaries

Let $G = (V, E)$ be an undirected graph, where V is a set of nodes and E is a set of undirected edges linking nodes. Two nodes $P, Q \in V$ are said to be neighbors if $\{P, Q\} \in E$. The set of P's neighbors is denoted by $N(P)$. The degree of P *i.e.,* $|N(P)|$, is denoted by δ_P. $G = (V, E)$ is a *tree* if it is connected and acyclic. A tree $\mathcal{T}$ can be *rooted* at some node, meaning that one of its nodes $Root$ is distinguished as the *root* (all other nodes are anonymous). In a rooted tree $\mathcal{T}$, we denote by $P.par$, the parent of node P in $\mathcal{T}$: If $P = Root$, then $P.par = P$; otherwise $P.par = Q$, where Q is the neighbor of P that is the closest from the root (in this case, P is said to be a *child* of Q in $\mathcal{T}$). Let $Chldrn(P) = \{Q \in N(P) : Q.par = P\}$, the *children* of P in the tree $\mathcal{T}$. An *ordered tree* is a rooted tree $\mathcal{T}$, together with an (local) order (called a left-to-right order) on the children of every node. We denote by $\prec_P$ the local order relation among the children of node P. Let $P_1, P_2, \ldots P_m$ be the children of the root of $\mathcal{T}$ in the left-to-right order. We denote by $\mathcal{T}_i$ be the subtree rooted at any P_i. Finally, we denote by $Q \in \mathcal{T}_i$ the fact that node Q is a node of $\mathcal{T}_i$.

We model our network topology as an ordered tree $\mathcal{T} = (V, E)$, where V is a set of n nodes representing processes and E is a set of edges, each representing the ability of two processes to communicate directly. (We will use the terms "node" and "process" interchangeably.) We denote by $h(P)$ the height of process P in $\mathcal{T}$, *i.e.*, its distance to the root. We denote by h the height of $\mathcal{T}$, *i.e.*, $\max_{P \in V} h(P)$.

2.1 Computational Model

We consider the locally shared memory model, introduced by Dijkstra [4]. In this model, communications are carried out by locally shared variables. Each process has

the finite set of shared variables (henceforth, referred to as variables) whose domains are finite. A process P can read its own variables and that of its neighbors, but can write only to its own variables. We assume that every process P can read the local names of its neighbors, so that if $Q \in N(P)$, P can tell, for example, whether $Q.par = P$. Each process writes its variables according to its (local) *program*. A *distributed algorithm* is a collection of n *programs*, each one operating on a single process. The *program* of each process is a finite set of actions $\langle label \rangle :: \langle guard \rangle \mapsto \langle statement \rangle$. *Labels* are only used to identify actions in the discussion. The *guard* of an action in the program of a process P is a Boolean expression involving the variables of P and its neighbors. The *statement* of an action of P updates one or more variables of P. An action can be executed only if it is *enabled*, *i.e.,* its guard evaluates to $true$. A process is said to be *enabled* if at least one of its actions is enabled.

Let $\mathcal{A}$ be a distributed algorithm operating of a network of topology G. The values of $\mathcal{A}$'s variables at some process P define $\mathcal{A}$'s *(local) state* of P in G. A configuration of $\mathcal{A}$ in G is an instance of $\mathcal{A}$'s states of all processes in G. In the following, if there is no ambiguity, configurations of $\mathcal{A}$ in G will be simply denoted by *configurations*.

Let $\mapsto$ be the binary relation over configurations of $\mathcal{A}$ in G such that $\gamma \mapsto \gamma'$ if and only if it is possible for the network of topology G to change from configuration γ to configuration γ' in one step of $\mathcal{A}$. An *execution* of $\mathcal{A}$ is a maximal sequence of configurations $\varrho = \gamma_0\gamma_1 \ldots \gamma_i \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration in which no action of any process is enabled. Each step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* processes executing an action. The evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step; this model is called *composite atomicity* [5].

We assume that each step from a configuration to another is driven by a *scheduler*, also called a *daemon*. If one or more processes are enabled, the scheduler selects at least one of these enabled processes to execute an action. We assume that the scheduler is *weakly fair*, meaning that, every continuously enabled process P is selected by the scheduler within finite time.

We say that a process P is *neutralized* in the step $\gamma_i \mapsto \gamma_{i+1}$ if P is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. The neutralization of a process represents the following situation: at least one neighbor of P changes its state between γ_i and γ_{i+1}, and this change effectively makes the guard of all actions of P false.

We use the notion of *round*. The first *round* of an execution ϱ, noted ϱ', is the minimal prefix of ϱ in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let ϱ'' be the suffix of ϱ starting from the last configuration of ϱ'. The second round of ϱ is the first round of ϱ'', the third round of ϱ is the second round of ϱ'', and so forth.

2.2 Self-stabilization and Silence

In the following, we define a *specification* as a set of executions. We said that an execution ϱ satisfies the specification SP if $\varrho \in SP$.

A distributed algorithm $\mathcal{A}$ is *self-stabilizing with respect to* the specification SP in a network of topology G if and only if there exists a set of configurations $\mathcal{C}$ such that:

1. Every execution of $\mathcal{A}$ in a network of topology G starting from a configuration in $\mathcal{C}$ satisfies SP (*closure*).
2. Every execution of $\mathcal{A}$ in a network of topology G eventually reaches a configuration in $\mathcal{C}$ (*convergence*).

All configurations of $\mathcal{C}$ are said to be legitimate, all other configurations are said to be illegitimate.

We say that an algorithm is *silent* [6] if each of its executions is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a configuration where no process is enabled.

2.3 Composition

To simplify the design of our algorithms, we use a variant of the well-known *collateral composition* [10]. Roughly speaking, when we collaterally compose two algorithms $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ and $\mathcal{B}$ run concurrently and $\mathcal{B}$ uses the outputs of $\mathcal{A}$ in its executions. In the variant we use, we modify the code of $\mathcal{B}$ so that a process executes an action of $\mathcal{B}$ only when it has no enabled action in $\mathcal{A}$.

Let $\mathcal{A}$ and $\mathcal{B}$ be two algorithms such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. The *hierarchical collateral composition* [3] of $\mathcal{A}$ and $\mathcal{B}$, noted $\mathcal{B} \circ \mathcal{A}$, is the algorithm defined as follows:

1. $\mathcal{B} \circ \mathcal{A}$ contains all variables of $\mathcal{A}$ and $\mathcal{B}$.
2. $\mathcal{B} \circ \mathcal{A}$ contains all actions of $\mathcal{A}$.
3. For every action "$L_i :: G_i \mapsto S_i$" of $\mathcal{B}$, $\mathcal{B} \circ \mathcal{A}$ contains the action "$L_i :: \neg D \wedge G_i \mapsto S_i$" where D is the disjunction of all guards of actions in $\mathcal{A}$.

The following sufficient condition is given in [3] to show the correctness of the composite algorithm:

Theorem 1. *The composite algorithm $\mathcal{B} \circ \mathcal{A}$ self-stabilizes to specification SP in a network of topology G assuming a weakly fair scheduler if the following conditions hold: (i) in a network of topology G, Algorithm $\mathcal{A}$ is a silent algorithm under a weakly fair scheduler; (ii) in a network of topology G, Algorithm $\mathcal{B}$ stabilizes to SP under a weakly fair daemon, starting from any configuration where no action of $\mathcal{A}$ is enabled.*

3 Computing Guide Pairs

3.1 Guide Pairs

Given an ordered tree $\mathcal{T}$, the guide pair of a node P in $\mathcal{T}$ is the pair of integers i and j such that i and j are, respectively, the rank of P in the *preorder* and *reverse postorder* traversal of $\mathcal{T}$. Below, we define these notions. Recall that we denote by $P_1, P_2, \ldots P_m$ the children of the root of $\mathcal{T}$ in the left-to-right order, and we denote by $\mathcal{T}_i$ be the subtree rooted at any P_i. The *preorder traversal* of $\mathcal{T}$ is defined, recursively, as follows:

1. Visit the root of $\mathcal{T}$.
2. For each i from 1 to m in increasing order, visit the nodes of $\mathcal{T}_i$ in *preorder*.

Postorder traversal $\mathcal{T}$ is similarly defined:

1. For each i from 1 to m in increasing order, visit the nodes of $\mathcal{T}_i$ in *postorder*.
2. Visit the root of $\mathcal{T}$.

Preorder traversal is top-down, while postorder traversal is bottom-up. However, we can also traverse $\mathcal{T}$ in *reverse postorder*, which is top-down, as follows.

1. Visit the root of $\mathcal{T}$.
2. For i from m to 1 in decreasing order, visit the nodes of $\mathcal{T}_i$ in *reverse postorder*.

If a node P is the i^{th} node of $\mathcal{T}$ visited in a preorder traversal of $\mathcal{T}$, we say that the *preorder rank* of P is i. If a node P is the j^{th} node of $\mathcal{T}$ visited in a reverse postorder traversal of $\mathcal{T}$, we say that the *reverse postorder rank* of P is j. Write *pre_ind*(P) and *post_ind*(P) for the *preorder* rank and *reverse postorder* rank of P, respectively. We define the *guide pair* of P to be the ordered pair *guide*$(P) =$ (*pre_ind*(P), *post_ind*(P)). Figure 1 shows an ordered tree where each process is labeled with its guide pair.

If (i, j) and (k, ℓ) are guide pairs, we write $(i, j) \leq (k, \ell)$ if $i \leq k$ and $j \leq \ell$. Thus, the set of guide pairs is partially ordered by $\leq$.

Remark 1. [Property 2 in [7]] If P and Q are nodes of an ordered tree $\mathcal{T}$, then *guide*$(P) \leq$ *guide*(Q) if and only if P is an ancestor of Q.

3.2 Algorithm GUIDE

Algorithm GUIDE is a hierarchical collateral composition of two algorithms: GUIDE = CGP $\circ$ COUNT, where both COUNT and CGP (for *Compute Guide Pairs*) use *P.par* as input in the program of every process P. Note that *P.par* either designates the actual parent link of P or is computed by a distributed spanning tree algorithm with which GUIDE must be composed using the hierarchical collateral composition.

Algorithm COUNT. COUNT acts as a bottom-up wave that computes the number of processes in each subtree. In COUNT, each process P has only one variable: *P.subcount*. Moreover, each process P can compute the following function: *Subcount*$(P) = 1 + \sum_{Q \in Chldrn(P)} Q.subcount$. Thus, the program of P consists of the following action:

$$\texttt{SetCnt} :: P.subcount \neq Subcount(P) \mapsto P.subcount \leftarrow Subcount(P)$$

Lemma 1. COUNT *is self-stabilizing and silent, converges within* $h+1$ *rounds from an arbitrary initial configuration to a legitimate configuration where* $P.subcount = |\{Q \in \mathcal{T}_P\}|$ *for all processes* P*, and works under the weakly fair scheduler.*

Algorithm CGP**.** Using the values of *subcount* computed by COUNT, each process P evaluates in CGP for each of its children Q the number of processes before Q in the *preorder* and *reverse postorder* traversal of the tree $\mathcal{T}$, respectively (using Actions `SetChldPrePred` and `SetChldPostPred`, respectively). Then, reading these values from its parent, each process, except the root, can compute its guide pair (using Actions `SetPreInd` and `SetPostInd`). The guide pair of the root is $(1, 1)$ (see Actions `SetPreInd` and `SetPostInd` for the root).

Variables of CGP. In CGP, each process maintains several variables. First, the following array variable enables each non-root process to know its index in the local left-to-right order of its parent:

1. $P.chld[i] \in N(P) \cup \{\perp\}$, for all $1 \leq i \leq \delta_P$. This array is maintained by Action `SetChld`. For all $1 \leq i \leq |Chldrn(P)|$, $P.chld[i]$ is set to the i^{th} child in P's local ordering of $N(P)$, while for all $|Chldrn(P)| < i \leq \delta_P$, $P.chld[i]$ is set to $\perp$.[1]

Then, CGP uses the following additional variables:

2. $P.pre\_ind$, $P.post\_ind$, integers. In stabilized state, they contain the *preorder* and *reverse postorder* ranks of P, respectively. Thus, we will write $P.guide = (P.pre\_ind, P.post\_ind)$, the guide pair of P.
3. $P.chld\_pre\_pred[i]$, $P.chld\_post\_pred[i]$, integer, defined for all $1 \leq i \leq |\delta_P|$:
   - For all $1 \leq i \leq |Chldrn(P)|$, $P.chld\_pre\_pred[i]$ is set to the number of predecessors of the i^{th} child of P (that is, $P.chld[i]$) in the *preorder* traversal of $\mathcal{T}$; and $P.chld\_post\_pred[i]$ is set to the number of predecessors of the i^{th} child of P in the *reverse postorder* traversal of $\mathcal{T}$.
   - For all $|Chldrn(P)| < i \leq \delta_P$, $P.chld\_pre\_pred[i]$ and $P.chld\_post\_pred[i]$ are set to 0.

   Hence, each process P computes its guide pair to be

$$(P.par.chld\_pre\_pred[j] + 1, P.par.chld\_post\_pred[j] + 1)$$

   where P is the j^{th} child of its parent in left-to-right order.

Functions of CGP. Based on the previous variables, each process P can compute the following functions:

- $my\_order(P)$. If P is not the root and there exists i, $1 \leq i \leq \delta_{P.par}$, such that $P.par.chld[i] = P$, then $my\_order(P)$ returns i. If the values of $P.par.chld$ did not stabilize, $my\_order(P)$ returns 1.
  Once the system has stabilized, $my\_order(P)$ returns the index of the non-root process P in the local left-to-right order of its parent.
- $Chld\_index(Q) = |\{Q' \in Chldrn(P) : Q' \prec_P Q\}| + 1$. It returns the index of the child Q of process P in the local left-to-right order of P.

[1] Actually, cells from index $|Chldrn(P)| + 1$ to δ_P are useless. However, as the tree may be obtained by a spanning tree construction, we cannot know the number of children of P in advance, but this number is bounded by δ_P.

- $Eval\_chld(i)$ returns the local name of the i^{th} child of P. That is, if $\exists Q \in \mathit{Chldrn}(P)$ such that $Chld\_index(Q) = i$, then $Eval\_chld(i)$ returns Q; otherwise, $Eval\_chld(i)$ returns $\perp$.
- $Eval\_chld\_pre\_pred(i)$. If $i = 1$, then $Eval\_chld\_pre\_pred(i)$ returns $P.\mathit{pre\_ind}$; else if $2 \leq i \leq |\mathit{Chldrn}(P)|$, then $Eval\_chld\_pre\_pred(i)$ returns $P.\mathit{chld\_pre\_pred}[i-1] + P.\mathit{chld}[i-1].\mathit{subcount}$; otherwise it returns 0.
  Once the system has stabilized, $Eval\_chld\_pre\_pred(i)$ returns the number of predecessors of the i^{th} child of P in the *preorder* traversal of $\mathcal{T}$.
- $Eval\_chld\_post\_pred(i)$. If $i = |\mathit{Chldrn}(P)|$, then $Eval\_chld\_post\_pred(i)$ returns $P.\mathit{post\_ind}$; else if $1 \leq i < |\mathit{Chldrn}(P)|$, then $Eval\_chld\_post\_pred(i)$ returns $P.\mathit{chld\_post\_pred}[i+1] + P.\mathit{chld}[i+1].\mathit{subcount}$; otherwise $Eval\_chld\_post\_pred(i)$ returns 0.
  Once the system has stabilized, $Eval\_chld\_post\_pred(i)$ returns the number of predecessors of the i^{th} child of P in the reverse postorder traversal of $\mathcal{T}$.

Actions of CGP. Actions of CGP are given below. To simplify the presentation, we assume priorities on actions, and list them below in the order from the highest to the lowest priority. If several actions are enabled simultaneously at a process, only the one of the highest priority can be executed. In other words, the actual guard of any action "$L :: G \mapsto S$" of process P is $\neg D \wedge G$, where D is the disjunction of the guards of all actions at P that appear before in the text.

For every process P:

`SetChld` :: $\exists i \in [1..\delta_P]$, $P.\mathit{chld}[i] \neq Eval\_chld(i)$ $\mapsto$ $\forall i \in [1..\delta_P]$, $P.\mathit{chld}[i] \leftarrow Eval\_chld(i)$

`SetChldPrePred` :: $\exists i \in [1..\delta_P]$, $P.\mathit{chld\_pre\_pred}[i] \neq Eval\_chld\_pre\_pred(i)$ $\mapsto$ $\forall i \in [1..\delta_P]$, $P.\mathit{chld\_pre\_pred}[i] \leftarrow Eval\_chld\_pre\_pred(i)$

`SetChldPostPred` :: $\exists i \in [1..\delta_P]$, $P.\mathit{chld\_post\_pred}[i] \neq Eval\_chld\_post\_pred(i)$ $\mapsto$ $\forall i \in [1..\delta_P]$, $P.\mathit{chld\_post\_pred}[i] \leftarrow Eval\_chld\_post\_pred(i)$

For the root process $Root$ only:

`SetPreInd` :: $Root.\mathit{pre\_ind} \neq 1 \mapsto Root.\mathit{pre\_ind} \leftarrow 1$

`SetPostInd` :: $Root.\mathit{post\_ind} \neq 1 \mapsto Root.\mathit{post\_ind} \leftarrow 1$

For every non-root process P only:

`SetPreInd` :: $P.\mathit{pre\_ind} \neq 1 + P.\mathit{par}.\mathit{chld\_pre\_pred}[\mathit{my\_order}(P)]$
$\mapsto$ $P.\mathit{pre\_ind} \leftarrow 1 + P.\mathit{par}.\mathit{chld\_pre\_pred}[\mathit{my\_order}(P)]$

`SetPostInd` :: $P.\mathit{post\_ind} \neq 1 + P.\mathit{par}.\mathit{chld\_post\_pred}[\mathit{my\_order}(P)]$
$\mapsto$ $P.\mathit{post\_ind} \leftarrow 1 + P.\mathit{par}.\mathit{chld\_post\_pred}[\mathit{my\_order}(P)]$

Overview of CGP. We now give an intuitive explanation of how CGP computes the values of $P.pre\_ind$ for all P. The values of $P.post\_ind$ are computed similarly.

Suppose that P is the i^{th} process visited in a *preorder* traversal of $\mathcal{T}$. Then i is the correct value of $P.pre\_ind$. CGP works by computing the number of predecessors of P, *i.e.,* the number of processes visited before P is visited. Let us call that number $Num\_Preorder\_Preds(P)$. It is the correct value of $P.pre\_ind - 1$.

$Num\_Preorder\_Preds(Root) = 0$; otherwise, $Num\_Preorder\_Preds(P)$ is computed by $P.par$ and stored in the variable $P.par.chld\_pre\_pred[j]$, where P is the j^{th} child of $P.par$ in left-to-right order. In order to compute these values for all its children, $P.par$ must have computed its own value of pre_ind, as well as the sizes of all of its subtrees. If $j = 1$, then $Num\_Preorder\_Preds(P) = P.par.pre\_ind$, since $P.par$ is the immediate predecessor of its leftmost child in the *preorder* visitation. Thus, $P.par.chld\_pre\_pred[1] \leftarrow P.par.pre\_ind$. $P.par.chld\_pre\_pred[2]$ is obtained by adding the size of the leftmost subtree of $P.par$ to $P.par.chld\_pre\_pred[1]$, since all members of that subtree are predecessors of the second child of $P.par$.

In general, the number of predecessors of P is equal to $P.par.pre\_ind$ plus the sum of the sizes of the leftmost $j - 1$ subtrees of $P.par$. The values of the array $P.par.chld\_post\_pred$ are computed from right to left, similarly. P then executes:

$$P.pre\_ind \leftarrow P.par.chld\_pre\_pred[j] + 1$$
$$P.post\_ind \leftarrow P.par.chld\_post\_pred[j] + 1$$

Theorem 2. GUIDE *is self-stabilizing and silent, computes the guide pairs of all processes in $O(h)$ rounds from an arbitrary initial configuration, and works under the weakly fair scheduler.*

4 Rank Ordering

In this section, we give an algorithm, RANK, that uses guide pairs to solve the *ranking problem* on an ordered tree, $\mathcal{T}$. We are given a value $P.weight$ for each process P in $\mathcal{T}$. (For convenience, we assume that the weights are integers.) The problem is to find the *rank* of each P. If $P_1, P_2, \ldots, P_n$ is the list of processes in $\mathcal{T}$ sorted by weight, then i is the rank of P_i. We allow ties to be broken arbitrarily, but deterministically.

Our algorithm RANK is a hierarchical collateral composition of two algorithms: RANK = CRK $\circ$ GUIDE. RANK computes the rank of each process P in $\mathcal{T}$, and sets the variable $P.rank$ to that value. RANK is self-stabilizing, and requires $O(n)$ rounds and $O(b + \delta_P \log n)$ space for each process P.

4.1 Overview of CRK

Flow of Packages. The key part of the algorithm CRK is the *flow of packages*. Each package is an ordered pair $x = (x.value, x.guide)$, where $x.value$ is its *value* and $x.guide$ is its *guide pair*. We identify a package with its *guide pair*. Moreover, for every

two packages, x and y, we have $x \geq y$ (resp. $x > y$) if and only if $x.value \geq y.value$ (resp. $x.value > y.value$).

Each package has a *home process* (the node from which the package is originally issued), although its location can be at any process in the chain between its home and the root. The guide pair of a package is the same as the guide pair of its home process, and its value is either the weight of its home process or the rank that CRK will assign to its home process.

Each process P initiates its flow of packages by creating an *up-package* whose value is $P.weight$. This up-package then moves to the root by successive copying. The flow of packages is organized so that packages with smaller weights reach the root before packages with larger weights, in a manner similar to the standard technique for maintaining min-heap order in a tree.

After the root copies an up-package from a child, it creates a *down-package* with the same home process as the up-package, but whose value is a number (a rank) in the range $1..n$. The root maintains a counter so that the first down-package it creates has value 1, the second value 2, and so forth. Each down-package then moves back to its home process by copying. When its home process copies a down-package, it assigns, or re-assigns, its rank to be the value of that package.

The purpose (in fact even the name) of the guide pair is now obvious. It is used to guide the down-package to its home process.

Since the root copies up-packages in weight order, it creates down-packages in that same order. The i^{th} down-package created by the root will carry rank i and will use the same guide pair as the i^{th} up-package copied by the root. Its home process will then be the process whose weight is the i^{th} smallest in $\mathcal{T}$.

When the root detects that it has created all down-packages, it initiates a broadcast wave which resets the variables of CRK (except the rank and weight variables) and starts a new epoch.

Redundant Packages. In our model of computation, if a variable of a process P is copied by a neighbor Q, it also remains at P. In the algorithm CRK, each process P can be home to at most one package, but we cannot avoid the existence of multiple copies of that package (up and/or down). We handle that problem by defining a package variable currently held by a process (not necessarily its home process, rather any process on the chain from its home to the root) as being either *active* or *redundant*. A redundant package can freely be overwritten, but not an active package.

If x is an up-package currently held by some process Q which is not the root, then x is redundant if x has already been copied by $Q.par$. If x is an up-package currently held by the root, then x is redundant if the root has already created a down-package with the same guide pair as x. Any other up-package is active.

If x is a down-package held by some process Q which is not its home process, then Q is redundant if it has been copied by some child of Q. (The child that copies x must be the process whose subtree contains the home process of x.) If x is a down-package held by its home process P, then x is redundant if $P.rank$ is equal to the value of x. This indicates that P has already copied its rank from x, or that $P.rank$ was correct before x arrived. Any other down-package is active.

Status Waves. As it is typical for distributed algorithms which are self-stabilizing, but not silent, CRK endlessly repeats the calculation of the ranks of the processes in $\mathcal{T}$. We call one (complete) pass through this cycle of computations an *epoch*. At the end of each epoch, the variables of CRK at all processes, other than the variables for weight and rank, are reset for the next epoch. If an epoch has a clean start, it will calculate the correct rank for each process. Subsequent epochs will simply recalculate the same value, and $P.rank$ will never change again.

On the other hand, in case of an arbitrary initial configuration, it is possible for incorrect values of rank to be calculated, but eventually a configuration will be reached when the next epoch will get a clean start.

This system is controlled by the *status* variables of the processes. At the beginning of an epoch, a broadcast wave starting from the root changes the status of every process from either 0 or 4 to 1, and all variables of CRK except rank and weight are set to their initial values. When this wave reaches the leaves of $\mathcal{T}$, a convergecast wave changes the status of all processes to 2. All computation of the ranking algorithm, as discussed above, takes place while processes have status 2. After the root has created the last down-package, it initiates a broadcast wave where the status of all processes changes to 3. The return convergecast wave then changes the status of all processes to 4, and when this wave reaches the root, the new epoch begins.

Status zero is used for error correction. If any process detects that the current epoch is erroneous, it changes its status to 0. Status 0 spreads down the tree, as well as up the tree unless it meets a process whose status is 1. If $Root.status$ becomes 0 (and all its children have status 0 or 4), then $Root$ initiates a status 1 broadcast wave starting a new epoch. However, this may cause an endless cycle of 0 and 1 wave, going up and down the tree, respectively. We solve this problem by adding a special rule for the non-root processes. If $P.status = 0$ and $P.par.status = 1$, the status 0 wave cannot move up; instead, the status 0 wave moves down followed by status 1 wave.

4.2 Formal Definition of CRK

Variables of CRK. Let P be any process. $P.par$, $P.guide$, and $P.weight$ are inputs of CRK. Then, the output of CRK is $P.rank$, an integer. To compute this output, P maintains the following additional variables:

1. $P.up\_pkg$ and $P.down\_pkg$ are respectively of package type (that is, a guide pair and an integer) or $\perp$ (undefined).
   If $P.up\_pkg$ (resp. $P.down\_pkg$) is defined, then its home process is some $Q \in \mathcal{T}_P$.
2. $P.started$, Boolean.
   This variable indicates whether P has already generated its up-package during this epoch. ($P.up\_pkg$ may or may not still contain that up-package.)
3. $P.up\_done$, Boolean.
   It indicates whether all processes in $\mathcal{T}_P$ have created their own up-package in the current epoch and whether $\mathcal{T}_P$ contains no active up-package. (Active up-packages whose home processes are in $\mathcal{T}_P$ could exist at processes above P.)
4. $P.status \in [0..4]$.
   Status variables are used to control the order of computation and to correct errors.

Finally, $Root$ contains the following additional variable:

5. $Root.counter \in \mathbb{N}$
   This incrementing integer variable assigns the rank to packages. It is initialized to be 0 every time a new epoch begins.

Predicates of CRK**.** The predicate $\mathit{Clean\_State}(P)$ below indicates if P is in a good initial or "clean" state.

$$\mathit{Clean\_State}(P) \equiv P.\mathit{up\_pkg} = \bot \wedge P.\mathit{down\_pkg} = \bot \wedge \neg P.\mathit{started} \wedge \neg P.\mathit{up\_done}$$

The four following predicates are used for error detection:

$$\mathit{Is\_Consistent}(P, g) \equiv g = P.\mathit{guide} \vee \exists Q \in \mathit{Chldrn}(P), g \geq Q.\mathit{guide}$$

$$\begin{aligned}\mathit{Guide\_Error}(P) \equiv\ & (P.\mathit{up\_pkg} \neq \bot \wedge \neg \mathit{Is\_Consistent}(P, P.\mathit{up\_pkg}.\mathit{guide})) \vee \\ & (P.\mathit{down\_pkg} \neq \bot \wedge \neg \mathit{Is\_Consistent}(P, P.\mathit{down\_pkg}.\mathit{guide}))\end{aligned}$$

$$\begin{aligned}\mathit{Status\_Error}(P) \equiv\ & (P.\mathit{status} \in \{1,3\} \wedge P.\mathit{par}.\mathit{status} \neq P.\mathit{status}) \vee \\ & (P.\mathit{status} \in \{2,4\} \wedge \exists Q \in \mathit{Chldrn}(P), Q.\mathit{status} \neq P.\mathit{status}) \vee \\ & (P.\mathit{status} \neq 0 \wedge P.\mathit{par}.\mathit{status} = 0) \vee \\ & (P.\mathit{status} \notin \{0,1\} \wedge \exists Q \in \mathit{Chldrn}(P), Q.\mathit{status} = 0)\end{aligned}$$

$$\begin{aligned}\mathit{Error}(P) \equiv\ & \mathit{Status\_Error}(P) \vee \\ & (\neg \mathit{Clean\_State}(P) \wedge P.\mathit{status} = 1) \vee \\ & (\mathit{Guide\_Error}(P) \wedge P.\mathit{status} = 2) \vee \\ & (P.\mathit{up\_done} \wedge \neg P.\mathit{started} \wedge P.\mathit{status} = 2) \vee \\ & (P.\mathit{up\_done} \wedge P.\mathit{status} = 2 \wedge \exists Q \in \mathit{Chldrn}(P), \neg Q.\mathit{up\_done})\end{aligned}$$

We say that a guide pair g is *consistent with* P if the predicate $\mathit{Is\_Consistent}(P, g)$ is true. If $\mathit{Is\_Consistent}(P, g)$ is false, g is the guide pair of no process in the subtree of P. $\mathit{Guide\_Error}(P) = \mathit{true}$ means that P holds a package whose home is not in the subtree of P. The predicate $\mathit{Status\_Error}(P)$ indicates whether P detects that its status is inconsistent with those of its neighbors. Status errors are always the result of arbitrary initializations; eventually, $\mathit{Status\_Error}(P)$ will become false and will remain false forever for all P. Finally, the predicate $\mathit{Error}(P)$ detects error in the context of the current wave.

The four following predicates are used for flow control:

$$\begin{aligned}\mathit{Up\_Redundant}(P) \equiv\ & (P \neq \mathit{Root} \wedge P.\mathit{up\_pkg} \neq \bot \wedge P.\mathit{par}.\mathit{up\_pkg} \neq \bot \wedge P.\mathit{par}.\mathit{up\_pkg} \geq P.\mathit{up\_pkg}) \vee \\ & (P = \mathit{Root} \wedge P.\mathit{up\_pkg} \neq \bot \wedge P.\mathit{down\_pkg} \neq \bot \wedge P.\mathit{down\_pkg}.\mathit{guide} = P.\mathit{up\_pkg}.\mathit{guide})\end{aligned}$$

$$\begin{aligned}\mathit{Down\_Ready}(P) \equiv\ & P.\mathit{down\_pkg} = \bot \vee (P.\mathit{down\_pkg} \neq \bot \wedge \\ & (P.\mathit{down\_pkg}.\mathit{guide} \neq P.\mathit{guide} \wedge \exists Q \in \mathit{Chldrn}(P), Q.\mathit{down\_pkg} = P.\mathit{down\_pkg}) \vee \\ & (P.\mathit{down\_pkg}.\mathit{guide} = P.\mathit{guide} \wedge P.\mathit{rank} = P.\mathit{down\_pkg}.\mathit{value}))\end{aligned}$$

$$\begin{aligned}\mathit{Can\_Start}(P) \equiv\ & \neg P.\mathit{started} \wedge (P.\mathit{up\_pkg} = \bot \vee \mathit{Up\_Redundant}(P)) \wedge \forall Q \in \mathit{Chldrn}(P), \\ & (\neg \mathit{Up\_Redundant}(Q) \vee Q.\mathit{up\_done}) \wedge (Q.\mathit{up\_pkg} > (P.\mathit{weight}, P.\mathit{guide}) \vee Q.\mathit{up\_done})\end{aligned}$$

$$\begin{aligned}\mathit{Can\_Copy\_Up}(P, Q) \equiv\ & Q \in \mathit{Chldrn}(P) \wedge (Q.\mathit{up\_pkg} \neq \bot \wedge \neg \mathit{Up\_Redundant}(Q)) \wedge \\ & (P.\mathit{up\_pkg} = \bot \vee \mathit{Up\_Redundant}(P)) \wedge \\ & (P.\mathit{started} \vee (P.\mathit{weight}, P.\mathit{guide}) > Q.\mathit{up\_pkg}) \wedge \forall R \in \mathit{Chldrn}(P), \\ & R.\mathit{up\_done} \vee (\neg \mathit{Up\_Redundant}(R) \wedge (R.\mathit{up\_pkg} \geq Q.\mathit{up\_pkg} \vee R.\mathit{up\_done})\end{aligned}$$

$P.up\_pkg$ is redundant if $Up\_Redundant(P)$ is true. $Down\_Ready(P)$ states whether $P.down\_pkg$ is redundant or undefined, and thus P can create or copy a new down-package. $Can\_Start(P)$ decides whether P can create its own package, that is, if P can set $P.up\_pkg$ to $(P.weight, P.guide)$. $Can\_Copy\_Up(P)$ indicates whether P can copy $Q.up\_pkg$ to $P.up\_pkg$. We note that P can evaluate $Up\_Redundant(Q)$ for any $Q \in Chldrn(P)$.

Predicate $Up\_Done(P)$ below decides whether all processes in $\mathcal{T}_P$ have created their own up-package in the current epoch and whether $\mathcal{T}_P$ contains no active up-package. The evaluation of $Up\_Done(P)$ gives the correct value for $P.up\_done$.

$$Up\_Done(P) \equiv P.started = true \ \wedge Up\_Redundant(P) \ \wedge \forall Q \in Chldrn(P), Q.up\_done$$

Actions of CRK**.** Actions of CRK are given below. To simplify the design, we assume that the actions of CRK use the same priorities as those of CGP.

For the root process $Root$ **only:**

`Err`	:: $Error(Root)$	$\mapsto Root.status \leftarrow 0$
`NewEpoch`	:: $Root.status \in \{0,4\} \wedge$ $\forall Q \in Chldrn(Root)$, $Q.status \in \{0,4\}$	$\mapsto Root.status \leftarrow 1$; $counter \leftarrow 0$ $Root.up\_pkg \leftarrow \perp$; $Root.down\_pkg \leftarrow \perp$ $Root.started \leftarrow false$; $Root.up\_done \leftarrow false$
`ConvCast`	:: $Root.status = 1 \wedge$ $\forall Q \in Chldrn(Root), Q.status = 2$	$\mapsto Root.status \leftarrow 2$
`CreateUpPkg`	:: $Root.status = 2 \ \wedge Can\_Start(Root)$	$\mapsto Root.up\_pkg.value \leftarrow Root.weight$ $Root.up\_pkg.guide \leftarrow Root.guide$ $Root.started \leftarrow true$
`CopyUpPkg`	:: $Root.status = 2 \wedge$ $\exists Q \in Chldrn(Root)$, $Can\_Copy\_Up(Root, Q)$	$\mapsto Root.up\_pkg \leftarrow Q.up\_pkg$, $Q = min_{\prec_{Root}}\{R \in Chldrn(Root), Can\_Copy\_Up(Root, R)\}$
`EndUpPkg`	:: $Root.started \ \wedge Up\_Redundant(Root) \wedge$ $\forall Q \in Chldrn(Root), Q.up\_done$	$\mapsto Root.up\_done \leftarrow true$
`CreateDownPkg`	:: $Down\_Ready(Root) \wedge$ $Root.up\_pkg \neq \perp \wedge$ $\neg Up\_Redundant(Root)$	$\mapsto counter \leftarrow counter + 1$ $Root.down\_pkg.value \leftarrow counter$ $Root.down\_pkg.guide \leftarrow Root.up\_pkg.guide$
`SetRank`	:: $Root.down\_pkg \neq \perp \wedge$ $Root.down\_pkg.guide = Root.guide \wedge$ $Root.down\_pkg.value \neq Root.rank$	$\mapsto Root.rank \leftarrow Root.down\_pkg.value$
`BroadCast`	:: $Root.status = 2 \wedge$ $Root.up\_done \ \wedge Down\_Ready(Root)$	$\mapsto Root.status \leftarrow 3$
`EndEpoch`	:: $Root.status = 3 \wedge$ $\forall Q \in Chldrn(Root), Q.status = 4$	$\mapsto Root.status \leftarrow 4$

For every non-root process P only:

`Err`	:: $Error(P)$	$\mapsto P.status \leftarrow 0$
`NewEpoch`	:: $P.par.status = 1 \wedge P.status \in \{0,4\} \wedge$ $\forall Q \in Chldrn(P), Q.status \in \{0,4\}$	$\mapsto P.status \leftarrow 1$ $P.up\_pkg \leftarrow \perp$ $P.down\_pkg \leftarrow \perp$ $P.started \leftarrow false$ $P.up\_done \leftarrow false$
`ConvCast`	:: $P.status = 1 \wedge \forall Q \in Chldrn(P), Q.status = 2$	$\mapsto P.status \leftarrow 2$
`CreateUpPkg`	:: $P.status = 2 \wedge Can\_Start(P)$	$\mapsto P.up\_pkg.value \leftarrow P.weight$ $P.up\_pkg.guide \leftarrow P.guide$ $P.started \leftarrow true$
`CopyUpPkg`	:: $P.status = 2 \wedge$ $\exists Q \in Chldrn(P), Can\_Copy\_Up(P, Q)$	$\mapsto P.up\_pkg \leftarrow Q.up\_pkg,$ $Q = \min_{\prec_P}\{R \in Chldrn(P),$ $Can\_Copy\_Up(P, R)\}$
`EndUpPkg`	:: $P.started \wedge Up\_Redundant(P) \wedge$ $\forall Q \in Chldrn(P), Q.up\_done$	$\mapsto P.up\_done \leftarrow true$
`CopyDownPkg`	:: $Down\_Ready(P) \wedge$ $P.par.down\_pkg \neq \perp \wedge$ $P.par.down\_pkg \neq P.down\_pkg \wedge$ $Is\_Consistent(P, P.par.down\_pkg)$	$\mapsto P.down\_pkg \leftarrow P.par.down\_pkg$
`SetRank`	:: $P.down\_pkg \neq \perp \wedge$ $P.down\_pkg.guide = P.guide \wedge$ $P.down\_pkg.value \neq P.rank$	$\mapsto P.rank \leftarrow P.down\_pkg.value$
`BroadCast`	:: $P.par.status = 3 \wedge P.status = 2 \wedge$ $\forall Q \in Chldrn(P), Q.status = 2 \wedge Down\_Ready(P)$	$\mapsto P.status \leftarrow 3$
`EndEpoch`	:: $P.status = 3 \wedge \forall Q \in Chldrn(P), Q.status = 4$	$\mapsto P.status \leftarrow 4$

The actions above achieve three tasks. They are (1) error correction, (2) epochs, and (3) ranking computation (using the flow packages).

Error Correction. Action `Err` performs the error correction. If one process detects any inconsistency among its state and that of its neighbors, it initiates a reset of the network by changing its status to 0. This reset is contagious as previously explained.

Epochs. A new epoch starts by a reset initiated by Action `NewEpoch` at the root: If $Root.status$ is either 0 or 4, and every child of $Root$ has status 0 or 4, then $Root$ broadcasts the status 1 and resets to a clean state.

When status 1 reaches the leaves, a convergecast wave starts and changes the status of all processes to 2 by Action `ConvCast`, so that actual ranks computation can begin.

When $Root$ detects that there are no more up-packages in the tree, and it already sent every down-package, it initializes a broadcast of status 3 by Action `BroadCast`. Note that there could still be active down-packages below $Root$, but there could not be any active up-packages. Thus, $Root$ is finished with its tasks for the current epoch. Non-root process P propagates the status 3 by Action `BroadCast` after sending all its down-packages. There could still be active down-packages below P, but no active up-packages. Since $P.par.status = 3$, P knows that its job for this epoch is done.

Once status 3 reaches the leaves, a convergecast of status 4 is initialized and propagated by Action `EndEpoch`. When $Root$ changes to status 4, the current epoch is done, and $Root$ initiates a new one.

Ranking computation. The computation of the ranking is bottom-up and starts when the convergecast of status 2 starts at the leaves. The flow of up-packages is organized using `CreateUpPkg` and `CopyUpPkg`, that is, a process either inserts its own package in the flow or copy some package coming from a child by ensuring that packages are moved up in ascending order of weight. Once a process P has detected that $\mathcal{T}_P$ has no active up-package, it sets $P.up\_done$ to true by Action `EndUpPkg`. $Root$ initializes the broadcast of status 3 only after $Root.up\_done$ switches to true.

Upon receiving a new up-package (that is, $Root.up\_pkg$ is active), if $Root.down\_pkg$ is available (that is, it is either $\perp$ or redundant), $Root$ is enabled to create a new down-package to send down to the home of its up-package by `CreateDownPkg`. If $counter = i$, then $Root.up\_pkg$ is the i^{th} up-package copied or created by $Root$, its weight is the i^{th} smallest weight in the network, and i will become the value of the down-package.

The new active down-package is propagated to its home process by successive copying using Action `CopyDownPkg`. When it reaches its home process P, the value field of that package contains the correct value of the rank of P, so P updates $P.rank$ using Action `SetRank`, if necessary.

Theorem 3. RANK *is self-stabilizing, computes the ranking of all processes in* $O(n)$ *rounds from an arbitrary initial configuration, and works under the weakly fair scheduler.*

References

1. Bein, D., Datta, A., Villain, V.: Snap-stabilizing optimal binary-search-tree. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, pp. 1–17. Springer, Heidelberg (2005)
2. Bourgon, B., Datta, A.K., Natarajan, V.: A self-stabilizing ranking algorithm for tree structured networks. In: Proceedings of the First Workshop on Self-Stabilizing Systems (WSS 1995), pp. 23–28 (1995)
3. Datta, A.K., Devismes, S., Heurtefeux, K., Larmore, L.L., Rivierre, Y.: Self-stabilizing small k-dominating sets. Tech. rep., VERIMAG (2011), `http://www-verimag.imag.fr/TR/TR-2011-6.pdf`
4. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of Computing Machinery 17, 643–644 (1974)
5. Dolev, S.: Self-Stabilization. The MIT Press, Cambridge (2000)
6. Dolev, S., Gouda, M., Schneider, M.: Memory requirements for silent stabilization. In: PODC 1996: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 27–34 (1996)
7. Flocchini, P., Enriques, A.M., Pagli, L., Prencipe, G., Santoro, N.: Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively. IEICE Transactions 89-D(2), 700–708 (2006)
8. Herman, T., Masuzawa, T.: A stabilizing search tree with availability properties. In: Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001), pp. 398–405 (2001)
9. Herman, T., Pirwani, I.: A composite stabilizing data structure. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 167–182. Springer, Heidelberg (2001)
10. Tel, G.: Introduction to distributed algorithms, 2nd edn. Cambridge University Press, Cambridge (2000)

Fault-Tolerant Algorithms for Tick-Generation in Asynchronous Logic: Robust Pulse Generation*

[Extended Abstract]

Danny Dolev[1], Matthias Függer[2], Christoph Lenzen[1], and Ulrich Schmid[2]

[1] Hebrew University of Jerusalem, Jerusalem, Israel
{dolev,clenzen}@cs.huji.ac.il
[2] Vienna University of Technology, Vienna, Austria
{fuegger,s}@ecs.tuwien.ac.at

Abstract. The advances of deep submicron VLSI technology pose new challenges in designing robust systems, which can in principle be addressed by approaches established in fault-tolerant distributed systems research. This paper is the first step in an attempt to develop a very robust high-precision clocking system for hardware designs like systems-on-chip for critical applications. It is devoted to the design and the correctness proof of a novel Byzantine fault-tolerant self-stabilizing pulse synchronization protocol, which facilitates a direct implementation in standard asynchronous digital logic. Despite the severe implementation constraints, it offers optimal resilience and smaller complexity than all existing pulse synchronization protocols.

Keywords: clock synchronization, Byzantine faults, self-stabilization.

1 Introduction and Related Work

With today's deep submicron technology running at GHz clock speeds [19], disseminating the high-speed clock throughout a *very large scale integrated* (VLSI) circuit, with negligible skew, is difficult and costly [2,3,11,23,27]. Systems-on-chip are hence increasingly designed *globally asynchronous locally synchronous* (GALS) [4], where different parts of the chip use different local clock signals. Two main types of clocking schemes for GALS systems exist, namely, (i) those where the local clock signals are unrelated, and (ii) multi-synchronous ones that provide a certain degree of synchrony between local clock signals [28,31].

GALS systems clocked by type (i) permanently bear the risk of *metastable upsets* when conveying information from one clock domain to another. To explain the issue, consider a physical implementation of a bistable storage element, like a register cell, which can be accessed by read and write operations concurrently.

* The full paper is available at the arxiv [9]. This work has been supported by the Swiss National Science Foundation (SNSF), by the Austrian Science Foundation (FWF) project FATAL (P21694), and by by the Israeli Science Foundation (ISF) Grant number 1685/07. Danny Dolev is Incumbent of the Berthold Badler Chair.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 163–177, 2011.

It can be shown that two operations occurring very closely to each other can cause the storage cell to attain neither of its two stable states for an unbounded time [22]. Although the probability of a single upset is very small, one has to take into account that every bit of transmitted information across clock domains is a candidate for an upset. Elaborate synchronizers [8,20,26] are the only means for achieving an acceptably low probability for metastable upsets here.

This problem can be circumvented in clocking schemes of type (ii): Common synchrony properties offered by multi-synchronous clocking systems are bounded *precision*, i.e., bounded maximum offset in the number of clock transitions of any two local clock signals, and bounded *accuracy*, i.e., bounded difference of the local clock rate and the rate of progress of real time. Type (ii) clocking schemes are particularly beneficial from a designers point of view, since they combine the convenient local synchrony of a GALS system with a global time base across the whole chip. It has been shown in [25] that these properties indeed facilitate metastability-free high-speed communication across clock domains.

The decreasing structure sizes of deep submicron technology also resulted in an increased likelihood of chip components failing during operation: Reduced voltage swing and smaller critical charges make circuits more susceptible to ionized particle hits, crosstalk, and electromagnetic interference [5,17]. *Fault-tolerance* hence becomes an increasingly pressing issue in chip design. Unfortunately, faulty components may behave non-benign in many ways. They may perform signal transitions at arbitrary times and even convey inconsistent information to their successor components if their outgoing communication channels are affected by a failure. This forces to model faulty components as unrestricted, i.e., Byzantine, if a high fault coverage is to be guaranteed.

The DARTS fault-tolerant clock generation approach [14,16] developed by some of the authors of this paper is a Byzantine fault-tolerant multi-synchronous clocking scheme. DARTS comprises a set of modules, each of which generates a local clock signal for a single clock domain. The DARTS modules (nodes) are synchronized to each other to within a few clock cycles. This is achieved by exchanging binary clock signals only, via single wires. The basic idea behind DARTS is to employ a simple fault-tolerant distributed algorithm [32]—based on Srikanth & Toueg's consistent broadcasting primitive [29]—implemented in asynchronous digital logic. An important property of the DARTS clocking scheme is that it guarantees that no metastable upsets occur during fault-free executions. For executions with faults, metastable upsets cannot be ruled out: Since Byzantine faulty components are allowed to issue unrelated read and write accesses by definition, the same arguments as for clocking schemes of type (i) apply. However, in [12], it was shown that the probability of a Byzantine component leading to a metastable upset of DARTS can be made arbitrarily small.

Although both theoretical analysis and experimental evaluation revealed many attractive additional features of DARTS, like guaranteed startup, automatic adaption to current operating conditions, etc., there is room for improvement. The most obvious drawback of DARTS is its inability to support late joining and restarting of nodes, and, more generally, its lack of self-stabilization properties.

If, for some reasons, more than a third of the DARTS nodes ever become faulty, the system cannot be guaranteed to resume normal operation even if all failures cease. Even worse, simple transient faults such as radiation- or crosstalk-induced additional (or omitted) clock ticks accumulate over time to arbitrarily large skews in an otherwise benign execution.

Byzantine-tolerant self-stabilization, on the other hand, is the major strength of a number of protocols [1,6,10,18,21] primarily devised for distributed systems. Of particular interest in the above context is the work on self-stabilizing *pulse synchronization*, where the purpose is to generate well-separated anonymous pulses that are synchronized at all correct nodes. This facilitates self-stabilizing clock synchronization, as agreement on a time window permits to simulate a synchronous protocol in a bounded-delay system. Beyond optimal (i.e., $\lceil n/3 \rceil - 1$, c.f. [24]) resilience, an attractive feature of these protocols is a small stabilization time [1,6,18,21], which is crucial for applications with stringent availability requirements. In particular, [1] synchronizes clocks in expected constant time in a synchronous system. Given any pulse synchronization protocol stabilizing in a bounded-delay system in expected time T, this implies an expected $(T + \mathcal{O}(1))$-stabilizing clock synchronization protocol.

Note that existing synchronization algorithms, in particular those that do not rely on pulse synchronization, have deficiencies rendering them unsuitable in our context. For example, they have exponential convergence time [10], require the relative drift of the nodes' local clocks to be very small [7,21], provide low synchronization precision [21] or make use of linear-sized messages [6]. Furthermore, standard distributed systems' models do not account for metastability.

In this paper, we describe and prove correct the novel FATAL pulse synchronization protocol, which facilitates a direct implementation in standard asynchronous digital logic. It self-stabilizes within $\mathcal{O}(n)$ time with probability $1 - 2^{n-f}$, in the presence of up to $f = \lceil n/3 \rceil - 1$ Byzantine faulty nodes, and is metastability-free by construction after stabilization in failure-free runs. While executing the protocol, non-faulty nodes broadcast a constant number of bits in constant time. In terms of distributed message complexity, this implies that stabilization is achieved after broadcasting $\mathcal{O}(n)$ messages of size $\mathcal{O}(1)$, improving by factor $\Omega(n)$ on the number of bits transmitted by previous algorithms.[1] The protocol can sustain large relative clock drifts of more than 10%, which is crucial if the local clock sources are simple ring oscillators (uncompensated ring oscillators suffer from clock drifts of up to 9% [30]). If the number of faults is not overwhelming, i.e., a majority of at least $n - f$ nodes continues to execute the protocol in an orderly fashion, recovering nodes and late joiners (re)synchronize in constant time. All this is achieved against a powerful adversary that, at time t, knows the whole history of the system up to time $t + \varepsilon$ (where $\varepsilon > 0$ is infinitesimally small) and does not need to choose the set of faulty nodes in advance.

[1] We remark that [21] achieves the same complexity, but considers a much simpler model. In particular, *all* communication is restricted to broadcasts, i.e., all nodes observe the same behaviour of a given other node, even if it is faulty.

2 Model

Our formal framework will be tied to the peculiarities of hardware designs, which consist of modules that *continuously*[2] compute their output signals based on their input signals. Following [13,15], we define (the trace of) a *signal* to be a timed event trace over a finite alphabet $\mathbb{S}$ of possible signal states: Formally, signal $\sigma \subseteq \mathbb{S} \times \mathbb{R}_0^+$. All times and time intervals refer to a global *reference time* taken from $\mathbb{R}_0^+$, that is, signals describe the system's behavior from time 0 on. The elements of σ are called *events*, and for each event (s,t) we call s the *state of event* (s,t) and t the *time of event* (s,t). In general, a signal σ is required to fulfill the following conditions: (i) for each time interval $[t^-,t^+] \subseteq \mathbb{R}_0^+$ of finite length, the number of events in σ with times within $[t^-,t^+]$ is finite, (ii) from $(s,t) \in \sigma$ and $(s',t) \in \sigma$ follows that $s = s'$, and (iii) there exists an event at time 0 in σ.

Note that our definition allows for events (s,t) and $(s,t') \in \sigma$, where $t < t'$, without having an event $(s',t'') \in \sigma$ with $s' \neq s$ and $t < t'' < t'$. In this case, we call event (s,t') *idempotent*. Two signals σ and σ' are *equivalent*, iff they differ in idempotent events only. We identify all signals of an equivalence class, as they describe the same physical signal. Each equivalence class $[\sigma]$ of signals contains a unique signal σ_0 having no idempotent events. We say that *signal* σ *switches to* s at time t iff event $(s,t) \in \sigma_0$. The *state of signal* σ at time $t \in \mathbb{R}_0^+$, denoted by $\sigma(t)$, is given by the state of the event with the maximum time not greater than t.[3] Because of (i), (ii) and (iii), $\sigma(t)$ is well defined for each time $t \in \mathbb{R}_0^+$. Note that σ's state function in fact depends on $[\sigma]$ only, i.e., we may add or remove idempotent events at will without changing the state function.

Distributed System. A distributed system is a finite set of n nodes $V = \{1,\dots,n\}$. Each node i comprises a number of *input ports*, namely $S_{i,j}$ for each node j, an *output port* S_i, and a set of *local ports*, introduced later on. An *execution* of the distributed system assigns to each port of each node a signal. For convenience of notation, for any port p, we refer to the signal assigned to port p simply by signal p. We say that *node* i *is in state* s at time t iff $S_i(t) = s$ and that *node* i *switches to state* s at time t iff signal S_i switches to s at time t.

To enable nodes to communicate (that is, exchange their state), we assume the existence of *channels* between them: for each pair of nodes i,j, output port S_i is connected to input port $S_{j,i}$ by a FIFO channel from i to j with *maximum delay* $d > 0$.[4] Note that this includes a channel from i to i itself. The *channel* from node i to j is said to be *correct* during $[t^-,t^+]$ iff there exists a function $\tau_{i,j} : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, called the channel's *delay function*, such that: (i) $\tau_{i,j}$ is continuous and strictly increasing, (ii) $\forall t \in [t^-,t^+] : 0 \leq \tau_{i,j}(t) - t < d$, and (iii) for each $t \in [t^-,t^+]$, $(s,t) \in S_{j,i} \Leftrightarrow (s,\tau_{i,j}^{-1}(t)) \in S_i$, where τ^{-1} is the inverse of $\tau|_{[t^-,t^+]}$, i.e., τ restricted to $[t^-,t^+]$. Node i *observes node* j *in state* s at time t if $S_{i,j}(t) = s$.

[2] In sharp contrast to classic distributed computing models, there is no computationally complex discrete zero-time state-transition here.

[3] Whenever referring to σ, we will talk of the signal, not the state function.

[4] W.r.t. $\mathcal{O}$-notation, we normalize $d \in \mathcal{O}(1)$, as all time bounds depend linearly on d.

Clocks and Timeouts. Nodes are never aware of the current reference time and we also do not require it to resemble Newtonian "real" time. Rather we allow for physical clocks that run arbitrarily fast or slow, as long as their speeds are close to each other in comparison. One may hence think of the reference time as progressing at the speed of the currently slowest correct clock. In this framework, nodes essentially make use of bounded clocks with bounded drift.

Formally, clock rates are within $[1, \vartheta]$ (with respect to reference time), where $\vartheta > 1$ is constant and $\vartheta - 1$ is the *(maximum) clock drift*. A *clock* C is a continuous, strictly increasing function $C : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ mapping reference time to local time. Clock C is said to be *correct* during $[t^-, t^+] \subseteq \mathbb{R}_0^+$ iff we have for any $t, t' \in [t^-, t^+]$, $t < t'$, that $t' - t \leq C(t') - C(t) \leq \vartheta(t' - t)$. Each node comprises a set of clocks assigned to it, which allow the node to estimate the progress of reference time.

Instead of directly accessing their clocks, nodes have so-called *timeout ports* of watchdog timers. A *timeout* is a triple (T, s, C), where $T \in \mathbb{R}^+$, $s \in \mathbb{S}$, and C is a clock, say of node i. Each timeout (T, s, C) has a corresponding timeout port $\mathrm{Time}_{T,s,C}$, being part of node i's local ports. Signal (T, s, C) is Boolean, that is, its possible states are from the set $\{0, 1\}$. We say that timeout (T, s, C) is *correct* during $[t^-, t^+] \subseteq \mathbb{R}_0^+$ iff clock C is correct during $[t^-, t^+]$ and the following holds:

1. For each time $t_s \in [t^-, t^+]$ when node i switches to state s, there is a time $t \in [t_s, \tau_{i,i}(t_s)]$ such that (T, s, C) is *reset*, i.e., $(0, t) \in \mathrm{Time}_{T,s,C}$. This is a one-to-one correspondence, i.e., (T, s, C) is not reset at any other times.
2. For a time $t \in [t^-, t^+]$, denote by t_0 the supremum of all times from $[t^-, t]$ when (T, s, C) is reset. Then it holds that $(1, t) \in \mathrm{Time}_{T,s,C}$ iff $c(t) - c(t_0) = T$. Again, this is a one-to-one correspondence.

We say that timeout (T, s, C) *expires* at time t iff $\mathrm{Time}_{T,s,C}$ switches to 1 at time t, and it *is expired* at time t iff $\mathrm{Time}_{T,s,C}(t) = 1$. We will omit the clock C from the notation and simply write (T, s) for both the timeout and its signal.

A *randomized timeout* is a triple $(\mathcal{D}, s, C)$, where $\mathcal{D}$ is a bounded random distribution on $\mathbb{R}_0^+$, $s \in \mathbb{S}$ is a state, and C is a clock. Its corresponding timeout port $\mathrm{Time}_{\mathcal{D},s,C}$ behaves very similar to the one of an ordinary timeout, except that whenever it is reset, the local time that passes until it expires next—provided that it is not reset again before that happens—follows the distribution $\mathcal{D}$. For the purpose of this abstract, we give a simplified formal definition here. A randomized timeout $(\mathcal{D}, s, C)$ is correct during $[t^-, t^+] \subseteq \mathbb{R}_0^+$, if C is correct during $[t^-, t^+]$ and the following holds:

1. For each time $t_s \in [t^-, t^+]$ when node i switches to state s, there is a time $t \in [t_s, \tau_{i,i}(t_s)]$ such that $(\mathcal{D}, s, C)$ is *reset*, i.e., $(0, t) \in \mathrm{Time}_{\mathcal{D},s,C}$. This is a one-to-one correspondence, i.e., $(\mathcal{D}, s, C)$ is not reset at any other times.
2. For a time $t \in [t^-, t^+]$, denote by t_0 the supremum of all times from $[t^-, t]$ when $(\mathcal{D}, s, C)$ is reset. Let $\mu : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ denote the density of $\mathcal{D}$. Then

$$P[\mathrm{Time}_{\mathcal{D},s,C} \text{ switches to 1 during } [t_0, t]] = \int_{t_0}^{t} \mu(C(t) - C(t_0))\, d\tau.$$

We will apply the same notational conventions to randomized timeouts as we do for regular timeouts.

We remark that these definitions allow for different timeouts to be driven by the same clock, implying that an adversary may derive some information on the state of a randomized timeout before it expires from the node's behaviour, even if it cannot directly access the values of the clock driving the timeout. This is crucial for efficient implementability, as nodes require one clock only.

Memory Flags. Another kind of node i's local ports are *memory flags*. For each state $s \in \mathbb{S}$ and each node $j \in V$, memory flag $\mathrm{Mem}_{i,j,s}$ is a local port of node i. It is used to memorize whether node i has observed node j in state s since the last reset of the flag. We say that node i *memorizes node* j *in state* s at time t if $\mathrm{Mem}_{i,j,s}(t) = 1$. Formally, we require that signal $\mathrm{Mem}_{i,j,s}$ switches to 1 at time t iff node i observes node j in state s at time t and $\mathrm{Mem}_{i,j,s}$ is not already in state 1. The times when $\mathrm{Mem}_{i,j,s}$ is *reset*, i.e., when signal $\mathrm{Mem}_{i,j,s}$ switches to 0, are specified by node i's state machine, which is introduced next.

State Machine. It remains to specify how nodes switch states and when they reset memory flags. We do this by means of state machines that may attain states from the finite alphabet $\mathbb{S}$. Node i's state machine is specified by (i) the set $\mathbb{S}$, (ii) a function tr, called the *transition function*, from $\mathcal{T} \subseteq \mathbb{S}^2$ to the set of Boolean predicates on the alphabet consisting of expressions "$p = s$" (used for expressing guards), where p is a local or input port of i and s is a possible state of signal p, and (iii) a function re, called the *reset function*, from $\mathcal{T}$ to the power set of the node's memory flags.

Intuitively, the transition function specifies the conditions (guards) under which a node switches states, and the reset function determines which memory flags to reset upon the state change. Formally, let P be a predicate on node i's input and local ports. We define P *holds at time* t by structural induction: If P is an element of the above alphabet, i.e., $p = s$, then P *holds at time* t iff $p(t) = s$. Otherwise, if P is of the form $\neg P_1$, $P_1 \wedge P_2$, or $P_1 \vee P_2$, we define P *holds at time* t in the straightforward manner.

We say node i *follows its state machine during* $[t^-, t^+]$ iff the following holds: Assume node i observes itself in state $s \in \mathbb{S}$ at time $t \in [t^-, t^+]$, i.e., $S_{i,i}(t) = s$. Then, for each $(s, s') \in \mathcal{T}$, both:

1. Node i switches to state s' at time t iff $tr(s, s')$ holds at time t and i is not already in state s'.[5]
2. Node i resets memory flag m at some time within $[t, \tau_{i,i}(t)]$ iff $m \in re(s, s')$ and i switches to state s' at time t. This correspondence is one-to-one.

A node may also run several state machines in parallel. In this case, S_i simply is the product of the individual machine's output signals, and the different state machines interact by means of the delayed signal $S_{i,i}$ only.

A node is defined to be *non-faulty* during $[t^-, t^+]$ iff during $[t^-, t^+]$ all its timeouts and randomized timeouts are correct and it follows (all of) its state

[5] If more than one guard $tr(s, s')$ can be true concurrently, break ties arbitrarily.

machine(s). In contrast, a faulty node may change states arbitrarily. While a faulty node may be forced to send consistent states to all other nodes if its channels remain correct, there is no way to guarantee that this still holds if channels are faulty.[6]

Metastability. In our discrete system model, the effect of metastability is captured by the lacking capability of state machines to instantaneously take on new states: Node i decides on state transitions based on the delayed status of port $S_{i,i}$ instead of its "true" current state S_i. This non-zero delay from S_i to $S_{i,i}$ bears the potential for metastability, as a successful state transition can only be guaranteed if the transition guard remains stable during this delay at least. Hence, we define that node $i \in V$ is *metastability-free during* $[t^-, t^+]$ *with respect to one of its state machines* M, iff for any time $t \in [t^-, t^+]$ when i switches to a state s of M, the infimum t' of times in $(t, t^+]$ when i switches to some state s' of M satisfies $t' > \tau_{i,i}(t)$.

3 The FATAL Pulse Synchronization Protocol

In this section, we present our self-stabilizing pulse generation algorithm, which will be stated in terms of state machines, as introduced in the previous section. Its goal is to generate synchronized, well-separated pulses that occur upon switching to a distinguished state *accept*. For a set of nodes $W \subseteq V$ and a set E of channels, we formally define an algorithm to be a (W, E)*-stabilizing pulse synchronization protocol with skew* Σ *and accuracy bounds* T^-, T^+ *stabilizing within time* T *with probability* p iff the following holds: Given that during $[t^-, t^+] \supseteq [t^-, t^- + T + \Sigma]$ nodes in W are non-faulty and channels in E correct, with probability at least p a time $t_s \in [t^-, t^- + T]$ exists so that, denoting by $t_i(k)$ the k^{th} time when node i switches to *accept* after t_s ($t_i(k) = \infty$ if no such time exists), for all $i, j \in W$, and $k \in \mathbb{N}$, *(i)* $t_i(1) \in (t_s, t_s + \Sigma)$, *(ii)* $|t_i(k) - t_j(k)| \leq \Sigma$ if $\max\{t_i(k), t_j(k)\} \leq t^+$, and *(iii)* $T^- \leq |t_i(k+1) - t_i(k)| \leq T^+$ if $t_i(k) + T^+ \leq t^+$.

Since the ultimate goal of the pulse generation algorithm is to stabilize a system of DARTS clocks, we introduce an additional port DARTS_i, for each node i, which is driven by node i's DARTS instance. As for other state signals, its output raises flag $\text{Mem}_{i,\text{DARTS}}$, to which for simplicity we refer to as DARTS_i as well. Note that the DARTS signals are of no concern to the liveliness or stabilization of the pulse algorithm itself; rather, they are control signals from the DARTS components that help in adjusting the frequency of pulses to the speed of the DARTS clocks once the system as a whole has become stable. Details can be found in [9].

Basic Cycle. The full algorithm makes use of a rather involved interplay between conditions on timeouts, states, and thresholds to converge to a safe state despite a limited number of faulty components. As our approach is difficult to present in a bulk, we break it down into pieces. Moreover, to facilitate giving intuition about the key ideas of the algorithm, in this section we assume that there are $f < n/3$ faulty nodes, and all the remaining $n - f$ nodes are non-faulty

[6] Note that a single physical fault may cause such a behaviour.

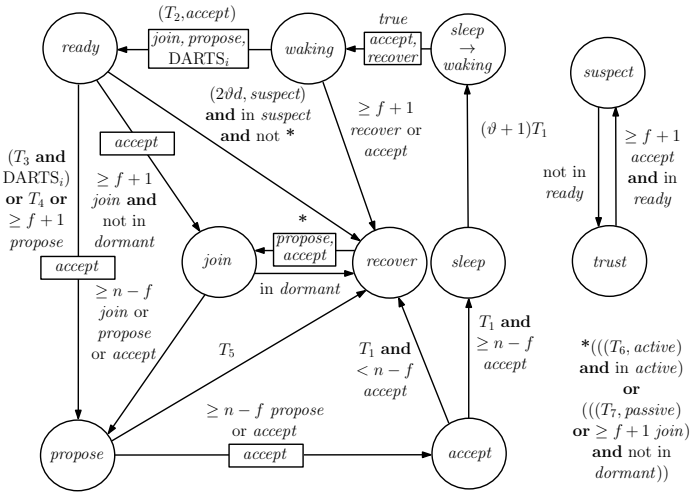


Fig. 1. Overview of the core routine of node i's self-stabilizing pulse algorithm

within $[0, \infty)$ (where of course the time 0 is unknown to the nodes). We further assume that channels between non-faulty nodes (including loopback channels) are correct within $[0, \infty)$. First, we present the *basic cycle* that is repeated every pulse once a safe configuration is reached. It consists of the states *accept*, *sleep*, *sleep* → *waking*, *waking*, *ready*, and *propose* in the given order (see Fig. 1).

We employ graphical representations of the state machine of each node $i \in V$. States are represented by circles containing their names, while transition $(s, s') \in \mathcal{T}$ is depicted as an arrow from s to s'. The guard $tr(s, s')$ is written as a label next to the arrow, and the reset function's value $re(s, s')$ is depicted in a rectangular box on the arrow. To keep labels simple we make use of abbreviations. We write T instead of (T, s) if s is the state that node i leaves if the condition involving (T, s) is satisfied. Threshold conditions like "$\geq f + 1$ s", where $s \in \mathbb{S}$, abbreviate Boolean predicates that reach over all of node i's memory flags $\mathrm{Mem}_{i,j,s}$, where $j \in V$, and are defined in a straightforward manner. If in such an expression we connect two states by "or", e.g., "$\geq n - f$ s or s'" for $s, s' \in \mathbb{S}$, the summation considers flags of both types s and s'. Thus, it is equivalent to $\sum_{j \in V} \max\{\mathrm{Mem}_{i,j,s}, \mathrm{Mem}_{i,j,s'}\} \geq f + 1$. For any state $s \in \mathbb{S}$, the condition $S_{i,j} = s$, (respectively, $\neg(S_{i,j} = s)$) is written in short as "j in s" (respectively, "j not in s"). If $j = i$, we simply write "(not) in s". We write "true" instead of a condition that is always true. Finally, $re(\cdot, \cdot)$ always requires to reset all memory flags of certain types, hence we write, for example, *propose* if all flags $\mathrm{Mem}_{i,j,propose}$ are to be reset.

We now briefly introduce the basic flow of the algorithm once it stabilizes, i.e., once all $n - f$ non-faulty nodes are well-synchronized. Recall that the remaining up to $f \leq \lfloor n/3 \rfloor$ faulty nodes may produce arbitrary signals on their outgoing channels. A pulse is locally triggered by switching to state *accept*. Thus, assume that at some time all non-faulty nodes switch to state *accept* within a time

window of $2d$, i.e., a valid pulse is generated. Supposing that $T_1 \geq 3\vartheta d$, these nodes will observe, and thus memorize, each other and themselves in state *accept* before T_1 expires. This makes timeout T_1 the critical condition for switching to state *sleep*. From state *sleep*, they will switch to states *sleep* $\rightarrow$ *waking*, *waking*, and finally *ready*, where the timeout (T_2, accept) is determining the time this takes, as it is considerably larger than $\vartheta(\vartheta+2)T_1$. The intermediate states serve the purpose of achieving stabilization, hence we leave them out for the moment. Note that upon switching to state *ready*, nodes reset their *propose* flags and DARTS_i. Thus, they essentially ignore these signals between the most recent time they switched to *propose* before switching to *accept* and the subsequent time when they switch to *ready*. This ensures that nodes do not take into account outdated information for the decision when to switch to state *propose*. Hence, it is guaranteed that the first node switching from state *ready* to state *propose* again does so because T_4 expired or because T_3 expired and its DARTS memory flag is true. Due to the constraint $\min\{T_3, T_4\} \geq \vartheta(T_2 + 4d)$, we are sure that all non-faulty nodes observe themselves in state *ready* before the first one switches to *propose*. Hence, no node deletes information about nodes that switch to *propose* again after the previous pulse. No non-faulty node can switch to state *accept* before it memorizes at least $n - f$ nodes in state *propose*, as the *accept* flags are reset upon switching to state *waking*. Therefore, at least $n - 2f \geq f + 1$ non-faulty nodes are in state *propose* when the first node switches to *accept* again. Hence, the rule that nodes switch to *propose* if they memorize $f + 1$ nodes in states *propose* will take effect, i.e., the remaining non-faulty nodes in state *ready* switch to *propose* after less than d time. Another d time later all non-faulty nodes in state *propose* will have become aware of this and switch to state *accept* as well, as the threshold of $n - f$ nodes in states *propose* or *accept* is reached. Thus the cycle is complete and the reasoning can be repeated inductively.

Main Algorithm. We proceed by describing the main routine of the pulse algorithm in full. Alongside the main routine, several other state machines run concurrently and provide additional information to be used during recovery.

The main routine is graphically presented in Fig. 1, together with a very simple second component whose sole purpose is to simplify the otherwise overloaded description of the main routine. Except for the states *recover* and *join* and additional resets of memory flags, the main routine is identical to the basic cycle. The purpose of the two additional states is the following: Nodes switch to state *recover* once they detect that something is wrong, that is, non-faulty nodes do not execute the basic cycle as outlined in Section 3. This way, non-faulty nodes will not continue to confuse others by sending for example state signals *propose* or *accept* despite clearly being out-of-sync. There are various consistency checks that nodes perform during each execution of the basic cycle. For example, no non-faulty node may be in state *propose* for more than a certain amount of time before switching to state *accept*. Therefore, nodes will switch from *propose* to *recover* when timeout T_5 expires. Similarly, when in state *ready*, nodes expect others not to be in state *accept* for more than a short period of time, as a non-faulty node switching to *accept* should imply that every non-faulty node switches to *propose* and then

to *accept* shortly thereafter. This is expressed by the second state machine comprising two states only. If a node is in state *ready* and memorizes $f + 1$ nodes in state *accept*, it switches to *suspect*. Subsequently, if it remains in state *ready* until a timeout of $2\vartheta d$ expires, it will switch to state *recover*.

Nodes can join the basic cycle again via the second new state, called *join*. Since the Byzantine nodes may "play nice" towards $f + 1$ or more nodes still executing the basic cycle, making them believe that system operation continues as usual, it must be possible to join the basic cycle again without having a majority of nodes in state *recover*. On the other hand, it is crucial that this happens in a sufficiently well-synchronized manner, as otherwise nodes could drop out again because the various checks of consistency detect an erroneous execution of the basic cycle.

In part, this issue is solved by an additional agreement step. In order to enter the basic cycle again, nodes need to memorize $n - f$ nodes in states *join* (the respective nodes detected an inconsistency), *propose* (these nodes continued to execute the basic cycle), or *accept* (there are executions where nodes reset their *propose* flags because of switching to *join* when other nodes already switched to *accept*). Since there are thresholds of $f + 1$ nodes memorized in state *join* both for leaving state *recover* and switching from *ready* to *join*, all nodes will follow the first one switching from *join* to *propose* quickly, just as with the switch from *propose* to *accept* in an ordinary execution of the basic cycle. However, it is decisive that all nodes are in states that permit to participate in this agreement step in order to guarantee success of this approach.

As a result, still a certain degree of synchronization needs to be established beforehand, both among nodes that still execute the basic cycle and those that do not. For instance, if at the point in time when a majority of nodes and channels become non-faulty, some nodes already memorize nodes in *join* that are not, they may switch to state *join* and subsequently *propose* prematurely, causing others to have inconsistent memory flags as well. Again, Byzantine faults may sustain this amiss configuration of the system indefinitely.

So why did we put so much effort in "shifting" the focus to this part of the algorithm? The key advantage is that nodes outside the basic cycle may take into account less reliable information for stabilization purposes. They may take the risk of metastable upsets (as we know it is impossible to avoid these during the stabilization process, anyway) and make use of randomization. In fact, to make the above scheme work, it is sufficient that all non-faulty nodes agree on a point in time to reset the memory flags for states *join* and *sleep* $\rightarrow$ *waking* as well as certain timeouts, while guaranteeing that no node is in these states close to the respective reset times. Except for state *sleep* $\rightarrow$ *waking*, all of these timeouts, memory flags, etc. are not part of the basic cycle at all, thus nodes may enforce consistent values for them when they agree on such a *resynchronization point*. Conveniently, the use of randomization also ensures that it is quite unlikely that nodes are in state *sleep* $\rightarrow$ *waking* close to a resynchronization point, as the consistency check of having to memorize $n - f$ nodes in state *accept* in order to switch to state *sleep* guarantees that the time windows during which non-faulty nodes may switch to *sleep* make up a small fraction of all times only.

Consequently, the remaining components of the algorithm deal with agreeing on resynchronization points and utilizing this information in an appropriate way to ensure stabilization of the main routine. We describe this connection to the main routine first. It is done by another, quite simple state machine, which runs in parallel alongside the core routine. It is the machine having three states that is depicted in the upper left corner of Fig. 2.

Its purpose is to reset memory flags in a consistent way and to determine when a node is permitted to switch to *join*. In general, a resynchronization point (locally observed by switching to state *resync*) triggers the reset of the *join* and *sleep* $\rightarrow$ *waking* flags. If there are still nodes executing the basic cycle, a node may become aware of it by observing $f+1$ nodes in state *sleep* $\rightarrow$ *waking* at some time. In this case it switches from state *passive*, which it entered at the point in time when it locally observed the resynchronization point, to state *active*, which enables an earlier transition to state *join*. This is expressed by the rather involved transition rule $tr(recover, join)$: T_6 is much smaller than T_7, but T_6 is of no concern until the node switches to state *active* and resets T_6.[7]

It remains to explain how nodes agree on resynchronization points.

Resynchronization Algorithm. The resynchronization routine is specified in Fig. 2 as well. It is a lower layer that the core routine uses for stabilization purposes only. It provides some synchronization that is very similar to that of a pulse, except that such "weak pulses" occur at random times, and may be generated inconsistently after the algorithm as a whole has stabilized. Since the main routine operates independently of the resynchronization routine once the system has stabilized, we can afford the weaker guarantees of the routine: If it succeeds in generating a "good" resynchronization point merely once, the main routine will stabilize deterministically.

Definition 1 (Resynchronization Points). *Given $W \subseteq V$, time t is a W-*resynchronization point *iff each node in W switches to state* supp $\rightarrow$ resync *in the time interval $(t, t + 2d)$. A W-resynchronization point is called* good *if no node from W switches to state* sleep *during $(t - (\vartheta + 3)T_1, t)$ and no node is in state* join *during $[t - T_1 - d, t + 4d)$.*

In order to clarify that despite having a linear number of states ($supp_i$, $i \in V$), this routine can be implemented using constant-bit channels only, we generalize our description of state machines as follows. If a state is depicted as a circle separated into an upper and a lower part, the upper part denotes the local state, while the lower part indicates the signal state to which it is mapped. A node's memory flags then store the respective signal states only, i.e., remote nodes do not distinguish between states that share the same signal.

The basic idea behind the resynchronization algorithm is the following: Every now and then, nodes will try to initiate agreement on a resynchronization point. This is the purpose of the small state machine in the lower left corner of Fig. 2. As

[7] The condition "not in *dormant*" ensures that the transition is not performed because of being in state *resync* a long time ago, while there was no recent switch to *resync*.

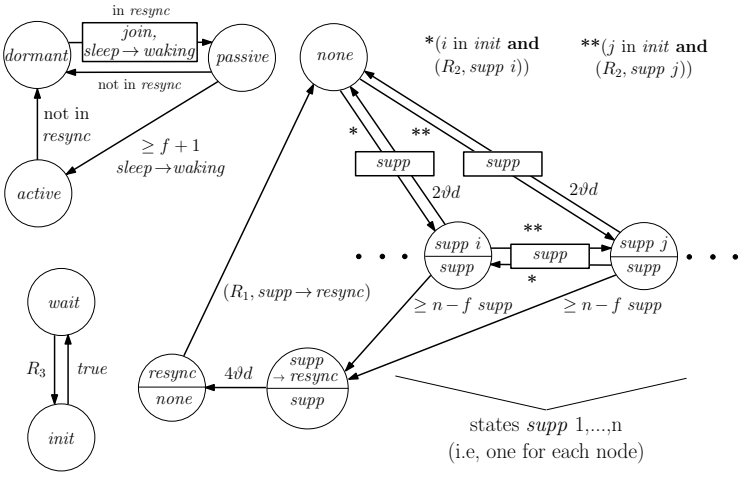


Fig. 2. Resynchronization algorithm of node i

the time when a node switches to *init* is determined by the randomized timeout R_3, which we choose to be distributed over a large interval, it is impossible to predict when it will expire, even with full knowledge of the execution up to the current point in time.

Consider now the state machine displayed on the right of Fig. 2. To understand how the routine is intended to work, assume that at the time t when a non-faulty node i switches to state *init*, all non-faulty nodes are not in any of the states *supp* $\rightarrow$ *resync*, *resync*, or *supp* i, and at all non-faulty nodes the timeout $(R_2, supp\ i)$ has expired. Then, no matter what the signals from faulty nodes or on faulty channels are, all non-faulty nodes will be in one of the states *supp* j, $j \in V$, or *supp* $\rightarrow$ *resync* at time $t + d$. Hence, they will observe each other (and themselves) in one of these states at some time smaller than $t + 2d$. These statements follow from the various timeout conditions of at least $2\vartheta d$ and the fact that observing node i in state *init* will make nodes switch to state *supp* i if in *none* or *supp* j, $j \neq i$. Hence, all of them will switch to state *supp* $\rightarrow$ *resync* during $(t, t + 2d)$, i.e., t is a resynchronization point. Since t follows a random distribution that is independent of the remaining algorithm and, as mentioned earlier, most of the times nodes cannot switch to state *sleep* and it is easy to deal with the condition on *join* states, there is a large probability that t is a good resynchronization point. Note that timeout R_1 makes sure that no non-faulty node will switch to *supp* $\rightarrow$ *resync* again anytime soon, leaving sufficient time for the main routine to stabilize.

The scenario we just described relies on the fact that at time t no node is in state *supp* $\rightarrow$ *resync* or state *resync*. We will choose $R_2 \gg R_1$, implying that $R_2 + 3d$ time after a node switched to state *init* all nodes have "forgotten" about this, i.e., $(R_2, supp\ i)$ is expired and they switched back to state *none* (unless other *init* signals interfered). Thus, in the absence of Byzantine faults, the above requirement is easily achieved with a large probability by choosing R_3 as a uniform distribution over some interval $[R_2 + 3d, R_2 + \Theta(nR_1)]$: Other

nodes will switch to *init* $\mathcal{O}(n)$ times during this interval, each time "blocking" other nodes for at most $\mathcal{O}(R_1)$ time. If the random choice picks any other point in time during this interval, a resynchronization point occurs. Even if the clock speed of the clock driving R_3 is manipulated in a worst-case manner (affecting the density of the probability distribution with respect to real time by a factor of at most ϑ), we can just increase the size of the interval to account for this.

However, what happens if only *some* of the nodes receive an *init* signal due to faulty channels or nodes? If the same holds for some of the subsequent *supp* signals, it might happen that only a fraction of the nodes reaches the threshold for switching to state *supp* $\rightarrow$ *resync*, resulting in an inconsistent reset of flags and timeouts across the system. Until the respective nodes switch to state *none* again, they will not support a resynchronization point again, i.e., about R_1 time is "lost". This issue is the reason for the agreement step and the timeouts $(R_2, \textit{supp } j)$. In order for any node to switch to state *supp* $\rightarrow$ *resync*, there must be at least $n - 2f \geq f + 1$ non-faulty nodes supporting this. Hence, all of these nodes recently switched to a state *supp* j for some $j \in V$, resetting $(R_2, \textit{supp } j)$. Until these timeouts expire, $f + 1 \in \Omega(n)$ non-faulty nodes will ignore *init* signals on the respective channels. Since there are $\mathcal{O}(n^2)$ channels, it is possible to choose $R_2 \in \mathcal{O}(nR_1)$ such that this may happen at most $\mathcal{O}(n)$ times in $\mathcal{O}(n)$ time. Playing with constants, we can pick $R_3 \in \mathcal{O}(n)$ maintaining that still a constant fraction of the times are "good" in the sense that R_3 expiring at a non-faulty node will result in a good resynchronization point.

Analysis and Results. Due to lack of space, we had to relegate the detailed formalization and analysis of our algorithm to [9]. We will hence only briefly summarize our major results, along with the required constraints. With $\lambda := \sqrt{(25\vartheta - 9)/(25\vartheta)} \in (4/5, 1)$, we need:

$$
\begin{array}{lll}
T_1 & \geq & \vartheta 4d \\
T_3 & \geq & (2\vartheta^2 + 3\vartheta - 1)T_1 - T_2 + \vartheta(T_6 + 5d) \\
T_5 & \geq & (\vartheta^2 + \vartheta - 2)T_1 + \vartheta(T_2 + T_4 + 9d) - T_6 \\
T_7 & \geq & (\vartheta - 1)T_2 + \vartheta((1 + 2/\vartheta - \vartheta)T_1 + T_4 + T_5 + T_6 + 10d) \\
R_2 & \geq & 2\vartheta(R_1 + (\vartheta + 2)T_1 + T_2/\vartheta + (8\vartheta + 9)d)(n - f)/(1 - \lambda) \\
\vartheta\lambda & \leq & (T_2 - (4\vartheta^3 + 28\vartheta^2 + 4\vartheta)d)/(T_2 - (8\vartheta^2 + \vartheta)d)
\end{array}
\qquad
\begin{array}{lll}
T_2 & \geq & (3\vartheta + 1 - 1/\vartheta)T_1 + T_5 \\
T_4 & \geq & T_3 \\
T_6 & \geq & \vartheta((\vartheta + 1)T_1 + T_2 + 6d) \\
R_1 & \geq & \vartheta(T_7 + (4\vartheta + 8)d) \\
R_3 & = & \text{uniformly distributed on} \\
 & & [\vartheta(R_2 + 3d), \vartheta(R_2 + 3d) + 8(1 - \lambda)R_2]
\end{array}
$$

This system is solvable for any $\vartheta < \vartheta_{\max} \approx 1.247$ with $T_1, \ldots, T_7, R_1 \in \mathcal{O}(1)$ and $R_2 \in \mathcal{O}(n)$. Furthermore, the system must satisfy the following property during a given time interval $[t^-, t^+]$: There is a subset $W \subseteq V$ of size at least $n - \lfloor n/3 - 1 \rfloor$ such that during $[t^- - (\vartheta(R_2 + 3d) + 8(1 - \lambda)R_2) - d, t^+]$ (i) all nodes $i \in W$ are non-faulty, and (ii) all channels $S_{i,j}$, $i, j \in W$, are correct.

A safe configuration is reached once all nodes in W switch to *accept* within $3d$: Time t is a *stabilization point (quasi-stabilization point)* iff all nodes $i \in W$ switch to *accept* within $[t, t + 2d)$ ($[t, t + 3d)$).

Theorem 1. *Suppose t is a quasi-stabilization point. Then (i) all nodes in W switch to* accept *exactly once within $[t, t+3d)$, and (ii) there will be a stabilization point $t' \in (t+(T_2+T_3)/\vartheta, t+T_2+T_4+5d)$ satisfying that no node in W switches to* accept *in the time interval $[t+3d, t')$ and that (iii) each node i's, $i \in W$, states of the basic cycle (*accept*,* sleep*,* sleep $\rightarrow$ waking*,* waking*,* ready*, and* propose*) are metastability-free during $[t + 3d, t' + 4d)$.*

By induction, we see that if for $|W| \geq n - \lfloor n/3 - 1 \rfloor$ we can show the existence of a quasi-stabilization point $t \in [t^-, t^+]$, then for any $E \supseteq W^2$ the protocol is (W, E)-stabilizing with skew $2d$ and accuracy bounds $(T_2 + T_3)/\vartheta - 2d$ and $T_2 + T_4 + 7d$. As it is impossible to guarantee metastability-freedom during stabilization, our remaining statements argue about metastability-free executions only, i.e., provided that metastability does not occur, the system will stabilize.

Theorem 2. *Denote by* $\hat{E}_3 := \vartheta(R_2 + 3d) + 8(1 - \lambda)R_2 + d$. *For any* $k \in \mathbb{N}$ *and any time* $t \in [t^-, t^+ - (k+1)\hat{E}_3]$, *with probability at least* $1 - (1/2)^{k(n-f)}$ *there will be a good* W*-resynchronization point in* $[t, t + (k+1)\hat{E}_3]$.

A good resynchronization point ensures sufficient consistency of nodes' memories for the main routine (Fig. 1) to stabilize deterministically.

Theorem 3. *Let* $T_k := (k+2)\hat{E}_3 + R_1/\vartheta$ *for* $k \in \mathbb{N}$. *Then, with probability at least* $1 - 1/2^{k(n-f)}$, *a stabilization point in* $[t^-, t^- + T_k]$ *exists, and the algorithm stabilizes within time* T_k *plus longest timeout.*

References

1. Ben-Or, M., Dolev, D., Hoch, E.N.: Fast self-stabilizing byzantine tolerant digital clock synchronization. In: Proc. 27th symposium on Principles of Distributed Computing (PODC), pp. 385–394 (2008)
2. Berman, A., Keidar, I.: Low-Overhead Error Detection for Networks-on-Chip. In: The 27th International Conference on Computer Design, ICCD (2009)
3. Bhamidipati, R., Zaidi, A., Makineni, S., Low, K., Chen, R., Liu, K.Y., Dalgrehn, J.: Challenges and Methodologies for Implementing High-Performance Network Processors. Intel Technology Journal 6(3), 83–92 (2002)
4. Chapiro, D.M.: Globally-Asynchronous Locally-Synchronous Systems. Ph.D. thesis, Stanford University (1984)
5. Constantinescu, C.: Trends and Challenges in VLSI Circuit Reliability. IEEE Micro 23(4), 14–19 (2003)
6. Daliot, A., Dolev, D.: Self-Stabilizing Byzantine Pulse Synchronization. CoRR abs/cs/0608092 (2006)
7. Daliot, A., Dolev, D., Parnas, H.: Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 32–48. Springer, Heidelberg (2003)
8. Dike, C., Burton, E.: Miller and Noise Effects in a Synchronizing Flip-Flop. IEEE Journal of Solid-State Circuits SC 34(6), 849–855 (1999)
9. Dolev, D., Függer, M., Lenzen, C., Schmid, U.: Fault-tolerant Algorithms for Tick-Generation in Asynchronous Logic: Robust Pulse Generation. CoRR abs/cs/1105.4708 (2011)
10. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. Journal of the ACM 51(5), 780–799 (2004)
11. Friedman, E.G.: Clock Distribution Networks in Synchronous Digital Integrated Circuits. Proceedings of the IEEE 89(5), 665–692 (2001)
12. Fuchs, G., Függer, M., Steininger, A.: On the Threat of Metastability in an Asynchronous Fault-Tolerant Clock Generation Scheme. In: Proc. 15th Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 127–136. Chapel Hill, N. Carolina (2009)

13. Függer, M.: Analysis of On-Chip Fault-Tolerant Distributed Algorithms. Ph.D. thesis, Technische Universität Wien, Institut für Technische Informatik (2010)
14. Függer, M., Dielacher, A., Schmid, U.: How to Speed-Up Fault-Tolerant Clock Generation in VLSI Systems-on-Chip via Pipelining. In: Proc. 8th European Dependable Computing Conference (EDCC), pp. 230–239 (2010)
15. Függer, M., Schmid, U.: Reconciling Fault-Tolerant Distributed Computing and Systems-on-Chip. Research Report 13/2010, Technische Universität Wien, Institut für Technische Informatik (2010)
16. Függer, M., Schmid, U., Fuchs, G., Kempf, G.: Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In: Proc. 6th European Dependable Computing Conference (EDCC), pp. 87–96 (2006)
17. Gadlage, M.J., Eaton, P.H., Benedetto, J.M., Carts, M., Zhu, V., Turflinger, T.L.: Digital Device Error Rate Trends in Advanced CMOS Technologies. IEEE Transactions on Nuclear Science 53(6), 3466–3471 (2006)
18. Hoch, E., 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)
19. Internat. Technology Roadmap for Semiconductors (2007), http://www.itrs.net
20. Kinniment, D.J., Bystrov, A., Yakovlev, A.V.: Synchronization Circuit Performance. IEEE Journal of Solid-State Circuits SC 37(2), 202–209 (2002)
21. Malekpour, M.: 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)
22. Marino, L.: General Theory of Metastable Operation. IEEE Transactions on Computers C 30(2), 107–115 (1981)
23. Metra, C., Francescantonio, S., Mak, T.: Implications of Clock Distribution Faults and Issues with Screening them During Manufacturing Testing. IEEE Transactions on Computers 53(5), 531–546 (2004)
24. Pease, M., Shostak, R., Lamport, L.: Reaching Agreement in the Presence of Faults. Journal of the ACM 27, 228–234 (1980)
25. Polzer, T., Handl, T., Steininger, A.: A Metastability-Free Multi-synchronous Communication Scheme for SoCs. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 578–592. Springer, Heidelberg (2009)
26. Portmann, C.L., Meng, T.H.Y.: Supply Noise and CMOS Synchronization Errors. IEEE Journal of Solid-State Circuits SC 30(9), 1015–1017 (1995)
27. Restle, P.J., et al.: A Clock Distribution Network for Microprocessors. IEEE Journal of Solid-State Circuits 36(5), 792–799 (2001)
28. Semiat, Y., Ginosar, R.: Timing Measurements of Synchronization Circuits. In: Proc. 9th Symposium on Asynchronous Circuits and Systems, ASYNC (2003)
29. Srikanth, T.K., Toueg, S.: Optimal Clock Synchronization. Journal of the ACM 34(3), 626–645 (1987)
30. Sundaresan, K., Allen, P., Ayazi, F.: Process and temperature compensation in a 7-MHz CMOS clock oscillator. IEEE J. Solid-State Circuits 41(2), 433–442 (2006)
31. Teehan, P., Greenstreet, M., Lemieux, G.: A Survey and Taxonomy of GALS Design Styles. IEEE Design and Test of Computers 24(5), 418–428 (2007)
32. Widder, J., Schmid, U.: The Theta-Model: Achieving Synchrony without Clocks. Distributed Computing 22(1), 29–47 (2009)

The South Zone: Distributed Algorithms for Alliances

M.C. Dourado[1], L.D. Penso[2], D. Rautenbach[2], and J.L. Szwarcfiter[1]

[1] Instituto de Matemática, NCE, and COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brazil
{mitre,jayme}@nce.ufrj.br
[2] Institut für Optimierung und Operations Research, Fakultät für Mathematik und Wirtschaftswissenschaften, Universität Ulm, Ulm, Germany
{lucia.penso,dieter.rautenbach}@uni-ulm.de

Abstract. We present novel results on and efficient deterministic as well as randomized synchronous message-passing distributed algorithms for generalized graph alliances, a new concept incorporating and expanding previous ones. An alliance is here a group of nodes of a connected network or a population fulfilling certain thresholds for their neighbourhood. More precisely, every node outside and inside the alliance must have a minimum number of neighbours inside the alliance. A threshold function defining this number may be specific to each node. We are interested in finding minimal alliances of generalized type: the threshold function might be any. We also investigate conditions in which it is possible to have anonymity, a praised property in population protocols.

Keywords: graph alliances, distributed algorithms, minimality, population protocols.

1 Introduction

In this paper we investigate the problem of determining minimal alliances in connected networks with bidirectional communication channels. We represent such alliances in networks by special sets of nodes of finite and simple connected graphs whose nodes are linked by undirected edges denoting processors. This problem comprises classical well-known problems as dominating sets [18, 20] in networks and arises every time it remains crucial for every node to have a minimum number of neighbours inside a group of nodes called the alliance. This number may be specific to each node. Thus, alliances expand the notion of dominating sets: for an alliance to be a dominating set, it suffices for each node outside the alliance to have at least a neighbour inside the alliance.

We introduce a new definition of alliance which incorporates and expands previous ones: (f, g)-alliance. More specifically, every node outside and inside an (f, g)-alliance must have a minimum number of neighbours inside the alliance. This number is defined by the threshold function f for nodes outside it and by the threshold function g for nodes inside it. Some examples of such alliances

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 178–192, 2011.

include previously studied ones such as the global defensive and global offensive alliances [17, 26]. The global defensive alliance sets f to 1 and g equal to the majority of neighbours, while the global offensive alliance sets f equal to the majority of neighbours and g to 0.

We direct our attention to distributed algorithms which enable us to identify those structural sets in an autonomic manner. Whenever possible, anonymity is adopted. To that end, we make use of randomization. We are mainly interested in retrieving minimal (f, g)-alliances.

Alliances have several applications which range from problems of population protocols [2] such as voting [21] and spread of disease [10] to server allocation in computer networks [13, 14] and replica caching in database and operating systems [15, 16]. Encountering such minimal structural sets is thus of utmost importance for a variety of problems in distributed computing, in particular those concerning population protocols [1] and optimal object placement [4].

Related Work

Here we concentrate on finding minimal alliances instead of minimum ones, as this is known to be a NP-complete task for many existing types of alliances [12, 17]. Though our alliance concept is here extended, one may easily construct a variation of the proofs of [12, 17], which only consider restricted classes of alliances.

In [26] Srimani and Xu use the classical self-stabilization paradigm proposed by Dolev, Dijkstra and Schneider [8, 9, 25] to derive protocols for two specific types of alliances: defensive and offensive alliances. Though efficient, both algorithms require a sequential model of computation which is neither autonomic nor distributed: an extern central daemon schedules one node at a time cyclically, that is, exactly one in each round according to some total ordering. To derive a distributed version, a transformer is required, but then both algorithms must be id-based. Not surprising as a total ordering of the nodes is needed. Besides, the defensive alliance algorithm provides only a 1-minimal alliance, which is not necessarily minimal, since the only given guarantee is that the removal of one single node makes the set not a defensive alliance anymore. Our work, however, has a total new approach and it is focused on finding minimal sets of (anonymous, if possible) nodes which correspond to a new concept expanding while including previous ones at the same time. Hence, instead of looking at self-stabilization algorithms for two particular types of alliances such as in [26], we focus on synchronous distributed algorithms specially built to discover minimal (f, g)-alliances, in an anonymous environment whenever possible.

There are also some special numbers which have a very close and direct relation to the structural properties presented by alliances. For instance, the minimum size of an (f, g)-alliance with $f = k$ and $g = 0$ for every node is known as the k-domination number, and was previously studied by researchers in [24]. There one may also verify results and references on the k-tuple domination number, which corresponds to a variation of an (f, g)-alliance where $f = k$ and $g = k - 1$.

Other examples of related special numbers are studied within the context of population protocols such as voting and spread of disease. For instance, in

[5, 6, 7], the authors define two numbers, $irr_f(G)$ and $r_f(G)$, which comprise the minimum cardinality of a set so that every other node converges to the very same initial proposed value (equal to 1) by that set. If a node is not allowed to change after converging to this value, then $irr_f(G)$ determines the minimum cardinality, otherwise $r_f(G)$ does. Clearly, the connectivity of nodes outside the set to nodes inside the set plays here a major role, just as f in (f, g)-alliances.

Our Contributions
We introduce a novel alliance definition which generalizes previous notions: the (f, g)-alliance. It relies on two threshold functions f and g which characterize the required number of neighbours inside the alliance. The new concept incorporates old ones, while broadening the possibilites. More precisely, we present in the next sections:

- an extended alliance definition called the (f, g)-alliance
- a sufficient property for minimal (f, g)-alliances
- an anonymous quadratic-time deterministic synchronous message-passing distributed algorithm to find some (f, g)-alliance for any f and any g
- an id-based quadratic-time deterministic synchronous message-passing distributed algorithm to find a minimal (f, g)-alliance for any f and any g
- an id-based linear-time deterministic synchronous message-passing distributed algorithm to find a minimal (f, g)-alliance where $f \geq g$
- an anonymous randomized synchronous message-passing distributed algorithm to find a minimal (f, g)-alliance where $f \geq g$

Our set of results shows how beautiful (f, g)-alliances might be.

Outline of Paper
In Section 2 we introduce our definitions and specify the model of computation. We also prove there a sufficient property for minimal (f, g)-alliances. In Section 3, we introduce an efficient quadratic id-based deterministic distributed algorithm for finding minimal (f, g)-alliances for any f and any g. In Section 4 we discuss why the results from Srimani and Xu [26] have their own limitations and show why deterministic distributed message-passing algorithms for minimal (f, g)-alliances cannot be anonymous if f and g are any, or even under the restriction $f \geq g$. This result is due to the minimality requirement, as verified previously in Section 3: an anonymous quadratic deterministic distributed algorithm exists if we simply want to find any (f, g)-alliance instead of a minimal one. In Section 5 we present a faster linear id-based deterministic distributed algorithm for finding minimal (f, g)-alliances with $f \geq g$ (thus including global offensive alliances as well as any alliance with $g = 0$). This algorithm is a warm-up for the next one in Section 6, where we finally show an anonymous randomized distributed algorithm for finding minimal (f, g)-alliances with $f \geq g$. We close with Section 7 where conclusions are drawn and some open problems are suggested.

2 Model and Definitions

We consider connected networks represented by graphs whose nodes denote processors and whose edges denote bidirectional communication links. The model of

computation is here a message-passing synchronous one: Each node operates in a round of same time size, computing locally, sending messages to its neighbours and receiving messages from its neighbours.

An *alliance* is nothing more than a group of nodes with enough influence outside and inside the group: There are enough neighbours belonging to the group for each node outside the group as well as for each node inside the group. Thus, an alliance comprises the key set of well-connected nodes for a successful performance. This notion can be precisely defined, as follows.

Let G be a finite, simple, connected and undirected graph, $V(G)$ be the set of vertices of G, $E(G)$ be the set of edges of G, $n(G) = |V(G)| = n$ be the order of G and $D(G)$ be the diameter of G. Let $N_G(u)$ be the open neighbourhood of vertex u of G (that is, u itself is not included) and $d_G(u) = |N_G(u)|$ be the degree of vertex u of G. For a set $S \subseteq V(G)$ and a vertex $u \in V(G)$, $d_G(u, S) = |N_G(u) \cap S|$ is defined to be the degree of vertex u regarding S.

The set of all functions $f : A \to B$ from a set A to a set B is here denoted by B^A. If $f, g \in B^A$ and $f(u) \geq g(u)$ for every u in A, then we simply write $f \geq g$. For a pair (f, g) of functions in $\mathbb{N}_0^{V(G)}$, a set $\emptyset \neq S \subseteq V(G)$ is defined to be an (f, g)-*alliance* if

- $d_G(u, S) \geq f(u)$ for every vertex u in $V(G) \setminus S$ *(property 1)* and
- $d_G(u, S) \geq g(u)$ for every vertex u in S *(property 2)*.

Hence, our definition makes use of two threshold functions: the first one f for nodes outside the alliance and the second one g for nodes inside the alliance. Clearly, if $d_G \geq g$, then the set $V(G)$ is itself an (f, g)-alliance for any f.

A *dominating set of a graph* G is a $(1, 0)$-alliance of G. A *global offensive alliance of* G is a $\left(\left\lceil \frac{d_G(u)+1}{2} \right\rceil, 0\right)$-alliance of G. A *global defensive alliance of* G is a $\left(1, \left\lceil \frac{d_G(u)+1}{2} \right\rceil\right)$-alliance of G. Hence, both global offensive and global defensive alliances of G are dominating sets of G. In fact, any (f, g)-alliance is a dominating set of G if and only if $f \geq 1$.

An (f, g)-alliance is *minimal*, if no proper subset is an (f, g)-alliance. Similarly, an (f, g)-alliance S is 1-*minimal*, if there is no vertex u in S for which $S \setminus \{u\}$ is an (f, g)-alliance. Clearly, every minimal (f, g)-alliance is 1-minimal.

In the next, we establish some results about minimal and 1-minimal (f, g)-alliances. Not every 1-minimal (f, g)-alliance is a minimal one. For instance, a global defensive alliance may be 1-minimal but not minimal, as remarked in [26]. However, it is possible to characterize some sufficient property on both threshold functions f and g of the (f, g)-alliance so that this actually holds, as we do in the following.

Lemma 1. *Let G be a graph and let $f, g \in \mathbb{N}_0^{V(G)}$. If $f \geq g$, then a 1-minimal (f, g)-alliance is minimal.*

Proof: Let $f \geq g$ and let S be a 1-minimal (f, g)-alliance. For contradiction, we assume that S is not minimal. Let the (f, g)-alliance S' be a proper subset of S. Let x be a vertex in $S \setminus S'$ and let $S'' = S \setminus \{x\}$. Every vertex u in $V(G) \setminus S''$

belongs to $V(G) \setminus S'$ and hence satisfies $d_G(u, S'') \geq d_G(u, S') \geq f(u)$. Every vertex u in S' satisfies $d_G(u, S'') \geq d_G(u, S') \geq g(u)$. Finally, every vertex u in $S'' \setminus S'$ belongs to $V(G) \setminus S'$ and hence satisfies $d_G(u, S'') \geq d_G(u, S') \geq f(u) \geq g(u)$. Hence S'' is an (f, g)-alliance, which implies the contradiction that S is not 1-minimal. □

The given sufficient property for 1-minimal (f, g)-alliances to be minimal is particularly interesting and useful: it has a practical impact. This result allows an improvement from quadratic to linear in the algorithmic complexity for finding minimal (f, g)-alliances, as to be seen in Section 3 and Section 5.

3 Minimal (f, g)-Alliances

In this section, we show how to efficiently get a minimal (f, g)-alliance out of any given (f, g)-alliance. This reduces the problem of finding efficiently a minimal (f, g)-alliance to the problem of finding efficiently an (f, g)-alliance.

Fortunately, as we discovered, one can design an anonymous pre-processing algorithm to efficiently find an (f, g)-alliance: we establish below how to construct an (f, g)-alliance out of $V(G)$. Of course, this pre-processing algorithm is just needed if it is not known a priori if $V(G)$ itself can be used. For instance, if $d_G \geq g$, then the set $V(G)$ is itself an (f, g)-alliance for any f and may be used as a starting point for encountering a minimal (f, g)-alliance in graph G.

In the next, we present both algorithms, which provide together an id-based quadratic synchronous message-passing distributed algorithms to find minimal (f, g)-alliances for any f and any g. Note that, as from Lemma 2 of Section 4, there is no anonymous deterministic synchronous message-passing distributed algorithm to find minimal (f, g)-alliances if f and g are any, or even if $f \geq g$. However, if the minimality (or the deterministic) requirement is removed, anonymity is possible: take as an example Theorem 1.a in the next (or the randomized algorithm in Theorem 3 of Section 6).

Theorem 1. *Let G be a graph and let $f, g \in \mathbb{N}_0^{V(G)}$.*

a) There is a quadratic-time anonymous synchronous message-passing distributed algorithm (i.e., the Quad_Get_Input_South_Zone) that computes deterministically an (f, g)-alliance contained in $V(G)$.

b) There is a quadratic-time id-based synchronous message-passing distributed algorithm (i.e., the Quad_Valid_Input_South_Zone) that computes deterministically a minimal (f, g)-alliance contained in a given (f, g)-alliance.

Proof: a) In the beginning, each node belongs to set T, initially equal to $V(G)$. At each round, every node $v \in T$ leaves T if and only if $d_G(v, T) < g(v)$. If v leaves T, it sends a broadcast to all other nodes. If at some point a node $v \in V(G) \setminus T$ is such that $d_G(v, T) < f(v)$, the algorithm ends after it broadcasts a message setting $T = \emptyset$: that is, there is no (f, g)-alliance. Otherwise, if this never happens, there is a round in which at every node of $V(G)$ no broadcast is received with

the announcement of the removal of a node from T, and the algorithm ends with T as the (f, g)-alliance. The algorithm is shown in the next.

Every phase is composed of the following two steps:

- [Step 1] *(Initialization, only in the very first phase:)*
  Every vertex v in G sends a message to all its neighbours.
  (So every vertex discovers its degree in the graph. If degrees are considered to be initially known, this step is not necessary.)
  (Initially, $T = V(G)$ and every node in G knows about it, as well as the order of G: $|V(G)| = n(G)$.)
- [Step 2]
  Every $u \in T$ that
  - satisfies $d_G(u, T) < g(u)$

  leaves T and updates all other nodes about its removal from T with a LEFT broadcast message. If no LEFT broadcast message is received during Step 2, all nodes in T send an YES broadcast message. The algorithm ends, and if some YES broadcast message was heard, all nodes know that $S \neq \emptyset$, otherwise all nodes know that $S = \emptyset$.
  Every $u \in V(G) \setminus T$ such that
  - satisfies $d_G(u, T) < f(u)$

  broadcasts a message NO expelling everybody out of T (so that we have $T = \emptyset$) and ending the algorithm.

Clearly, if a node does not satisfy $d_G(v, T) < g(v)$ when T is set to $V(G)$, it must be outside any (f, g)-alliance as $d_G(v, V(G)) \geq d_G(v, T)$. Thus, after the first round, any (f, g)-alliance must be contained in T. This argument is iterated as many times as it gets necessary to satisfy $d_G(v, T) \geq g(v)$ (*property 2*) to every node $v \in T$ or to break $d_G(v, T) \geq f(v)$ (*property 1*) to some node $v \in V(G) \setminus T$. If there is no (f, g)-alliance in G, the algorithm ends with $T = \emptyset$. If there is any (f, g)-alliance S in G, at some round no broadcast annoucing the removal of a node from T is heard and the algorithm ends with $S = T$. Since the algorithm can clearly be implemented to run in quadratic time as a broadcast takes $O(D(G)) \leq O(n)$ time and there are at most n phases, the proof is complete. □

b) Let S be a given (f, g)-alliance. For every vertex $u \in S$, let the set S_u arise from $S \setminus \{u\}$ by iteratively removing vertices v from the current set T_u (starting with $S \setminus \{u\}$) that satisfy $d_G(v, T_u) < g(v)$. Now we compute if there is a proper set of S which is an (f, g)-alliance as well. (Note that the computation to verify if this possible proper set of S belongs to any of the different sets T_u is done simultaneously for every different $u \in V(G)$.) The algorithm is as follows.

Every phase is composed of the following two steps:

- [Step 1] (*Initialization, only in the very first phase:*)
  Every vertex v in G sends to all nodes its ID and if it belongs to S.
  For all $u \in S$, every vertex v in G initially belongs to T_u if it belongs to S and $v \neq u$. Otherwise, it does not.
  (Thus, every vertex v in G for all $u \in S$ knows whether or not it belongs to T_u as well as whether or not every other vertex in G belongs to T_u.)

– [Step 2]
  For every u in $V(G)$, every vertex $v \in V(G) \setminus T_u$ such that
  • satisfies $d_G(v, T_u) < f(v)$
  broadcasts a message ending the algorithm negatively for $S_u = T_u$.
  For every u in $V(G)$, every vertex v in T_u that
  • satisfies $d_G(v, T_u) < g(v)$
  leaves T_u and updates all other nodes about its removal from T_u with a broadcast message.
  If $T_u = \emptyset$, the algorithm ends negatively for $S_u = T_u$.
  If no broadcast message concerning a removal from T_u is received during Step 2, the algorithm ends positively for $S_u = T_u$.

If the algorithm ends positively for S_u^* for some $u^* \in S$, then it is restarted now with $S = S_u^*$ as initial input. That means S_u^* is the new candidate for a minimal (f, g)-alliance.

If it ends negatively for S_u for all $u \in S$, then S is a minimal (f, g)-alliance.

Clearly, every (f, g)-alliance S' that is a subset of $S \setminus \{u\}$ must also be a subset of S_u. Since $d_G(\cdot, S_u) \geq d_G(\cdot, S')$, this implies that $S \setminus \{u\}$ contains an (f, g)-alliance if and only if S_u is an (f, g)-alliance. Altogether, we obtain that if S_u is not an (f, g)-alliance for every u in S, then S is a minimal (f, g)-alliance and if S_u^* is an (f, g)-alliance for some u^* in S, then S is not a minimal (f, g)-alliance. In this last case, to find the minimal (f, g)-alliance we have to iterate the algorithm using S_u^* instead of S and so on. Again, since the algorithm can clearly be implemented to run in quadratic time as a broadcast takes $O(D(G)) \leq O(n)$ time and there are at most n phases, the proof is complete. □

Corollary 1. *Let G be a graph and let $f, g \in \mathbb{N}_0^{V(G)}$. There is an efficient quadratic-time id-based synchronous message-passing distributed algorithm (i.e., the Quad_South_Zone) that computes deterministically a minimal (f, g)-alliance contained in $V(G)$.*

Proof: The proof follows immediately from Theorem 1: the algorithm is given by the combination of the one which finds an (f, g)-alliance out of $V(G)$, namely *Quad_Get_Input_South_Zone* in Theorem1.a, with the one which calculates a minimal (f, g)-alliance given an (f, g)-alliance, namely *Quad_Valid_Input_South_Zone* in Theorem1.b. Though the earlier is anonymous, *Quad_South_Zone* is id-based as the later is eponymous. □

4 Srimani and Xu's Algorithm

Srimani and Xu proposed in [26] two self-stabilizing deterministic distributed algorithms under a weak daemon model called the central daemon [3], where only one processor may execute an atomic step at one time, for the computation of

– a minimal global offensive alliance and in a graph G with anonymous nodes and
– a 1-minimal global defensive alliance in a graph G with nodes uniquely identified.

Both algorithms share the same problem of working only under the central daemon. Furthermore, while there are transformers preserving the self-stabilization property [3] which translate the central daemon into a distributed daemon, where any number of processors may execute an atomic step simultaneously as in a distributed algorithm, those require unique identifiers.

Hence, our objective in the next sections is to present distributed algorithms (that is, under the distributed daemon) to find minimal alliances (instead of 1-minimal) of a more general and broader type ((f, g)-alliances) in network graphs where, if possible, node identifiers are not needed (that is, nodes are anonymous). More precisely, the distributed algorithms are to be run in a message-passing synchronous model of computation. Anonymity is a cherished property in various applications, including in population protocols. As nothing in life is for free, a new element is introduced to achieve that deed: as is to be seen, in the following sections we make use of randomization in order to avoid unique identifiers so that we are able to build such distributed algorithms on anonymous networks.

We will now explain in detail why such a restriction to the central daemon applies for the algorithm computing a minimal global offensive alliance in [26]. A similar argument may be achieved for the 1-minimal global defensive alliance algorithm in [26]. The algorithm maintains a set S and relies on two rules that are iteratively applied by the individual vertices of the graph:

- Rule R1, which causes a vertex u that does not belong to S to join S, and
- Rule R2, which causes a vertex u that belongs to S to leave S.

The proof of the convergence statement (cf. Theorem 2 in [26]) considers the number X of edges uv of G with $|\{u, v\} \cap S| = 1$. The authors claim that one application of R1 does not decrease X while one application of R2 increases X. Since X is bounded by the number $m(G)$ of edges of G, the number of applications of R2 is bounded by $m(G)$ and since an application of R1 increases the cardinality of S by 1, the number of applications of R1 after the last application of R2 is bounded by the number $n(G)$ of vertices.

The problem with this argument is that it implicitly assumes that the nodes act one after the other, i.e. it implicitly assumes some underlying schedule which breaks the symmetry of the nodes. If nodes are allowed to act simultaneously, the claimed properties of R1 and R2 do no longer hold and the algorithm fails. The simplest example is $G = K_2$ and $S = V(G)$. If both vertices of G act simultaneously, they will both apply R1, then R2, then R1, and so forth, i.e. X does not change and the algorithm never stabilizes.

In fact one can easily construct the following result for synchronous distributed systems and daemons, which corresponds to the impossibility of leader election.

Lemma 2. *There is no (self-stabilizing or not) deterministic, anonymous, and synchronous message-passing distributed algorithm for the computation of a minimal global offensive (resp. defensive) alliance.*

Proof: Applied to the complete graph of order n in which all vertices are initially in the same state, the algorithm would have to select a set of order exactly $\lceil \frac{n}{2} \rceil$. Since all vertices are initially in the same state and dispose of the very same local information, they will behave exactly the same and hence all remain in the same state. Therefore, the algorithm can never complete its task. □

5 Faster Id-Based Deterministic Synchronous Distributed Minimal (f, g)-Alliance Algorithm for $f \geq g$

In this section we describe a faster linear id-based deterministic synchronous distributed algorithm that determines a minimal (f, g)-alliance in G for any $f \geq g$. Hence, both global offensive alliances and dominating sets are comprised as special cases. This algorithm, named *Linear_$f \geq g$_South_Zone*, does not need so many iterations as *Quad_South_Zone* from *Corollary 1*, ending up much more earlier, more precisely, in linear time. To that end, we take advantage of the restriction $f \geq g$, under which 1-minimality implies minimality by *Lemma 1*.

A set S that is an (f, g)-alliance is maintained. Initially, S is simply $V(G)$ if $d_G \geq g$, as in the case of global offensive alliances where $g = 0$. If not, then it is not possible to find such an (f, g)-alliance in G, as $f \geq g$, and the algorithm terminates negatively with the empty set. Note that *Quad_Get_Input_South_Zone* may be used for that purpose: if $f \geq g$, it even has a linear complexity due to the existence of only a single phase. (If $f \geq g$, in the first phase, either it terminates with all nodes of $V(G)$ staying in T or it terminates with $T = \emptyset$.) Nevertheless, instead of calling it, we incorporate the necessary checks in the initialization.

We assume that each vertex u of G has an individual ID number $\mathrm{id}(u) \in \mathbb{N}_0$ such that the IDs define a proper colouring of G, i.e. adjacent vertices have distinct IDs. Hence, all ids must not be distinct, but rather there has to be enough different ids to define a colouring.

The algorithm *Linear_$f \geq g$_South_Zone* proceeds in synchronous message-passing rounds which we call here *phases*. In each phase either S is not minimal and at least one vertex u in S leaves S such that $S \setminus \{u\}$ is still an (f, g)-alliance or S is minimal and nothing changes. There are exactly n phases.

Every phase is composed of the following seven steps:

- [Step 1] *(Only in the very first phase:)*
  For every $u \in V(G)$ that
  - satisfies $d_G(u) \geq g(u)$

  sends to all its neighbours its ID number,
  and every $u \in V(G)$ that
  - satisfies $d_G(u) < g(u)$ (and thus, $d_G(u) < f(u)$, as $f \geq g$)

  broadcasts a message ending the algorithm negatively with $S = \emptyset$.
  (If there is no broadcast message terminating the algorithm with $S = \emptyset$, then $S = V(G)$ initially, and all nodes know their neighbours' ID numbers.)

- [Step 2]
  Every vertex $u \in S$ that
  - satisfies $d_G(u, S) \geq f(u)$

  sends to all its neighbours (inside and outside S) a CONSENT message signaling that it wants to leave S.
- [Step 3]
  Every $u \in S$ that
  - satisfies $d_G(u, S) > g(u)$

  sends an OK message to all neighbours from which it received a CONSENT in Step 2.
  Every $u \notin S$ that
  - satisfies $d_G(u, S) > f(u)$

  sends an OK message to all neighbours from which it received a CONSENT in Step 2.
- [Step 4]
  Every $u \in S$ that
  - receives OK messages from all its neighbours (inside and outside S) in Step 3

  sends to all its neighbours (inside and outside S) a NEW_CONSENT message.
- [Step 5]
  Every $u \in S$ sends NEW_OK messages to as most as possible from its $d_G(u, S) - g(u)$ neighbours inside S of smallest ID numbers from whom a NEW_CONSENT message was received in Step 4.
  Every $u \notin S$ sends NEW_OK messages to as most as possible from its $d_G(u, S) - f(u)$ neighbours inside S of smallest ID numbers from whom a NEW_CONSENT message was received in Step 4.
- [Step 6]
  Every $u \in S$ that
  - receives NEW_OK messages from all its neighbours (inside and outside S) in Step 5

  sends to all its neighbours in S its ID number.
- [Step 7]
  Every $u \in S$ that
  - sent its ID number during Step 6 and
  - did not receive during Step 6 an ID number that is smaller than his own

  leaves S and updates all neighbours about its removal from S.
  After n phases, the algorithm ends.

Theorem 2. *Let G be a graph and let $f, g \in \mathbb{N}_0^{V(G)}$. Let S be an (f, g)-alliance in G with $f \geq g$. The algorithm Linear_$f \geq g$_South_Zone is such that:*

a) If S is minimal, then applying the seven steps of a phase in constant time changes nothing (S remains the same) and the algorithm ends in linear time. (Termination)

b) If S is not minimal, then applying the seven steps of a phase in constant time produces a proper subset of S which is itself an (f, g)-alliance in G. (Correctness)

Proof: a) If the (f,g)-alliance S is minimal, then every node $u \in S$ is necessary in S to guarantee: $d_G(v,S) \geq f(v)$ for some vertex v in $V(G) \setminus S$ *(property 1)* or $d_G(v,S) \geq g(v)$ for some vertex v in S *(property 2)*. That means that every removal of a $u \in S$ either breaks *(property 1)* or *(property 2)*. If *(property 1)* breaks with the removal of u, then u either does not satisfy $d_G(u,S) \geq f(u)$ in Step 2 or does not receive either an OK message in Step 3 or a NEW_OK message in Step 5 from some neighbour outside S, so u does not send its ID number in Step 6. If *(property 1)* holds but *(property 2)* breaks with the removal of u, then u does not receive either an OK message in Step 3 or a NEW_OK message in Step 5 from some neighbour inside S. Anyway, no node $u \in S$ sends its ID number during Step 6. Hence, S remains the same from that point on until the n-th phase is reached in linear time, as each phase is done in constant time since nodes only communicate with neighbours. □

b) If the (f,g)-alliance S is not minimal, due to *Lemma 1*, where we determined that if $f \geq g$ every 1-minimal alliance is also minimal, we have that: at least one node, namely the node u^* in S such that $S \setminus \{u^*\}$ is an (f,g)-alliance in G and $\mathrm{id}(u^*)$ is minimum among its neighbours in S, leaves S. That follows from Step 2 and Step 3, where all nodes $u \in S$, such that $S \setminus \{u\}$ satisfies both *property 1* and *property 2*, what includes u^*, are considered: $d_G(u,S) = d_G(u, S \setminus \{u\}) \geq f(u)$ (Step 2) and $d_G(v, S \setminus \{u\}) \geq f(v)$ for all $v \in N_G(u) \cap V(G) \setminus S$ as well as $d_G(v, S \setminus \{u\}) \geq g(v)$ for all $v \in N_G(u) \cap S$ (Step 3) . (Remember: the neighbourhood is open.) Besides, node u^* is also considered in Step 5, which *takes into consideration smallest IDS* to verify both *property 1* and *property 2* for $S \setminus \{u^*\}$: $d_G(v, S \setminus \{u^*\}) \geq d_G(v, S \setminus A) \geq f(v)$ for all $v \in N_G(u^*) \cap V(G) \setminus S$ where $\{u^*\} \subseteq A \subseteq N_G(v) \cap S$ as well as $d_G(v, S \setminus \{u^*\}) \geq d_G(v, S \setminus A) \geq g(v)$ for all $v \in N_G(u^*) \cap S$ where $\{u^*\} \subseteq A \subseteq N_G(v) \cap S$. This is due to the minimality of $\mathrm{id}(u)$ and again the fact that $S \setminus \{u^*\}$ is an (f,g)-alliance (thus satisfying not only *property 1* but also *property 2*). Hence, node u^* itself is included in the considered nodes in Step 6. Finally, u^* leaves S due to the minimality of $\mathrm{id}(u^*)$ in Step 7, each step being done in constant time due to neighbour communication. □

Corollary 2. *Let G be a graph and let $f, g \in \mathbb{N}_0^{V(G)}$. There is a fast linear-time id-based synchronous message-passing distributed algorithm (i.e., the Linear_$f \geq g$_South_Zone) that computes deterministically a minimal (f,g)-alliance contained in $V(G)$ for any $f \geq g$.*

Proof: The proof follows immediately from Theorem 2 and Theorem 1.a: due to Theorem 1.a, after linear time the algorithm either terminates with no (f,g)-alliance if it is not possible to have one, or provides an initial $S = V(G)$ which is an (f,g)-alliance in G with $f \geq g$. Theorem 2 then guarantees us that in constant time, if S is not minimal, a phase produces a proper subset of S, and otherwise, if S is minimal, nothing changes. Thus, as there are n phases, after a linear time the algorithm ends with a minimal S. □

6 Anonymous Randomized Synchronous Distributed Minimal (f, g)-Alliance Algorithm for $f \geq g$

In this section we describe *Random_$f \geq g$_South_Zone*, an anonymous randomized and synchronous message-passing distributed algorithm that determines a minimal (f, g)-alliance in G for any $f \geq g$, including global offensive alliances.

Initially, as in the last section, S is simply $V(G)$ if $d_G \geq g$, as in the case of global offensive alliances where $g = 0$. If not, as before, such an (f, g)-alliance does not exist. As in the last section, the algorithm maintains a set S that is an (f, g)-alliance. However, contrary to the algorithm in Section 5, we will not make use of ID numbers but assume instead that each vertex has a random bit generator that produces "1" and "0" with probability 1/2.

The algorithm proceeds in synchronoud rounds, which we call here *phases*. In each phase either S is not minimal and with positive probability at least one vertex u in S leaves S such that $S \setminus \{u\}$ is still an (f, g)-alliance or S is minimal and nothing changes. Furthermore, the algorithm ends if no broadcast message with the removal of a node is heard during the last X phases, where X may be either a known constant or better, a polynomial in the order of $V(G)$. The larger X, the more probable the algorithm ends with a minimal (f, g)-alliance.

Every phase is composed of the following six steps:

- [Step 1] *(Only in the very first phase:)*
  For every $u \in V(G)$ that
  - satisfies $d_G(u) < g(u)$ (and thus, $d_G(u) < f(u)$, as $f \geq g$)

  broadcasts a message ending the algorithm negatively with $S = \emptyset$.
  (If there is no broadcast message terminating the algorithm with $S = \emptyset$, then $S = V(G)$ initially.)
- [Step 2]
  Every $u \in S$ that
  - satisfies $d_G(u, S) \geq f(u)$

  produces a random bit x, and if $x = 1$, sends to all its neighbours (inside and outside S) a CONSENT message signaling that it wants to leave S.
- [Step 3]
  Every $u \in S$ sends an OK message to a random set with the largest possible size up to $d_G(u, S) - g(u)$ of neighbours in S from whom a CONSENT message was received.
- [Step 4]
  Every $u \notin S$ sends an OK message to a random set with the largest possible size up to $d_G(u, S) - f(u)$ of neighbours in S from whom a CONSENT message was received.
- [Step 5]
  Every u in S that
  - receives an OK message from all its neighbours (inside and outside S)

  sends to all its neighbours in S a message.
- [Step 6]
  Every vertex u in S that
  - sent a message during Step 5 and
  - did not receive during Step 5 a message

leaves S and updates all other nodes about its removal from S with a broadcast message.

If no broadcast message is received during Step 6 since the last X phases, the algorithm ends.

Theorem 3. *Let G be a graph and let $f, g \in \mathbb{N}_0^{V(G)}$. Let S be an (f,g)-alliance in G with $f \geq g$. The algorithm Random_$f \geq g$_South_Zone is such that:*

a) If S is minimal, then applying the six steps of a phase ends the algorithm after the X next phases. (Termination)
b) If S is not minimal, then with positive probability applying the six steps of a phase produces a proper subset of S. (Correctness With Positive Probability)

Proof: Analogous to *Theorem 2.* □

7 Conclusions and Open Problems

In this paper we investigated (f,g)-alliances: alliances under a general definition which make use of two threshold functions, f and g. As seen, an alliance is nothing more than a central group of nodes to which every other node should be minimally connected to, more precisely, by having a minimum number of neighbours inside the alliance. This group of nodes thus play a more powerful and influential role inside the network, being core for example to problems such as voting and disease spread, where connectivity to a main group makes a difference.

It remains as an open problem how results on (f,g)-alliances would get impacted if a distance of k would to be considered instead of a distance of 1 [11]. For instance, determining fast distributed algorithms for certain types of alliances could prove useful in multicast systems [27] and message routing in sparse tables [22]. A few partial results for some specific alliances are already known. For instance, the problem of finding a k-dominating set, which extends the problem of finding a dominating set, has already been investigated by some researchers. Though finding such a minimum k-dominating set is NP-Hard [12], some charming ideas for efficient distributed algorihms were proposed in [19, 23]. It would be interesting to know as well a bit more about the effect of the k-distance on (f,g)-alliances or any other of its subtypes, including the global defensive and global offensive alliances.

Acknowledgments. This work was supported in part by DAAD/CAPES PPP PROBRAL *'Cycles, Convexity and Searching in Graphs'* grant number 50724218.

References

1. Angluin, D., Aspnes, J., Eisenstat, D.: Fast computation by population protocols with a leader. Distributed Computing 21(3), 183–199 (2008)
2. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007); PODC 2006, Special Issue

3. 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)
4. Bar-Ilan, J., Kortsarz, G., Peleg, D.: How to allocate network centers. Journal of Algorithms 15, 385–415 (1993)
5. Centeno, C., Dourado, M.C., Penso, L.D., Rautenbach, D., Szwarcfiter, J.L.: Irreversible Conversions on Graphs. Accepted in Theoretical Computer Science (to appear)
6. Dourado, M.C., Penso, L.D., Rautenbach, D., Szwarcfiter, J.L.: On Reversible and Irreversible Conversions. In: International Symposium on Distributed Computing (DISC 2010), pp. 395-397 (September 2010)
7. Dourado, M.C., Penso, L.D., Rautenbach, D., Szwarcfiter, J.L.: Reversible Iterative Graph Processes (under submission)
8. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17, 643–644 (1974)
9. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
10. Dreyer Jr., P.A., Roberts, F.S.: Irreversible k-Threshold Processes: Graph-Theoretical Threshold Models of the Spread of Disease and of Opinion. Discrete Applied Mathematics 157(7), 1615–1627 (2009)
11. Goddard, W., Hedetniemi, S.T., Jacobs, D.D., Trevisan, V.: Distance-k Information in Self-Stabilizing Algorithms. In: Flocchini, P., Gąsieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 349–356. Springer, Heidelberg (2006)
12. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman & Co., New York (1979)
13. Gupta, A., Maggs, B.M., Oprea, F., Reiter, M.K.: Quorum Placement in Networks to Minimize Access Delays. In: ACM Symposium on Principles of Distributed Computing, PODC 2005 (July 2005)
14. Golovin, D., Gupta, A., Maggs, B.M., Oprea, F., Reiter, M.K.: Quorum Placement in Networks: Minimizing Network Congestion. In: ACM Symposium on Principles of Distributed Computing, PODC 2006 (July 2006)
15. Herlihy, M.: Concurrency and Availability as Dual Properties of Replicated Atomic Data. Journal of the ACM 37(2), 257–278 (1990)
16. Hassin, Y., Peleg, D.: Average Probe Complexity in Quorum Systems. In: ACM Symposium on Principles of Distributed Computing (PODC 2001), pp. 180–189 (August 2001)
17. Jamieson, L.H.: Algorithms and Complexities for Alliances and Weighted Alliances of Various Types. Clemson University (May 2007)
18. Kuhn, F., Wattenhofer, R.: Constant-time distributed dominating set approximation. In: ACM Symposium on Principles of Distributed Computing, PODC 2003 (July 2003)
19. Kutten, S., Peleg, D.: Fast Distributed Construction of Small k-Dominating Sets and Applications. Journal of Algorithms 28(1), 40–66 (1998)
20. Lenzen, C., Wattenhofer, R.: Minimum Dominating Set Approximation in Graphs of Bounded Arboricity. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 510–524. Springer, Heidelberg (2010)
21. Peleg, D.: Majority Voting, Coalitions and Monopolies in Graphs. In: International Symposium in Structured Information and Communication Complexity (SIROCCO 1996), pp. 152–169 (June 1996)

22. Peleg, D., Upfal, E.: A tradeoff between size and efficiency for routing tables. Journal of ACM 36, 510–530 (1989)
23. Penso, L.D., Barbosa, V.: A Distributed Algorithm for Finding k-Dominating Sets. Discrete Applied Mathematics 141(1-3), 243–253 (2004)
24. Rautenbach, D., Volkmann, L.: New Bounds on the k-domination number and on the k-tuple domination number. Applied Mathematics Letters 20(1), 98–102 (2007)
25. Schneider, M.: Self-Stabilization. ACM Computing Surveys 25(1), 45–67 (1993)
26. Srimani, P.K., Xu, Z.: Distributed Protocols for Defensive and Offensive Alliances in Network Graphs Using Self-Stabilization. In: International Conference on Computing: Theory and Applications (ICCTA 2007), pp. 27–31 (2007)
27. Wittmann, R., Zitterbart, M.: Multicast Communication: Protocols and Applications. Morgan Kaufmann, San Francisco (2001)

Social Market: Combining Explicit and Implicit Social Networks

Davide Frey, Arnaud Jégou, and Anne-Marie Kermarrec

INRIA-Rennes Bretagne Atlantique, Rennes, France

Abstract. The pervasiveness of the Internet has lead research and applications to focus more and more on their users. Online social networks such as Facebook provide users with the ability to maintain an unprecedented number of social connections. Recommendation systems exploit the opinions of other users to suggest movies or products based on our similarity with them. This shift from machines to users motivates the emergence of novel applications and research challenges.

In this paper, we embrace the social aspects of the Web 2.0 by considering a novel problem. We build a distributed social market that combines interest-based social networks with explicit networks like Facebook. Our Social Market (SM) allows users to identify and build connections to other users that can provide interesting goods, or information. At the same time, it backs up these connections with trust, by associating them with paths of trusted users that connect new acquaintances through the explicit network. This convergence of implicit and explicit networks yields TAPS, a novel gossip protocol that can be applied in applications devoted to commercial transactions, or to add robustness to standard gossip applications like dissemination or recommendation systems.

1 Introduction

The advent of Online Social Networks (OSN) has shifted the core of Internet applications from devices to users. Explicit social networks like *Facebook*, or *LinkedIn* enable people to exploit real-world connections in an online setting. Collaborative tagging applications such as *delicious*, *CiteULike*, or *flickr* form dynamic implicit networks of users on the basis of their online activities, interest profiles, or search queries. Users can not only access and introduce new available content but they become themselves accessible through the online infrastructure. Existing online social networks can be grouped into two main categories: explicit and implicit. In explicit networks, users explicitly determine which other users they should be connected to. In *Facebook* or *MySpace*, they issue and accept friendship requests. In *Twitter*, they decide that they wish to follow the tweets of specific users. In all cases, the topology of the resulting network reflects the choices of users and often consists of links that already exist between real people. Explicit networks are therefore very useful in reinforcing and exploiting existing connections but provide little support for discovering new content [3,1]. Implicit networks fill this gap by taking an opposite approach which allows users to discover new content, and acquaintances [4].

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 193–207, 2011.

Specifically, implicit networks form dynamic communities by collecting information about collateral activities of users, such as browsing websites or tagging documents, URLs, or pictures. A given user may or may not be aware of the other members of her own communities depending on the target application. Other users should be clearly visible if the purpose of the application is to discover new people, while they may be hidden for the sake of privacy when they are simply being used as proxies to access new interesting content. In either case, the ability to establish new social connections is key to identifying new and useful data.

Recent years have seen the emergence of a significant number of research efforts and applications exploring the power of each of these two paradigms. Nonetheless, a lot more can be achieved if both approaches are combined into a single framework. Consider the following example. John, who lives in London, bought two electronic tickets for a classical-music concert in Paris, a concert version of Berenice by Handel, but an unexpected event makes him and his friend unable to travel to Paris to attend the concert. The concert is tomorrow and John would like to sell the tickets to someone who can actually attend the event. Unfortunately, while John has many friends interested in classical music, they are all based in the United Kingdom. He does know a few people in Paris, but they are mostly colleagues or friends he met while traveling and who do not share his musical tastes. He tries calling a few of them but his best bet is Joseph, who claims to have a friend whose parents often go to classical-music concerts. Unfortunately, this friend of Joseph's is out of town and Joseph does not know how to reach his parents. As a last resort, John posts a message on a French classical music forum, linking to an EBay ad. However, none of the classical music fans on the forum are responsive enough and some of them even become suspicious that the electronic ticket being sold by this new forum user is actually a fake.

John's problem would be very easy to solve if he was able to contact someone that, albeit not knowing him directly, was at the same time interested in the concert and would trust him enough to buy an electronic ticket from him. This is exactly what can be achieved by combining the discovery potential of implicit networks, with the real-world guarantees provided by trusted social links in explicit ones. While implicit networks do not convey any kind of trust, explicit links almost always carry some kind of trust properties resulting from being friends, coworkers and so on.

In our example, the implicit network allows John to identify people that could be interested in the concert. Among these, he discovers François, a music teacher from Paris who is trying to buy two tickets for one of his students and himself. A cross check on the explicit network then allows John to assess François's trustworthiness. He finds out that François is actually the cousin of a French colleague of his wife. This allows the two to gain confidence in each other and thus complete a safe transaction without external help.

Combining the discovery capabilities of implicit networks with the trust and confidence that are inherent in friendship relationships is useful not only in

the context of commercial transactions but also in applications like information dissemination, or recommendation. Consider a distributed news dissemination system [6]. Users receiving an unusual news item will be more likely to be interested in it if the source or the user that forwarded it is associated with some level of trust. Similarly, recommendation systems can take into account the trust of users in other people's opinions.

The power of combining explicit and implicit social networks, however, has a cost. First, the enormous amount of information necessary to manage user profiles requires significant storage capacity as well as computing power to determine who the best users are for a given transaction. Second, the associated costs are equally enormous and can only be afforded by a few very large companies, and this, in turn, brings significant privacy problems. Modifications in the terms of service of websites like Facebook have already caused public uproars in multiple occasions. Most people are rightfully upset at the idea that their personal data may be collected by a company and sold to third parties for whatever reasons. The most promising way to continue to use personalized information is therefore to develop scalable decentralized solutions.

A class of protocols that appear to be particularly suited for this purpose are those based on the gossip paradigm. Initially introduced in the context of distributed databases, gossip protocols have rapidly shown their applicability in a large number of applications including data dissemination, overlay maintenance, and more recently social networks. In this paper, we extend existing work on interest-based gossip overlays [4] and propose *Social Market*, a solution for the identification of trusted social acquaintances.

Our main contribution in Social Market is TAPS (Trust-Aware Peer Sampling), a novel protocol that operates by directly incorporating trust relationships extracted from an explicit social network into the gossip-based overlay. This provides each user with a set of neighbors that are not only useful but that can also benefit from a high degree of trust. Through extensive simulations, we show that Social Market and TAPS achieve performances that are comparable to those obtained by protocols equipped with global system knowledge, while limiting the diffusion of sensitive trust information. This makes our solution directly applicable to situations like the social transaction example described above. Moreover, our results open new directions for making existing gossip-based applications more robust in the presence of unreliable users.

2 System Model and Problem Definition

Social Market (SM) is a novel distributed application enabling users to identify previously unknown social acquaintances that, at the same time, are similar to them and can be trusted through a chain of explicit social connections, the *trusted path*. Selecting similar users is crucial when searching for the right people for a given transaction, but also when building recommendation or data-dissemination systems. Trust enables the implementation of transactions without external help and increases users' confidence in recommendation results.

We consider a system consisting of a set of users equipped with interconnected computing devices that enable them to exchange information in the form of messages. Each user is associated with a *user profile* that characterizes her interest, her past behavior, her geographical location, and whatever other information the user wishes to add. The profile is essentially a vector of strings that can represent, for example, URLs, words, or phrases. We refer to each such string as a *keyword*. Each *keyword* in a profile is also associated with a counter, its *weight*, which counts how many times the keyword has been added to the profile. The weight basically measures how relevant a given keyword is with respect to the other keywords in the profile. Keywords can be added by the user, they can be extracted from her browsing history, as well as from her interaction with Social Market. To simplify notation, we refer to a user and her profile with the same symbol $u \in U$, where U is the universe of all user profiles. Profiles can be compared with each other using a standard similarity metric. Even though our solution can operate using any similarity metric, in the following, we make use of the well known *cosine similarity* [19], which measures normalized overlap.

$$Sim(u_1, u_2) = cos(u_1, u_2) = \frac{u_1 u_2}{||u_1|| \cdot ||u_2||}$$

Users interact with Social Market by proposing items that they wish to exchange with other users. An item can be, for example, an object to sell, an object to buy, but it can also be a question that is being asked and that needs an answer. When a user u creates an item, she associates it with an item profile, similar to what is done in [2]. Structurally, an item profile is identical to a user profile. Upon creation, the system initializes the item profile to the corresponding user profile. The user then completes the creation by selecting which keywords from this profile clone should be kept, which should be removed, and which, if any, new keywords should be added.

In the example of Section 1, John creates an item for the Handel concert in Paris. Prompted with a profile that contains, among other things: *computer science*, *cycling*, *mountaineering*, *violin*, *rock*, and *classical music*, John decides to keep only *violin* and *classical music* in the item profile. He then adds two more keywords, *Paris* and *Handel*, and decides to keep only the latter in his user profile as he's not interested in being notified about other items associated with Paris. Once an item has been created, the goal of Social Market is to lead this item to *meet* other users who

- (i) are interested in the item,
- (ii) can be trusted and can trust the creator of the item,
- (iii) can be reached through a trusted path on the social network (Figure 1a).

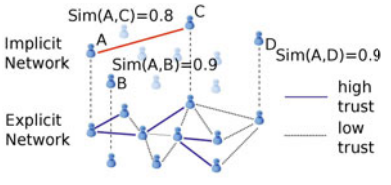
To make this possible, Social Market users can create explicit social links. While similar to friendship links in systems like Facebook, SM links also have an additional feature: *trust*. Upon establishing a link, users declare how much they trust each other by specifying a value in $(0, 1]$. The value of the trust link is similar to the degree of friendship/confidence specified in some existing social networks

such as CouchSurfing[1]. In particular, if user A assigns a value close to 0 to the link to a user B, it does not necessarily mean that A completely distrusts B, but it may simply mean that A does not know B enough to express a positive opinion.

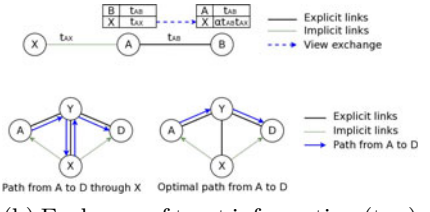
In the following, we present our solution to address Social Market's goals. In this version of our work, we assume that two explicit friends always agree on a trust value for the link they share. This yields an undirected social graph with arc weights between 0 and 1. Extensions to the directed case as well as mechanisms to guarantee high levels of privacy and resilience to attacks are outside the scope of this paper and will be the subject of our future work.

3 Social Market

Identifying users that, at the same time, are interested in an item and towards whom it is possible to identify a trusted path requires an effective protocol to group users according to these two conflicting requirements. Recent research on gossip-based protocols has shown their effectiveness in building overlays that cluster nodes or users according to some distance function or similarity metric. In this section, we extend this research by presenting a novel protocol capable to provide each user or item with a set of neighbors that have highly similar profiles and to which there is a trusted social path. Because users and items are treated identically by our protocol, we refer to both with the term *node*.



(a) Importance of a trusted path: A selects C, rather than B or D as a neighbor even if it has a lower similarity value because it is reachable through a trusted path (high-trust links).



(b) Exchange of trust information (top) and Short circuiting of trusted paths (bottom)

Fig. 1

3.1 Background: Gossip-Based Implicit Networks

Protocols for gossip-based clustering generally consist of two layers. The bottom one is a random peer sampling protocol (RPS) [11] which provides each node with a continuously changing view of the network. The properties of the RPS are such that the resulting topology, that is the union of all the RPS views, can

[1] www.couchsurfing.org is a social community supporting the exchange of accommodation between its users.

be assimilated to a random graph. This makes the RPS effective in maintaining a connected overlay in the presence of disconnections and arrivals of new nodes. The top layer is also a gossip-based overlay maintenance protocol and is based on a variation of [20]. Specifically, at each gossip exchange, a node selects its neighbors by choosing those with the best similarity values.

Social Market exploits a similar protocol structure. Instead of a random peer-sampling layer, it applies our novel trust-aware peer-sampling protocol, TAPS. TAPS provides the clustering protocol with candidate nodes that not only have similar interests but that can also be trusted. The clustering protocol then uses this information to select those that offer the best compromise between trust and interest similarity as shown in Figure 1a.

To summarize, each node maintains three data structures: the explicit view, the TAPS view, and the CLUSTER view. The explicit view contains the node's explicit friends, while the other two are maintained by the TAPS and clustering protocols as described in the following.

3.2 Trust-Aware Peer Sampling

TAPS follows the general structure of a peer-sampling protocol described in [11]. Its goal is to populate the TAPS view with an ever-changing set of references to other nodes. Periodically, each node contacts a node selected from its TAPS view and the two exchange subsets of their respective views.

In a standard peer-sampling protocol, each view entry contains at least information on how to contact the corresponding node (eg. IP address and port), and an age or timestamp value indicating when the information in the entry was generated. In TAPS, we introduce additional fields. First, as in [4], we add a user profile. This makes it possible to cluster similar nodes together by computing the cosine similarity of their profiles. In addition, we include an *inferred trust value*, and a *trusted path*. The *inferred trust value* indicates the trust that a node can have in another node. If the other node is a friend in the explicit network, then the trust value corresponds to the one agreed upon when the friendship relationship was established. Otherwise, it is an inferred value that depends on the path that the trust information has taken during the gossip dissemination, the *trusted path*.

Trust Propagation. Each edge in a trusted path carries some amount of uncertainty about the trustworthiness of the target node even if all the nodes in the path fully trust each other. To model this, we define the *inferred trust* of a path as the product of the trust values of its edges, weighted by a *trust transitivity* coefficient, τ, expressing how much a node values other nodes' recommendations. Given a path $u_1, u_2..., u_n$ with trust values $t_{1,2}, t_{2,3}..., t_{n-1,n}$, the inferred trust between u_1 and u_n is

$$\tilde{t}_{1,n} = \tau^{n-2} \prod_{i=1}^{i=n-1} t_{i,i+1},$$

Lower τ values cause trust to decay faster with path length. For example, with $\tau = 0.7$ trust decays from 1 to 0.49 in only three hops.

The ability to perform trust inference is what makes TAPS the perfect candidate for building our Social Market application. In the following we describe the mechanisms that enable trust inference to be implemented in the context of a gossip-based overlay protocol.

Initialization and View Exchanges. An inference process can only produce results if it starts from some reliable initial values. In TAPS, these values are those that have been agreed upon at the creation of explicit friendship relationships. We therefore initialize the TAPS view by inserting one entry for each explicit neighbor. During the course of the protocol, these entries are exchanged with entries received from other nodes. Initially nodes providing new entries will be the node's explicit friends, then the node's friends' friends, and so on.

As the gossip process evolves, nodes collaborate in computing inferred trust values. Consider a node A exchanging profiles with another node B as shown in the top diagram of Figure 1b. A sends B a subset of its view as well as the value it has for t_{AB}, the inferred trust between A and B. B uses this information to update the inferred trust values before adding the nodes to its own view. Specifically, let t_{AX} be the trust A has in X, then B computes its own trust for X, t_{BX}, as follows. First it verifies if it already has a value for t_{AB}. If so, it keeps the highest value between the received one and its own. It then uses the selected t_{AB}, to compute t_{BX} as

$$t_{BX} = \tau t_{AB} t_{AX}.$$

During the course of dissemination, a node A will inevitably receive multiple references for the same node X. Each of these may arrive from a different neighbor and carry a different trust value. In the presence of multiple trust values for the same reference, a node always selects the largest. However, two references for the same node may also contain slight differences in the node's profile, and in this case, the node should keep the most recent information. To balance these two needs, when A receives a reference to a node X, it chooses the highest trust value between the one in its view and the one it received, and combines it with the most recent profile. Because $\tau < 1$, choosing the largest trust value guarantees that nodes always selects the most direct trust values, thus converging towards shorter and shorter trusted paths. Nonetheless, some nodes may still be unable to infer the trust of other nodes through the best social paths.

To enhance the chances of a successful trust inference, a node initiates gossip exchanges not only towards nodes in its TAPS view but also to those in its explicit neighbors. These additional gossip exchanges are executed every T_{exp} TAPS cycles and cause nodes to exchange entries from the unions of their TAPS and explicit views. Explicit-view exchanges increase the probability that nodes can infer trust through the shortest available paths.

Managing Trust Paths. When exchanging trust information with each other, nodes also update the associated path information. Specifically, when a node A receives a reference to node X from node B, it computes the associated path p_{AX} by concatenating p_{AB} and p_{BX}. Maintaining paths up to date is not only

necessary to enable users to enforce correct transactions in the social market application, but it is also useful to correct the degradation of trust values caused by possible loops in the inferred paths.

Consider the situation depicted in the bottom diagram of Figure 1b. Node X holds a reference to D with a trust value t_{XD} and a path going through Y and sends it to A. Node A should combine this with the trust value it has for node X, t_{AX}, also obtained through Y. However, this would lead to a path that uselessly goes twice through node Y and once through X.

Node A can prevent this by short circuiting the path, thus computing a more accurate trust value. The problem is that the information received from X only contains the identifiers of the nodes in the path and not their trust values. Including such values in the path would allow the re-computation of trust, but at the cost of disclosing trust information to third parties.

We therefore replace the re-computation of trust with the computation of a lower bound. Specifically, A knows that the aggregated impact on trust of the segment YX cannot be greater than τ^n, n being the number of useless links in the path, each link being counted once for each time it is traversed (n=2 in this case). It can therefore conclude that the trust value of node D as seen from A, T_{AD} is at least:

$$t_{AD} \geq \frac{t_{XD}t_{AX}}{\tau^n}$$

This makes it possible to increase the accuracy of trust inference without disseminating private trust information to third parties.

3.3 Clustering Trusted Nodes

Our Trust-Aware Peer Sampling protocol provides each node with a continuously changing set of nodes along with their inferred trust values. This constitutes a source of information for our clustering protocol, a variation of the well-known Vicinity protocol [20]. This protocol maintains a CLUSTER view that collects the nodes that offer the best compromise between trust and similarity.

Filling the Cluster View. In order to fill its CLUSTER view, each node periodically selects the node with the oldest timestamp and exchanges the content of its view with it. Upon receipt of another node's view, a node X combines the received view, its own view, and its own TAPS view, and selects the entries that are associated with the best trade-off between similarity and trust. Specifically, for each entry N, X computes a score s_{XN} as follows

$$s_{XN} = Sim(X,N)^{2-\epsilon}{t_{XN}}^{\epsilon},\ \epsilon \in [0,2]$$

where ϵ, the *trust weight*, determines the importance of trust in the trade-off.

To speed up convergence, nodes also update their CLUSTER views when they receive new information through the TAPS protocol. In this case, they simply combine the received TAPS view with the current CLUSTER view and extract the nodes with the best trade-off between trust and similarity. The selected nodes replace those that were previously in the CLUSTER view.

Trust Verification. The decentralized nature of our trust-inference protocol can allow nodes to cheat on their trust values when communicating with nodes that are not direct friends. For example, in Figure 1b (bottom), node D could try to enter A's cluster by making A believe that it has a high-trust link to node Y.

To prevent this, each node verifies the trust values of the entries in its CLUSTER view, once they have remained in the view for at least c cycles.[2] To achieve this, A asks D to forward a *verification message* back to A along the trusted path. The message starts with a trust value of 1. Nodes along the path multiply the message's value by τ and by their trust for the node they received the message from. Y thus multiplies 1 by τt_{YD} in the example. In the absence of colluding nodes, this process causes the *verification message* to reach A with the correct value for t_{AD} thereby invalidating D's cheating attempts.

4 Evaluation

We evaluated the effectiveness of our approach by means of extensive simulations on several data traces obtained from real social networks. In the following we first present the details of our setting, and then discuss our results.

4.1 Setting

We evaluated our protocol on real data traces consisting of 3000 users extracted from the Facebook and Digg social websites. The Facebook trace[3] contains friendship links and a list of social interactions. To obtain a treatable subset for our experiments, we first cleaned up the trace by removing all users that had only one friendship link, as they would be too isolated to benefit from our social platform. We then selected the user with the largest number of interactions and proceeded in a spiral fashion by selecting her friends, then the friends of her friends, and so on, until we reached 3000 users. We associated each of these users with a random user profile from the Digg social network. We obtained these profiles by crawling Digg in late 2010.

Friendship links in the Facebook trace and profiles in the Digg trace provided the base explicit links and profiles for our experiments. On top of them, we built several traces by varying the trust patterns between the nodes. We distinguish our traces into two groups: binary and multi-valued. In both, we made the assumption that the number of interactions between two nodes is a measure of trust (this assumption is not part of the protocol itself).

Binary Traces. In binary traces we assigned a binary trust value to each link in the data set. Specifically, we sorted the friends of each user according to the number of interactions she had with them. Then, for a user with $|N|$ friends, we assigned a trust value of 1 to the $\beta|N|$ directed links with the largest number of

[2] Similar to [4], we choose $c = 5$.
[3] Network A at http://current.cs.ucsb.edu/facebook/index.html

interactions. Because, this process creates asymmetric trust values, we then set the symmetric trust value of each link as the logical OR of the two asymmetric values. These lead to the proportions of trusted links depicted in Figure 2a.

Multivalued Traces. While binary traces provide a simple experimental setting, reality tends to be more complex. Thus, we also considered traces with trust values of 1, 0.8, 0.5, and 0. Similar to the binary case, we sorted each user's friends by the number of interactions and assigned a trust value of 1 to the top $\gamma_1|N|$, a value of 0.8 to the following $\gamma_{0.8}|N|$ and so on, leading to the three traces shown in Figure 2b.

4.2 Terms of Comparison

We compared the performance of our Social-Market solution against several alternatives. First, we considered *best*, an ideal system that, powered with global knowledge, always provides each user with the set of neighbors that offer the best combination of similarity and trust. This allows us to assess TAPS's ability to reach similar results in a decentralized way. Then we considered *oracle*: this consists of a standard similarity-based implicit network [4], augmented with an oracle that provides each user with the best trusted path to her neighbors. *Oracle* therefore maximizes profile similarity at the cost of possibly lower trust values.

We also compared against two variations of our protocol. *Full-trust* is a version in which nodes exchange complete trust information along with trusted paths. This makes it possible to short-circuit long paths more accurately than as described in Section 3.2 at the cost of disclosing trust values. *Full-Trust* is only shown in multi-valued traces as it is equivalent to TAPS in binary ones. *TAPS no-bound* is instead a version of TAPS in which nodes do not attempt to short-circuit cycles thus yielding less accurate trust computation. Finally, unless otherwise specified, we ran our simulations using default values for τ and ϵ. $\tau = 0.75$ provides a good balance between trust decay and path length (0.56 at 3 hops, 0.23 at 6). $\epsilon = 1$ gives instead a fair tradeoff between trust and similarity.

4.3 Results

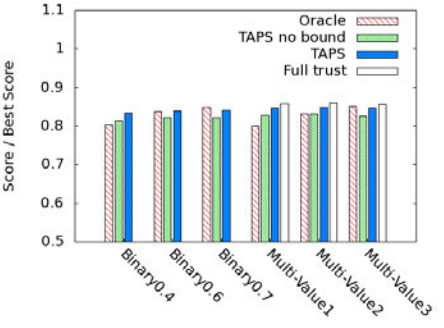
Impact of Trust Density. We start our performance comparison by examining the results obtained in the various traces with a trust transitivity of $\tau = 0.75$. Our results, depicted in Figure 2c, show the average score values in the CLUSTER views with TAPS as well as with its competitors as percentages of *best*'s scores. TAPS's performance is either comparable or better than that obtained by *Oracle* with the use of global knowledge. As expected, TAPS is particularly good whenever the social network has a limited proportion of high-trust links (binary-0.4 and multi-valued-1 and -2). This can be explained by observing that *Oracle* always selects the nodes with the best cosine similarity and is therefore penalized in networks where the density of high-trust links is lower. With smaller τ values, we observed that TAPS outperforms *Oracle* in all the considered traces.

Name	β	% 1	% 0
Binary-0.4	0.4	53.5	46.5
Binary-0.6	0.6	71.3	28.7
Binary-0.7	0.7	80.7	19.3
Binary-0.8	0.8	89.3	10.7

(a) Binary Traces

Name	% 1	% 0.5	% 0.25	% 0
MultiValued-1	41.3	23.8	23.4	11.5
MultiValued-2	57.4	27.9	13.3	1.4
MultiValued-3	76.2	12.3	10.1	1.4

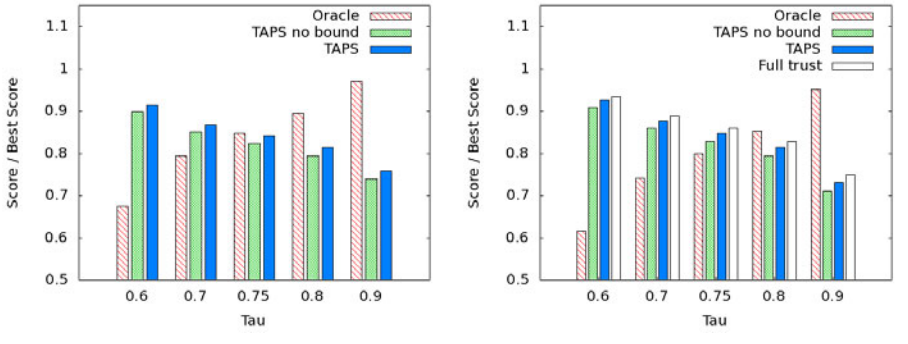
(b) Multi-valued Traces



(c) Impact of trust density

Fig. 2. Trust values distribution for different traces (left), impact of trust density (right)

Impact of Trust Transitivity. Figure 3 confirms the above observation by examining the impact of trust transitivity, τ, on performance in the binary-0.8 (left) and in the multi-value-1 traces. Results show that the performance of TAPS is particularly good when trust decays faster. A faster decay gives more importance to nodes that are closer in the social network even if they may have poorer similarity values. This suggests that a protocol like TAPS becomes more and more important when it is being used for important transactions and in situations in which people can tolerate only limited amounts of risk.

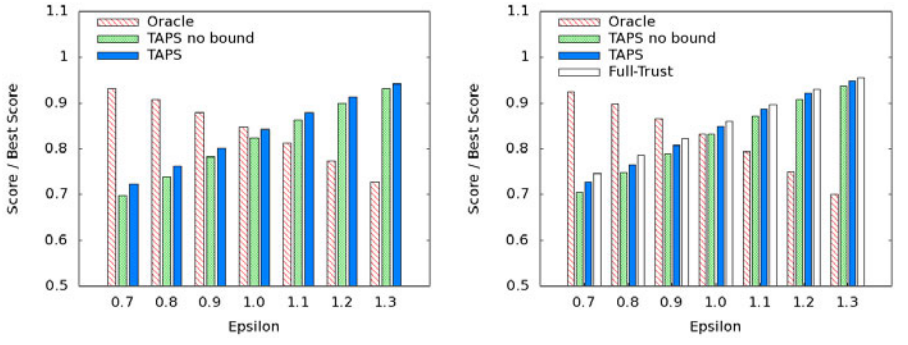


(a) binary (b) multi-valued

Fig. 3. Impact of τ in the binary (left) and multi-valued (right) traces

With very high values of τ trust decays more slowly. In this case, an protocol like *Oracle* that first finds the most similar nodes and then searches for a trusted path may be viable. However, it should be noted that *Oracle* achieves this through global knowledge. A distributed protocol to compute trusted paths would probably be either very costly or ultimately equivalent to TAPS in networks characterized by a high trust density. Moreover, such a protocol would remain inapplicable in situations where the density of trusted links is low, as shown in Figure 2c.

Impact of Trust Weight. Next, we evaluate the impact of the *trust-weight* factor on the performance of TAPS and its competitors. Figure 4 shows the result in the binary (left) and multi-valued (right) traces respectively. Both plots show that the benefits of a protocol like TAPS become more important as more weight is placed on the trust between nodes. With values of ϵ above 1, TAPS performs better than *Oracle* even when considering its *no-bound* version.



(a) binary (b) multi-valued

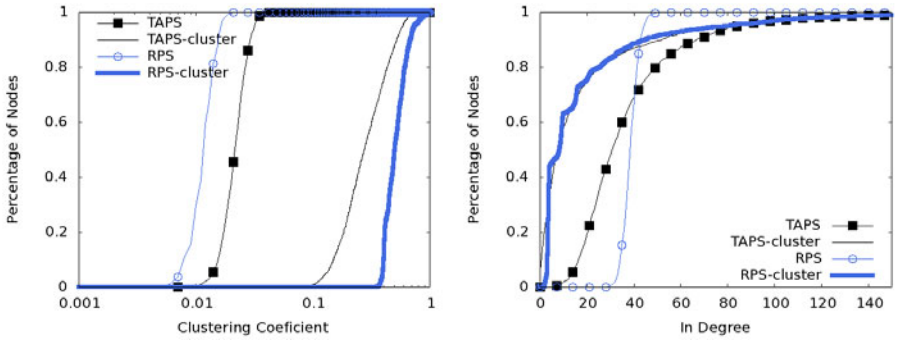
Fig. 4. Impact of ϵ in the binary (left) and multi-valued (right) traces

In addition, the plots show that the importance of short-circuiting cyclic paths is greater when the value of ϵ is small. This is because smaller values make it possible for the protocol to select nodes that are farther away in the social network, which, in turn, makes the presence of cycles more likely.

Graph Properties of TAPS. To better understand the behavior of TAPS, we conclude our evaluation by examining the properties of the TAPS overlay from a graph-theoretical perspective. First we observe that convergence speed is comparable to that obtained with standard protocols: views reach 90% of their scores within 15 cycles and completely converge after 80. Then we examine our TAPS and clustered overlays in terms of clustering coefficient and in-degree distribution to assess how close they are, respectively, to a random and a clustered graph.

Figure 5a shows the cumulative distributions of the local clustering coefficients of the nodes in our TAPS, and CLUSTER views (TAPS and TAPS-cluster) and compares them to those of a standard peer sampling protocol (RPS) and a standard clustering protocol that does not consider trust (RPS-cluster). The plot (in logarithmic scale) shows that, according to expectations, the TAPS topology is slightly more clustered, and thus less uniformly random, than a standard RPS topology. Conversely, our CLUSTER topology is slightly less dense than the one based on pure similarity.

The in-degree distribution shown in Figure 5b also shows some differences with respect to traditional protocols. In this case, the difference is more accentuated between TAPS and the RPS than for the corresponding clustered overlays. The in-degree distribution of TAPS is in fact slightly skewed because nodes that are



(a) clustering coefficient (log scale) (b) in-degree

Fig. 5. Cumulative distributions of the local clustering coefficients (left) and of the in-degree (right) distributions for TAPS-based and standard protocols

not trusted by many others tend to have fewer neighbors. This is indeed a desirable property as untrusted nodes could potentially harm the system through illicit behaviors. Moreover, while these differences are inherent in the trust-dependence of the TAPS overlay, their small absolute value suggests that TAPS could probably replace traditional protocols in a number of applications.

5 Related Work

The concept of trust in explicit social networks has been exploited in domains ranging from peer-to-peer security to recommendation systems. SybilGuard [23] and SybilLimit [22] propose protocols that exploit trust relationships between friends to protect peer-to-peer systems from sybil attacks. Reliable Email [9] uses a similar approach to build an email-whitelisting system based on friend-to-friend relationships, while Ostra [18] exploits social trust to limit the incidence of unwanted communication in messaging and content-sharing systems.

NABT [15] proposes the use of trust between friends to prevent freeriding behaviors using a more efficient form of tit-for-tat based on indirect trust relationships. NABT's credit-based approach can be viewed as a basic form of trust inference between friends of friends. A more advanced approach to trust-inference is adopted by SUNNY [14], a centralized protocol that takes into account both trust and confidence to build a Bayesian network. Even if SUNNY is centralized, its confidence-based idea could lead to interesting improvements for TAPS.

A number of research efforts have instead investigated the use of trust links to improve the performance of recommendation systems. [21] uses an approach similar to that of [14], while TrustWalker [10] combines trust and item-based collaborative filtering. TaRS [17] builds a recommendation system capable of operating both with global and with local trust metrics. Global trust metrics [7] predict a global reputation value for each node. Local trust metrics [16], on the other hand, take an approach similar to ours and compute trust values that are dependent on the target user.

Despite the mole of work on social trust, Social Market is, to the best of our knowledge, the first system to propose the use of trust relationships to build a decentralized interest-based marketplace. Similarly, TAPS is the first attempt to combine explicit and implicit social networks into a single gossip protocol. Existing research on gossip protocols has addressed a number of problems including data dissemination [5], aggregation [13], and overlay construction and maintenance [12,20]. In this context, the two contributions that are most closely related to our work are [8], which uses gossip to disseminate news in explicit networks, and [4], which proposed the use of gossip for implicit ones.

6 Conclusions and Future Directions

We presented Social Market (SM), a novel distributed application designed to enable trusted collaborative actions between similar people in a social-network environment. We proposed a solution to the challenges posed by SM in the form a novel trust-aware peer-sampling protocol, TAPS, which creates an RPS-like overlay taking into account the mutual trust expressed by users when joining an explicit social network. Our experimental results show that, combined with a clustering protocol, TAPS is highly effective in providing users with high-quality implicit acquaintances that not only share similar interests but are also reachable through a verifiable trusted path. This makes Social Market a promising platform for the development of decentralized user-to-user transactions and warrants further investigation into the use of trust to secure existing gossip protocols.

Our promising results encourage us to extend Social Market and TAPS in a number of ways. First, we are examining the possibility of using asymmetric trust values as opposed to symmetric ones. This would make it possible to render trust information more private as users would not need to disclose to their friends the trust they have for them. Second, even though the gossip nature of TAPS makes it inherently self healing, we are considering solutions to maintain the quality of trust paths even during churn by having nodes rely on trusted peers to disseminate information while disconnected. Finally, we plan to explore the use of multiple redundant trusted paths as a way to limit the effects of colluding nodes, which cannot be tolerated by this version of the protocol. These improvements will enable us to integrate TAPS in our existing prototype applications as a way to reinforce the trust of users in disseminated information, recommendations, and ultimately in the implementation of a real-world social-market platform.

Acknowledgments. This work is supported by the ERC Starting Grant GOSSPLE number 204742. We are grateful to Guang Tan for his initial contributions, as well as to the anonymous reviewers for their constructive suggestions.

References

1. Ahn, Y.Y., Han, S., Kwak, H., Moon, S., Jeong, H.: Analysis of topological characteristics of huge online social networking services. In: WWW (2007)
2. Bai, X.: Personalized top-k processing: from centralized to decentralized systems. Ph.D. thesis, INSA Rennes, France (2010)

3. Bender, M., Crecelius, T., Kacimi, M., Miche, S., Xavier Parreira, J., Weikum, G.: Peer-to-peer information search: Semantic, social, or spiritual? IEEE Database Engineering Bulletin 30(2) 2007
4. Bertier, M., Frey, D., Guerraoui, R., Kermarrec, A.-M., Leroy, V.: The gossple anonymous social network. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 191–211. Springer, Heidelberg (2010)
5. Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM TOCS 17(2) (1999)
6. Boutet, A., Frey, D., Guerraoui, R., Kermarrec, A.-M.: Whatsup: News, from, for, through, everyone. In: P2P, Delft (2010)
7. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems 30(1-7) (1998)
8. Datta, A., Sharma, R.: Godisco: Selective gossip based dissemination of information in social community based overlays. In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 227–238. Springer, Heidelberg (2011)
9. Garriss, S., Kaminsky, M., Freedman, M.J., Karp, B., Mazières, D., Yu, H.: Re: Reliable email. In: NSDI 2006, San Jose, CA (2006)
10. Jamali, M., Ester, M.: Trustwalker: A random walk model for combining trust-based and item-based recommendation. In: KDD, Paris (2009)
11. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., van Steen, M.: Gossip-based peer sampling. ACM TOCS 25(3) (2007)
12. Jelasity, M., Babaoglu, O.: T-Man: Gossip-based overlay topology management. In: Brueckner, S.A., Di Marzo Serugendo, G., Hales, D., Zambonelli, F. (eds.) ESOA 2005. LNCS (LNAI), vol. 3910, pp. 1–15. Springer, Heidelberg (2006)
13. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM TOCS 23(3) (2005)
14. Kuter, U., Golbeck, J.: Sunny: A new algorithm for trust inference in social networks using probabilistic confidence models. In: AAAI. AAAI Press, Menlo Park (2007)
15. Liu, Z., Hu, H., Liu, Y., Ross, K.W., Wang, Y., Mobius, M.: P2p trading in social networks: the value of staying connected. In: INFOCOM, San Diego, CA (2010)
16. Massa, P., Avesani, P.: Controversial users demand local trust metrics: an experimental study on epinions.com community. In: AAAI. AAAI Press, Menlo Park (2005)
17. Massa, P., Avesani, P.: Trust-aware recommender systems. In: RecSys. ACM, New York (2007)
18. Mislove, A., Post, A., Druschel, P., Gummadi, K.P.: Ostra: leveraging trust to thwart unwanted communication. In: NSDI, Berkeley, CA, USA (2008)
19. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York (1986)
20. Voulgaris, S., van Steen, M.: Epidemic-style management of semantic overlays for content-based searching. In: Europar, Lisbon (2005)
21. Wang, Y., Vassileva, J.: Bayesian network trust model in peer-to-peer networks. In: Moro, G., Sartori, C., Singh, M.P. (eds.) AP2PC 2003. LNCS (LNAI), vol. 2872, pp. 23–34. Springer, Heidelberg (2004)
22. Yu, H., Gibbons, P.B., Kaminsky, M., Xiao, F.: Sybillimit: a near-optimal social network defense against sybil attacks. IEEE/ACM TON 18(3) (2010)
23. Yu, H., Kaminsky, M., Gibbons, P.B., Flaxman, A.D.: Sybilguard: defending against sybil attacks via social networks. IEEE/ACM TON 16(3) (2008)

TRUMANBOX: Improving Dynamic Malware Analysis by Emulating the Internet

Christian Gorecki[1,⋆], Felix C. Freiling[2], Marc Kührer[3], and Thorsten Holz[3]

[1] AGT Germany
[2] Friedrich-Alexander-University Erlangen-Nuremberg
[3] Ruhr-University Bochum

Abstract. Dynamic analysis of malicious software (*malware*) is a powerful tool in countering modern threats on the Internet. In dynamic analysis, a malware sample is executed in a controlled environment and its actions are logged. Through dynamic analysis, an analyst can quickly obtain an overview of malware behavior and can decide whether or not to indulge into tedious manual analysis of the sample. However, usual dynamic analysis exposes the Internet to the threats of an executed malware (like portscans) because advanced concealment techniques of malware often require full Internet access. For example, a missing link to the Internet or the unavailability of a specific server often causes the malware to not trigger its malicious behavior. In this paper, we present TRUMANBOX, a technique to emulate relevant parts of the Internet to enhance dynamic malware analysis. We show that TRUMANBOX not only prevents many threats but also enlarges the scope of the types of malware that can be analyzed dynamically.

1 Introduction

Since the amount of malicious programs is increasing day by day, many different approaches have been developed to analyze malware. Until recently, the standard approach to malware analysis was based on *static* methods, i.e., disassembling the binary and manual reverse engineering. Modern malware increasingly thwarts such analysis through refined encryption and obfuscation methods. Therefore, *dynamic analysis* methods offer a promising approach to analyze malicious software.

In dynamic analysis, the actual execution of a malware sample is an important part of the analysis process. However, executing malware exposes the runtime environment to malicious activity. Especially malicious outgoing network traffic induced by the corresponding malware samples in execution is a critical issue. Simply preventing any access to the Internet, however, is also not feasible in many cases since malware often needs Internet access to trigger and exhibit its malicious behavior (that is to be analyzed). For example, the unavailability of a specific command and control server often causes certain bots to remain silent.

1.1 Related Work

The issues of preventing outgoing malicious traffic from an analysis platform are similar to those experienced with *honeypots* [10,12]. Briefly spoken, a honeypot is electronic

⋆ Work by Christian Gorecki was performed while being with University of Mannheim.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 208–222, 2011.

bait that can be used for malware capture, i.e., a resource on the Internet whose use lies in the fact that it is compromised. Especially *high-interaction* honeypots, i.e., those that emulate full systems, are a threat to an analysis network. To protect the environment from malicious traffic, a honeynet is separated from the Internet by a *Honeywall* [3]. A Honeywall typically is an OSI layer-2 bridging device with different capabilities in logging and filtering. Often, such a device is called an *extrusion prevention system*.

HONEYD is a simulation framework that allows to instrument thousands of IP addresses with virtual honeypots running corresponding network services [11]. Due to the simplicity of configuration, simulation of huge networks can be set up within minutes and huge networks can be simulated with high-performance. Even though HONEYD is very flexible in its configuration, it is static in the way that services are only listening on predefined IP addresses and on static ports that need to be specified in advance.

As mentioned above, we focus on extrusion prevention not during malware capture but rather during malware analysis. In particular, we are interested in dynamic malware analysis, i.e., running malware within a protected "sandbox" environment like CWSANDBOX [13] and ANUBIS [2,7] which log system calls and thereby create traces of the visible behavior of that malware. As mentioned above, running malware makes it necessary to simulate parts of the network to stimulate certain behavior. One system that does this is the *Botnet Evaluation Environment* (BEE) [1], a testbed for evaluating bots and botnets in a self-contained environment based on *Emulab* [5]. Emulab is a platform for creating virtual network nodes, which can emulate operating systems or applications after being equipped with a corresponding image. Overall, BEE offers services such as IRC, DNS, and DHCP. These services are available on certain IP addresses connected to the virtual network, and outgoing traffic is redirected to these services. For example, any IRC request independent from the original destination address will contact the IRC server deployed within the network. As this network translation is implemented on the client, this approach requires control of the clients. We overcome this drawback in our work, as we will see later on.

Last we want to mention INetSim [6], a very powerful network simulation suite, particularly regarding the number of different services implemented. However, INetSim assumes that protocols always use their "correct" port (e.g., HTTP on port 80), which is not a valid assumption when analyzing malware.

1.2 Contributions

The challenge is to find a trade-off between taking the risk of infecting third party systems and a reduced behavior of malware by preventing or restricting outgoing traffic. In this paper, we present the design, implementation, and evaluation of TRUMANBOX, a system that provides novel flexibility in malware analysis.

The idea of TRUMANBOX is to emulate the most commonly used Internet services on one physical machine in an easily configurable way. Like Honeywall, the system is inserted into the Internet connection as a transparent network bridge. Like BEE, the system emulates network nodes to provide services like IRC. However, unlike Honeywall, BEE, and INetSim, the system (1) dispatches all supported service requests – independent from destination IP address and TCP port – to the corresponding local service, and

(2) uses a heuristic pattern matching approach to identify protocols when they are used on non-standard ports. This means that with TRUMANBOX, any service can be bound to every available port.

TRUMANBOX also offers a flexible set of working modes that range from a behavior analogous to Honeywall to a full Internet emulation that does not need real Internet connectivity at all. The overall goal of these functions is to keep malware running as long as possible, without actually contacting the Internet, or at least restricting/controlling outgoing traffic, to prevent malicious interactions with third-party systems. From the view of an analyst, especially the full emulation mode is a notable feature of TRUMANBOX, since it allows comprehensive malware analysis without Internet access.

To summarize, our contributions are twofold:

- We present the design and implementation of TRUMANBOX, a novel and flexible tool to support dynamic malware analysis. TRUMANBOX has been implemented under Linux and is freely available [4].
- We evaluate TRUMANBOX and show that it offers additional flexibility to malware analysis not offered by other existing tools. For example, in full emulation mode, we found that in 98 out of 154 test cases (malware samples) the traffic reports include at least the same information we would obtain by using a sandbox environment like ANUBIS and CWSANDBOX. Furthermore, 36 TRUMANBOX reports even included network traffic which was not logged by these sandbox environments at all. Note that these results were achieved *without* connection to the Internet.

1.3 Roadmap

This work is outlined as follows: In Section 2, we provide background on different techniques used in our implementation. In Section 3, we describe the actual system. We evaluate the system in Section 4 and conclude in Section 5.

2 Technical Background

We now give some technical background that is important for the understanding of TRUMANBOX in Section 3.

2.1 Naming Conventions

Since TRUMANBOX is a bridge with emulation capabilities, we use two network interface cards (NICs) in our machine, one to the client side we want to "provide" with the emulation and one to the outside network infrastructure, e.g., the Internet. We name those NICs `eth1` and `eth0`, respectively (see Fig. 1). The logical bridge device containing these interfaces as so-called bridge-ports is called `br0`. The *client* is usually a computer running the malware (possibly within a malware analysis sandbox), however, the network structure on the client side can vary.

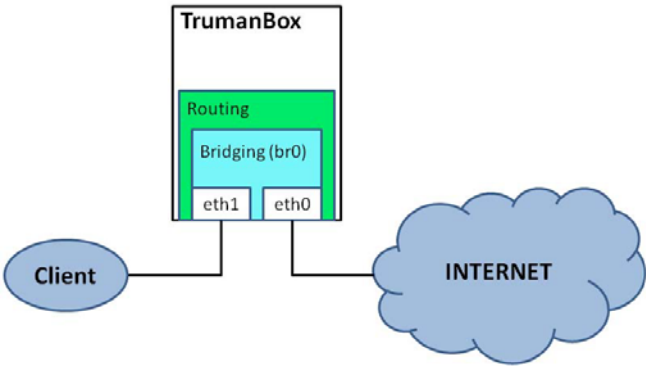


Fig. 1. Naming conventions

2.2 Bridging and (Re-)directing Data

While technically TRUMANBOX is a transparent bridge, it also has to behave as a router if traffic is redirected within the Internet emulation. A transparent bridge has usually no IP address of its own and thus cannot be detected by common port or network scanning tools, enabling a *stealthy* and *transparent* device for intercepting network data traffic.

Fig. 2 provides a simplified view about how a packet passes through the Linux kernel for the case of bridging the two available network interfaces. First, the incoming packets pass the *BROUTING* chain in the *broute* table, where, based on the destination MAC address, it will be decided to pass the packet to OSI layer-3 (IP protocol), to drop it, or to bridge it, i.e., which is an OSI layer-2 forwarding to the other interface. This decision depends on whether the destination MAC aims at connecting to our machine, a machine on the same side, or one that is known to be on the other side of the bridge, respectively. In case the location of the system with the corresponding MAC address is unknown, the packet is flooded over all forwarding bridge ports. Given this complex chain, there are multiple ways in which interception can be done, as we now explain.

We chose to intercept packets at the end of the *PREROUTING* chain in the *nat* table. The next hook, i.e., the *FORWARD* chain in the *filter* table of ebtables, is already part of the forwarding mechanism, hence it would be too late to intercept the packets using this or one of the following hooks. So we finally use appropriate iptables rules to create the possibility of data redirection.

However, using the iptables command, our bridge needs an IP address to which data can be redirected. Therefore, we have to configure an IP address to our logical interface `br0`. Since bridges usually do not have an IP address, we also have to take further steps for maintaining stealthiness.

Since the IP address of TRUMANBOX is needed only for internal packet forwarding, no other machine needs access to our system using the IP protocol. Therefore, we drop all incoming ARP broadcasts by using ebtables. In this way, our system stays invisible to the client side, while we can still access it remotely from the other interface pointing to the Internet. This is important, as we will need outgoing communication for certain modes of our implementation, as discussed later on.

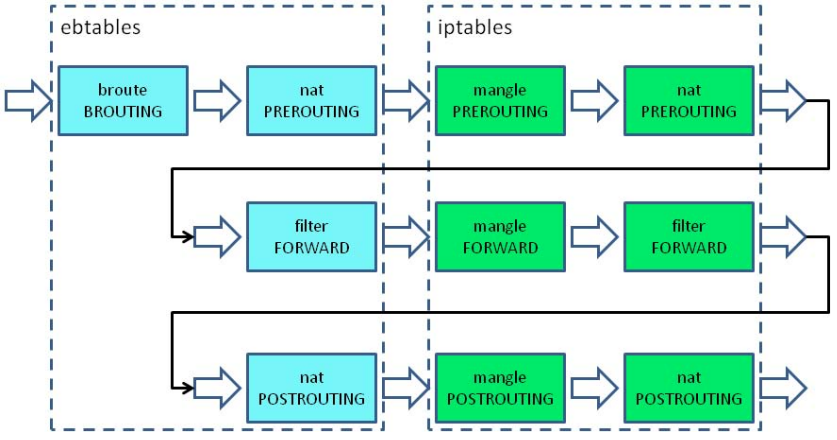


Fig. 2. Simplified data-packet traverse through the Linux kernel of a bridge

2.3 Extracting Non-altered Header Information

Combining the techniques introduced so far, we obtain a transparent bridge that can redirect bypassing traffic to itself. Unfortunately, we loose original TCP header information during this process because the destination IP address and (optionally) the TCP port are changed. So we had to employ a technical trick: We kept the *PREROUTING* chain in the *mangle* table (see Fig. 2) in which the non-altered header information is still visible. We then use the *QUEUE* target in iptables to hand data-packets to the userspace. To access the data (in particular the original header information of the packet), we use the library *libipq* [8] and then hand back the packet to the kernel. Furthermore, we store the information about original destination in a global variable and can use this information in the dispatching phase.

3 TRUMANBOX

3.1 Approach

Combining the techniques presented in Section 2, we can setup TRUMANBOX as a transparent (stealthy) bridge, redirecting selected (or all) bypassing traffic to itself. Thus, any outgoing malicious traffic can be avoided. In order to trick malware and have its malicious activities still being performed, we need to provide the impression of Internet connectivity. For this purpose, standard services like HTTP, FTP, IRC, and SMTP servers are provided (almost) in their standard configuration, e.g., standard listening port, etc., locally on TRUMANBOX.

To overcome static protocol identification driven by the TCP destination port, we provide more flexibility by a hybrid protocol identification. Certain malicious programs use non-standard ports for their communication, some malware samples even use standard protocols on a different standard port, e.g., HTTP on port 21 or FTP on port 80. To overcome this problem, we have a dispatching service running on a certain port

where all bypassing connections are redirected to. The dispatching service maps new connections to the local standard services, as described in Section 3.2, depending on the protocol in use. Information on the original destination, e.g., IP address, TCP port, etc., are extracted for further processing. The connection mapping/dispatching is done in respect to preserving the possibility of intercepting and manipulating payloads.

To improve the emulation provided by TRUMANBOX, we allow adjustment of the amount of outgoing traffic by providing two different modes (see Fig. 3):

1. *Full Emulation* requires no Internet access at all and provides the client with a rather static and simple emulation of the Internet.
2. *Half Proxy* requires access to the Internet; however, TRUMANBOX only uses this to gather additional information to improve the emulation, e.g., by fetching banner information from the originally targeted destination.

Independent of the actual mode in use, the client never interacts with the Internet directly, but only communicates with services provided locally on the TRUMANBOX. DNS is not affected by the mode. Instead, it can be configured separately in order to en-/disable proper domain name resolution to corresponding IP addresses, during the analysis process.

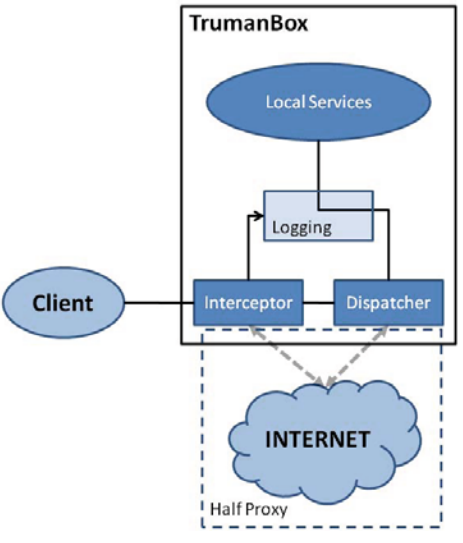


Fig. 3. Connection modes

Even though the *half proxy* mode appears to be an enhanced *full emulation*, there are reasons for having both modes implemented. In certain environments, execution of malware on a computer with full Internet access might be no option due to legal obligations. While execution without Internet access might result into no malicious actions being triggered, TRUMANBOX in *full emulation* may satisfy trigger conditions to render dynamic analysis being successful. Another scenario is dynamic analysis of, for

instance, malicious software trying to reach for a command and control server that is not online anymore. Again, *full emulation* can improve dynamic analysis results. However, *half proxy* mode might be adequate when we are aware of the fact that investigated malware samples do not perform any malicious actions but downloading files and contacting the command and control servers.

More details on the applications of TRUMANBOX are available in Section 4.

3.2 Implementation Details

Header Extraction. Even though our dispatching approach, which will be discussed in detail in the following, has some advantages, e.g., in terms of easily redirecting traffic, there is also one major drawback: it does neither preserve the original destination IP address nor the destination port. However, this information is important in postprocessing of dynamic malware analysis. Therefore, we have to extract the data as described in Section 2.3. Once, the information is extracted, it is used for logging and (in *half proxy* mode) for gathering further information from the original destination host, as for example service banners.

Dispatching. The core of our implementation is the dispatching function. Its processing starts as soon as a new connection has been established to the local port it is listening on, e.g., TCP/UDP port 400. Right after accepting the new connection request (A), a second connection (B) is established depending on the protocol of the incoming connection and the mode the TRUMANBOX is currently running. In the following, payloads are forwarded between A and B. Depending on the mode of operation, certain alterations of payload and execution of additional functions are triggered by a successful match of predefined patterns against the payload. We will explain these different manipulations later when discussing the mode of operation they occur in.

Hybrid Protocol Identification. In order to handle more than only one connection, we create a new process for every incoming connection. Since establishing the second connection (B) also depends on the protocol of the incoming connection (A), we somehow have to identify it. The common identification by considering the TCP destination port is not strong enough for our purposes because data connections initiated by malware often use non-standard ports. Therefore, we strengthen the traditional protocol identification by implementing a payload protocol identification which is applied first. These two algorithms in combination form our *hybrid protocol identification*: First we identify new connections by observing the payload; if this fails, we examine the destination port to determine the protocol in use.

To use the first payload of a connection for protocol identification, we first have to distinguish between server-first-sending (SFS) and client-first-sending (CFS) protocols. For example, in HTTP and IRC usually the client sends the first payload, hence these are CFS protocols. SMTP and FTP start with a payload sent by the server (often called the “banner” or “welcome message”), hence, these are examples for SFS protocols.

Given the first payload of a new connection, we use the patterns listed in Table 1 to determine the protocol. Here we do not distinguish between SFS and CFS protocols, since this would be an unnecessary restriction. Rather, we stay generic to also be

capable of recognizing possible modifications of SFS protocols to corresponding CFS versions. If all these methods fail, we try to determine the protocol by considering the destination port even though, particularly in malware analysis, this is not reliable in all cases.

Table 1. Payload patterns determining the protocol

Protocol	Pattern (at the beginning)	Pattern (somewhere)
HTTP	`"GET /"`	
IRC	`"NICK "`	
FTP	`"220 "`	`"ftp"` (case insensitive)
SMTP	`"220"`	`"mail"` OR `"smtp"` (case insensitive)

Sometimes there is no payload coming over the incoming connection which makes protocol identification harder. We can alleviate this in half proxy mode of TRUMANBOX, as we explain below.

Full Emulation Mode. The goal of full emulation mode is to emulate the Internet without any access to the Internet. This is probably the most challenging of our modes: we want to prevent any kind of data connection (apart from possibly DNS) to the Internet without letting the client (i.e., the malware) become aware of it. Therefore, we provide local services as generic as possible and redirect the connection requests of the client to our machine. If outgoing DNS traffic has to be avoided as well, we are forced to also set up a local DNS service which resolves requested hostnames to predefined IP addresses.

Since responses of our generic services are static, it is quite easy to fingerprint TRUMANBOX, so this mode is not meant to trick a human intruder or some malware which uses methods to verify the authentication of the server to contact. For instance, a rather simple method of detecting TRUMANBOX is to send requests to unreachable servers. In full emulation mode, TRUMANBOX always responds with a valid reply as if the originally targeted server is online. Since the attacker does not expect a response at all, he might realize that some interception is going on. A further approach could be to investigate the content of received responses. Whenever an attacker requests a specific file from a webserver, e.g. an update of a malware executable, TRUMANBOX returns fake content since in full emulation mode, it is not allowed to contact the Internet to get the originally requested file. As a consequence, TRUMANBOX returns data the attacker most probably does not expect. If the attacker examines the returned data, e.g. by calculating the hash value of the received content and comparing it with the expected value, he will notice that the received data is not what he requested. In case the attacker detects that TRUMANBOX is in place, he might simply stop his actions on the compromised system so that we would not be able to gather any more information. One possible countermeasure, at least during malware investigation, is to set up a communication channel between TRUMANBOX and the system performing malware samples. Whenever a sample does not perform any activity after a reply of TRUMANBOX, the malware is restarted and TRUMANBOX acts in a different way, e.g. does not grant access to an FTP server this time.

In summary, the advantage of full emulation is clearly that we do not need a connection to the Internet. Hence, we could also use this mode in an offline analysis environment.

Half Proxy Mode. All services running within TRUMANBOX are set up in their basic configuration. Thus, server responses will correspond to the defaults which most likely differ from the original server a certain connection request was aiming to contact. In full emulation mode, an attacker might therefore try to fingerprint the network environment. To overcome this problem, we introduced the half proxy mode by substituting certain service features using access to the Internet. For example, if there is no initial payload on the client side, we establish a connection to the Internet trying to fetch the banner from the original server. In case of FTP, the last step is extended to also test if anonymous login is provided. The resulting information is then used for protocol identification.

As a result, an attacker is not able to identify TRUMANBOX by sending requests to unreachable servers since TRUMANBOX contacts the originally targeted servers and acts exactly the same way. Yet, detecting TRUMANBOX by investigating received responses is still possible because the current implementation of TRUMANBOX does not forward the original file content to the attacker. However, future versions of TRUMANBOX will support this feature so that attackers always get the expected file content and investigations of that data will not reveal any information whether TRUMANBOX is in place.

3.3 Extensions to Half Proxy Mode

The following functionality is mainly based on half proxy mode. These functions show how incremental improvements can be done in certain situations to improve the "reality" of the emulation.

Replaying FTP and SMTP Banners: Assuming RFC conform behavior, FTP or SMTP are SFS protocols where payloads are sent first from server to client. Thus, we already fetch banner or welcome messages from the original server during the protocol identification by payload. All gathered banners are stored locally in a cache (a plain text file named with the IP address and the TCP port number of the corresponding server). By checking the cache before contacting the original server, we avoid unnecessary data traffic.

Emulation Efforts on FTP and HTTP: FTP or HTTP connections are particularly challenging since the corresponding servers usually provide a complex and unknown directory structure that the client may wish to access. For example, a client may try to access `http://example.org/path/to/file`. We handle this as follows: After accepting the incoming connection, we identify the protocol by matching the `GET /` pattern in the first payload sent by the client. After parsing the URL, we on the fly create a filesystem structure in our webserver's base directory according to the request we just received. We do not aim at serving the content the client expects in the requested path, even though this could be easily be done in half proxy mode. Our interest is to track what a client is doing. With our approach, we get an overview of the client's behavior

on the contacted server. Assuming the client does not send false requests to verify being connected to the server it was intending to connect to, we will learn something about the remote side and the filesystem structure provided there.

FTP User Authentication vs. Anonymous Login: Supporting FTP in our emulation, we need to reason whether to grant anonymous login or rather require a valid username/password combination. In the latter case, we also have to define which credentials are to be accepted and which ones are invalid.

Reviewing our aim to provide an emulation as close to reality (here: the Internet) as possible and also driving a mode that allows us to access the original server, we decide to provide both anonymous login and user authentication, depending on what the original server requires. This is simply done by extending our "fetch the banner" function by a check if, in case of an FTP server, anonymous login is granted. If so, we provide the same and let anonymous login attempts pass. Otherwise, we grant access with any arbitrary login. Even though this might lead to logging a lot of invalid FTP logins we cannot use for further investigation, we hope to also gather valid account data we can use to login ourselves into the original server, for example, to fetch further malicious binaries for analyses. Again, this can be done either manually or in an automated manner by extending our program with functions accordingly. Note that manually processed investigations may be restricted to a certain time slot because logins might expire or get deactivated if the client becomes aware of our interception.

In case of a non-anonymous login, we stick to our approach of not using highly customized configurations but rather extending our payload alteration. Therefore, we just setup our FTP server with one valid login, where the corresponding username and password is known to TRUMANBOX. Any login attempt is then altered to the valid username and password combination. By logging the unmodified login data, we try to gather valid account data for the original server we may use for further investigation. Unfortunately, this approach lacks in handling false login testing, i.e., it is easy for the malware to fingerprint TRUMANBOX by using random combinations of login and password. We can detect such behavior by extending the login payload alteration to recognize if interaction just stops after a successful login and during the next execution of the malware preventing a login to the same server using the same credentials. This idea requires conditionally repeated execution of a malware on the client side.

IRC Session Logging and Emulation: For IRC sessions, issues regarding logging have to be considered. When logging the whole communication the log can become very long. In particular long sequences of PINGs and PONGs which are often used for alive testing during an IRC session do not improve our analyses. The alternative is to use a whitelist of patterns. Both logging techniques enable us to replay the login process on the original server and thus gather information, for example, about botnets. One possible application is to feed botnet monitoring programs like botspy [9] with the collected information.

3.4 Logging

During the runtime of TRUMANBOX, all incoming payloads and header information are logged to plaintext files named with the IP address and TCP port of the destination

target. For this reason, we have to perform outgoing DNS traffic, namely DNS requests, to get this information accurately. If the requested full qualified domain name (FQDN) cannot be resolved, we are faced with a new challenge, but for now we assume that the domain can be resolved. The log files are stored in the subfolders *ftp*, *http*, *irc*, and *smtp* – respectively the protocol of the connection. At the beginning of every new connection attempt, we add a timestamp to the log file that helps us to determine time and date of the logged information. Later, we can use the connection information specified in the filename in combination with the corresponding file content to replay a login or just analyze the traffic directed to that machine.

Provided with the logging structure just outlined, we now have to decide which information to log. To be prepared for different logging strategies, we have implemented a switch deciding to either log everything or just log those information matching certain patterns, where patterns are basically protocol directives we expect to appear together with interesting information, e.g., the patterns `NICK`, `USER`, and `PASS`.

4 Evaluation

We now investigate the practical use of the TRUMANBOX approach.

4.1 Testbed and Evaluation Setting

We set up a testbed to analyze the capability of TRUMANBOX to detect malicious network traffic generated by 300 randomly selected samples taken from the ANUBIS [2,7] database. Our testbed consists of a Windows 7 system which executes samples in a virtual Windows XP machine and a system running Ubuntu 11.04 which is exclusively used for TRUMANBOX and the local HTTP, FTP, IRC, and SMTP services. The first network interface of TRUMANBOX is connected to the Internet, while the second interface points to the Windows 7 system. The two interfaces of TRUMANBOX are bridged so that all outgoing traffic caused by the ANUBIS samples has to pass TRUMANBOX and can be investigated.

We used this setting to run TRUMANBOX in full emulation and half proxy mode to observe network traffic from malware samples executed on the virtual machine within a two minute lifespan. In the evaluation, we compared the TRUMANBOX logs with the ANUBIS sandbox reports from the database that were created running the malware sample with full Internet access.

4.2 Full Emulation Mode

Our main focus was to evaluate the full emulation mode because it does not require an Internet connection. Interestingly, we were able to show that TRUMANBOX is still able to collect valuable information from the network traffic to improve analysis of malware. We ran the sample set of 300 samples from the ANUBIS database in our virtual Windows XP machine. Table 2 outlines the results we observed.

Out of the 300 randomly selected samples, merely 154 samples generated network traffic at all. The other 146 samples did not attempt to perform any network activities.

Table 2. Evaluation results of the 300 investigated ANUBIS samples

Positive network traffic	**#**
TRUMANBOX log and ANUBIS report cover the same information	98
Additional HTTP information in TRUMANBOX log	12
Additional IRC information in TRUMANBOX log	4
Additional FTP information in TRUMANBOX log	20
Total positive	134
Negative network traffic	**#**
No IRC but HTTP traffic in TRUMANBOX log	1
No FTP but HTTP and IRC traffic in TRUMANBOX log	3
Undetected/Invalid IRC format	9
Less HTTP information in TRUMANBOX log	7
Total negative	20
Sum positive and negative	154
Overall	**#**
Number of samples generating network traffic	154
Number of samples not generating network traffic	146
Overall number of samples	300

We can only guess that either these samples do not generate network traffic at all or the network functions were just not triggered for unknown reasons.

When taking a look at the TRUMANBOX log files of the remaining 154 samples performing network activities, our tool was able to collect the same information which is stated in the ANUBIS reports for 98 samples. These samples cover various HTTP requests, FTP connections, and IRC traffic. However, the TRUMANBOX log files do not include one particular type of data: since we do not access the Internet in full emulation mode and thus cannot fetch information from the originally targeted servers, all IRC channel topics are unknown to TRUMANBOX. Channel topics of command and control servers often include URLs which are downloaded once malware joins the channel. The majority of ANUBIS reports cover network traffic of such HTTP and FTP downloads. In full emulation, TRUMANBOX certainly is not able to provide these download URLs to the investigated samples, thus the log files do not include this type of traffic. Nevertheless, all other network activities targeting HTTP, FTP, and IRC were logged accordingly and resemble the data in the ANUBIS reports.

One of the 154 samples did not establish an IRC connection next to HTTP GET requests although the ANUBIS report included information about a command and control server. Furthermore, three of the 154 samples did not establish FTP connections next to HTTP and IRC traffic, although the ANUBIS reports covered details about FTP connections. We can only guess that the IRC and FTP functions were not triggered for unknown reasons.

Furthermore, nine samples did not use the proper IRC protocol so that TRUMANBOX either did not redirect the connection to the local running IRC server or the IRC server refused the connection. Some of the samples used commands like

```
CK [DE|WinXP|15718] 0 * :mIRC 6.17
```

instead of NICK and USER, others simply did not send the NICK command. We speculate that the malware samples perform this non-standard conform communication in order to bypass protocol-based detection of malware communication.

Seven of the 154 samples generated fewer HTTP requests as stated in the ANUBIS reports. On the one hand, we guess that some of the HTTP GET requests were just not triggered for unknown reasons. On the other hand, some malware expects the downloaded content to be valid, e.g., executable files. However, we do not provide the malware with the exact data it expects since we would need to connect to the Internet to get those files. If a malware tries to download a file, performs a check, and notices that it did not get the proper file, it might stop performing further HTTP requests. Yet, some of the investigated malware samples exactly did the opposite and tried to connect to various other download locations. As a result, 12 of the 154 ANUBIS samples performed more requests than stated in the ANUBIS reports, and we were able to extract information about these backup web and command and control servers in an automated way. This includes details about several other domains providing the same malicious file, but also information about completely different filenames and URLs.

Furthermore, four of the 154 samples revealed *more* details about IRC connections compared to the corresponding ANUBIS reports. The TRUMANBOX log files provide more information about IRC channels (because of network downtime of the originally targeted server) and commands performed by the malware. Even a connection to a previously unknown command and control server was established.

Overall, we achieved the best results with the FTP protocol. 20 out of the 154 TRUMANBOX log files provided more information about FTP connections than the corresponding ANUBIS reports. In the majority of cases, more uploads, downloads, and other file operations were logged by TRUMANBOX. Some samples sent the wrong FTP credentials to the target server, so ANUBIS was not able to log any FTP traffic except the unsuccessful login attempt. Since TRUMANBOX redirects the FTP connection to our local FTP server which accepts all login information regardless of the credentials sent by the malware, the samples were able to log in and perform various file operations. A further advantage of running a local FTP server lies in the fact that we can access and analyze all uploaded files in the FTP directory without extracting the file content from the network traffic logs.

Unfortunately, connection attempts to SMTP servers were not initiated by any of the 300 randomly selected samples, so neither ANUBIS reports nor TRUMANBOX log files included any details about SMTP connections. However, we expect to achieve similar results compared to the other network protocols.

4.3 Half Proxy Mode

In half proxy mode, TRUMANBOX redirects all traffic coming from the virtual machine to our local services, whereas TRUMANBOX itself tries to connect to the originally targeted servers to fetch banners and other information. Thus, the malware receives exactly the same information as it would get when it is directly connected to the Internet. As a consequence, TRUMANBOX log files contain exactly the information which is already covered by the ANUBIS reports. However, in some cases, TRUMANBOX log files include even more information, for example, when a command and control server is not

reachable anymore. Using ANUBIS, the malware sample does not perform any communication when the connection attempt failed. In contrast, the connection is redirected to our local running IRC service when using TRUMANBOX, which is available the whole time. Hence, the malware performs its regular communication with our service which is then logged to the TRUMANBOX report. We found that operating TRUMANBOX in half proxy mode results in receiving at least the same information we would obtain by using a "sandbox" environment like ANUBIS and CWSANDBOX. In special cases like downtimes of command and control servers or invalid FTP credentials sent by the malware, TRUMANBOX is capable of obtaining more information about the performed steps of a malware sample.

5 Conclusion

We presented TRUMANBOX, an approach to improve dynamic malware analysis by emulating the Internet. We presented the design and implementation of TRUMANBOX and performed an evaluation. In conclusion, the full emulation mode offered by TRUMANBOX provides us with the opportunity to obtain all information retrieved by "sandbox" environments in most cases. Due to the fact that we do not access the Internet at all, some of the reports, however, might only include parts of the traffic logged by tools accessing the Internet. Yet, TRUMANBOX is effective when the originally targeted servers are unavailable or when malware samples perform failed login attempts.

There are many possible improvements that may be implemented in the freely available source code of TRUMANBOX [4]. For example, full emulation mode can be improved by making it less static, e.g., by randomizing server responses. Also, to thwart fingerprinting attempts by random login and password guessing, a mechanism for conditionally repeated execution of a malware on the client side would be useful. However, this would require a communication channel to the client side. Also, to improve the analysis of IRC channels, TRUMANBOX might send self-generated `NICK random` commands when it detects the lack of `NICK` in the first payload sent to the IRC server.

Acknowledgments. This work was supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant 315-43-02/2-005-WFBO-009), the Federal Ministry of Education and Research (Grant 01BY1020 – MobWorm), and the German Bundesamt für Sicherheit in der Informationstechnik (BSI). We thank Truman Burbank, Peter Weir, and the anonymous reviewers for their valuable insights and comments.

References

1. Barford, P., Blodgett, M.: Toward botnet mesocosms. In: Proceedings of the USENIX First Workshop on Hot Topics in Understanding Botnets, HotBots I (April 2007)
2. Bayer, U., Moser, A., Krügel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology 2(1), 67–77 (2006)
3. Chamales, G.: The Honeywall CD-ROM. IEEE Security & Privacy Magazine 2(2), 77–79 (2004)

4. Gorecki, C.: TrumanBox – Internet Emulation (2011), http://trumanbox.s6y.org
5. Flux Group. Emulab: Network emulation testbed home, Internet: http://www.emulab.net (accessed July 2007)
6. Hungenberg, T., Eckert, M.: INetSim – Internet Simulation (2011), http://www.inetsim.org
7. International Secure Systems Lab. Anubis: Analyzing unknown binaries (2011), http://anubis.iseclab.org
8. Morris, J.: libipq: iptables userspace packet queuing library, Internet: https://svn.netfilter.org/netfilter/trunk/iptables/libipq (accessed July 2007)
9. Claus, R.F.: Overbeck. Botspy – efficient observation of botnets. Presentation at Hack.lu (October 2007)
10. The Honeynet Project. Know Your Enemy: Learning About Security Threats, 2nd edn. Addison-Wesley, Reading (2004)
11. Provos, N.: A Virtual Honeypot Framework. In: Proceedings of the 13th USENIX Security Symposium, pp. 1–14 (2004)
12. Provos, N., Holz, T.: Virtual Honeypots: From Botnet Tracking to Intrusion Detection, 1st edn. Addison-Wesley, Reading (2007)
13. Willems, C., Holz, T., Freiling, F.C.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security & Privacy Magazine 5(2), 32–39 (2007)

Rendezvous Tunnel for Anonymous Publishing: Clean Slate and Tor Based Designs*

Ofer Hermoni[1], Niv Gilboa[2,3],
Eyal Felstaine[1], Yuval Elovici[1,3], and Shlomi Dolev[2]

[1] Department of Information Systems Engineering
Ben-Gurion University of the Negev, Israel
[2] Department of Computer Science
Ben-Gurion University of the Negev, Israel
[3] Deutsche Telekom Labs at
Ben-Gurion University of the Negev, Israel

Abstract. Anonymous communication, and in particular anonymous Peer-to-Peer (P2P) file sharing systems, have received considerable attention in recent years. In a P2P file sharing system, there are three types of participants: publishers that insert content into the system, servers that store content, and readers that retrieve content from the servers. Existing anonymous P2P file sharing systems confer partial anonymity. They provide anonymity to participant pairs, such as servers and readers or publishers and readers, but they do not consider the anonymity of all three types of participants.

In this work we propose two solutions for anonymous P2P file sharing systems. *Both of our solutions provide anonymity to all three types of participants*. The proposed solutions are based on indexing by global hash functions (rather than an index server), dispersal of information, and three anonymity tunnels. Each anonymity tunnel is designed to protect the anonymity of a different user (publisher, server, or reader). In both solutions the reader and publisher tunnels are sender anonymity tunnels. In the first solution the third tunnel is a rendezvous tunnel, constructed by means of a random walk and terminating at the server. In the second solution, which is based on Tor, the third tunnel is built using Tor's hidden services.

The first solution preserves anonymity in the presence of a semi-honest adversary that controls a limited number of nodes in the system. The second solution is based on Tor primitives, coping with the same adversary as that assumed in Tor. The second solution enhances Tor, ensuring publisher, server, and reader anonymity.

* This research has been supported by the Ministry of Science and Technology (MOST), the Israel Internet Association (ISOC-IL), the Lynne and William Frankel Center for Computer Science at Ben-Gurion University, Rita Altura Trust Chair in Computer Science, the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Microsoft, US Air-Force, *Israel Science Foundation* (grant number 428/11), Verisign 25th Anniversary of .COM grant and Deutsche Telekom Labs at BGU. A poster presenting preliminary results of this work was presented in CCS '10 [13].

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 223–237, 2011.

1 Introduction

On-line communication occupies a great part of the daily lives of many people, increasing the popularity of peer-to-peer (P2P) systems, as we use them to chat, talk, share files, etc. Anonymous systems have thus become an important area of research [1], [4], [6], [7], [9], [14], [23] and [24] and implementation [8]. Anonymity in P2P file sharing means that an adversary cannot link participating users to the content they share. Note that an adversary can act as a user, and therefore, the users must also remain hidden with respect to one another.

A weakness common to many existing anonymity solutions is that they do not provide (or even consider) anonymity for *all* participating users, namely, for the publisher, the server and the reader. Solutions such as those given in [4], [22] and [26] provide anonymity to the reader but they do not protect the server that stores the document. Publishing solutions such as [7] and [16] provide anonymity for the publisher and the reader, but they do not protect the server(s) that stores the index.

In this paper we describe two anonymous P2P file sharing networks. Our solutions are based on indexing by a global hash function, dispersal of information, and *anonymity tunnels*. The indexing mechanism allows us to find documents without the need for an index server. Dispersal of information makes our solutions more resistant to node failure and denial of service attacks. Anonymity tunnels are widely used both in theory and in practice. Tor [8], for example, uses circuits (i.e. tunnels) to provide anonymity. A tunnel is an ordered set of nodes where each node has auxiliary information identifying its predecessor and successor in the tunnel.

The first solution – *Rendezvous Tunnel for Anonymous Publishing* (RTAP) – is extensively explained in this paper. In RTAP, the anonymity of all users is achieved by using three anonymity tunnels (Figure 1), separated into two types of anonymity tunnels.

The publishing and reading tunnels are sender anonymity tunnels. A sender anonymity tunnel is designed to protect the anonymity of the message's sender. For example, in Figure 1, the entrance node U_1 does not know the identity of the publisher P (or the reader R), since they communicate through a sender anonymity tunnel. However, the publisher P (or the reader R) knows the identity of the entrance node U_1. Anonymity of a sender anonymity tunnel is usually achieved using a Mix [4] based protocol. The third tunnel is a rendezvous tunnel, which is quite different. U_1 initiates the tunnel and each node extends the tunnel by randomly choosing its successor. Hence, the initiator of the tunnel does not know the identity of the user at the end of the tunnel and vice versa (U_1 does not know the identity of the server, and the server does not know the identity of U_1). The reading and the publishing tunnels protect the anonymity of the reader and the publisher, respectively, whereas the rendezvous tunnel protects the anonymity of the server. This solution holds for a semi-honest adversary.

Servers in RTAP store shares of documents, such that each share is published and retrieved through a rendezvous tunnel constructed between the server and

an address given by a hash of the document's name. To publish a document, the publisher first divides the document into shares. For each share, the publisher finds the address of the entrance node to the tunnel. Next, the publisher uses anonymous communication (publishing tunnel) to send the share to the entrance node of the rendezvous tunnel. Then a random walk is used to build the rendezvous tunnel and to send the share to the server. The reader that wants to retrieve the document behaves similarly. The reader finds the addresses of the entrance nodes to the rendezvous tunnels by hashing the document's name and the shares' numbers. The reader then uses anonymous communication (reading tunnel) to reach the entrance of the tunnels, retrieves the shares anonymously, and reconstructs the document.

The second solution uses Tor [8] to provide anonymity in a P2P file sharing network. Similar to RTAP, in this solution (which is outlined in Section 5), documents are divided into shares. Each share is published and retrieved through three anonymity tunnels. The publishing and the reading tunnels are regular Tor tunnels, whereas the rendezvous (server) tunnel is a Tor's hidden services tunnel. This scheme copes with the same adversary as that assumed in Tor.

The main advantage of the first solution over the second solution is that in addition to publisher, server, and reader anonymity, the first solution provides document anonymity. Document anonymity (see definition in Section 4) means that the server does not have information on the content it holds. The main advantage of the second solution compared to the first solution is that it is resistant to attacks by a more powerful adversary. Specifically, the second solution maintains anonymity against the same adversary that is assumed in Tor.

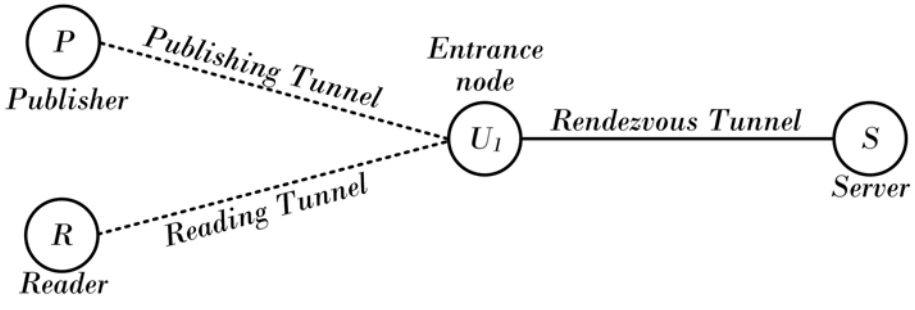


Fig. 1. Three-tunnel system. The publishing and the reading tunnels are sender anonymity tunnels.

Related Work. The core concept of providing Internet anonymity goes back to the early days of the public network and has been extensively studied ever since. Chaum [4] proposed using an intermediary proxy (relay server or Mix) whose aim is to hide reader identity from the server. Later works developed Chaum's approach by deploying predefined or ad-hoc paths.

Previous works can be divided into two categories. The first category includes approaches that provide a high degree of anonymity but whose costs include high communication overhead (e.g. [1], [5] and [9]) to ensure predefined traffic statistics. In practice, such solutions require continuous communication, hence the statistics do not reveal activities.

The Dining Cryptographers network [5] is an anonymous broadcast protocol based on the dining cryptographers problem. DC-net provides very strong sender and receiver anonymity but at the cost of broadcast. Due to this expensive broadcast communication, the scheme suffers from poor scalability and is unsuitable for large-scale use. The amount of communication can be reduced relative to DC-nets by using Xor-Trees [9].

Buses [1] also consider the anonymity of the sender and the receiver. Inspired by the observation that public transit buses hide the movement patterns of passengers. In the buses scheme, passengers are pieces of information that are allocated seats in a bus that traverses the network. Buses aim to hide the traffic patterns and to prevent an external adversary from forming a link between two communicating parties. This scheme provides sender-receiver unlinkability at the cost of traffic even when no information is transferred. However, the sender must know the intended receiver and vice versa, and therefore, the anonymity of both sender and receiver with respect to each other are not supported.

The second category consists of approaches based on the assumption that traffic patterns do not reveal information, and they provide different levels of anonymity with correspondingly different levels of communication overhead. The Java Anon Proxy [2] (JAP or WebMIXes) provides anonymity by using *cascades* – fixed tunnels that are shared among all users. A user of JAP selects a cascade to communicate with another user. As such, users are aggregated into larger anonymity sets. However, user anonymity is not well maintained outside of the cascade.

Onion Routing [26] uses a fixed, predefined path that is essentially a list of intermediate proxies leading to the destination. The major advantage of onion routing is that relays cannot unravel the information received or determine the destination address. Tor [8], is an advanced, low-latency scheme that improves the original Onion Routing, is a tunnel-based system that constructs circuits in stages, extending the circuit one hop at a time. Tor provides sender anonymity and sender-receiver unlinkability. In addition, Tor provides receiver anonymity through *rendezvous points* and *hidden services*. The main shortcoming of Tor's hidden services is that they are provided by the content owner, i.e., Tor does not consider publishing. In this work we address this limitation by publishing the content. Another related solution is the tunnel-based scheme CIAS [24], a low latency P2P scheme that provides anonymity for senders and receivers while adhering to strict and low bounds of delay, communication and bandwidth overheads. However, CIAS also does not consider publisher anonymity.

The works described above were designed to provide anonymity and unlinkability to the reader and to the server of a message. But none of them consider P2P networks with anonymous publishing phases. Freenet [6], a solution that

addresses this problem, provides some degree of server and publisher anonymity. When a user requests a document, it uses a document identifier to send the query without being aware of the server's identity or location. The weakness of Freenet is that the tunnel length is not controlled by the user. The tunnel may be as short as one node, and an adversary that controls the tunnel can revoke the anonymity of the reader, the server and the publisher.

A different technique used to provide anonymity employs secret sharing schemes to break data items into several parts and distribute them among different servers. In Publius [16], the content is encrypted by a key and stored in a fixed set of servers. The encryption key is shared by Shamir's secret sharing and distributed to the servers. Retrieval comprises reconstructing the key, retrieving the encrypted document, and decrypting it. Publius does consider publisher anonymity, however, the documents are stored in a static list of available servers, and the index is not protected. Another approach that uses secret sharing is Deniability [14], taking a different approach, instead of trying to hide the identity of the users, the deniability technique blurs the connection between pieces of information and their meanings.

Free Haven [7] is an anonymous publishing system, comprises several servers, known as servnets, which agree to store and provide documents for anyone. The identities of these servnets are publicly known, and communication is carried out over a Mix-based communication layer. Free Haven provides a certain level of publisher, reader and document anonymity, but it lacks server anonymity.

Organization. Section 2 discusses the system settings while Section 3 describes the solution. We give an anonymity analysis of our system in Section 4. A description and analysis of the Tor-based solution is briefly given in Section 5.

2 Settings and Requirements

Participants. In P2P file sharing networks, information is stored in units called *documents*. The *publisher* of a document is the entity that placed the document in the system. The *server* of a document is an entity that stores and distributes the document. Readers retrieve documents from servers.

Name-Index. Document retrieval in any system relies on a mapping from the name of the retrieved object to the index of its location in the system. The name-index mapping enables a user to find the location of a specific document. Usually the name-index mapping is stored in a database or databases called *index server(s)*. However, the index server is a threat to anonymity on several levels. First, if the adversary monitors the name-index server, the anonymity of the users that communicate with it (the publisher that publishes a new document in the index and the reader that searches for a document) may be compromised. Moreover, if the name-index mapping includes controversial content, legal procedures may be invoked against the owner of the index server.

Our system design has no name-index server, index mapping is performed locally by the publishers and the readers themselves. The documents in the system are divided into shares, such that each share in the system has a three-parameters index entry. First, the entrance node to a tunnel, denoted by U_1. Second, U_1's session identification tag, denoted by ID_1. And the reader's seed (which is used to encrypt the share) denoted by s_0. The publisher (and later the reader) calculates the index entry according to the document name doc_{name} and the share number (see Algorithm 1).

The seed and the session identification tag are calculated by using a global hash function, whereas the the entrance node is calculated by a *Distributed Hash Table* (DHT), e.g., chord [25]. DHT receives a key as an input and outputs an identifier of a node. The DHT in our system maps the document name to the entrance node rather than to the location of the server itself (the conventional way to use DHT), in order to provide server anonymity.

Anonymity Model. In this work, we design a system that provides anonymity to all participants in a P2P file-sharing network: the publishers, the servers, and the readers. Several types of anonymity can be defined with respect to P2P networks. We adopt the definitions and terms of Pfitzmann and Hansen [20] combined with the terminology of Dingledine at al. [7] to define the anonymity of participating users (see formal definitions in Section 4). *Reader anonymity* means that an adversary has no way of knowing which reader on the network has retrieved a particular document. *Server anonymity* means an adversary has no way of knowing which server on the network has served this document or currently stores it. *Document anonymity* means that a server does not know which documents it is storing. *Publisher anonymity* means that an adversary has no way of knowing which user on the network has published a particular document. Note that the users also maintain their anonymity with respect to each other. For example, the reader maintains its anonymity even when retrieving a document from a server, such that the server does not know the reader's identity and vice versa. In particular, a server functioning as a reader does not know whether the document it retrieved came from itself.

According to the terminology of [20], anonymity means that a user is unidentifiable within a group of users, the *anonymity set*. The unlinkability of items (e.g., users, messages) means that an attacker cannot sufficiently distinguish whether the items are related. Sender-receiver unlinkability protects against the adversary that is neither the sender nor the receiver of the message.

Adversary Model. In anonymous P2P networks, the participating entities, the server, the reader, and the publisher do not want their identities to be revealed. The adversary's goal is to link specific content to a participating user in the system and as such to identify the user.

Similar to other schemes for anonymous networks, the adversary in our model is assumed to be *semi-honest*, which means that the adversary can control nodes in the network but is obligated to follow the algorithm. We assume that the

adversary can control at most t network nodes, where t is smaller than the total number of nodes. We also assume that the communication patterns and statistics do not reveal information, i.e., the statistics and patterns of message traffic are distributed in a fixed (say normal) distribution. Such an assumption is used when mixes schemes [4] are employed, for example, when mixes are used and only one message is sent in the entire network; in such a case, the identities of the sender and of the receiver are obviously revealed.

3 RTAP Architecture

In this section we explain our solution via the steps the publisher takes to publish a document (Algorithms 1-2) and the steps the reader takes to retrieve a document (Algorithms 3-6). For each algorithm we explain in short the goals of the algorithm and we give a pseudo code. In Section 4 we analyze and prove the anonymity of the solution.

There are two phases in this solution. The first phase is the *publication*, in which the publisher sends encrypted shares to the servers. During the second phase, *retrieval*, the reader retrieves the shares from the servers and reconstructs the document.

The overall flow of the solution is as follows. First, the publisher divides the document into shares using an (n, k) Information Dispersal Algorithm (IDA) [21]. IDA divides a document into n shares, such that each subset of k shares out of the n shares ($n > k$) is sufficient to reconstruct the document. For each share, the publisher then uses hash functions to build the index mapping. Next, the publisher encrypts the share and sends it to the entrance node through a sender anonymity tunnel, the publishing tunnel. The entrance node encrypts the share, initiates a random walk, and sends the encrypted share through the rendezvous tunnel to the server. Each node along the rendezvous tunnel encrypts the share and forwards it to the server.

A reader, that wants to retrieve the document operates in a similar manner. For each share, the reader constructs the index in the same way, and then uses a sender anonymity tunnel (the reading tunnel) to query the entrance node, which forwards the query message to the server through the rendezvous tunnel. As soon as the query arrives at the server, the server sends the share back along the rendezvous tunnel. Each user along the rendezvous tunnel decrypts the share. Using the reading tunnel, the entrance node then sends the share to the reader, which decrypts the share. As soon as enough shares have been successfully retrieved, the reader reconstructs the document.

3.1 Publication

Dividing and Sending the Document: In the first part of the publication phase (Algorithm 1) the publisher divides the document into shares, for each share it creates an index entry, finds the entrance node, encrypts the share, and by using a sender anonymity tunnel, it sends the encrypted share to the entrance node.

Algorithm 1. Dividing and sending the document

Publisher:
$\{sh_1, sh_2, \cdots, sh_n\} \leftarrow IDA(d)$ {dividing the document into n shares}
for $j = 1$ to n **do**
  $U_1 \leftarrow DHT(doc_{name}||j)$; {entrance node}
  $ID_1 \leftarrow hash_1(doc_{name}||j)$; {entrance node's ID}
  $s_0 \leftarrow hash_2(doc_{name}||j)$; {reader seed}
  $\hat{sh}_j \leftarrow sh_j \oplus G(s_0)$; {encrypt the share with reader seed}
  $anon\_send(insertShareMsg(ID_1, \hat{sh}_j, length))$ *to* U_1; {send the encrypted share to U_1 through a publishing tunnel, the rendezvous tunnel is of length $length$}
end for

Insert Share Message: The goal of the second part of the publication phase (Algorithm 2) is to build the rendezvous tunnel. During construction of the rendezvous tunnel, the share is sent to the server. Each node i encrypts the share and forwards the message to the next node along the rendezvous tunnel until the share arrives at the server.

Algorithm 2. Insert Share Message

Node *i*:
$receive(insertShareMsg(ID, \hat{sh}, length)$;
if $length > 0$ **then**
  $lastNode \leftarrow FALSE$; {this is not the last node}
  $s_i \leftarrow random()$; {node's session key}
  $\hat{sh} \leftarrow \hat{sh} \oplus G(s_i)$; {encrypt the share with session key}
  $U_{i+1} \leftarrow DHT(random())$; {choose at random the next node along the rendezvous tunnel}
  $ID_{i+1} \leftarrow hash_1(random())$; {next node's session ID}
  $send(insertDocMsg, ID_{i+1}, \hat{sh}, length - 1)$ *to* U_{i+1}; {forward the message along the rendezvous tunnel}
else
  $lastNode \leftarrow TRUE$; {this is the server}
  save $\hat{sh}$; {the server saves the encrypted share}
end if

3.2 Retrieval

The retrieval phase comprises four stages (Algorithms 3-6) – First the reader recalculates the index and sends a query to the entrance node of each share. Second, the query traverses the tunnel to the server. Then the server replies and sends the share to the reader. Finally, when enough shares have been collected, the reader decrypts the shares and reconstructs the document.

Initiate Queries: The reader calculates the index and sends query messages to the entrance nodes through the reading tunnels.

Algorithm 3. Initiate Queries

1: **Reader:**
2: **for** each share j **do**
3: $U_1 \leftarrow DHT(doc_{name}||j)$;
4: $ID_1 \leftarrow hash_1(doc_{name}||j)$;
5: $s_0 \leftarrow hash_2(doc_{name}||j)$; {the index entry}
6: $anon\_send(queryMsg, ID_1)$ *to* U_1; {send a query message to U_1 through a reading tunnel}
7: **end for**
8: GoTo Algorithm 6;

Query Message: Each node i along the rendezvous tunnel forwards the query message to the server.

Algorithm 4. Query Message

1: **Node *i*:**
2: $receive(queryMsg)$;
3: **if** $lastNode == FALSE$ **then**
4: $send(queryMsg, ID_{i+1})$ *to* U_{i+1}; {forward the query message to the next node in the tunnel}
5: **else**
6: GoTo Algorithm 5;
7: **end if**

Share Response Message: The server sends the share to the reader through the rendezvous tunnel. The entrance node sends the share to the reader.

Algorithm 5. Share Response Message

1: **Node *i*:**
2: **if** $lastNode == TRUE$ **then**
3: $send(shareRspnsMsg, ID_i, \hat{sh})$ *to* U_{i-1};
4: **else**
5: $receive(shareRspnsMsg)$;
6: $\hat{sh} \leftarrow \hat{sh} \oplus G(s_i)$; {remove the session key s_i}
7: $send(shareRspnsMsg, ID_i, \hat{sh})$ *to* U_{i-1};. {nodes forward the message back to U_1. Node U_1 sends the share to the reader through the reading tunnel}
8: **end if**

Reconstruct the Document: The reader collects the shares and reconstructs the document as soon as enough (k) shares were collected.

Algorithm 6. Reconstruct the Document

1: **Reader:**
2: $receive(shareRspnsMsg)$;
3: $sh \leftarrow \hat{sh} \oplus G(s_0)$; {the reader reconstructs the shares}
4: insert sh to $\{shares\}$ {the reader collects the shares}
5: **if** $|\{shares\}| = k$ **then**
6: $d \leftarrow IDA^{-1}(\{shares\})$ {the reader reconstructs the document}
7: **end if**

4 Anonymity Analysis

In this Section we provide a short anonymity analysis. The anonymity definitions are derived from Pfitzmann and Hansen [20] and the Free haven project [7]. A more complete analysis appears in a technical report [12].

Definitions

• **Anonymity Set** – Anonymity set is the set of all possible subjects. The anonymity set of the publisher is the group of all *currently active* publishers in the system. The anonymity set of the reader is the the group of all *currently active* readers in the system. The anonymity set of the server is the the group of all *currently active* servers in the system.

• **Publisher Anonymity** – The publisher of a document d is anonymous if any polynomially bound adversary, given the choice of two possible active publishers, cannot identify the correct publisher with probability greater than $1/2+\varepsilon$, where ε is a negligible function of the adversary's input length.

• **Reader Anonymity** – The reader of a document d is anonymous if any polynomially bound adversary, given the choice of two possible active readers, cannot identify the correct reader with probability greater than $1/2 + \varepsilon$.

• **Server Anonymity** – The server of a document d is anonymous if any polynomially bound adversary, given the choice of two possible active servres, cannot identify the correct server with probability greater than $1/2 + \varepsilon$.

• **Document Anonymity** – The server maintains document anonymity if the server can not deduce the content of the document it stores.

Anonymity Framework. We prove that our scheme provides anonymity to publisher, reader, and server given the following assumptions.

A path between a publisher (or reader) and a server can be logically divided into two parts: the nodes from the publisher (or reader) to U_1 and the nodes from U_1 to the server. Let t be a system parameter, and assume each of the two parts in each path includes at least $t+1$ nodes. Thus, if a path has T nodes then $T \geq 2t + 2$.

Assume that an adversary controls at most t nodes along the path between the publisher (or reader) and the server. Furthermore, assume that the adversary is *semi-honest* and *static*. In other words, the adversary follows protocol specifications exactly and controls the same subset of parties throughout the

execution of a protocol. Assume further, that the adversary obtains information only through the execution of the protocols of Section 3.

Given such an adversary, we prove the anonymity of each user even if the adversary controls t nodes, including the other two users. For example, we prove the anonymity of the server even if the adversary controls both the publisher and the reader. We prove anonymity by showing that the adversary cannot distinguish between a user that processes (i.e. serves or reads) a document that the adversary recognizes and a user that processes a completely unrelated document.

Note that a more powerful adversary, e.g. one that correlates the timing of packets in different parts of a path between publisher, reader, and server may be able to learn information that is unavailable to our adversary.

Notation. We follow standard notation and definitions for multi-party secure computation, [10], [11]. Let Π be a protocol for T parties to compute a function g. The input of the i-th party is denoted x_i and the output of the i-th party is $g_i(x_1, \ldots, x_T)$. An adversary controls a set of parties denoted by I and receives the "view" of every party in I. The view of a party includes its input, output and all intermediate messages that it receives.

In the case we investigate, $T \geq 2t + 2$, the publisher's input is (d, d_{name}), the reader's input is d_{name} and all other parties do not have an input. The output of a reader is d. For every node U_i in the rendezvous tunnel nodes, the output includes (i, s_i, ID_i, ID_{i+1}) and the length of the tunnel. The output of the server is $d \bigoplus_i G(s_i)$.

Let X and Y be two ensembles of probability distributions on binary strings. We say that X and Y are *computationally indistinguishable* and use the notation $X \stackrel{c}{\equiv} Y$ if any polynomial time algorithm can distinguish between X and Y with only negligible probability.

Anonymity Claims. Regarding which nodes the adversary controls, consider several nodes which are contiguous along the tunnel. The protocol ensures that if the adversary controls these nodes then it knows that they are part of the same tunnel. However, the tunnel has at least $t+1$ nodes on each of its legs (publisher to U_1, reader to U_1 and U_1 to server). If all the nodes that an adversary controls are contiguous then the adversary can not be sure that two users (either publisher and server or reader and server) are communicating and thus can not link them. Furthermore, in this case the adversary does not have any information on a user beyond the fact that it is acting as a publisher or a reader or a server. Thus, when all corrupted nodes are contiguous then anonymity is maintained.

Theorem 1. *Assume the existence of pseudo-random generators. Let an adversary control at most t nodes in the network. For any two nodes that the adversary controls, which are not connected by contiguous corrupt nodes in a single tunnel, the adversary can not distinguish whether the nodes are part of the same publishing / reading tunnel for a document d or part of two tunnels for a document d and a document* $\overline{d}$.

Theorem 2. *Assume the existence of pseudo-random generators. Let an adversary control at most t nodes in a network. Given the view of its nodes during the Algorithms 1-2 for a document d, the adversary does not obtain any information on d beyond the information it had prior to the protocol.*

Theorem 3. *Assume the existence of pseudo-random generators and let an adversary control at most t nodes in a network. Given view of the adversary's nodes during execution of all protocols in Section 3 for a document d the scheme maintains publisher, reader, server and document anonymity.*

Proof Sketch. Publisher and reader anonymity are naturally derived from the sender anonymity tunnels they use. The publisher and the reader that communicate with the entrance node U_1, always do so behind the protection of a sender anonymity tunnel. If we assume a strong sender anonymity tunnel such as Tor [8], the anonymity of the reader and the publisher is as strong as the tunnel they use to communicate with U_1.

We prove server anonymity based on Theorems 1 and the discussion at the beginning of the section. If the adversary controls t contiguous nodes in the rendezvous tunnel of d it is unable to identify the server since it has no information on the server's actions. In this case the anonymity set includes all servers in the network, not just the active servers.

If the adversary's nodes are not contiguous along the tunnel then Theorem 1 claims that the adversary can not distinguish between the messages of two different documents. Hence, the identity of a server serving d remains hidden.

Document anonymity is proved by Theorem 1 and Theorem 2. Document anonymity is a relevant concept only when the adversary controls the server. Theorem 2 claims that the adversary does not obtain any knowledge about the document during the publication phase. Since the adversary controls the server, it does not control all the nodes along the tunnel and is unable to distinguish between d and $\overline{d}$ in the server. Hence we obtain document anonymity.

5 Tor Based Solution

RTAP provides anonymity in a semi-honest model. We now describe a solution based on Tor that provides anonymity to three participants in a P2P file sharing network against a more powerful adversary. Specifically, our scheme gives the same anonymity assurance as Tor itself and is secure against the same adversary assumed in Tor. Tor's adversary is assumed to be able to observe, generate, modify, delete, or delay some fraction of network traffic. Moreover, the adversary can operate or compromise some fraction of the onion routers (for more details, see [8] Subsection 3.1 and Section 7). Later works showed some of the vulnerabilities inherent in Tor [18], [19] and [15].

Tor has two operating modes, the first of which is the *sender anonymity tunnel* that was designed to protect the anonymity of the sender of a message. In a sender anonymity tunnel where recipient identity is known by the sender,

anonymity is achieved by using a Mix [4] based protocol. The second operating mode of Tor is its *hidden services*, which were designed to allow content owners to provide their content anonymously. A content owner that wants to provide its content builds a Tor tunnel to an introduction point and publishes the introduction point in a global directory. Note that in Tor's hidden services, the publisher and the server are the same entity while in a P2P file sharing network, the publisher and the server may be separate entities.

Dissociation of the publisher from the server that stores its content is accomplished by each server in the system building one (or more) hidden services tunnel to an introduction point (see the right part of Figure 2). The servers then publish the introduction points for use by publishers.

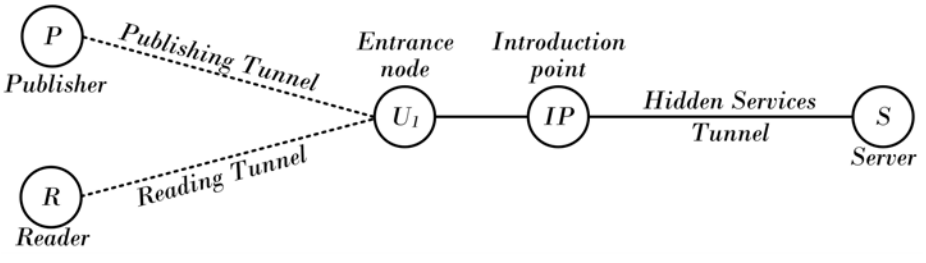


Fig. 2. Three-tunnel system. The publishing and the reading tunnels are sender anonymity tunnels, while the server uses Tor's hidden services.

The overall flow of the solution is as follows. First, the publisher divides the document into shares using an (n, k) IDA [21]. Then, for each share, the publisher uses hash functions and a DTH to build the index mapping. Next, the publisher encrypts the share and sends it to the entrance node, U_1, through a Tor sender anonymity tunnel, the publishing tunnel. The entrance node encrypts the share, selects at random an introduction point and sends the encrypted share through the hidden services tunnel to the server. The server stores the share.

A reader that wants to retrieve the document operates in a similar manner. For each share, the reader constructs the index using hash functions and a DHT, and uses a sender anonymity tunnel (the reading tunnel) to query the entrance node. The entrance node forwards the query message to the server through the hidden services tunnel. As soon as the query arrives at the server, it sends the share back along the tunnel. The entrance node decrypts the share, uses the reading tunnel, and sends the share to the reader. The reader then decrypts the share. As soon as enough shares have been successfully retrieved, the reader reconstructs the document. Note that the publisher and the reader perform essentially the same algorithms used in RTAP.

Discussion. The Tor based solution provides anonymity to all three participants in the P2P file sharing network. This solution copes with the adversary assumed in Tor. Publisher anonymity is achieved by the publishing tunnel while reader anonymity is achieved by the reading tunnel. Finally, server anonymity is

achieved by the hidden services tunnel. An adversary that can compromise the anonymity of the publisher/reader/server in the proposed solution can compromise the anonymity of a Tor user.

As described above, a DHT is used to determine U_1, the entrance node to the server (U_1 forwards the messages to the server through the introduction point). If we do not use U_1, the publisher (and the server) must communicate directly with the introduction point. In this case, our efforts to cope with an inefficient exhaustive dictionary attack may lead to the server deducing the content it stores. In such a event, document anonymity does not hold as the server determines the introduction point during the hidden services tunnel construction. The introduction of U_1 forces a more complicated attack against document anonymity, combining control of U_1 and the server and an inefficient exhaustive dictionary attack. Note that in RTAP document anonymity cannot be compromised.

References

1. Beimel, A., Dolev, S.: Buses for anonymous message delivery. Journal of Cryptology 16(1), 25–39 (2003)
2. Berthold, O., Federrath, H., Köpsell, S.: Web-MIXes: A System for Anonymous and Unobservable Internet Access. In: Federrath, H. (ed.) Designing Privacy Enhancing Technologies. LNCS, vol. 2009, pp. 115–129. Springer, Heidelberg (2001)
3. Boneh, D.: The decision diffie-hellman problem. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 48–63. Springer, Heidelberg (1998)
4. Chaum, D.: Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM 4(2) (February 1981)
5. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untranceability. Communication of the ACM 24(2) (1988)
6. Clarke, I., Sandberg, O., Wiley, B., Hong, T.: Freenet: A distributed anonymous information storage and retrieval system. In: Federrath, H. (ed.) Anonymity. LNCS, vol. 2009, pp. 46–66. Springer, Heidelberg (2001)
7. Dingledine, R., Freedman, M.J., Molnar, D.: The free haven project: Distributed anonymous storage service. In: Federrath, H. (ed.) Anonymity. LNCS, vol. 2009, pp. 67–95. Springer, Heidelberg (2001)
8. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. In: Proceedings of the 13th USENIX Security Symposium (August 2004)
9. Dolev, S., Ostrovsky, R.: Xor-trees for efficient anonymous multicast and reception. ACM Transactions on Information and System Security 3(2), 63–84 (2000)
10. Goldreich, O.: Foundations of Cryptography: Basic Tools. Cambridge University Press, New York (2000)
11. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press, New York (2004) [9]; O. Goldreich
12. Hermoni, O., Gilboa, N., Felstaine, E., Elovici, Y., Dolev, S.: Rendezvous Tunnel for Anonymous Publishing: Clean Slate and TOR Based Designs. TR 11-09 Department of Computer Science, Ben Gurion University of the Negev, Israel (2011)
13. Hermoni, O., Gilboa, N., Felstaine, E., Elovici, Y., Dolev, S.: Rendezvous Tunnel for Anonymous Publishing. In: CCS 2010, pp. 690–692 (2010)
14. Hermoni, O., Gilboa, N., Felstaine, E., Shitrit, S.: Deniability - an alibi for users in p2p networks. In: COMSWARE, pp. 310–317 (2008)

15. Ling, Z., Luo, J., Yu, W., Fu, X., Xuan, D., Jia, W.: A new cell counter based attack against tor. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009), pp. 578–589 (2009)
16. Marc Waldman, A.R., Cranor, L.: Publius: A robust, tamper-evident, censorship-resistant and source-anonymous web publishing system. In: Proceedings of the 9th USENIX Security Symposium, pp. 59-72 (August 2000)
17. Mittal, P., Borisov, N.: ShadowWalker: peer-to-peer anonymous communication using redundant structured topologies. In: Al-Shaer, E., Jha, S., Keromytis, A.D. (eds.) ACM Conference on Computer and Communications Security, pp. 161–172. ACM, New York (2009)
18. Murdoch, S.J., Danezis, G.: Low-cost traffic analysis of Tor. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, pp. 183–195. IEEE, Los Alamitos (2005)
19. Overlier, L., Syverson, P.: Locating Hidden Servers. In: IEEE Symposium on Security and Privacy, pp. 100–114 (2006)
20. Pfitzmann, A., Hansen, M.: A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management, v0.34 (August 2010)
21. Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. J. ACM 36(2), 335–348 (1989)
22. Reiter, M.K., Rubin, A.D.: Crowds: anonymity for Web transactions. ACM Transactions on Information and System Security 1(1), 66–92 (1998)
23. Serjantov, A.: Anonymizing censorship resistant systems. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, p. 111. Springer, Heidelberg (2002)
24. Shitrit, S., Felstaine, E., Gilboa, N., Hermoni, O.: Anonymity scheme for interactive p2p services. Journal of Internet Technology 10, 299–312 (2009)
25. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M., Dabek, F., Balakrishnan, H.: Chord: A Acalable Peer-to-Peer Lookup Protocol for Internet Applications. IEEE/ACM Transactions on Networking 11(1), 17–32 (2003)
26. Syverson, P., Goldsclag, D., Reed, M.: Anonymous connections and onion routing. In: Proceedings of the IEEE 18th Annual Symposium on Security and Privacy, Oakland, California, pp. 44–54 (1997)

Snake: Control Flow Distributed Software Transactional Memory

Mohamed M. Saad and Binoy Ravindran

ECE Dept., Virginia Tech
{msaad,binoy}@vt.edu

Abstract. Remote Method Invocation (RMI), Java's remote procedure call implementation, provides a mechanism for designing distributed Java technology-based applications. It allows methods to be invoked from other Java virtual machines, possibly at different hosts. RMI uses lock-based concurrency control, which suffers from distributed deadlocks, livelocks, and scalability and composability challenges. We present *Snake-DSTM*, a distributed software transactional memory (D-STM) that is based on the RMI as a mechanism for handling remote calls and transactional memory for distributed concurrency control, as an alternative to RMI/locks. Critical sections are defined as atomic transactions, in which reads and writes to shared, local and remote objects appear to take effect instantaneously. The novelty of Snake-DSTM is in manipulating transactional memory by moving control to remote nodes, rather than remote nodes' data being copied to the node at which the transaction runs. Transaction metadata is detached from the transactional context, and the dynamic two phase commitment protocol (D2PC) is employed to coordinate the voting process among participating nodes toward making distributed transactional commit decisions. We propose a simple programming model using (Java 5) annotations to define critical sections and remote methods. Instrumentation is used to generate code at class-load time, which significantly simplifies user-space end code. No changes are needed to the underlying virtual machine or compiler. We describe Snake-DSTM's architecture and implementation, and report on experimental studies comparing it against competing models including RMI with mutual exclusion and read/write locks, distributed shared memory (DSM), and dataflow-based D-STM. Our studies show that Snake-DSTM outperforms competitors by up to 12× on different workloads using a 120-node system.

1 Introduction

Lock-based concurrency control suffers from drawbacks including deadlocks, livelocks, lock convoying, and priority inversion. In addition, it has scalability and composability challenges [10]. These difficulties are exacerbated in distributed systems with nodes, possibly multicore, interconnected using message passing links, due to additional, distributed versions of their centralized problem counterparts [12]. Transactional memory (TM) promises to alleviate these difficulties.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 238–252, 2011.

In addition to providing a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking [13,11]. In TM, atomic sections are defined as *transactions* in which reads and writes to shared objects appear to take effect instantaneously. A transaction maintains its read set and write set, and at commit time, checks for conflicts on shared objects. If conflicts are detected, the transaction rolls-back its changes and retries; otherwise, the changes are made to take effect. Numerous multiprocessor TM implementations have emerged in software (STM) [29], in hardware (HTM) [11], and in a combination (Hybrid TM) [17]. Distributed STM (or D-STM) implementations also exist. Examples include Cluster-STM [5], D^2STM [7], DiSTM [14], and Cloud-TM [22]. Communication overhead, balancing network traffic, and network failures are additional concerns for D-STM.

Previous research on D-STM has largely focused on the dataflow model [32,18], in which objects are replicated (or migrated) at multiple nodes, and transactions access local object copies. Using cache coherence protocols [12,8,35], consistency of the object copies is ensured. However, this model is not suitable in applications (e.g., P2P), where objects cannot be migrated or replicated due to object state dependencies, object sizes, or security restrictions. A control flow model, where objects are immobile and transactions invoke object operations via remote procedure calls (RPCs), is appropriate in such instances.

This paper focuses on the design and implementation of D-STM based on Java's Remote Method Invocation (RMI) mechanism. We are motivated by the popularity of the Java language, and the need for building distributed systems with concurrency control, using the control flow model. Support for distributed computing in Java is provided using RMI since release 1.1. However, distributed concurrency control is (implicitly) provided using locks. Besides, the RMI architecture lacks the transparency required for distributed programming, supporting a remote method requires defining an interface, skeleton and stub objects, plus changing the prototype to throw remote exceptions and extending special base class `UnicastRemoteObject`. We present *Snake-DSTM*, an RMI/D-STM implementation that uses D-STM for distributed concurrency control in (RMI's) control flow model, and exports a simpler programming model with transparent object access. Using (Java 5's) annotations, and our instrumentation engine, a programmer can define *remote* objects (or methods), and define *atomic* sections as transactions, in which reads and writes to shared (local and remote) objects appear to take effect instantaneously. Distributed atomicity, object registration, and remote method declarations are handled transparently without any changes to the underlying virtual machine or compiler. Our experimental studies show that Snake-DSTM outperforms RMI with read/write locks by as much as 12*times* on a broad range of transactional workloads, and shows comparable performance to distributed shared memory, and dataflow D-STM. To the best of our knowledge, this is the first D-STM design and implementation in the control flow model, and constitutes the paper's contribution.

Snake-DSTM is freely available as part of the HyFlow project [23,24], which is producing a Java D-STM framework for the design, implementation, and

evaluation of D-STM algorithms and mechanisms, under both control flow and dataflow. We hope this will increase momentum in the TM community in D-STM research.

The rest of the paper is organized as follows. We overview past and related efforts in Section 2. In Section 3, we detail the Snake-DSTM design and implementation and underlying mechanisms. In Section 5, we experimentally evaluate Snake-DSTM against competing distributed programming models and report results. We conclude in Section 6.

2 Related Work

The high popularity of the Java language for developing large, complex systems has motivated significant research on distributed and concurrent programming models. DISK [30] is a distributed Java Virtual Machine (DJVM) for network of heterogenous workstations, and uses a distributed memory model using multiple-writer memory consistency protocol. Java/DSM [34] is a DJVM built on top of the TreadMarks [2] DSM system. JESSICA2 [36] provides transparent memory access for Java applications through a single system image (SSI), with support for thread migration for dynamic load balancing. These implementations facilitate concurrent access for shared memory. However, they rely on locks for distributed concurrency control, and thereby suffer from (distributed) deadlocks, livelocks, lock-convoying, priority inversion, non-composability, and the overhead of lock management.

TM, proposed by Herlihy and Moss [11], is an alternative approach for shared memory concurrent access, with a simpler programming model. Memory transactions are similar to database transactions: a transaction is a self-maintained entity that guarantees atomicity (all or none), isolation (local changes are hidden till commit), and consistency (linearizable execution). TM has gained significant research interest including that on STM [29], HTM [11], and HyTM [17]. STM has relatively larger overhead due to transaction management in software and architecture-independence. HTM has the lowest overhead, but assumes architecture specializations. HyTM seeks to combine the best of HTM and STM.

Similar to multiprocessor STM, D-STM was proposed as an alternative to lock-based distributed concurrency control. In [12], Herlihy *et. al.* classified distributed execution models into control-flow and dataflow models. In the control-flow model [4,16,31], objects are immobile and transactions invoke object operations through remote calls, resulting in a distributed locus of control flow movement — "distributed thread" [21] — for a transaction. On the other hand, in the dataflow model [32,18], objects are replicated (or migrated) at multiple nodes, and transactions access local copies. While the dataflow model preserves the locality of reference principle, it is not applicable in many cases in which objects cannot be transferred due to state, size, or security restrictions. Example dataflow D-STM implementations include Cluster-STM [5], D^2STM [7], DiSTM [14], and Cloud-TM [22]. Communication overhead, balancing network traffic, and network failure models are additional concerns for such designs. These

implementations are mostly specific to a particular programming model (e.g., the partitioned global address space or PGAS model [1]) and often need compiler or virtual machine modifications (e.g., JVSTM [6]), or assume specific architectures (e.g., commodity clusters). While dataflow D-STM has been intensively studied, relatively little efforts have focused on applying TM concepts under the control-flow model.

Snake-DSTM is a control-flow D-STM implementation, based on the Java RMI mechanism for supporting remote procedure calls. Unlike [1,6], it doesn't require any changes to the underlying virtual machine or compiler, as it uses embedded library as a JVM agent, which is loaded at runtime.

3 System Overview

3.1 System Model

We consider an asynchronous distributed system model, similar to Herlihy and Sun [12], consisting of a set of N nodes $N_1, N_2, ..., N_n$, communicating through weighted message-passing links. We assume that each shared object has an unique identifier. We use a grammar similar to the one in [9], but extend it for distributed systems.

A transaction is a sequence of instructions that are guaranteed to be executed atomically. Any object changes within transactional code must appear to take effect instantaneously. Each transaction has an unique identifier, and is invoked by a node (or process) in a distributed system of N nodes. A transaction can be in one of three states: *active*, *busy*, and *aborted*, or *committed*. When a transaction is aborted, it is retried by the node again using a different identifier.

Objects are resident at their originating nodes. Every object has, one "owner" node that is responsible for handling requests from other nodes for the owned object. Any node that wants to read from, or write to an object, contacts the object's owner using a remote call. A remote call may in turn make other remote calls, which construct, at the end of the transaction, a global graph of remote calls. We call this graph, a *call graph*.

3.2 Programming Model

The Java RMI specifications require defining a `Remote` interface for each remotely accessible class, and modifying class signatures to throw *remote* exceptions. Server side should register the implementation class, while client uses a delegator object that implements the desired `Remote` interface.

In our model, a programmer annotates remotely accessible methods with the `@Remote` annotation, and critical sections are defined as methods annotated with `@Atomic`. An object that contains at least one `@Remote` method is named *remote object*, and it must implement the `IDistinguishable` interface to provide our registry with a unique object identifier. Remote objects register themselves automatically at construction time, and are populated to other node registries. A transactional object is one that defines one (or more) `@Atomic` methods. Atomic

```
1 public class SearchAgent implements IDistinguishable {
2  public Object getId() {
3    return id;
4  }
5    @Remote
6    @Atomic{retries = 10}
7    public List search(String keyword) {
8      List found = new LinkedList();
9        // search at neighbors
10     for(String neighbor: neighbors){
11       SearchAgent remoteAgent = Locator.open(neighbor);
12       found.addAll( remoteAgent.search(keyword) );
13     }
14     .... // search at local database
15     return found;
16   }
17 }
```

Fig. 1. A P2P agent using an atomic remote TM method

annotation can be, optionally, parametrized by the maximum number of transactional retries. Currently, we support the *closed nesting* model [20], which extends the isolation of an inner transaction until the top-level transaction commits. We "flatten" nested transactions into the top-level one, resulting in a complete abort on conflict, or allow partial abort of inner transactions.

Transactional or remote objects are accessed using *locators*. Traditional object references cannot be used in a distributed environment. Further, locators monitor object accesses and act as early detectors for possible transactional conflicts. Objects can be located (or opened) in read-only or read-write modes. This classification permits concurrent access for concurrent read transactions.

Figure 1 shows a distributed transactional code example. A peer-to-peer (P2P) file sharing agent atomically searches for resources and return a list of resources owners to the caller node. The agent may act recursively and propagate the call to a set of neighbor agents. At the programming level, no locks are used, the code is self-maintained by retrying on failures, and atomicity, consistency, and isolation are guaranteed (for the `search` transaction). Composability is also achieved: any other atomic method can be called within the higher-level atomic `search` operation. A conflicting transaction is transparently retried. Note that the location of the agents is hidden from the program. It is worth noting that other distributed programing models such as DSM or dataflow D-STM cannot be used in such applications, as an agent must search files at its node. This is an example of objects with system-state property.

4 Implementation

Figure 2 shows a layered architecture of our implementation. Similar to the official RMI design, we have the three layers of: 1) *Transport Layer*, where

actual networking and communication handling is performed, 2) *Remote Reference Layer*, which is responsible for managing the "liveliness" of the remote objects, and 3) *Stub/Skeleton Layer*, which is responsible for managing the remote object interface between hosts. Additionally, we define an *Object Access Layer*, which provides the required transparency to the application layer. Local and remote objects are accessed in an uniform manner, and a dummy object is created to delegate calls to the RMI stub. Transactional code is maintained by a *Transaction Manager* module, which provides distributed atomicity and memory consistency for applications. As described in Section 4.1, an *Instrumentation Engine* is responsible for load-time code modifications, which is required for the Transaction Manager and Object Access Layer.

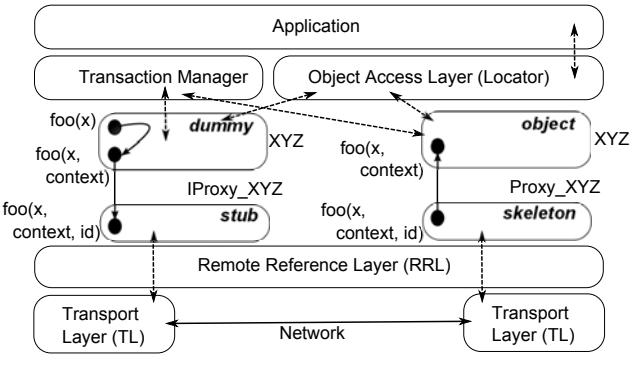


Fig. 2. Snake-DSTM layered architecture overview

4.1 Instrumentation Engine

Java *Instrumentation* provides a run-time ability to modify and generate bytecode at class load-time. We exploited this feature to modify class code at runtime, add new fields, modify annotated methods to support remote and transactional behavior, and generate helper classes. We built our engine as an extension of the Deuce (multiprocessor) STM [15]. We consider a Java method as the basic annotated block. This approach has two advantages. First, it retains the familiar programming model, where `@Atomic` replaces `synchronized` methods and `@Remote` substitutes for RMI calls. Secondly, it simplifies transactional memory maintenance, which has a direct impact on performance. Transactions need not handle local method variables as part of their read or write sets.

Our Instrumentation Engine works in two phases; the first phase processes *remote objects*. For any class with one (or more) methods annotated as `@Remote`, a `Remote` interface is generated with the remote method's signature. Further, a delegator class that implements the `Remote` interface is generated to work as the RMI-client stub. The original class constructors are modified to register objects at the object registry and populate object IDs to other nodes. That has two purposes: i) objects are accessed with a reference of the same type, so objects and object proxies are treated equally and transparently; and ii) no changes

to remote method signatures are required, as the modified signature versions are defined by delegator generated code. This phase simplifies the way remote objects are accessed, and reduces the burden of writing complex code.

The second phase handles transactional code generation. This transformation occurs as follows:

- ***Classes.*** A synthetic field is added to represent the state of the object as local or remote. The class constructor(s) code is modified to register the object with the Directory Manager at creation time.
- ***Fields.*** For each instance field, setter and getter methods are generated to delegate any direct access for these fields to the transaction context. Class code is modified accordingly to use these methods.
- ***Methods.*** Two versions of each method are generated. The first version is identical to the original method, while the second one represents the transactional version of the method. During the execution of transactional code, the second version of the method is used, while the first version is used elsewhere.
- ***@Atomic methods.*** Atomic methods are duplicated as described before, however, the first version is not similar to the original implementation. Instead, it encapsulates the code required for maintaining transactional behavior, and it delegates execution to the transactional version of the method.

Figure 3 shows part of the instrumented version of a `SearchAgent` class defined in Figure 1.

4.2 Distributed Software Transactional Memory

Supporting shared memory-like access in distributed systems requires an additional level of indirection. Each transaction must preserve memory consistency, and must expose its local changes instantaneously. In order to do that, old or new values of modified objects must be stored at local-transaction buffers till commit time. Two strategies can be used to achieve this: i) *undo-log* [17], where changes are made to the main object, while old values are stored in a separate log; and ii) *write-buffer* [10], where changes are made to transaction-local memory and written to the main object at commit time. Both strategies are applicable in the distributed context. However, (distributed) transactions cannot move between nodes during their execution with all these metadata (undo-logs or write-buffers) due to high communication costs. Instead, transaction metadata must be detached from the transaction context, while keeping the minimal information mobile with the transaction. In Snake-DSTM, we implemented both approaches. Using a distributed mechanism for storing transaction read-set and write-set, distributed transactions are managed with minimum amount of mobile data (e.g. transaction id, priority). The complete algorithm and more implementation details are available in a technical report [26].

Before (and after) accessing any transactional object field, the transaction is consulted for read (or written) value. A transaction builds up its write and read sets, and handles any private buffers accordingly. At commit time, a distributed

```
1 // Generated Remote interface
2 interface $HY$_ISearchAgent
3       extends Remote, Serializable{
4   public List search(Object id, ControlContext context, String
        keyword) throws RemoteException;
5   ....
6 }
7 // Generated Proxy delegator stub
8 class $HY$_Proxy_SearchAgent
9       extends UnicastRemoteObject
10          implements $HY$_ISearchAgent{
11        ....
12  }
13 public class SearchAgent implements IDistinguishable {
14  // Remote Proxy referece
15  $HY$_ISearchAgent $HY$_proxy;
16  // Modified constructor
17  SearchAgent(String id){
18    ....
19    DirectoryManager.register(id, this);
20  }
21  // Synthetic duplicate method
22    public List search(String keyword, Context c) {
23      if($HY$_proxy!=null)     //Invoke remote call
24        return $HY$_proxy.search(id, c, keyword);
25      ....     //execute call locally
26    }
27    // Original method instrumented
28    public List search(String keyword) {
29      //Transaction active thread
30      Context context = ContextDelegator.getInstance();
31    boolean commit = true;
32    List result = null;
33      for (int i=10; i>0; --i) {
34        //Initialize transaction
35      context.init();
36      try{
37        result=search(keyword, context);  //Try execute
38      } catch(TransactionException ex) {
39        commit = false;  //Aborted
40      } catch(Throwable ex) {
41        throwable = ex; //Application Exception
42      }
43      if(commit){
44        if (context.commit()) {
45          if (throwable == null)
46            return result; //Committed
47          throw (IOException)throwable; //Rethrow Exception
48        }
49     }else{
50        context.rollback(); //Rollback
51        commit = true;
52     }
53    }
54    throw new TransactionException();  //Maximum Retries
55    }
56 }
```

Fig. 3. Instrumented version of SearchAgent class

Undo Log (Eager-Pess)

On Write
If(owned) resolve
set owned by me
Backup and Change in master copy
On Read
If(owned) resolve
Read value and version
Try Commit
Validate reads (version < current)
On Commit
Increment owned versions
Release owned
On Rollback
Undo changes for owned
Release owned

Write Buffer (Lazy-Opt)

On Write
Change in private copy
On Read
If(in Write Set) read local value
else read master copy value
Read version
Try Commit
Acquire ownership of write-set
Validate reads (version < current)
On Commit
Write values to main copy
Increment owned versions
Release owned
On Rollback
Discard local changes

Fig. 4. Snake D-STM implementations

validation step is required to guarantee consistent memory view. In this phase, transaction originator nodes trigger a voting request to the participating nodes. Each node uses its portion of write and read sets to make its local decision. If validation succeeds on all nodes, the transaction is committed; otherwise, an abort handler rolls-back the changes. During the validation phase, the transaction state is set to *busy*, which ensures that a transaction cannot be aborted. This helps in ensuring the correctness of the validation (i.e., all nodes unanimously agree on the transaction to be committed and the transactions to be aborted), and also, it prevents transactions at later stages from being aborted by newly started ones.

Figure 4 shows our two implementations of the Snake-DSTM: write-buffer and undo log. Objects use verrsioned lock to enable ownership and validation. `Try Commit` procedure is used during the voting to make sure that all nodes are ready to commit.

4.3 Distributed Contention Management

Two transactions conflict if they concurrently access the same object, and one of them is a write transaction. Upon detecting a conflict, a *contention management policy* (CM) is used to resolve this situation (arbitrarily or priority-based) e.g., one of the transactions is stalled or aborted and retried. A wide range of transaction contention management policies has been studied for non-distributed STM [28,27]. We classify CMs into three categories: 1. *Incremental CM* (e.g., Karma, Eruption, Polka), where the CM builds up the priorities of the transactions during transaction execution; 2. *Progressive CM* (e.g., Kindergarten, Priority, Timestamp, Polite), which ensures a system-wide progress guarantee (i.e., at least one transaction will proceed to commit); and 3. *Non-Progressive*

CM (e.g., Backoff, Aggressive), which assumes that conflicting transactions will eventually complete, however, livelock situations can occur.

As mentioned earlier, in the control flow model, a distributed transaction T_x is executed over multiple nodes. Under Incremental CM, T_x can have different priorities at each node. This is because, a transaction builds its priority during its execution over multiple nodes. Under this behavior, a live-lock situation can occur. Consider transactions T_x and T_y with priorities P_x, P'_y and P'_x, and P_y at nodes N_1 and N_2, respectively. It is clear that, if $P'_x > P_y$ and $P'_y > P_x$, then both transactions will abort each other, and this will continue forever. The lack of a central store for transactional priorities causes this problem. However, having such a central store will significantly increase the communication overhead during transaction execution, causing a system bottleneck. Non-Progressive CM shows comparable performance for non distributed STM [3]. Nevertheless, our experiments show that it cannot be extended for D-STM due to the expensive cost of retries (see [26]).

4.4 Global Commitment Protocol

In the control flow model, a remote call on an object may trigger another remote call to a different object. The propagated access of objects forms a *call graph*, which is composed of nodes (i.e., sub-transactions) and undirected edges (i.e., calls). This graph is essential for making a commit decision. Each participating node may have a different decision (on which transaction to abort/commit) based on conflicts with other concurrent transactions. Thus, a voting protocol is required to collect votes from nodes, and the originating transaction can commit only if it receives an "yes" message from all nodes. By default, we implement the D2PC protocol [19], however, any other protocol may substitute it. We choose D2PC, as it yields the minimum possible time for collecting votes [19], which reduces the possibility of conflicts and results in the early release of acquired objects. Furthermore, it balances the overhead of collecting votes by having a variable coordinator for each vote.

5 Experimental Evaluation

Distributed Benchmarks. We developed a set of distributed benchmarks to evaluate Snake-DSTM against competing models including: i) classical RMI, which uses mutual exclusion locks and read/write locks with random timeout mechanism to handle deadlocks and livelocks; ii) distributed shared memory (DSM), which uses the Home directory protocol such as Jackal [33]; and iii) distributed dataflow STM implementation [25]. Our benchmark suite includes a distributed version of the vacation benchmark from the STAMP benchmark suite [37] (vacation) and two monetary applications (bank and loan).

Testbed. We conducted our experiments on a multiprocessor/multicomputer network comprising of 120 nodes, each of which is an Intel Xeon 1.9GHz processor, running Ubuntu Linux, and interconnected by a network with $1ms$ end-to-end delay. Each node invokes 50-200 sequential transactions. In a single

experiment, we thus executed 6-24 thousands transactions, and measured the throughput for each concurrency model, for each benchmark. Our experiments shows that Snake-DSTM write-buffer implementation outperforms undo-log implementations under all benchmarks. The reason for this is that undo-log pessimistic approach incur relatively larger number of retries, which in turn increases objects requests over the network. In this section we focus on Snake-DSTM write-buffer results against other concurrency models.

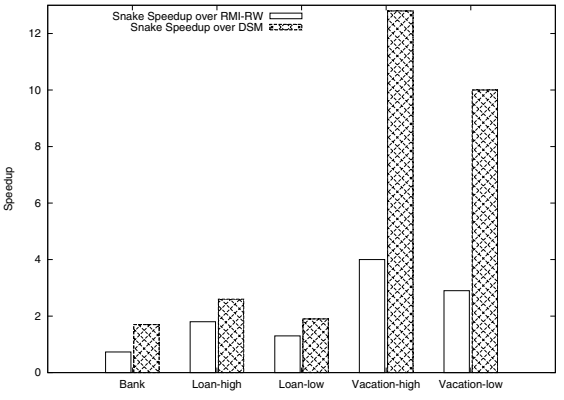


Fig. 5. Snake-DSTM speedup for a distributed benchmark suit over 120-node system

Evaluation. Figure 5[1] shows the relative throughput speedup achieved by Snake-DSTM over other concurrency models on the benchmarks. We observe that Snake-DSTM outperforms all other models under loan and vacation (the speedup ratio ranges between 1.3× and 12.8×). Under Bank benchmark only two nodes are involved into the transfer transaction, so Snake-DSTM overhead (voting, validation and versioning) outweighs the performance gain for this simple transaction, relative to RMI.

Using the Loan benchmark, transaction execution time was $200ms$ under ideal conditions. Six different objects were accessed per each transaction, issuing twenty remote calls. Figure 6(a) shows the scalability of Snake-DSTM under increasing number of nodes, and using 50% and 10% read-only transactions. Figure 6(b) shows the throughput under increasing number of participating objects in each transaction (transaction execution time under no contention is $350ms$ in this experiment). Greater the number of accessed objects, higher the algorithm overhead, and higher the number of remote calls per each transaction (e.g., a transaction of twelve objects issues 376 remote object calls during its execution).

From Figure 6(a), we observe that Snake-DSTM outperforms classical RMI using mutual exclusion locks (RMI-Locks), and also using read/write locks

[1] High and low indicate the benchmark contention which is controlled by either increasing write transactions or reducing the number of objects

(RMI-R/W), by 180% at high contention (10% reads), and by 150% at normal contention (50% reads). Though RMI with read/write locks shows better performance at a single point (6 nodes) due to the voting protocol overhead, yet, it suffers from performance degradation at increasing loads. It worth noting that the y-axis represents the nodal throughput, which means that in Figure 6(a) Snake-DSTM sustains almost the same nodal throughput with increasing the objects contention.

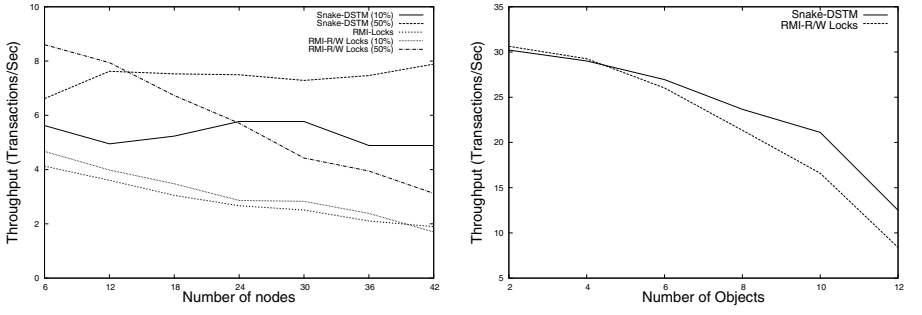


Fig. 6. Throughput of Loan benchmark: a) under increasing number of nodes, b) using pure read transactions over 12 nodes, and variable object count per transaction

Figure 6(b) uses the no-contention situation (100% reads) to compare the overhead of Snake-DSTM and RMI-R/W. At small number of shared objects per transaction, the TM overhead outweighs the provided concurrency, and both Snake-DSTM and RMI-R/W incur the same overhead. With increasing number of objects, Snake-DSTM outperforms RMI-R/W by 50%. Notice that the throughput degradation is not due to contention (100% reads), but it is because the transaction execution time is different (more objects at each data point), so the relevance of this figure is the relative implementation overhead of RMI and Snake-DSTM appraoches.

Figure 7(a) compares control-flow and dataflow D-STM implementations using the Bank Benchmark, where the end-to-end latency is changed, due to network conditions or object size. Figure 7(b) shows the effect of increasing the number of calls per a single remote object on Snake-DSTM throughput. This experiment illustrates the trade-off between employing locality of reference under dataflow model, and invoking remote calls at immobile objects using control-flow model. The best strategy is application based, which leaves a space for having both models in use.

End-to-end delay plays an important role in the design of distributed systems. We can decompose it into: network delay (propagation, processing, transmission, and queuing delay) and JVM delay (serialization, marshaling, and type checking). We define the object-to-parameter ratio (ρ) as the ratio of the end-to-end delay incurred in sending an object to the end-to-end delay incurred in sending the remote call parameters for this object. For example, ρ=2, when sending an object requires double the end-to-end delay of sending parameters of a remote call.

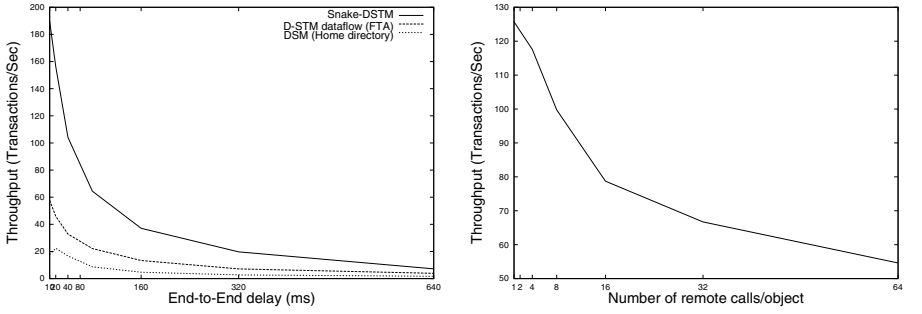


Fig. 7. Throughput of Bank benchmark: a) under dataflow and control-flow models using different end-to-end delay (ρ=1 and #calls/object=1), b) Snake-DSTM throughput under increasing number of calls per object

Figure 7(a) shows the effect of end-to-end delay on throughput when ρ=1 (i.e., sending the object is equivalent to sending the remote call parameters). Here, only one call is issued per any remote object within a transaction, which means that, for an application with ρ=4, the throughput of the dataflow flow model should be compared to the control-flow throughput multiplied by four. Similarly, for an application that invokes four calls per each object within a transaction, the equivalent control-flow throughput is divided by four.

Figure 7(b) demonstrates the effect of not employing locality of reference: in the control flow model, each remote call incurs a round-trip network delay. As shown in the figure, it reduces throughput by 25% for four to eight calls. This should be considered in environments with high link latency.

6 Conclusions

We presented Snake-DSTM, a high performance, scalable, distributed STM based on the control flow execution model. Our experiments show that Snake-DSTM outperforms other distributed concurrency control models, with acceptable number of messages and low network traffic. Control flow is beneficial under non-frequent object calls or when objects must be immobile due to object state dependencies, object sizes, or security restrictions. Our implementation shows that Snake-DSTM provides comparable performance to classical distributed concurrency control models, and exports a simpler programming interface, while avoiding dataraces, deadlocks, and livelocks.

The HyFlow project provides a testbed for the TM research community to design, implement, and evaluate algorithms for D-STM. HyFlow is publicly available at *hyflow.org*.

Acknowledgements. This work is supported in part by US National Science Foundation under Grant 0915895, and NSWC under Grant N00178-09-D-3017-0011.

References

1. Partitioned Global Address Space, PGAS (2003)
2. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared memory computing on networks of workstations. IEEE Computer (29) (1996)
3. Ansari, M., Kotselidis, C., Luján, M., Kirkham, C., Watson, I.: Investigating contention management for complex transactional memory benchmarks. In: MULTIPROG (2009)
4. Arnold, K., Scheifler, R., Waldo, J., O'Sullivan, B., Wollrath, A.: Jini Specification. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
5. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP 2008, NY, USA, pp. 247–258 (2008)
6. Cachopo, J.A., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63, 172–185 (2006)
7. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: Dependable distributed software transactional memory. In: PRDC 2009 (2009)
8. 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)
9. Guerraoui, R., Kapalka, M.: The semantics of progress in lock-based transactional memory. SIGPLAN Not. 44, 404–415 (2009)
10. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory, pp. 253–262. ACM, NY (2006)
11. Herlihy, M., Moss, J.E.B., Eliot, J., Moss, B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA 1993, pp. 289–300 (1993)
12. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 324–338. Springer, Heidelberg (2005)
13. Johnson, T.: Characterizing the performance of algorithms for lock-free objects. IEEE Transactions on Computers 44(10), 1194–1207 (1995)
14. 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 (2008)
15. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with Java STM. In: Third Workshop on Programmability Issues for Multi-Core Computers, MULTIPROG (2010)
16. Liskov, B., Day, M., Herlihy, M., Johnson, P., Leavens, G.: Argus reference manual. Technical report, Cambridge University, Cambridge, MA, USA (1987)
17. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: HPCA 2006 (2006)
18. Philippsen, M., Zenger, M.: Java Party transparent remote objects in Java. concurrency practice and experience (1997)
19. Raz, Y.: The Dynamic Two Phase Commitment (D2PC) Protocol. In: ICDT 1995, London, UK, pp. 162–176 (1995)
20. Reed, D.P.: Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA (1978)
21. Reynolds, F.: An architectural overview of alpha: A real-time distributed kernel. In: Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, Berkeley, CA, USA, pp. 127–146 (1992)

22. Romano, P., Rodrigues, L., Carvalho, N., Cachopo, J.: Cloud-TM: harnessing the cloud with distributed transactional memories. SIGOPS Oper. Syst. Rev. 44, 1–6 (2010)
23. Saad, M.M., Ravindran, B.: Hyflow: A high performance distributed software transactional memory framework. In: HPDC 2011 (2011)
24. Saad, M.M., Ravindran, B.: Supporting STM in Distributed Systems: Mechanisms and a Java Framework. In: TRANSACT 2011, San Jose, California, USA (2011)
25. Saad, M.M., Ravindran, B.: Transactional Forwarding Algorithm: Technical Report. Technical report, ECE Dept., Virginia Tech (2011)
26. Saad, M.M., Ravindran, B.: RMI-DSTM: Control Flow Distributed Software Transactional Memory: Technical Report. Technical report, ECE Dept., Virginia Tech (2011)
27. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC 2005, New York, NY, USA, pp. 240–248 (2005)
28. Scherer III, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: PODC 2004, NL, Canada (2004)
29. Shavit, N., Touitou, D.: Software transactional memory. In: PODC 1995, New York, USA, pp. 204–213 (1995)
30. Surdeanu, M., Moldovan, D.: Design and performance analysis of a distributed java virtual machine. IEEE Trans. Parallel Distrib. Syst. 13, 611–627 (2002)
31. Tatsubori, M., Sasaki, T., Chiba, S., Itano, K.: A bytecode translator for distributed execution of legacy Java software. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 236–255. Springer, Heidelberg (2001)
32. Tilevich, E., Smaragdakis, Y.: J-Orchestra: Automatic Java application partitioning. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 178–204. Springer, Heidelberg (2002)
33. Veldema, R., Bhoedjang, R.A.F., Bal, H.E.: Distributed shared memory management for java. In: ASCII 2000, p. 256 (2000)
34. Yu, W., Cox, A.: Java/DSM: A platform for heterogeneous computing. Concurrency: Practice and Experience 9(11), 1213–1224 (1997)
35. Zhang, B., Ravindran, B.: Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In: OPODIS 2009, Berlin, Heidelberg, pp. 48–53 (2009)
36. Zhu, W., Wang, C.-L., Lau, F.: Jessica2: a distributed java virtual machine with transparent thread migration support. In: CC 2002 (2002)
37. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008 (2008)

POLISH: Proactive Co-operative LInk Self-Healing for Wireless Sensor Networks

Tatsuro Iida, Atsuko Miyaji, and Kazumasa Omote

Japan Advanced Institute of Science and Technology (JAIST)
{s0910001,miyaji,omote}@jaist.ac.jp

Abstract. In this paper we propose the first proactive co-operative link self-healing (POLISH) scheme, in which the secure link compromised in WSNs automatically self-heals with time, without the help of a server. Our scheme updates a secure link using the random data transmitted from the neighboring sensor nodes, based on the idea of the POSH scheme. It is necessary to newly take the security of a link between sensors into consideration in our scheme since such security is not considered in the POSH scheme. We conduct analytical evaluation and a simulation experiment for our scheme, and the results indicate that our scheme is very effective in self-healing.

1 Introduction

Wireless Sensor Networks (WSNs) consist of small, battery-operated, limited memory and limited computational power sensor nodes. Hence, most of existing pairwise key establishment schemes in WSNs are not based on public key cryptography. One of the most popular schemes, referred to as *RKP* (Random Key Pre-distribution) in this paper, was first proposed by Eschenauer and Gligor [4] and has been applied to many schemes. These basic probabilistic schemes are pairwise key pre-distribution schemes based on symmetric key cryptography. However, the security of the whole network in such schemes degrades with time when there is an attacker. An attacker who corrupts several sensors can obtain a set of the pairwise symmetric keys. If the attacker is continuously corrupting sensors, they will eventually learn all the pairwise symmetric keys, and all newly deployed sensors will establish links that will immediately be compromised. This is a non-desirable property.

The WSNs are usually deployed to operate for a long period of time. Availability is very important to long-term use of WSNs under the presence of an attacker. Actually, we can find several schemes [2,5,6,10], which maintain availability of the secure link. Link composed of a pairwise symmetric key in WSNs. These schemes are called *resilient multiphase WSNs*, in which a link self-heals against node-capture attacks by redeploying a sensor node when the battery of a sensor is depleted. However, as far as we know, any efficient scheme which maintains availability of the secure link between sensor nodes requires the help of a server. It is thus desirable that the link *self-heal* against node-capture attacks to

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 253–267, 2011.

maintain availability without the help of a server. Self-healing of a secure link is the property that the compromised link recovers in a WSN.

In this paper, we propose the first proactive co-operative link self-healing (POLISH) scheme, in which the secure link compromised in a WSN automatically self-heals with time, without the help of a server. Our scheme updates a secure link using the random data transmitted from the neighboring sensor nodes, based on the idea of the POSH scheme. The POSH scheme self-heals the secret key for encrypting the sensed data on a sensor node for the purpose of data survival. In our scheme, a link self-heals in two steps: first, two neighboring sensors are self-healed, and then the link between these sensors is self-healed. It is necessary to newly take the security of a link between sensors into consideration in our scheme since such security is not considered in the POSH scheme. Furthermore, our scheme has an advantage that the probability of establishing a secure link is 100%. In addition, we conduct analytical evaluation and a simulation experiment for our scheme, and the results indicate that the proposed scheme is very effective in self-healing. Our scheme is both effective and efficient, as supported by analytical and simulation results.

The rest of this paper is organized as follows. In the next section we present related work on pairwise key distribution schemes with self-healing property for WSNs. Some preliminaries are provided in Section 3, and we review the POSH scheme in Section 4. We explain our scheme in detail in Section 5, analyze its security and efficiency in Section 6, and compare the POLISH scheme with the previous RoK scheme in Section 7. We finally conclude this paper in Section 8.

2 Related Work

One of the most popular schemes, referred to as *RKP* in this paper, was proposed by Eschenauer and Gligor [4], which has been applied to many schemes. These probabilistic pairwise key pre-distribution schemes are efficient because they are not based on public key cryptography. However, these schemes do not have self-healing feature of a link, and thus the ratio of the compromised links reaches 100% as time passes against node-capture attacks.

Castelluccia and Spognardi [2] have proposed the RKP scheme with self-healing property, named *RoK* scheme, for *multiphase WSNs*, in which a link self-heals against node-capture attacks by redeploying a sensor node (with server's help) when the battery of a sensor is depleted. The RoK scheme improves the security of the RKP scheme by limiting the lifetime of the keys, and by refreshing keys. Some recent schemes improve the resiliency of the RoK scheme. Yilmaz et al. [10] proposed a more resilient scheme than the RoK scheme to speed up the self-healing process. Kalkan et al. [6] proposed a zone-based RKP (Zo-RoK) scheme which combines the best parts of Du et al.'s scheme [3] and the RoK, and improves the resiliency of the RoK. Furthermore, Ito et al. [5] proposed a strongly-resilient polynomial-based random key pre-distribution scheme for multiphase WSNs (RPoK): a private sub-key is not directly stored in each sensor node by applying the polynomial-based scheme to the RoK scheme.

There is another drawback in the RKP scheme; the probability of establishing a secure link is not 100%. We recall the basic pairwise key pre-distribution scheme, *polynomial-based key pre-distribution scheme* [1] which was proposed prior to the RKP scheme and which maintained the probability of establishing a secure link at 100%. To pre-distribute pairwise keys in the polynomial-based key pre-distribution scheme, a setup server randomly generates t-degree $f(x, y)$ over a finite field $\mathbb{F}_q$, where it has the symmetrical property of $f(x, y) = f(y, x)$.

As for self-healing of the secret key for the purpose of data survival, the POSH scheme [8] and the DISH scheme [7] have been proposed by Pietro et al. and Ma et al., respectively. These schemes use key evolution and sensor cooperation to self-heal the secret key which encrypts the sensed data on a sensor node, for the purpose of data survival. These schemes involve each sensor sharing an initial key with the sink (base station). At any time, sensors are either occupied (red), sick (yellow) or healthy (green). The self-healing of a sensor means that a sick sensor becomes healthy. The POSH and the DISH schemes update a secret key using the random data transmitted from other sensor nodes. That is, if at least one of the sensor nodes which send random data is not corrupted, the compromised secret key is updated and then is self-healed.

3 Preliminaries

3.1 Notation

n : Total number of sensors (i.e., Size of network)
s_i : Sensor i
ID_i : Index of s_i
m : Number of links with neighboring sensors
r : Round index (i.e., fixed-length time slot)
$K_{i,j}^r$: Pairwise symmetric key (secure link) between s_i and s_j at round r
S_i^r : Seed of s_i at round r
$c_{i\ell}^r$: ℓ-th *contribution* received by s_i at round r
G^r : Set of green sensors at round r
Y^r : Set of yellow sensors at round r
R^r : Set of red sensors ($= k$) at round r
GL^r : Set of green links at round r
RL^r : Set of red links at round r
q : Large prime number
H : Secure hash function $H : \{0,1\}^* \rightarrow \{0,1\}^q$
$f(x, y)$: Bivariate t-degree polynomial at over a finite field $\mathbb{F}_q$

3.2 Requirements

The following requirements need to be considered when designing a link self-healing scheme in WSNs.

Highly-Secure Connectivity. After deployment, two sensors share a pairwise symmetric key to establish a secure link. This probability is called secure connectivity. Highly-secure connectivity is required in a pairwise symmetric key scheme in WSNs.

High Resiliency. Sensor nodes may be deployed in public or hostile locations in many applications. The resiliency (self-healing) means that the ratio of compromised links is suppressed low even if the adversary regularly/continuously corrupts sensor nodes of the network. This feature is achieved by security properties, *forward and backward secrecy*[1]. Resiliency is estimated by the ratio of links that are not compromised by the capture of nodes.

Restricted Resources. It is required that the WSNs consist of small, battery-operated devices with limited memory and limited computational power.

3.3 System and Network Assumptions

Time is divided into equal and fixed rounds. Round synchronization can be implemented. The network is connected at all times. Any two sensors can communicate either directly or indirectly, via other sensors. Each sensor can perform cryptographic hashing and polynomial execution and has a unique ID. Also, each sensor has a Pseudo-Random Number Generator (PRNG) initialized with a unique secret seed. A sensor re-initializes secret seed values in any round.

3.4 Adversarial Model

We refer to the adversary as ADV from here on. ADV's main goal is to learn as many sensor secrets (keys or other key material) as possible, and hence ADV is only interested in learning the secrets of sensors when it compromises/captures. ADV knows the entire topology of the WSNs. Such adversary model is usually employed in the previous schemes [4,2,10,6,5]. ADV can create a table of sensor secrets and share it. This might be later used to decrypt encrypted communication. Furthermore, ADV does not stay at one local place for stealthy operation and then does not interfere with sensor's behavior, i.e., it does not delete, delay or introduce messages. ADV also eavesdrops on communication from the sensor through wireless transmission. Note that ADV leaves no trace behind (e.g., he does not establish an sniffing tool somewhere in WSNs).

Time is divided into equal and fixed rounds. At the end of each round, ADV randomly picks a subset of k sensors to be compromised in the following round (ADV preferentially aims at a green sensor). At the start of each round, the ADV releases the subset from the previous round and compromises the new subset. ADV is unable to monitor and record all communication at the same time as described in [7,8].

[1] These security properties are defined in [8]. Forward secrecy means that ADV cannot learn any keys used to decrypt and/or authenticate before compromise, and backward secrecy means that ADV cannot learn any keys used to decrypt and/or authenticate after compromise.

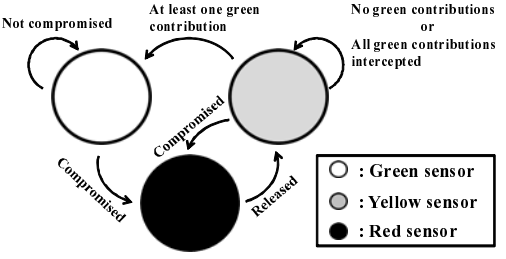


Fig. 1. Sensor state transition diagram [8]

4 The POSH Scheme

4.1 Overview

Pietro et al. have proposed a proactive co-operative self-healing (POSH) scheme for data survival on a sensor. A sensor s_i encrypts its sensed data in round r by using secret key K_i^r. A sensor whose current key is known to ADV can regain security and compute a new key unknown to ADV, if it obtains at least one "infusion" of secure randomness from a peer sensor whose randomness is not currently compromised. Each sensor shares a secret key K_i^1 with the sink in the first round 1. ADV breaks into $k = |R^r|$ sensors and reads all keys. At any time, we identify three sets of sensors:

- *Red sensors* (R^r) are currently occupied by ADV in round r.
- *Yellow sensors* (Y^r) are those that have been compromised in some round $r' < r$, and their current keys are known to ADV in round r.
- *Green sensors* (G^r) are those that have either never been compromised, or which have regained their security by round r.

The main point of the POSH scheme is for sensors, at each round, to provide each other with contributions derived from their PRNG-s. Each sensor, having received some such contributions, uses them together with its prior keys to compute a key for the next round. Specifically, each sensor produces a certain number of contributions and recipient IDs using its PRNG, and sends each value to them. In more detail, to update its key at the end of round r, s_i computes:

$$K_i^{r+1} = H(K_i^r || c_{i_1}^r || ... || c_{i_\sigma}^r), \ r \geq 1, \tag{1}$$

where σ is the number of received contributions, and $c_{i_\ell}^r$ is the ℓ-th contribution received during current round. This key evolution holds both forward and backward security. Note that all contributions generated by red and yellow sensors are known to ADV. On the other hand, contributions by green sensors are unknown to ADV. Thus, if a yellow sensor receives a single contribution from a green sensor, ADV cannot learn the former's next key. Note that a green sensor cannot become a yellow sensor directly. The state transition diagram of a single sensor is shown in Fig. 1.

4.2 Boundary of Security Evaluation in POSH

To evaluate the healing rate of secret key for data encryption, the POSH scheme analyzes the number of green sensors in any round. The secret key K_i^r is used as a secure link between a sensor s_i and the sink at round r since the sink knows all the secret keys of sensors. However, we cannot directly achieve the secure link between sensors by the POSH scheme, since the security of a link between sensors is not considered in the POSH scheme.

5 The Proposed Scheme

In this section we propose the POLISH (Proactive co-Operative LInk Self-Healing) scheme. The primary aim of our scheme is to decrease the compromised ratio of links against node-capture attacks without help of a server, that is, links compromised in WSNs automatically self-heal with time. Our scheme updates a link using the random data transmitted from the neighboring sensors, based on the idea of the POSH scheme. Although our protocol is very simple like POSH, more importantly, our security evaluation is not achieved easily, i.e., it is necessary to newly take the security of a link between sensors into consideration in our scheme since such security is not considered in the POSH scheme.

A link self-heals in two steps: first two neighboring sensors are self-healed, and then the link between these sensors is self-healed. A major difference between POSH and POLISH is the security analysis of a link. While the POSH scheme in a sense treats the secure link between a sensor and a powerful sink, the POLISH scheme treats the secure link between sensors. In addition, our scheme uses a bivariate t-degree polynomial, and thus an attacker has to capture $(t+1)$ polynomial shares during a limited period of time (i.e., at round 1) in order to corrupt a link.

ADV breaks into $k = |R^r|$ sensors to read the pairwise symmetric keys and secret seeds of PRNG in R^r, and to monitor all the communication of R^r. At any time, we identify three sets of sensors (refer to Section 4) and two sets of links, as follows:

- *Red links* (RL^r) are those that have been compromised in some round $r' < r$ and the pairwise symmetric key of the link is known to ADV in round r.
- *Green links* (GL^r) are those that have either never been compromised or regained their security in round r.

Note that, in our scheme, a red sensor s_i at round r means that ADV knows a seed S_i^r. If s_i becomes red in round r' and is self-healed at the end of round $r > r'$, then ADV can compute the contributions of s_i from round r' to r.

5.1 The Protocol

Setup. To predistribute pairwise keys, the setup server randomly generates a bivariate t-degree polynomial $f(x, y)$ over a finite field $\mathbb{F}_q$, such that it has the

property of $f(x, y) = f(y, x)$. For each sensor s_i, the setup server computes a polynomial share of $f(x, y)$, that is, $f(x, ID_i)$. Each sensor can use a secure hash function, a polynomial and a PRNG with a unique secret seed. Note that the secure degree t of polynomial is dependent on the number of adversary at each round. For instance, if we set $t \geq 10$ as the secure degree of polynomial when we assume $k = 10$, then ADV cannot recover $f(x, y)$.

Key Establishment. For any two sensors s_i and s_j, the sensor s_i can compute the key $f(ID_j, ID_i)$ by evaluating $f(x, ID_i)$, and the sensor s_j can compute the same key $f(ID_i, ID_j) = f(ID_j, ID_i)$ by evaluating $f(x, ID_j)$. As a result, sensors s_i and s_j can establish a pairwise symmetric key $K^1_{i,j} = f(ID_i, ID_j)$ in the first round (round 1). After key establishment, s_i deletes all the coefficients of a polynomial.

Key and Seed Update. The neighboring sensors s_i and s_j have a pairwise symmetric key $K^1_{i,j}$ (secure link) when they are deployed at the beginning of the first round (round 1). At the beginning of round r, s_i produces m pseudo-random values (contributions) using its PRNG for m neighboring sensors, and sends them to the neighboring sensors using a secure link. Note that all the contributions that s_i sends are different. Then, each sensor receives contributions from the neighboring sensors during round r. The recipient uses two contributions as inputs to the secure hash function used for key update. To update the secure link at the end of round r, s_i computes:

$$K^{r+1}_{i,j} = H(K^r_{i,j} || c^r_{i_\eta} || c^r_{j_\lambda}), \tag{2}$$

where $c^r_{i_\eta}$ is the η-th contribution that s_i received at round r and $c^r_{j_\lambda}$ is the λ-th contribution that s_j received at round r. Both s_i and s_j delete $K^r_{i,j}$ after key updating.

Furthermore, each sensor updates a seed of PRNG using m contributions, which are all contributions received by the neighboring sensors. To update the seed S^r_i at the end of round r, s_i computes[2]:

$$S^{r+1}_i = H(S^r_i || c^r_{i_1} || \cdots || c^r_{i_m}) \tag{3}$$

After seed updating, s_i deletes S^r_i. A seed is updated in every round, and then m contributions are generated by PRNG with such new seed.

Remark. In the POSH scheme, each sensor receives contributions from sensors which are randomly chosen in WSNs. On the other hand, in our scheme, each sensor receives contributions from neighboring sensors. The probability that a contribution will be intercepted on the way by ADV may become high in the POSH scheme, since a contribution can be sent from a sensor which is far from the recipient.

5.2 The Link State

A link self-heals in two steps: first two neighboring sensors are self-healed, and then the link between them is self-healed. A sensor state follows the transition

[2] The update of a PRNG seed is similar to [9].

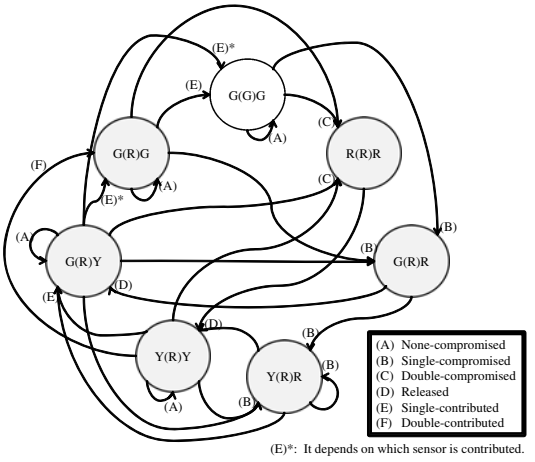


Fig. 2. Link state transition diagram

diagram in Fig.1. According to the state of a sensor we can generate the seven kinds of link states as described in Fig.2 (i.e., $GL^r = \{\text{G(G)G}\}$ and $RL^r = \{\text{G(R)G, G(R)Y, Y(R)Y, Y(R)R, G(R)R, R(R)R}\}$). A link state is constituted by a pair of sensors and their common link. For example, G(R)Y means that two neighboring sensors of green and yellow are connected by the red link. The conditions of transition are as follows:

1. *Double-compromised* condition means that both of two neighboring sensors are compromised.
2. *Single-compromised* condition means that either of two neighboring sensors is compromised.
3. *None-compromised* condition means that neither of two neighboring sensors is compromised.
4. *Single-contributed* condition means that either of two neighboring sensors receives at least one "secure contribution".
5. *Double-contributed* condition means that both of two neighboring sensors receive at least one secure contribution.

Note that the secure contribution is a green contribution which is not intercepted by ADV.

A red link remains red if a red sensor is within the wireless communication range of both of two sensors which constitute the red link. On the other hand, a green link remains green as long as both of two sensors which constitute the green link are green. We notice that even if two sensors are green, the link between them can be also red (i.e., G(R)G). A green link (G(G)G) can be changed from two states G(R)G and G(R)Y when single-contributed. G(R)G becomes G(G)G when one of two neighboring sensors receives a secure contribution from the other. G(R)Y becomes G(G)G when the yellow sensor Y receives a secure contribution from this green sensor G. G(R)Y becomes G(G)G when Y receives at least one secure contribution from other green sensors except this G.

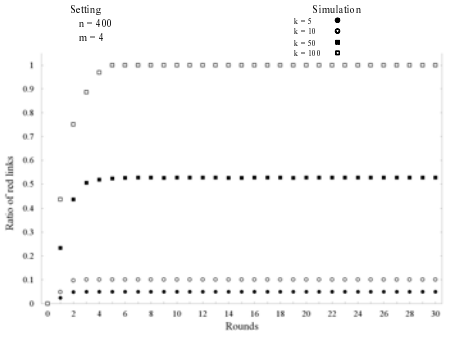


Fig. 3. Simulation results against continuous attackers

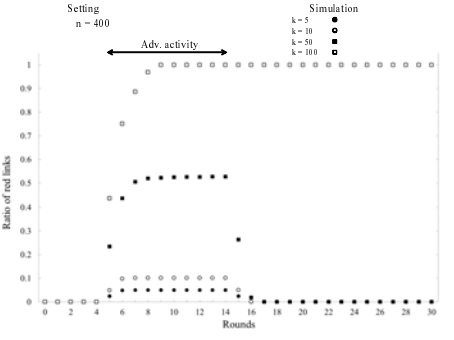


Fig. 4. Simulation results against temporary attackers

6 Analysis

6.1 Evaluation by Simulation

We evaluate the ratio of red links against *continuous* attackers to show the resiliency of our scheme, and we also evaluate the ratio of red links against *temporary* attackers to show the self-healing capabilities of our scheme. For ease of exposition and without loss of generality, we assume that the time slots (rounds) when sensors are compromised have the same duration and are synchronized.

Simulation Setup. The simulations are implemented in C on Windows XP SP3. All the simulations are repeated 1,000 times, and the results show the average values. To simplify the security analysis, we modeled the network as a grid of sensors of size $n = 400$ (20×20). We assume that the number of neighbors of each sensor is constant and equal to four (m=4). We can imagine a torus structure. Thus, the number of all links in WSN becomes 800. We also assume that the network topology does not change over time. The number k of ADV is 5, 10, 50 and 100 in every round.

Simulation Details. We evaluate the security of our scheme by the number of red links when ADV can compromise k sensors from the set G^r in any round. At the first round (round 1), n green sensors are deployed. We consider two different types of attackers: continuous attackers and temporary attackers. A continuous attacker keeps compromising sensors at constant rate from the deployment of the first round of sensors to the end of the network. In contrast, temporary attacker compromises sensors during a limited period of time, from round 5 to round 14 in our simulations. We then counted, in each round, the number of red links and computed the ratio. With the continuous attacker, we ran the simulation until: (1) the WSN has no more green sensors or (2) $|R^r|$ reaches a steady state.

Simulation Results. This section presents the results of our simulations for the different types of attackers. Fig.3 displays the ratio of red links against continuous attackers. The ratio of red links reaches 100% when $k \geq 62$. For example, the ratio of red links is suppressed to 5.1% with $k = 5$, 10% with $k = 10$, 52% with $k = 50$ and 100% with $k = 100$, depicted in Fig.3. The results for the temporary attacker are collected in Fig.4. The action interval of the attacker (from generation 5 to generation 14) is denoted with the label "Adv. activity". We simulate a network with the same settings as the network used for the continuous attacker. Fig.4 illustrates the self-healing property of our scheme as soon as ADV stops its activity, and the ratio of the red links starts decreasing at once. A link self-heals in only about three rounds. Note that once the ratio of red links becomes 1, the ratio remains 1 even when ADV stops its activity.

6.2 Analytical Model

Unlike the POSH scheme, a sensor in our scheme receives contributions from neighboring sensors, that is, a sensor receives m contributions. Note that the state transition of a sensor is the same as in the POSH scheme. In our scheme, it is necessary to consider the contributions from two-hop neighboring sensors. The contributions from neighboring sensors may be eavesdropped on by two-hop neighboring sensors. In this case, a green sensor is not self-healed even if it gets a contribution from a green sensor (refer to an example in Appendix **??**). Let $(1-(1-p_{R^r})^{m-1})$ be the probability that at least one sensor of two-hop neighboring sensors is red, that is, the probability that a green sensor's contribution is eavesdropped on by ADV (i.e., red sensor) which is within the wireless communication range of the green sensor. To become a green sensor (from yellow), the yellow sensor needs to be linked with at least one green sensor among neighboring sensors, and also a red sensor must not be within the wireless communication range of that green sensor. Thus, the probability of a yellow sensor not becoming green can be expressed as follows:

$$Pr' = \sum_{i=0}^{m} \binom{m}{i} p_{G^r}^{i} (1-p_{G^r})^{m-i} (1-(1-p_{R^r})^{m-1})^{i}, \tag{4}$$

where $p_{G^r} = \frac{|G^r|}{n-1}$, $p_{Y^r} = \frac{|Y^r|}{n-1}$ and $p_{R^r} = \frac{|R^r|}{n-1}$. The expected number of green sensors at round r is the same as in the POSH scheme, as follows[3]:

$$E[|G^{r+1}|] = |G^r| + (1-Pr')|Y^r| - |R^r| \tag{5}$$

To evaluate the link-healing rate of our scheme, we analyze the number of green links by evaluating the state of sensors in any round, i.e., the number of G(G)G in Fig.5. The partial state transition diagram of a link is shown in Fig.5, in which only the transition required to analyze the number of green links is depicted.

[3] Since we assume that ADV corrupts only the green sensor (i.e., INF-ADV in [8]), we can use not inequality but an equation.

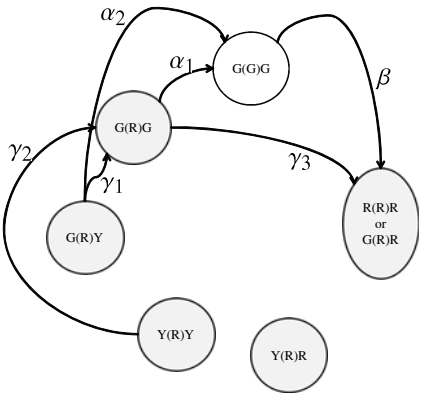


Fig. 5. Partial link state transition diagram

That is, we consider only the input and the output of G(G)G and G(R)G. Let α_1, α_2, β, γ_1, γ_2 and γ_3 be the number of link state transition (use not probability but a number.) and let $RL^r_{G(R)G} \subset RL^r$ be a set of the link state G(R)G. This figure shows that the expected number of green links in round r is:

$$E[|GL^{r+1}|] = |GL^r| + \alpha_1 + \alpha_2 - \beta, \tag{6}$$

where $\alpha_1 = (1 - (1 - (1 - p_{R^r})^{m-1})^2)|RL^r_{G(R)G}|$, $\alpha_2 = (1 - Pr')|Y^r|p_{\alpha_2}$ and $\beta = |R^r|p_\beta$. α_1 is the number of green links between two green sensors changed from $RL^r_{G(R)G}$. This transition occurs if neither of the green sensors is linked with a red sensor. Let p_{α_2} be the probability that a sensor needs to be linked with at least one green sensor of the neighboring sensors, and also that a red sensor must not be within the wireless communication range of that green sensor. α_2 is the number of green links between two green sensors, changed from red links between a green sensor and a yellow sensor. Let p_β be the probability that at least one green sensor in GL^r is corrupted. Hence, β is the number of red links between two red sensors, or between a yellow sensor and a red sensor changed from GL^r, since ADV corrupts only the green sensors and the number of ADV is $|R^r|$ in any round.

The number of red links between two green sensors is estimated in Fig. 5 as follows:

$$E[|RL^{r+1}_{G(R)G}|] = |RL^r_{G(R)G}| - \alpha_1 + \gamma_1 + \gamma_2 - \gamma_3, \tag{7}$$

where $\gamma_1 = (1 - Pr')|Y^r|p_{\gamma_1}$, $\gamma_2 = (1 - Pr')|Y^r|p_{\gamma_2}$ and $\gamma_3 = |R^r|p_{\gamma_3}$. Let p_{γ_1} be the probability that a sensor is linked with a green sensor, and also that a red sensor must not be within the wireless communication range of that green sensor. Let p_{γ_2} be the probability that a sensor is linked with a yellow sensor which becomes green. Moreover, let p_{γ_3} be the probability that at least one green sensor in $RL^r_{G(R)G}$ is corrupted.

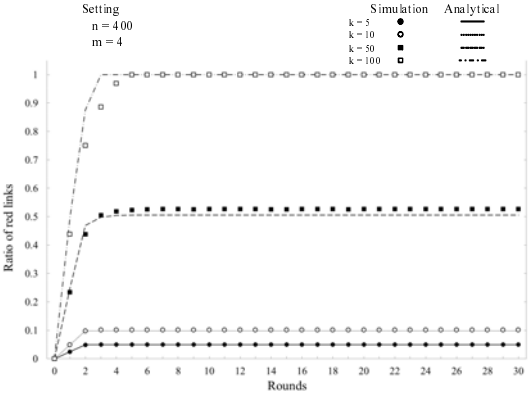


Fig. 6. Comparison of the analytical results and simulation results against continuous attackers

Let μ be the ratio of $|GL^r|$ in a set of two neighboring green sensors which are linked each other, i.e., $\mu = \frac{|GL^r|}{|GL^r|+|RL^r_{G(R)G}|}$. We show the probability of p_{α_2}, p_β, p_{γ_1}, p_{γ_2} and p_{γ_3} as follows:

$$p_{\alpha_2} = \sum_{i=0}^{m} \binom{m}{i} (p_{G^r}(1-p_{R^r})^{m-1})^i (1-p_{G^r}(1-p_{R^r})^{m-1})^{m-i} i$$

$$p_\beta = \sum_{i=0}^{m} \binom{m}{i} (p_{G^r}\mu)^i (1-p_{G^r}\mu)^{m-i} i$$

$$p_{\gamma_1} = \sum_{i=0}^{m} \binom{m}{i} (p_{G^r}(1-(1-p_{R^r})^{m-1}))^i (1-p_{G^r}(1-(1-p_{R^r})^{m-1}))^{m-i} i$$

$$p_{\gamma_2} = \sum_{i=0}^{m} \binom{m}{i} ((1-Pr')p_{Y^r})^i (1-(1-Pr')p_{Y^r})^{m-i} i$$

$$p_{\gamma_3} = \sum_{i=0}^{m} \binom{m}{i} (p_{G^r}(1-\mu))^i (1-p_{G^r}(1-\mu))^{m-i} i$$

Fig. 6 shows a comparison of the analytical results and the simulation results. Note that the simulation results are the same as in Fig. 3. We found that our analytical results well matched the simulation results of our scheme.

6.3 Secure Connectivity

Our scheme has an advantage that the probability of establishing a secure link is 100%, since a sensor s_i has a polynomial $f(x, ID_i)$ and also shares the pairwise symmetric key $K^r_{i,j} = f(ID_j, ID_i)$ with s_j in the first round (round 1). After that the pairwise symmetric key of each link is updated, and hence the secure connectivity is 100% at every round.

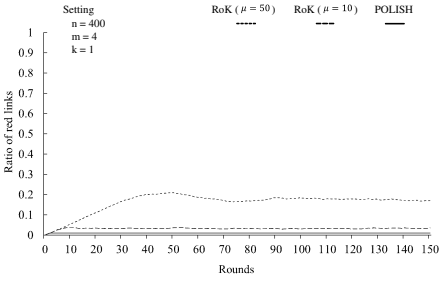


Fig. 7. Comparison of the simulation results against continuous attackers ($k = 1$)

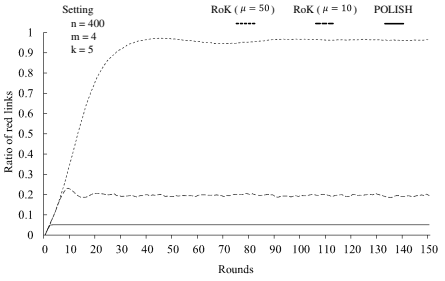


Fig. 8. Comparison of the simulation results against continuous attackers ($k = 5$)

6.4 Efficiency of POLISH

We discuss computational cost, communication cost and the size of memory in the POLISH scheme. Let R and H be the PRNG and hashing operations, respectively, and also let $|q|$ be the size of the contribution, ID, the output of hashing and the coefficient of a polynomial. The computational cost of each sensor in a round is $mR + (m + 1)H$. Note that it is required for s_i to assign the value of ID_j to a polynomial $f(x, ID_i)$ to share the pairwise symmetric key only in the first round (round 1). The communication cost of each sensor in a round is $2m|q|$ which includes the contributions of transmission and reception. Note that s_i needs to obtain IDs from m neighboring sensors in round 1. In order to setup data of a sensor at the first round, s_i requires a seed, the ID and the coefficient of a polynomial, i.e., the size of memory on a sensor requires $(t+3)|q|$ in total. After key establishment, s_i deletes all the coefficients of a polynomial, but m pairwise symmetric keys whose sizes are $m|q|$ are generated. Thus, the amount of memory on a sensor can save $(t + 1 - m)|q|$. This means that s_i can keep the contributions of transmission and reception if $(t + 1) \geq 3m$. Therefore, our scheme is efficient and is suitable for WSNs which constitute sensors with both limited memory and limited computational power.

7 Comparison

The POLISH scheme is the first proactive co-operative link self-healing scheme, without the help of a server. On the other hand, the previous link self-healing schemes need server's help, called resilient multi-phase WSNs. A multiphase WSN is a network where a sensor is redeployed with server's help when its battery is depleted. Although multiphase WSNs and our scheme are quite different in that the help of a server is necessary, we dare to compare our scheme with the previous scheme. We especially compare the RoK scheme with our scheme regarding security (the ratio of sick links) and efficiency since the RoK scheme is efficient and representative in multiphase WSNs.

7.1 Security

We compare the resiliency of our scheme with that of the RoK scheme by simulation experiments. For a fair comparison, the same size of memory is assumed between both schemes. When the length of key ring is 250 (i.e., $\omega = 250$), the total number of keys in the RoK scheme is just 500 which is the same parameters as [2]. When each size of key is 160 bits ($|q| = 160$ bits), at least 10kB memory is required in the RoK scheme. In our scheme, $(t + 1 - m)|q|$ bits memory is required as described in Section 6.4, more concretely, $20(t - 3)$ bytes memory is necessary when we follow the same simulation parameters as Section 6.1. We set $t = 497$ as the degree of polynomial from the standpoint of fairness. Actually, we can set $t = 5$ as the secure degree of polynomial when $k = 5$. In this case ADV cannot recover the polynomial.

We evaluate the ratio of red links against continuous attackers. A red link implies the compromised link in the RoK scheme. The network topology of both simulation is the same, in which a grid of sensors of size $n = 400$ and a sensor does not move over time. Fig. 7 and Fig. 8 are the comparisons of the simulation results between the RoK scheme and our scheme when the number of ADV is 1 and 5, respectively. In the RoK scheme, we simulate sensors expiration by assigning to each sensor a random expiration date, chosen according to a Gaussian distribution with mean $\mu = 50.0$ rounds and with standard deviation $\sigma = 16.7$ whose parameters are the same as [2], and also with $\mu = 10.0$ and with $\sigma = 3.33$. When the sensor expiration becomes short in the RoK scheme, the ratio of red links also decreases as described in Fig. 7 and Fig. 8. However, it is difficult to lower the ratio of red links substantially since this is contrary to the battery extension of sensor life which the WSNs aim at.

Remark. In [2] the compromised ratio (i.e., the ratio of red links) is evaluated as a probability that a link is "indirectly" compromised by ADV. On the other hand, we evaluate all the number of red links in both schemes in this simulation. Since we re-evaluate the RoK scheme by the total red links, the ratio of red links of the RoK scheme in Fig. 7 and Fig. 8 becomes a little higher than the original results.

7.2 Efficiency of Key Update

Both the RoK scheme and our scheme update keys and key materials at every round. We think that it is important to especially reduce the size of memory since the amount of computations and communications is not so frequent (i.e., they are executed at the beginning of each round). While the key materials are updated only by each sensor in the RoK scheme, keys and key materials are updated by cooperation of the neighboring sensors in our scheme. Hence more communications are required in our scheme but more memory is required in the RoK scheme.

Since most links are compromised in the RoK scheme when $k = 5$ as described in Fig. 8, we consider $k \leq 5$ as the number of ADVs. We thus set $t = 5$ in order that at most five ADVs cannot recover the polynomial of our scheme.

Furthermore, we set $m = 4$, $|q| = 160$ bits and $\omega = 250$. The computational cost, the communication cost and the size of memory in our scheme are $mR+(m+1)H$, $(t + 1 - m)|q|$ and $2m|q|$, respectively, by Section 6.4. On the other hand, the computational cost and the size of memory in the RoK scheme are $2H$ and $2w|q|$, respectively, and then the communication cost is not necessary for the update of key materials. As a result, although a little computation and a little communication are required in our scheme, the size of memory is much lower than the RoK scheme.

8 Conclusion

We have proposed the POLISH scheme, in which together with key evolution, our scheme provides both forward and backward security of a link in the presence of an adversary. Both analytical and simulation results show that our scheme is very effective. Our simulation shows that POLISH based network that is continuously attacked is resilient when $k < 62$ ($n = 400$ and $m = 4$). Our simulation also shows that a network that is temporarily attacked automatically self-heals in only about three rounds. Furthermore, we found that our analytical results well matched the simulation results of our scheme.

References

1. Blundo, C., Santis, A.D., Herzberg, A., Kutten, S., Vaccaro, U., Yung, M.: Perfectly-secure key distribution for dynamic conferences. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 471–486. Springer, Heidelberg (1993)
2. Castelluccia, C., Spognardi, A.: Rok: A robust key pre-distribution protocol for multi-phase wireless sensor networks. In: SecureComm 2007, pp. 351–360 (September 2007)
3. Du, W., Deng, J., Han, Y.S., Chen, S., Varshney, P.K.: A key management scheme for wireless sensor networks using deployment knowledge. In: INFOCOM 2004, pp. 586–597 (March 2004)
4. Eschenauer, L., Gligor, V.D.: A key-management scheme for distributed sensor networks. In: CCS 2002, pp. 41–47 (November 2002)
5. Ito, H., Miyaji, A., Omote, K.: RPoK: A strongly resilient polynomial-based random key pre-distribution scheme for multiphase wireless sensor networks. In: Globecom, pp. 1–5 (December 2010)
6. Kalkan, K., Yilmaz, S., Yilmaz, O.Z., Levi, A.: A highly resilient and zone-based key predistribution protocol for multiphase wireless sensor networks. In: Q2SWinet 2009, pp. 29–36 (October 2009)
7. Ma, D., Tsudik, G.: DISH: Distributed self-healing. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 47–62. Springer, Heidelberg (2008)
8. Pietro, R.D., Ma, D., Soriente, C., Tsudik, G.: POSH: Proactive co-operative self-healing in unattended wireless sensor networks. In: SRDS 2008, pp. 185–194 (2008)
9. Pietro, R.D., Oligeri, G., Soriente, C., Tsudik, G.: Intrusion-Resilience in Mobile Unattended WSNs. In: INFOCOM, pp. 2303–2311 (2010)
10. Yilmaz, O.Z., Levi, A., Savas, E.: Multiphase deployment models for fast self healing in wireless sensor networks. In: SECRYPT, pp. 136–144 (July 2008)

The Weakest Failure Detector to Implement a Register in Asynchronous Systems with Hybrid Communication

Damien Imbs[2] and Michel Raynal[1,2]

[1] IUF
[2] IRISA, Université de Rennes 1, France
{Damien.Imbs,raynal}@irisa.fr

Abstract. This paper introduces an asynchronous crash-prone hybrid system model. The system is hybrid in the way the processes can communicate. On the one side, a process can send messages to any other process. On another side, the processes are partitioned into clusters and each cluster has its own read/write shared memory. In addition to the model, a main contribution of the paper concerns the implementation of an atomic register in this system model. More precisely, a new failure detector (denoted $M\Sigma$) is introduced and it is shown that, when considering the information on failures needed to implement a register, this failure detector is the weakest. To that end, the paper presents an $M\Sigma$-based algorithm that builds a register in the considered hybrid system model and shows that it is possible to extract $M\Sigma$ from any failure detector-based algorithm that implements a register in this model. The paper also (a) shows that $M\Sigma$ is strictly weaker than Σ (which is the weakest failure detector to implement a register in a classical message-passing system) and (b) presents a necessary and sufficient condition to implement $M\Sigma$ in a hybrid communication system.

Keywords: Asynchronous message-passing system, Atomic register, Distributed algorithm, Failure detector, Fault-tolerance, Hybrid communication, Necessity proof, Process crash, Shared memory system, Weakest failure detector.

1 Introduction

1.1 Atomic Register

Among the objects that allow concurrent processes to exchange information and cooperate to a common goal, the *atomic register* is certainly the most fundamental. Such an object (let us denote it REG) provides the processes with two operations $REG.\mathsf{read}()$ and $REG.\mathsf{write}(v)$. The read operation provides the invoking process with the value of the object, while the write operation associates a new value v with the object.

Atomicity [9,11] means that the (possibly concurrent) read and write operations issued on a register appear as if they have been executed sequentially, and this "witness sequence" is (1) legal (a read returns the value written by the closest write that precedes it in this sequence) and (2) respects the real time occurrence order on the operations (if the operation $op1$ terminates before an operation $op2$ starts, $op1$ appears before $op2$ in the witness sequence). Let us observe that concurrent operations can be ordered in any way as long as the legality property stated in item (1) is satisfied.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 268–282, 2011.

1.2 Building an Atomic Register in a Message-Passing System

Simulating a register in an asynchronous system. In an asynchronous message-passing system, the processes communicate by sending and receiving message through channels and there are assumptions neither on the speed of processes nor on message transmission delays.

If the system is reliable, it is easy to build an atomic register on top of an asynchronous message-passing system. This is no longer the case if processes can crash. Let n be the number of processes that compose the system and t be a model parameter that defines an upper bound on the number of processes that may crash. Algorithms that build an atomic register object despite asynchrony and up to $t < n/2$ process crashes are described in [1].

An important result is proved in [1], namely, there is no algorithm implementing an atomic register in asynchronous message-passing systems where $t \geq n/2$. The intuition that underlies this impossibility is that, due to asynchrony and the fact that $t \geq n/2$, the system can appear as being partitioned, in such a way that each partition considers that the processes in the other partition have crashed (while they actually have not). The reader interested by a pedagogical introduction to these issues will consult [2,12,13].

The failure detector approach to circumvent the "$t \geq n/2$" impossibility The failure detector approach [4,5] has been introduced to circumvent impossibility results. It consists in enriching each process of an unreliable asynchronous system with an additional device (sometimes called "oracle") that provides it with hints on process failures. According to the type and the quality of these hints, several classes of failure detectors can be defined.

The class of *quorum* failure detectors, denoted Σ, has been introduced by Delporte-Gallet, Fauconnier and Guerraoui in [6]. (A quorum is a set of processes. Quorums have first been introduced by Gifford [7].) It is shown in [3,6] that Σ is the weakest class of failure detectors that allow building an atomic register object in asynchronous message passing systems despite any number of process crashes (i.e., in systems where $t = n - 1$). "Weakest" means that Σ captures the minimal information on failures that has to be known by the processes in order to implement a register. The definition of Σ is given below. It is important to notice that, due to the results of [1] and [6], it follows that Σ cannot be implemented in asynchronous message-passing systems despite any number of crashes.

1.3 Content of the Paper

Towards new system models. The advent of multicore architectures where processors share a common memory and the design of clusters (where, for example, each cluster is a multicore system) communicating by message-passing opens the door for the design of new computing models where processes communicate both by shared memory (intra-cluster communication) and message passing (point-to-point communication).

Context and content of the paper. This paper is on the construction of atomic registers in hybrid models (such as the one previously described). It has several contributions.

- The paper first introduces a simple asynchronous crash-prone model which captures the previous intra-cluster and point-to-point communication types (the meaning of m will be defined later). This system model is denoted $SM\_MP_{n,m}[\emptyset]$.
- The paper then introduces a new failure detector, denoted $M\Sigma$, and
  - Presents and proves correct an algorithm that builds an atomic register in the system model $SM\_MP_{n,m}[M\Sigma]$ ($SM\_MP_{n,m}[\emptyset]$ enriched with $M\Sigma$),
  - Shows that $M\Sigma$ is the weakest information on failures $SM\_MP_{n,m}[\emptyset]$ has to be enriched with in order an atomic register can be implemented.
- The paper finally shows that $M\Sigma$ is strictly weaker than Σ. It also presents a necessary and sufficient condition to implement $M\Sigma$ in a hybrid communication system.

Roadmap. The paper is made up of 7 sections. Section 2 presents the computation model $SM\_MP_{n,m}[\emptyset]$. The new failure detector class $M\Sigma$ is introduced in Section 3. Then, Section 4 presents an algorithm that builds an atomic register in $SM\_MP_{n,m}[M\Sigma]$ and Section 5 shows that $M\Sigma$ is optimal. Section 6 presents a necessary and sufficient condition to implement $M\Sigma$ in a hybrid communication system. Finally, Section 7 concludes the paper.

2 A Hybrid Communication System Model

2.1 System Model with Hybrid Communication

Process model. The system comprises n processes denoted $p_1, \ldots, p_n$. Each process p_i is asynchronous (i.e., it proceeds to an arbitrary speed) and sequential (it executes one step -base action- at a time). $\Pi = \{1, \ldots, n\}$ is the set of process identities.

A process can crash. A crash is a premature halt (after it has crashed, if it ever does, a process issues no more step). Let t be the upper bound on the number of processes that are allowed to crash. We assume here $t = n - 1$ (this is sometimes the *wait-free* process model).

Progress condition. In the following we are interested in a system model whose algorithms satisfy the *wait-freedom* progress condition [8]. When considering an algorithm implementing an atomic register REG, this means that a process that does not crash must return from all its invocations of the operations $REG.\mathsf{read}()$ and $REG.\mathsf{write}()$.

Message-passing communication Processes can send and receive messages through reliable channels. It is assumed that any pair of processes is connected by a bidirectional channel. Channels are reliable but asynchronous. Reliable means that messages are neither corrupted, nor duplicated nor lost. Asynchronous means that, albeit finite, message transfer delays are arbitrary.

The sending and the reception of a message are atomic steps. The processes can also use a broadcast operation, but this operation is not atomic (if a process crashes during a broadcast, an arbitrary subset of the processes receive the corresponding message).

Partially shared memory communication. The n processes are partitioned into m, $1 \leq m \leq n$, non-empty subsets $P[1], \ldots, P[m]$ called clusters (i.e., $\cup_{1 \leq x \leq m} P[x] = \Pi$ and $\forall x, y : (x \neq y) \Rightarrow (P[x] \cap P[y] = \emptyset)$).

Inside each cluster x, $1 \leq x \leq m$, the processes in $P[x]$ share a common read/write memory denoted MEM_x. MEM_x is composed of a set of atomic 1WMR (single-writer/multi-reader) registers (this assumption is without loss of generality as multi-writer/multi-reader atomic registers can be built on top single-writer/multi-reader atomic registers [2, 11, 12]). For notational convenience, we use an index/array notation for every register of MEM_x: if $i \in P[x]$, $MEM_x[i]$ can be written only by p_i and read by all processes in $P[x]$ (if $i \notin P[x]$, $MEM_x[i]$ is meaningless and p_i cannot access MEM_x).

Two examples of partially shared memory are depicted in Figure 1 where the communication channels are not depicted. In both cases, we have $n = 7$ and $m = 3$ but the partitions are different.

Fig. 1. Two examples of partially shared memories

Notation. As already indicated in the introduction, $SM\_MP_{n,m}[\emptyset]$ is used to denote the previous base wait-free hybrid distributed computing model. In the following $\emptyset$ will be replaced by a failure detector to denote the corresponding enriched model. In Figure 1 we have two instances of $SM\_MP_{7,3}[\emptyset]$.

Two particular cases. The two extreme cases $m = 1$ and $m = n$ are particularly interesting. The case $m = 1$ corresponds to the case where all processes share a common read/write memory. In that case, as the read/write communication model is stronger than the message-passing model, message-passing communication becomes useless and, consequently, $SM\_MP_{n,1}[\emptyset]$ is the classical shared memory model.

When $m = n$, there is a single process in each partition and for each x, $1 \leq x \leq n$, MEM_x boils down to the local memory of a single process. Hence, $SM\_MP_{n,n}[\emptyset]$ is the classical send/receive message-passing model.

2.2 An Atomic Register Cannot Be Built in $SM\_MP_{n,m}[\emptyset]$ when $m > 1$

Theorem 1. *$1 < m \leq n$. It is impossible to build an atomic register in $SM\_MP_{n,m}[\emptyset]$.*

Proof. The proof is a simple reduction to the impossibility theorem stating that there is no wait-free implementation of a register in an asynchronous send/receive message-passing system [1, 2, 13].

To that end, let us assume that there is an algorithm A that builds a register in $SM\_MP_{n,m}[\emptyset]$ and consider its executions in $SM\_MP_{n,m}[\emptyset]$ where, in each partition, all processes but one crash before taking any step. As A is wait-free, it follows that the $m > 1$ remaining processes implement an atomic register in the system model $SM\_MP_{m,m}[\emptyset]$, i.e., in a pure message-passing system model. This contradicts the existence of algorithm A and concludes the proof. $\square_{Theorem\ 1}$

3 A New Failure Detector Class

3.1 Failure Pattern and Failure Detector

The underlying *time model* is the set $\mathbb{N}$ of natural integers. This time notion is not accessible to the processes. It can only be used from an external observer point of view to state or prove properties. Time instants are denoted τ, τ', etc.

Formal definitions. The notions introduced here are from [5].

A *failure pattern* is a function $F()$ such that $F(\tau)$ denotes the set of processes that have crashed by time τ. As crashes are stable, we have $\forall\ \tau$: $F(\tau) \subseteq F(\tau+1)$. Given a run, let $\mathcal{F}$ be the set of processes that crash in that run (these are the faulty processes) and $\mathcal{C}$ the set of processes that do not crash (these are the correct processes). We have $\mathcal{F} = \cup_\tau F(\tau)$ and $\mathcal{C} = \Pi \setminus \mathcal{F}$.

A *failure detector history* H with range $\mathcal{R}$ is a function from $\Pi \times \mathbb{N}$ to $\mathcal{R}$ whose meaning can be interpreted as follows: $H(i,\tau)$ is the output of the considered failure detector at process p_i at time τ.

A *failure detector (FD)* $\mathcal{D}$ *with range* $\mathcal{R}$ is a function that maps each failure pattern $F()$ to a non-empty set of failure detector histories with range $\mathcal{R}$. $\mathcal{D}(F)$ is the set of behaviors (possible failure detector histories) that $\mathcal{D}$ can exhibit when the failure pattern is F.

On the operational side. From an algorithm point of view, a failure detector can be seen as a distributed device that provides each process p_i with a read-only local variable whose value at time τ is $H(i,\tau)$.

3.2 The Failure Detector Class Σ

As already indicated, this failure detector class [6] is the class of the weakest failure detectors that allow an atomic register to be implemented in the base send/receive message-passing system model. Using the formalism introduced in Section 2, this means that $SM\_MP_{n,n}[\Sigma]$ is the weakest failure detector-based system model in which an atomic register can be built.

The range of Σ is the set of all non-empty subsets of processes ($2^\Pi \setminus \emptyset$). Let Σ_i be the read-only local variable provided to p_i by Σ. Such a local output is called a *quorum*. This failure detector class is defined by the two following properties where Σ_i^τ denotes the value of Σ_i at time τ, i.e., $\Sigma_i^\tau = H(i,\tau)$.

– Intersection. $\forall\, i,j \in \Pi,\ \forall\, \tau,\tau' :\ \Sigma_i^\tau \cap \Sigma_j^{\tau'} \neq \emptyset$.
– Liveness. $\exists\, \tau :\ \forall\, \tau' \geq \tau :\ \forall\, i \in \mathcal{C} :\ \Sigma_i^\tau \subseteq \mathcal{C}$.

The intersection property states that any two quorums taken at any times intersect. This property prevents partitioning and is used to maintain the consistency of the atomic register. The liveness property states that eventually a quorum contains only correct processes. This property is used to allow a process to stop waiting for messages from crashed processes. Because any two majorities always intersect, it is easy to see that Σ can be easily implemented in a message-passing system in which a majority of process never crashes.

3.3 The Failure Detector Class $M\Sigma$

Definition. This failure detector class is for the system model $SM\_MP_{n,m}[\Sigma]$. It consists of all the failure detectors that satisfy the following properties where the quorum $M\Sigma_i$ is the local output at process p_i and $M\Sigma_i^\tau$ its value at time τ.

– Intersection. $\forall\, i,j \in \Pi,\ \forall\, \tau,\tau' :$
$\exists x,k,\ell :\ (x \in [1..m]) \wedge (k \in M\Sigma_i^\tau) \wedge (\ell \in M\Sigma_j^{\tau'}) \wedge (k,\ell \in P[x])$.
– Liveness. $\exists\, \tau :\ \forall\, \tau' \geq \tau :\ \forall\, i \in \mathcal{C} :\ M\Sigma_i^\tau \subseteq \mathcal{C}$.

The liveness property is the same as the one of Σ. The intersection property is more general. It states that any pair of quorums (whose values are taken at any times) is such that each one contains a process and these two, processes share the same common memory. This can be seen as an "indirect" intersection: $M\Sigma_i$ and $M\Sigma_j$ are not required to intersect "directly" but must include processes that share the same memory.

Particular cases. Let us first consider the case $m = 1$ (the model is then the classical base read/write shared memory model). In that case, there is a single shared memory (MEM_1) and taking always $M\Sigma_i = \{i\}$ for each p_i, both properties are always satisfied. Hence, there is a trivial implementation of $M\Sigma$ in $SM\_MP_{n,1}[\emptyset]$ which means that $M\Sigma$ adds no computational power when $m = 1$. This is in perfect agreement with the fact that $SM\_MP_{n,1}[\emptyset]$ is the base read/write shared memory model in which atomic register are given for free.

Let us now consider the case $m = n$ (the model is then the classical send/receive message-passing model). In that case, there is a single process per cluster x (e.g., $P[x]$ contains only p_x). It follows that, for the intersection property to be true, we need to have $\forall\, i,j \in \Pi,\ \forall\, \tau,\tau' : \exists k : (k \in M\Sigma_i^\tau) \wedge (k \in M\Sigma_j^{\tau'})$, i.e., $\forall\, i,j,\ \forall\, \tau,\tau' : M\Sigma_i^\tau \cap M\Sigma_j^{\tau'} \neq \emptyset$. Hence, when considering $m = n$, $M\Sigma$ boils down to Σ, which means that $SM\_MP_{n,n}[M\Sigma]$ and $SM\_MP_{n,n}[\Sigma]$ define the same computational model.

4 $M\Sigma$ is Sufficient: Building an Atomic Register in $SM\_MP_{n,m}[M\Sigma]$

This section presents and proves correct an algorithm that builds an 1WMR atomic register in $SM\_MP_{n,m}[M\Sigma]$. The writer is denoted p_w. The atomic register that is constructed is denoted REG.

The algorithm, described in Figure 2, is a simple adaptation to the hybrid model of the algorithm described in [1] that builds an atomic register in a message-passing system where a majority of processes are correct. As already indicated, while the operation send is atomic, the operation broadcast is not.

This algorithm is not designed with efficiency in mind. Its aim is only to show that an atomic register can be built in $SM\_MP_{n,m}[M\Sigma]$, and consequently show that $M\Sigma$ is sufficient. (Let us remember that, when $m = 1$, the underlying message-passing system can be easily simulated on top of the shared memory.)

The variables implementing the atomic register REG. Let p_i be a process and x its cluster (i.e., $i \in P[x]$). Process p_i stores its "local copy" of REG in $MEM_x[i]$. More precisely, this base register has two fields $MEM_x[i].val$ (which stores the last value of REG know by p_i) and $MEM_x[i].sn$ (which stores the corresponding sequence number).

The variables in italics with subscript s are variables that are local to process p_s. These local variables are used to generate local sequence numbers.

*The operation REG.*write(v). This operation (which can be issued only by p_w) first associates a new sequence number (sn_w) with its current invocation (line 01). Then, it sends the message WRITE(v, sn_w) to all the processes to inform them on the new write (line 02). When, $M\Sigma_w$ is such that p_w has received a matching acknowledgment from each of its processes, the operation returns ok and terminates (lines 02-04).

Let p_i be a process such that $i \in P[x]$. When p_i (p_i can be p_w) receives a message WRITE$(v, seqnb)$ from a process p_j it updates $MEM_x[i]$ if this message carries a more recent write (line 12). Moreover, p_i always sends by return an acknowledgment carrying $seqnb$ (line 13) to inform p_j that its "local copy" of REG has now a sequence number which is $\geq seqnb$.

*The operation REG.*read$()$. This operation proceeds in two phases. In the first phase (lines 05-08), p_i broadcasts a message READ(r_sn_i) where r_sn_i is used to identify all its read invocations (lines 05-06) and waits until $M\Sigma_i$ contains only processes from which p_i has received a matching acknowledgment (line 07).

When a process p_k receives such a message READ(r_sn) from a process p_j, it computes the most recent value of REG stored in the cluster shared memory MEM_x, i.e., such that $k \in P[x]$ (lines 14-15) and sends back to p_j this most recent value (line 16).

Finally, p_i determines the most recent value of REG it has received from the processes in $M\Sigma_i$ (line 08). That value will be returned by the read operation (line 11), but before, p_i has to execute the second phase (lines 08-10) whose aim is ensure that no overwritten value is ever returned by a read operation. To that end, p_i simulates a write of the value it is about to return.

Theorem 2. *Let $1 \leq m \leq n$. The algorithm described in Figure 2 is wait-free construction of a 1WMR atomic register in $SM\_MP_{n,m}[M\Sigma]$.*

Due to page limitation, the proof can be found in [10].

operation REG.write(v): % This code is only for the single writer p_w %
(01) $sn_w \leftarrow sn_w + 1$;
(02) broadcast WRITE(v, sn_w);
(03) **wait until** $\big(M\Sigma_w$ is such that $\forall\, j \in M\Sigma_w$: p_w has received ACK(sn_w) from $p_j\big)$;
(04) return(ok).

% The code snippets that follow are for every process p_i, $1 \leq i \leq n$ %
% Moreover, the value x denotes p_i's partition number i.e., x is such that $i \in P[x]$ %

operation REG.read():
(05) $r\_sn_i \leftarrow r\_sn_i + 1$;
(06) broadcast READ(r_sn_i);
(07) **wait until** $\big(M\Sigma_i$ is such that $\forall\, j \in M\Sigma_i$: p_i has rec. VAL(v, sn, r_sn_i) from $p_j\big)$;
(08) $\langle v, sn\rangle \leftarrow (\langle v, sn\rangle \mid$ VAL(v, sn, r_sn_i) rec. $\wedge\, \nexists\, sn' > sn :$ VAL$(-, sn', -)$ rec.$)$;
(09) broadcast WRITE(v, sn);
(10) **wait until** $\big(M\Sigma_i$ is such that $\forall\, j \in M\Sigma_i$: p_i has received ACK(sn) from $p_j\big)$;
(11) return(v).

Task T1: when WRITE$(v, seqnb)$ **is received from** p_j:
(12) **if** $(MEM_x[i].sn < seqnb)$ **then** $MEM_x[i] \leftarrow \langle v, seqnb\rangle$ **end if**;
(13) send ACK$(seqnb)$ to p_j.

Task T2: when READ(r_sn) **is received from** p_j:
(14) $mem \leftarrow \{MEM_x[k]$ such that $k \in P[x]\}$;
(15) $\langle v, sn\rangle \leftarrow (mem[k] \mid \nexists \ell : mem[\ell].sn > mem[k].sn)$;
(16) send VAL(v, sn, r_sn) to p_j.

Fig. 2. Building an atomic 1WMR register $SM\_MP_{n,m}[M\Sigma]$

5 $M\Sigma$ is Necessary

5.1 $M\Sigma$ is the Weakest FD for a Register in a Hybrid Communication Model

The previous section has shown that an atomic register can be built in $SM\_MP_{n,m}[M\Sigma]$, thereby showing that enriching $SM\_MP_{n,m}[\emptyset]$ with $M\Sigma$ is sufficient (from an "information on failures" point of view) when one wants to build an atomic register. This section addresses the necessity side. It shows that any failure detector D such that an atomic register can be built in $SM\_MP_{n,m}[D]$ provides enough information on failures in order $M\Sigma$ can be built in $SM\_MP_{n,m}[D]$.

Let D be any failure detector such that there is an algorithm A that allows building an atomic register in $SM\_MP_{n,m}[D]$. The proof of the "necessity" part consists in showing that it is possible to build a failure detector of the class $M\Sigma$ from A executed in $SM\_MP_{n,m}[D]$. In the failure detector parlance, we say that it is possible to "extract" Σ from A. In a very interesting way, the proposed extraction algorithm is the one we have presented in [3,13] (for Σ) but its proof is different. Hence, the current paper shows that the extraction algorithm introduced in [3] has a generic dimension with respect to failure detectors.

5.2 Bonnet-Raynal's Extraction Algorithm

This section presents Bonnet-Raynal's extraction algorithm introduced in [3] where it is assumed that the underlying D-based algorithm A builds an atomic register in $SM\_MP_{n,m}[D]$. Albeit not new, this presentation is needed for completeness of the minimality proof.

Arrays of atomic registers. Let Q be a non-empty set of processes, and $REG_Q[1..n]$ an array of n atomic registers (initialized to $[\bot, \ldots, \bot]$), such that each atomic register $REG_Q[x]$ is implemented by the n-process algorithm A executed only by $|Q|$ threads, each one associated with a process of Q.

A simple register-based algorithm. Let WR_Q be the following register-based algorithm (also called a task) where each process p_i such that $i \in Q$ executes the following algorithm (where $reg_i[1..n]$ is an array local to p_i):

algorithm WR_Q:
$REG_Q[i].\mathsf{write}(\top)$; **for each** $x \in \{1, ..., n\}$ **do** $reg_i[x] \leftarrow REG_Q[x].\mathsf{read}()$ **end for**.

The process p_i first writes the value $\top$ in its entry of the array REG_Q, and then reads asynchronously all its entries. The $REG_Q[i].\mathsf{write}(\top)$ and $REG_Q[x].\mathsf{read}()$ operations are provided to the processes by the previous algorithm A. (Let us notice that the value obtained by a read is irrelevant. As we will see, what is important is the fact that $REG_Q[x]$ has been written or not.) A corresponding run of WR_Q is denoted E_Q. In that run, no process outside Q sends or receives messages related to the task WR_Q.

Let us remember that $\mathcal{C}$ is the set of identities of the processes that are correct in the considered run. Let us observe that, as the underlying failure detector-based algorithm A that builds a register is correct, if the set Q contains all the correct processes (i.e., $\mathcal{C} \subseteq Q$), E_Q is such that every correct process terminates the task WR_Q. In the other cases, i.e., for the tasks WR_Q such that $\neg(\mathcal{C} \subseteq Q)$, E_Q is such that a process of Q either terminates WR_Q, or blocks forever, or crashes. (This depends on the actual failure pattern, the outputs of the underlying failure detector D used by the algorithm A, and the code of A. As an example, let us consider the task WR_Q, and two correct processes p_i and p_j such that $i \in Q$ and $j \notin Q$. Let $th_{i,Q}$ be the thread of p_i involved in Q. As $j \notin Q$, the thread $th_{j,Q}$ does not exist. The thread $th_{i,Q}$ can block forever when it executes A to read or write a register of $REG_Q[1..n]$ if, due to the output of D and the code of A, it is directed to wait for a message from $th_{j,Q}$ -that does not exist-[1]).

Running concurrently $2^n - 1$ *tasks.* The extraction algorithm considers the $2^n - 1$ distinct tasks WR_Q where Q is a non-empty set of 2^{Π}. To that end, each process p_i manages 2^{n-1} threads, one for each subset Q such that $i \in Q$. Let us notice that the crash of a process p_i entails the crash of all its threads.

[1] A similar blocking can happen when the processes use an underlying Ω-based algorithm [4] and, in the considered run, the correct process that is eventually elected as a leader does not participate in the algorithm.

The extraction algorithm. The algorithm that extracts Σ is described in Figure 3. Let us recall that the aim is to provide each process p_i with a local variable Σ_i such that the $(\Sigma_x)_{1\leq x\leq n}$ variables satisfy the intersection and liveness properties of Σ.

To that end, each process p_i manages two local variables: a set of sets of process identities, denoted $quorum\_sets_i$, and a queue denoted $queue_i$. The aim of the set of sets $quorum\_sets_i$ is to contain all the sets Q such that p_i terminates WR_Q (task $T1$), while $queue_i$ is managed in such a way that eventually the correct processes appear in it before the faulty processes (tasks $T2$ and $T3$).

The idea is to select an element of $quorum\_sets_i$ as the current output of Σ_i. As we will see in the proof, given any pair of processes p_i and p_j, any quorum in $quorum\_sets_i$ has a non-empty intersection with any quorum in $quorum\_sets_j$, thereby supplying the required intersection property.

The main issue is to ensure the liveness property of Σ_i (eventually Σ_i has to contain only correct processes) while preserving the intersection property. This is realized with the help of the local variable $queue_i$ as follows: the current output of Σ_i is the set (quorum) of $quorum\_sets_i$ that appears as being the "first" in $queue_i$. The formal definition of "first element of $quorum\_sets_i$ with respect to $queue_i$" is stated in the task $T4$. To make it easy to understand, let us consider the following example. Let $quorum\_sets_i = \{\{3,4,9\},\{2,3,8\},\{1,2,4,7\}\}$, and $queue_i =< 4,8,3,2,7,5,9,1,\cdots >$. The set $S = \{2,3,8\}$ is the first set of $quorum\_sets_i$ with respect to $queue_i$ because each of the other sets $\{3,4,9\}$ and $\{1,2,4,7\}$ includes an element (9 and 7, respectively) that appears in $queue_i$ after the elements of S. (In case several sets are "first", any of them can be selected). The notion of *first quorum* is used to ensure the liveness of Σ, i.e., the set Σ_i of any correct process p_i eventually contains only correct processes.

Remark 1. Initially $quorum\_sets_i$ contains the set $\{1,\ldots,n\}$. As no set of processes is ever withdrawn from $quorum\_sets_i$ (task $T1$), $quorum\_sets_i$ is never empty. Moreover, it is not necessary to launch the task $WR_{\{1,\ldots,n\}}$ in which all the processes

Init: $quorum\_sets_i \leftarrow \{\{1\ldots,n\}\}$; $queue_i \leftarrow < 1,\ldots,n >$;
for each $Q \in \left(2^{\Pi} \setminus \{\emptyset,\{1,\ldots,n\}\}\right)$ **do**
if $(i \in Q)$ **then** launch a thread associated with the task WR_Q **end if end for**.
% Each process p_i participates concurrently in all the tasks WR_Q such that $i \in Q$ %

Task $T1$: **when** p_i terminates in the task WR_Q: $quorum\_sets_i \leftarrow quorum\_sets_i \cup \{Q\}$.

Task $T2$: **repeat periodically** broadcast ALIVE(i) **end repeat**.

Task $T3$: **when** ALIVE (j) **is received**:
suppress j from $queue_i$; enqueue j at the head of $queue_i$.

Task $T4$: **when** p_i reads Σ_i:
let $m = \min_{Q\in quorum\_sets_i}(\max_{x\in Q}(rank[x]))$ where $rank[x]$ = rank of x in $queue_i$;
return (a set Q such that $\max_{x\in Q}(rank[x]) = m$).

Fig. 3. Extracting Σ from a FD-based algorithm A that implements a register (code for p_i)

participate. This is because, as the underlying failure detector-based algorithm A (that implements a register) is correct, it follows that all the correct processes terminate in the task $WR_{\{1,\ldots,n\}}$. This case is directly taken into account in the initialization of $quorum\_sets_i$ (thereby saving the execution of the task $WR_{\{1,\ldots,n\}}$).

Remark 2. A simple examination of the extraction algorithm shows that (1) both the variables $queue_i$ and $quorum\_sets_i$ are bounded, and (2) messages carry bounded values, from which it follows that the construction is bounded.

5.3 Minimality of $M\Sigma$

Theorem 3. *Let $1 \leq m \leq n$. $M\Sigma$ is the weakest failure detector $SM\_MP_{n,m}[\emptyset]$ has to be enriched with in order an atomic register can be built.*

Proof. Let D be any failure detector such that there is an algorithm A that builds an atomic register in $SM\_MP_{n,m}[D]$. The proof consists in showing that, given such an algorithm A, the algorithm described in Figure 3 builds a failure detector $M\Sigma$.

Proof of the intersection property
The proof is by contradiction. Let us first observe that the set Σ_i returned to a process p_i is a set of $quorum\_set_i$ (that contains the set $\{1,\ldots,n\}$ -initial value- plus all the sets Q such that p_i terminates WR_Q). Let us assume that there are two sets Q_1 and Q_2 such that (1) $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$, and (2) $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k,\ell\} \not\subseteq P[x])$. Let us notice that the first item means that Q_1 and Q_2 can be returned to some processes as their local value for Σ. The second item means that at least one of k and ℓ is not in $P[x]$, from which we conclude that the processes in Q_1 and Q_2 cannot communicate via the shared memory cluster $P[x]$.

Let p_i be a process that terminates WR_{Q_1} and p_j a process that terminates WR_{Q_2} (due to the "contradiction" assumption, such processes do exist). Using the fact that the system is asynchronous, let us construct the runs E_{Q_1} and E_{Q_2} associated with WR_{Q_1} and WR_{Q_2} as follows. If any, the messages sent by the processes of Q_1 to the processes of Q_2, when they execute A to implement each register of the array REG_{Q_1}, are delayed for an arbitrarily long period (until p_i has added Q_1 to $quorum\_set_i$ and p_j has added Q_2 to $quorum\_set_j$). And similarly for the messages sent by the processes of Q_2 to the processes of Q_1 when they execute A for each register of the array REG_{Q_2}.

Let us observe that, in the concurrent runs E_{Q_1} and E_{Q_2}, the algorithm A that is executed only by (1) the processes of Q_1 in E_{Q_1} to build the registers $REG_{Q_1}[1..n]$, and (2) only the processes of Q_2 in E_{Q_2} to build the registers $REG_{Q_2}[1..n]$, is fed with the same outputs of the underlying failure detector D. Since (a) $p_i \in Q_1$ and $p_j \in Q_2$, and (b) $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k,\ell\} \not\subseteq P[x])$, p_i does not write to $REG_{Q_2}[i]$ and p_j does not write to $REG_{Q_1}[j]$. Thus, p_i reads $\perp$ from $REG_{Q_1}[j]$, and p_j reads $\perp$ from $REG_{Q_2}[i]$.

Let us construct a run $E_{Q_{12}}$, where $Q_{12} = Q_1 \cup Q_2$, that is a simple merge of E_{Q_1} and E_{Q_2} defined as follows. In this run, the algorithm A (that involves only the processes in Q_{12} and implements the array of registers $REG_{Q_{12}}[1..n]$) is fed with the same failure detector outputs as the ones supplied to the concurrent runs E_{Q_1} and E_{Q_2}.

Moreover, the messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed as in E_{Q_1} and E_{Q_2}. So, p_i (resp., p_j) receives the same messages and the same outputs from the underlying failure detector in $E_{Q_{12}}$ and E_{Q_1} (resp., E_{Q_2}).

- On the one side, we have the following. As the process p_i receives the same messages and the same failure detector outputs in both $E_{Q_{12}}$ and E_{Q_1}, it follows that $REG_{Q_1}[1..n]$ and $REG_{Q_{12}}[1..n]$ contain the same values. Consequently, p_i reads $\perp$ from $REG_{Q_{12}}[j]$. Similarly, p_j reads $\perp$ from $REG_{Q_{12}}[i]$.
- On the other side we have the following. In $E_{Q_{12}}$, the process p_i writes $\top$ into $REG_{Q_{12}}[i]$ and the process p_j writes $\top$ into $REG_{Q_{12}}[j]$. Moreover, one of these operations terminates before the other. Without loss of generality, let us assume that the write by p_i terminates before the write by p_j. Consequently, p_j reads $REG_{Q_{12}}[i]$ after it has been written. Due to the atomicity of that register, it follows that p_j obtains the value $\top$ when it reads $REG_{Q_{12}}[i]$.

The second item contradicts the first one. It follows that the initial assumption (existence of a failure detector-based algorithm A that builds a register, $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ and $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$) is false, from which we conclude that at least one of the two assertions stated at the beginning of the proof (namely (1) $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ and (2) $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$) is false, which completes the proof of the intersection property of $M\Sigma$.

Proof of the liveness property

As far as the liveness property is concerned, let us consider the task $WR_{\mathcal{C}}$ (recall that $\mathcal{C}$ is the set of correct processes). As the underlying failure detector-based algorithm A that implements the registers $REG_{\mathcal{C}}[1..n]$ is correct (assumption), each correct process p_i terminates its $REG_{\mathcal{C}}[i].\mathsf{write}(\top)$ and $REG_{\mathcal{C}}[x].\mathsf{read}()$ operations in $E_{\mathcal{C}}$. Consequently, in the extraction algorithm, the variable $quorum\_set_i$ of each correct process p_i eventually contains the set $\mathcal{C}$.

Moreover, after some finite time, each correct process p_i receives $\text{ALIVE}(j)$ messages only from correct processes. This means that, at each correct process p_i, all the correct processes eventually precede the faulty processes in $queue_i$. Due to the definition of "first set of $quorum\_set_i$ with respect to $queue_i$" stated in the task $T4$, it follows that, from the time $\mathcal{C}$ has been added to $quorum\_set_i$, the quorum Q selected by the task $T4$ is always such that $Q \subseteq \mathcal{C}$, which proves the liveness property of $M\Sigma$. $\square_{Theorem\ 3}$

5.4 $M\Sigma$ is Strictly Weaker Than Σ When $m < n$

Theorem 4. *Let $m < n$. The model $SM\_MP_{n,m}[M\Sigma]$ is strictly weaker than the model $SM\_MP_{n,m}[\Sigma]$.*

Proof. To prove the theorem we have to show that (a) it is possible to build $M\Sigma$ in $SM\_MP_{n,m}[\Sigma]$ and (b) it is impossible to build Σ in $SM\_MP_{n,m}[M\Sigma]$ when $m < n$.

- Proof of Item (a). For any i and τ, let us define $M\Sigma_i^\tau = \Sigma_i^\tau$. As $\Sigma_i^\tau \cap \Sigma_j^{\tau'} \neq \emptyset$, it follows that that $\exists k \in M\Sigma_i^\tau \cap M\Sigma_j^{\tau'}$ and there is trivially a partition x such that

$k \in P[x]$ which proves the intersection property of $M\Sigma$. The liveness property of $M\Sigma$ follows directly from its Σ counterpart.

– Proof of Item (b). The proof is by contradiction. let us assume that there is a wait-free algorithm A that builds Σ in $SM\_MP_{n,m}[M\Sigma]$ when $m < n$.
As $m < n$, there is a partition $P[x]$ and a pair of processes p_i and p_j such that $i, j \in P[x]$ (i.e., p_i and p_j belong to the same memory partition x). Let us consider a run in which p_i and p_j are correct and all the other processes crash before taking any step. As $i, j \in P[x]$, $\forall\, \tau, \tau'$, $M\Sigma_i^{\tau} = \{i\}$ and $M\Sigma_j^{\tau'} = \{j\}$ are correct local outputs of $M\Sigma$ (they satisfy its intersection and liveness properties).
Let us suppose that, while it is executing A, p_j pauses during an arbitrary long but finite period during which p_i runs solo and (due to asynchrony) receives no message from p_j. As $\forall\, \tau$ we have $M\Sigma_i^{\tau} = \{i\}$, p_i cannot distinguish this execution of A from the one in which it is the only correct process. Hence, after some finite time, because it is wait-free, A has to output $\{i\}$ at p_i in order the liveness property of Σ be satisfied. Hence, there is a time τ_i such that $\Sigma_i^{\tau_i} = \{i\}$.
Let us now suppose that, after time τ_i, p_i pauses for an arbitrary long but finite period during which p_j runs solo and (due to asynchrony) receives no message from p_i. It follows from the same reasoning as before that there is a time τ_j at which we have $\Sigma_j^{\tau_j} = \{j\}$.
It follows that $\Sigma_i^{\tau_i} \cap \Sigma_j^{\tau_j} = \emptyset$, and the intersection property of Σ is violated which concludes the proof of the theorem. $\square_{Theorem\ 4}$

Remark. Let us observe from the second part of the previous proof (Item b) that, when the processes of all but one memory clusters crash, $M\Sigma$ is too week to give information on failures. Moreover, the next corollary follows from the previous theorem when we consider the case $m = 1$ (read/write shared memory model in which $M\Sigma$ can be trivially implemented).

Corollary 1. *Σ cannot be built from atomic registers only.*

6 On the Implementability of $M\Sigma$ Despite Asynchrony and Failures

When $m = n$ (pure asynchronous message-passing system), $M\Sigma$ boils down to Σ and it is known that Σ can be implemented in a pure message-passing asynchronous system where a majority of processes are correct. Hence, the question: Is there a necessary and sufficient condition C on n, m and a system parameter associated with failures such that $M\Sigma$ can be implemented in $SM\_MP_{n,m}[C]$ (where $SM\_MP_{n,m}[C]$ denotes the system model $SM\_MP_{n,m}[\emptyset]$ restricted to the runs where C is satisfied)? This section presents such a necessary and sufficient condition C.

Notion of a faulty cluster. Let us say that a cluster x is faulty in a run if all processes of $P[x]$ are faulty in that run. Let t, $1 \leq t < m$ be the upper bound on the number of faulty clusters.

The next theorem shows that $C \equiv (t < m/2)$ is a necessary and sufficient condition to implement $M\Sigma$ in $SM\_MP_{n,m}[\emptyset]$.

Theorem 5. *Let COND be the set of all the predicates on n, m and t, $C' \in COND$ and $C = (t < m/2)$. $M\Sigma$ can be built in $SM\_MP_{n,m}[C']$ if and only if $C' \Rightarrow C$.*

Proof. The proof of the theorem is made up of two parts: (a) $M\Sigma$ can be built in all the runs in which C is satisfied and (b) $M\Sigma$ cannot be built in all the runs in which C is not satisfied.

Proof of Item (a). The algorithm described in Figure 4 builds $M\Sigma$ in $SM\_MP_{n,m}[t < m/2]$. Initially, each process p_i initializes $M\Sigma_i$ to Π (the set of all process identities). Then, repeatedly, p_i broadcasts a message ALIVE(i), waits until it has received a message from $(m - t)$ processes belonging to different clusters and sets $M\Sigma_i$ to set of processes. It is easy to show that the intersection and liveness properties of $M\Sigma$ are satisfied.

- Let us first observe that, due to the assumption $t < m/2$, no correct process remains blocked forever in the **wait** statement. Moreover, after some finite time, a correct process receives message only from correct processes. It follows directly from these two observations that, after some finite time, $M\Sigma_i$ contains only correct processes which is the liveness property of $M\Sigma$.
- $\forall\, i, j \in \Pi$, $\forall\, \tau, \tau'$, let us consider the values of $M\Sigma_i^\tau$ and $\Sigma_j^{\tau'}$. It follows from $t < m/2$ that Σ_i^τ contains processes belonging to a majority of clusters, and similarly for $\Sigma_j^{\tau'}$. As any two majorities intersect, we conclude that there is cluster x such that $k \in M\Sigma_i^\tau \wedge \ell \in M\Sigma_j^{\tau'} \wedge \{k, \ell\} \subseteq P[x]$ which proves the intersection property of $M\Sigma$.

```
M Σ_i ← Π;
repeat forever
    broadcast ALIVE(i);
    wait until (messages received from processes in (m − t) different clusters);
    M Σ_i ← the set of processes from which messages have been received at previous line
end repeat.
```

Fig. 4. Building $M\Sigma$ in $SM\_MP_{n,m}[t < m/2]$ (code of p_i)

Proof of Item (b). Considering that $t \geq m/2$, let us partition the set of clusters in two sets QC_1 and QC_2 (i.e., $QC_1 \cap QC_2 = \emptyset$ and $QC_1 \cup QC_2 = \cup_{1 \leq x \leq m} P[x]$). Due to asynchrony it is possible to delay for an arbitrary long period all the messages from the processes in QC_1 to the processes in QC_2 and all the messages from the processes in QC_2 to the processes in QC_1. Then, the processes in QC_1 cannot distinguish the case where the processes in QC_2 have crashed or are only very slow and similarly for the processes of QC_2 with respect to the processes of QC_1. The impossibility follows from this classical partitioning argument. $\square_{Theorem\ 5}$

7 Conclusion

This paper has introduced a new distributed computing model with hybrid communication (any pair of processes can communicate by asynchronous message-passing

and processes are partitioned into clusters in which they can communicate through a read/write shared memory). The paper has investigated the minimal information on failures that allows an atomic register to be implemented in such a hybrid communication model. This minimal information on failures is captured by a new failure detector denoted $M\Sigma$ (which generalizes the failure detector Σ). The paper has also presented a necessary and sufficient condition on the number of faulty shared memory clusters that, when satisfied, allows $M\Sigma$ to be implemented despite the net effect of asynchrony and failures. The paper has also shown that, while Σ is the weakest failure detector that allows a register to be implemented in a pure asynchronous message-passing system, it cannot be implemented from registers only.

As suggested by a referee, it would be interesting to investigate a parameterized definition of Σ where the parameter would be a cluster partition of the system.

Acknowledgments. We would like to thank the anonymous referees whose suggestions helped us improve the paper. This work has been realized in the context of the TRANSFORM Marie Curie project (638239) of the European Community (2009-2013).

References

1. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message Passing Systems. Journal of the ACM 42(1), 121–132 (1995)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn., 414 pages. Wiley-Interscience, Hoboken (2004)
3. Bonnet, F., Raynal, M.: A Simple Proof of the Necessity of the Failure Detector Σ to Implement an Atomic Register in Asynchronous Message-passing Systems. Information Processing Letters 110(4), 153–157 (2010)
4. Chandra, T., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus. Journal of the ACM 43(4), 685–722 (1996)
5. Chandra, T., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
6. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Tight Failure Detection Bounds on Atomic Object Implementations. Journal of the ACM 57(4) Article 22, 32 pages (2010)
7. Gifford, D.K.: Weighted Voting for Replicated Data. In: Proc. 7th ACM Symposium on Operating System Principles (SOSP 1979), pp. 150–172. ACM Press, New York (1979)
8. Herlihy, M.P.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
9. Herlihy, M.P., Wing, J.L.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
10. Imbs, D., Raynal, M.: The Weakest Failure Detector to Implement a Register in Asynchronous Systems with H ybrid Communication. Tech Report #1792, 11 pages, IRISA, Université de Rennes, France (2011)
11. Lamport, L.: On Interprocess communication. Part I: Formalism. Part II: Algorithms. Distributed Computing 1-2(2), 87–103 (1986)
12. Lynch, N.A.: Distributed Algorithms, 872 pages. Morgan Kaufmann Pub, San Francisco (1996)
13. Raynal, M.: Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems, 251 pages. Morgan & Claypool Pub (2010) ISBN 978-1-60845-293-4

Price Stabilization in Networks — What Is an Appropriate Model ?

Jun Kiniwa[1] and Kensaku Kikuta[2]

[1] Department of Applied Economics, University of Hyogo,
8-2-1 Gakuen nishi-machi, Nishi-ku, Kobe-shi, 651-2197 Japan
kiniwa@econ.u-hyogo.ac.jp
[2] Department of Strategic Management, University of Hyogo,
kikuta@biz.u-hyogo.ac.jp

Abstract. We consider a simple network model for economic agents where each can buy commodities in the neighborhood. Their prices may be initially distinct in any node. However, by assuming some rules on new prices, we show that the distinct prices will converge to unique by iterating buy and sell operations. First, we present a protocol model in which each agent always bids an arbitrary price in the difference between his own price and the lowest price in the neighborhood, called max price difference. Next, we derive the condition that price stabilization occurs in our model. Furthermore, we consider game (auction) theoretic price determination by assuming that each agent's value is uniformly distributed over the max price difference. Finally, we perform a simulation experiment. Our model is suitable for investigating the effects of network topologies on price stabilization.

Keywords: multiagent model, price determination, game (auction) theory, Bayesian-Nash equilibrium.

1 Introduction

Motivation. Conventionally, the topics of price determination have been discussed in the context of microeconomics approach. Figure 1 shows supply and demand curves, where an equilibrium occurs at the intersection of them — if the price is higher (resp. lower) than the equilibrium, there is excess supply (resp. excess demand) and thus the price moves to the equilibrium. At the equilibrium price, the quantity of goods sought by consumers is equal to the quantity of goods supplied by producers. Neither consumers nor producers have an incentive to alter the price or quantity at the equilibrium. Thus, the price determination has been considered as an abstract, theoretical model which explains a market mechanism. To know a detailed process to the equilibrium, we need more sophisticated model, e.g., multiagent approach, which gives us another insight into the price determination.

Recently, the topics of price fluctuation have been discussed in the context of multiagent approach. In minority games [3], or artificial financial markets [20],

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 283–295, 2011.

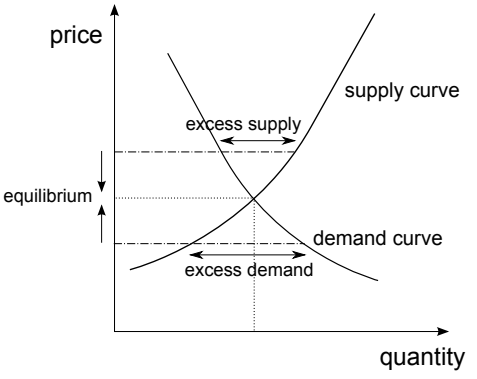


Fig. 1. Supply and Demand

the macroscopic behavior arises from interactions governed by micro-rules of agents. Such an idea is reasonable because economic phenomena are brought about by each person's business activities. Thus, the price fluctuation has been considered as an abstract, experimental model which explains a market mechanism. The price determination, however, has not been theoretically revealed by the multiagent approach as far as we know.

We can construct a price determination model by applying the idea of stabilization to the multiagent approach. The self-stabilization has been originally studied as the recovery from transient faults in distributed systems. From any initial state, self-stabilizing algorithms eventually lead to a legitimate state without any aid of external actions. We notice that the properties of self-stabilization resemble those of price determination in convergence to a stable state without external influences. For example, a self-stabilizing consensus algorithm is associated with the price determination because every agent eventually has the same agreement.

If we consider the behavior of the economic agents as a distributed system, we can develop a new model for the economy. So we construct a network model consisting of nodes and edges as cities and their links to neighbors, respectively. Each node contains an agent which represents people in the city. Any interaction among agents is governed by micro-rules, that is, the agents who want to buy a commodity make bids to their neighboring nodes. Then, the agents who want to sell the commodity accept the highest bid, like an auction. By iterating these rules, the prices will reach an equilibrium.

Related Work. The classical theory of price determination in microeconomics is introduced, e.g., in [21,22]. We review the theory from multiagent points of view. Though several economic network models have been already known [2,10,14], such models contain a bipartite structure [10,14] or traders who play intermediary roles [2]. Agent-based stabilization has been discussed in [1,6,11,13]. Unlike our *staying* agents, their ideas are to use mobile agents for the purpose of stabilization. The agents in our model may need to know how to win in an auction. The auction theory using bidding mechanisms is argued in [17,19]. It is useful

in designing protocols by what price we should make a bid. Several kinds of game theoretic flavors have appeared in self-stabilization, e.g., time complexity analysis [9], strategies with optimal complexity [5], relationships between Nash equilibria and stabilization [7,12], and strategies in algorithms [15]. This paper appends an auction approach to such a trend. Our protocol in Section 3 can be considered as a kind of consensus algorithm. The consensus algorithm in decentralized systems is described in [18], and its self-stabilizing version is described in [8,4].

Contributions. We consider a new network model for economic agents where each can buy and sell commodities in the neighborhood. Such a network model is useful for investigating price behavior in microeconomics. First, we present a protocol in which each agent always offers a fixed price without considering other bidders' strategies. Then, we consider how to determine a bidding price by using Bayesian-Nash equilibrium. Finally, we perform a simulation experiment which reveals the effects of network topologies.

The rest of this paper is organized as follows. Section 2 states our model. Section 3 shows that a condition such that our protocol can stabilize distinct commodity prices. Section 4 investigates the bidding rate by game (auction) theory. Then, Section 5 shows some simulation results which reveal the effects of network topologies. Finally, Section 6 concludes the paper.

2 Model

Our system can be represented by a connected network $G = (V, E)$, consisting of a set of nodes V and edges E, where the nodes represent cities and a pair of neighboring nodes (cities) is linked by an edge. Let N_i be a set of neighboring nodes of $i \in V$, and let $N_i^+ = N_i \cup \{i\}$. We assume that each node $i \in V$ has a commodity and its initial price may be distinct. Let $P_i(t)$, or denoted by P_i, be the commodity price at node i for the time step $t \in T = (0, 1, 2, \ldots)$, Each node $i \in V$ has exactly one representative agent a_i who always stays at i and can buy commodities in the neighborhood N_i. The *buy operation* is executed as follows.

Each agent a_i assigns a *value* $v_i^k(t)$, or denoted by v_i^k, to the commodity of any neighboring node k, where the value means the maximum amount an agent is willing to pay. Agent a_i compares its own commodity price P_i with the neighboring price P_k. If the cheapest price in N_i is P_k $(< P_i)$, the agent a_i wants to buy it and submits a bid $b_i^k(t)$, or denoted by b_i^k, to node k, where b_i^k is determined by a *strategy* $S(v_i^k)$. We call such $P_i - P_k$ a *max price difference*. We consider $v_i^k(t) = P_i(t)$ for any $k \in N_i$ because he can buy it at the price $P_i(t)$ in his node.

After accepting bids from N_k, agent a_k *contracts* with all agents who submitted the highest price. Then, a_k passes the commodity to the contracted agents and sets $P_k(t+1)$ to the highest price, called a *sell operation*. We do not take the carrying cost of commodities into consideration but focus on the change of prices. In this way, at every time, any price is updated if necessary. The state Σ_i of each node $i \in V$ is represented by the commodity price $P_i(t)$.

We assume a *synchronous model*, that is, every agent periodically (for each *round*) exchanges messages and knows the states of neighboring agents. The global state of all nodes is called a *configuration*. The set of all configurations is denoted by $\Gamma = \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_{|V|}$. An *atomic step* consists of reading the states of neighboring agents, a buy/sell operation, and updating its own state. Then, a configuration is changed from $\boldsymbol{c}_j \in \Gamma$ into $\boldsymbol{c}_{j+1} \in \Gamma$ (or $\boldsymbol{c}_{j+1}$ is reached from $\boldsymbol{c}_j$) by the atomic step. An *execution* E is a sequence of configurations $E = \boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_j, \boldsymbol{c}_{j+1}, \ldots$ such that $\boldsymbol{c}_{j+1} \in \Gamma$ is reached from $\boldsymbol{c}_j \in \Gamma$.

3 Protocol Design

In this section, we consider a protocol model, called **ArbitraryBid**, in which each agent a_i always makes a bid b_i^k $(P_k(t) \leq b_i^k \leq P_i(t))$ to an agent $a_k \in N_i^+$ with the lowest price in the neighborhood.

ArbitraryBid

- Each agent a_i makes a bid with an integer price

$$b_i^k(t) \in [P_k(t), P_i(t)]$$

  to node $k \in N_i^+$ which has the lowest-price commodity in N_i^+. The agent a_k contracts with the neighboring a_i who has submitted the highest bid. That is, the commodity price at node k becomes

$$P_k(t+1) := \max_{i \in N_k} b_i^k(t).$$

- If a_i accepts no bidding from N_i and a_i's bid is accepted by some neighboring a_k, the price at time $t+1$ will be cut to

$$P_i(t+1) := b_i^k(t).$$

- If several agents make bids to node k with the same highest price, agent a_k makes deals with all of them.

Example 1. Figure 2 shows an example of our network system consisting of 4 nodes $V = \{0, 1, 2, 3\}$. At time t, the prices of commodities are $(P_0(t), P_1(t), P_2(t), P_3(t)) = (50, 10, 110, 70)$ as shown in Figure 2(a). Each agent a_i wants to buy the commodity if its price is lower than $P_i(t)$, i.e., $P_i(t) > \min_{j \in N_i} P_j(t)$. Thus, agent a_2 makes a bid to node 0 with price $80 \in [50, 110]$. Likewise, agents a_0 and a_3 make bids to node 1 and node 0, respectively. Then, a_2's bid and a_0's bid are successful, a_2 (resp. a_0) makes a contract with a_0 (resp. a_1).

At time $t+1$, the prices become $(P_0(t+1), P_1(t+1), P_2(t+1), P_3(t+1)) = (80, 30, 80, 70)$ as shown in Figure 2(b). Since agent a_2 accepted no bids from N_2 and a_2's bid b_2^0 was accepted by agent 0, the price is cut to 80 $(= b_2^0(t))$ at time $t+1$. □

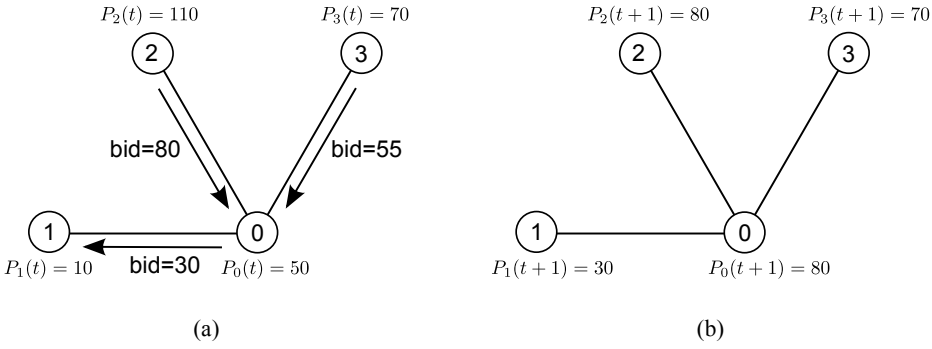


Fig. 2. An Illustration of Protocol **ArbitraryBid**

We are concerned with whether or not the commodity prices eventually converge to the same one even if they are initially distinct. So we define the legitimacy of a configuration as follows.

Definition 1 (legitimate configuration). *A configuration is legitimate if every commodity has the same price.* □

In [16], we developed a protocol in which every agent bids the half of max price difference in his neighborhood. Then, we showed that the protocol achieves price stabilization. Here, we consider a condition such that any protocol satisfying the framework of **ArbitraryBid** achieves price stabilization.

Let $C_t \subseteq V$ be the set of nodes that have updated their prices from time t to $t+1$. Let the highest price be $P^{max}(t) = \max_{i \in C_t} P_i(t)$, and the lowest price be $P^{min}(t) = \min_{i \in C_t} P_i(t)$. Furthermore, let $diff(t) = \max_{i \in C_t} P_i(t) - \min_{i \in C_t} P_i(t)$.

The following lemma proves that the protocol **ArbitraryBid** is free from deadlocks.

Lemma 1. *The protocol* **ArbitraryBid** *is deadlock-free. That is, there exist some nodes in C_t as long as the configuration is illegitimate.*

Proof. Suppose that the configuration is illegitimate at time t. Then, there is a pair of neighboring nodes $i, j \in V$ such that $P_i(t) = \max_{k \in N_j} P_k(t)$ and $P_j(t) = \min_{k \in N_i} P_k(t)$, where $P_i(t) - P_j(t)$ is the max price difference, are satisfied. In this case, agent a_i makes a bid to node j and agent a_j accepts the price. Since $P_j(t)$ is increased at time $t+1$, $j \in C_t$ holds. □

Suppose that agents a_i and a_j make bids to node k. We say that *bids have the same order as values* if $v_i^k \leq v_j^k$ implies $b_i^k \leq b_j^k$ for the commodity of node k. Next, we show that the bids having the same order as values is necessary for price stabilization.

Lemma 2. *If bids do not always have the same order as values, price stabilization is not guaranteed.*

Proof. Suppose that there is an agent a_i who has the maximum value for the commodity in the node k. That is, $v_i^k \geq v_j^k$ holds for any node $j \in N_k$. If he always bids with the lowest price in the neighborhood ($b_i^k < b_j^k$), he loses and thus the price P_i does not change. So the price stabilization is not guaranteed. □

The following theorem further shows that an additional condition leads to the price stabilization.

Theorem 1. *Suppose that bids have the same order as values. If any contract price lies between buyer's price and seller's price, price stabilization occurs.*

Proof. Let $v_i^k(t) = P_i(t) = P^{max}(t)$. Let node k with $\min_{k \in N_i} P_k(t)$ have the minimum price in the neighborhood. Since bids have the same order as values, the bidding price of agent a_i is the highest one in N_k. Thus agent a_i can contract with a_k. It means $P^{max}(t) = P_i(t) > P_i(t+1)$. Since no other agents make bids greater than $P^{max}(t)$, we have $P^{max}(t) > P^{max}(t+1)$. The similar argument holds for $P^{min}(t)$. Thus we have

$$\mathit{diff}\,(t) > \mathit{diff}\,(t+1). \tag{1}$$

This means price stabilization occurs. □

If any contract price does not lie between buyer's price and seller's price, price stabilization is not guaranteed. It is intuitively clear because the inequality (1) may not be satisfied.

4 Best Bidding Price

In this section, we consider how to determine a bidding price by using Bayesian-Nash equilibrium [17,22], which is useful for the game with incomplete information, e.g., a sealed-bid auction. Suppose that each agent's value is uniformly distributed on $[\alpha, \beta]$ (independent and identically distributed). Then, the distribution function is $F(x) = h(x - \alpha)$, where $h = 1/(\beta - \alpha)$. Let Y be the highest of $B-1$ values. Then, Y is the highest order statistic of the values, and thus the distribution function of Y, denoted by $G(x)$, is $G(x) = F(x)^{B-1}$. In addition, let $g(x)$ be the density function of $G(x)$.

Agent a_i's strategy $S(v_i^k)$ against $B-1$ bidders, where B at time t is denoted by $B(t)$, is derived as follows. For simplicity, v_i^k is denoted by v here. Notice that the expression (2), introduced in [17], is described in our appendix. We consider that the Krishna's solution [17] for the case of single-unit demand can be applicable to our model. This is because an agent a_i at node i actually means people in the city i and each of them has a single-unit demand.

$$
\begin{aligned}
S(v) &= \frac{1}{G(v)} \int_{\alpha}^{v} yg(y)dy \qquad (2)\\
&= \frac{1}{\{h(v-\alpha)\}^{B-1}} \int_{\alpha}^{v} yh^{B-1}(B-1)(y-\alpha)^{B-2}dy \\
&= \frac{B-1}{(v-\alpha)^{B-1}} \left\{ \left[\frac{y(y-\alpha)^{B-1}}{B-1} \right]_{\alpha}^{v} - \int_{\alpha}^{v} \frac{(y-\alpha)^{B-1}}{B-1} dy \right\} \\
&= \frac{1}{(v-\alpha)^{B-1}} \left\{ v(v-\alpha)^{B-1} - \left[\frac{(y-\alpha)^{B}}{B} \right]_{\alpha}^{v} \right\} \\
&= v - \frac{1}{B(v-\alpha)^{B-1}} \{(v-\alpha)^{B} - 0\} \\
&= v - \frac{v-\alpha}{B}
\end{aligned}
$$

We would like to examine whether or not above strategy satisfies our condition in Theorem 1.

First, for the orders of bids and values, suppose $v_i^k \leq v_j^k$ holds.

$$
\begin{aligned}
b_j^k - b_i^k &= S(v_j^k) - S(v_i^k) \\
&= (v_j^k - \frac{v_j^k - \alpha}{B}) - (v_i^k - \frac{v_i^k - \alpha}{B}) \\
&= (v_j^k - v_i^k)(1 - \frac{1}{B}) \geq 0.
\end{aligned}
$$

Thus $b_i^k \leq b_j^k$ holds.

Next, for the bidding price,

$$
S(v_i^k) - \alpha = v_i^k - \frac{v_i^k - \alpha}{B} - \alpha = (v_i^k - \alpha)(1 - \frac{1}{B}) > 0.
$$

On the other hand, it is clear that

$$
v_i^k - S(v_i^k) = v_i^k - (v_i^k - \frac{v_i^k - \alpha}{B}) = \frac{v_i^k - \alpha}{B} > 0.
$$

Thus, we have $\alpha < S(v_i^k) < v_i^k$. Now we obtain the following theorem.

Theorem 2. *In our protocol* **ArbitraryBid**, *suppose that each agent a_i confronting $B-1$ bidders repeatedly makes a bid to the lowest-price node $k \in N_i$ by strategy*

$$
S(v_i^k(t)) = v_i^k(t) - \frac{v_i^k(t) - P_k(t)}{B} = P_i(t) - \frac{P_i(t) - P_k(t)}{B}.
$$

Then, price stabilization occurs. □

Actually, the problem in our model is to know the precise number of bidders B. Though the maximum number of bidders to node i is N_i, some of them place bids to other neighboring nodes.

Estimation Methods. We compare the following two methods which enable us to estimate the number of bidders B.

1. A method is to use B in the previous step by assuming that the number of bidders to the neighboring node can be known.
2. Another method is to estimate B by assuming that the value of each agent is uniformly distributed over the same interval.

Method 1 uses the previous information based on the idea that the situation does not suddenly change in time. For example, suppose agent a_i wants to estimate $B(t)$ at node j. Then a_i just substitutes $B(t-1)$ for $B(t)$. Method 2 uses the neighboring information of prices based on the idea that the situation does not suddenly change in location. More precisely, let gap_i be the difference between the maximum price in N_i and the minimum price in N_i. Let g_j be the difference between the maximum price in N_i and the price of node j. Then, agent a_i estimates the number of bidders at node j to be

$$B = int\left(\frac{g_j}{\max(gap_i, 1)} \cdot N_j\right),$$

where "*int*" means the truncation to integer and $gap_i = 1$ is substituted for $gap_i = 0$.

5 Simulation

Here we present some simulation results. Our questions are :

- How different topologies have influence on price stabilization ?
- Which method is more appropriate for the estimation of B, Method 1 or Method 2 ?

Table 1 describes some parameters. We repeated the experiment up to $Tr = 500$ trials, where a trial ends with one equilibrium, and obtained averaged results. A varying parameter is the number of nodes, changed from 100 to 500, and others are fixed.

Table 1. Parameters

Meaning	Symbol	Values
Number of trials	Tr	500
Number of nodes	$\|V\|$	100 — 500
Agent i's initial price	$P_i(0)$	random integer in $[1, 100]$
Agent i's bid	b_i^k	(max price difference)/2

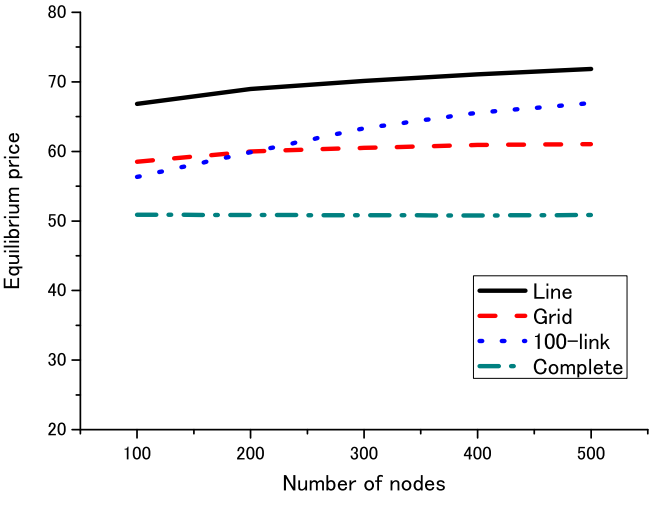


Fig. 3. Equilibrium prices

Effects of Topologies For the first question, we consider four kinds of graphs, a line graph, a grid graph, a random k-link graph, and a complete graph. Then, we compare the equilibrium prices and the number of steps until stabilization on the graphs. The random k-link graph is defined as a circle with randomly selected k edges.

Figure 3 shows the equilibrium prices for four kinds of graphs. In general, large-degree nodes tend to reach convergence because they can easily find a maximal-price node and a minimal-price node in the neighborhood. Then, they rapidly converge to an equilibrium.

The low price nodes can easily win contracts, while the high price nodes cannot easily win contracts because there are not so many agents who want to buy at the nodes. Thus, the equilibrium price tends to become higher and higher if the stabilization time is long.

Figure 4 shows the number of steps for stabilization for the four kinds of graphs. Obviously, a line graph needs much longer time than other graphs to reach an equilibrium. The reason is that the path for spreading the price is limited. It is also reasonable that a grid graph for a large number of nodes needs longer time than complete/random 100-link graphs. Since the grid graph has only local links, it takes somewhat long time to propagate the price movement for a large number of nodes.

Estimation of B. For the second question, we consider two kinds of graphs, a random k-link graph, and a complete graph. To evaluate the methods, we use the following expression

$$U = \sum_{i \in V} \left(\frac{e_i(B) - a_i(B)}{N_i} \right)^2 ,$$

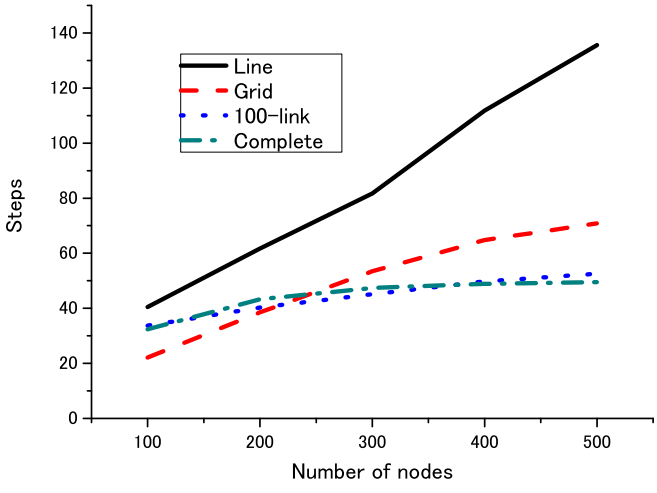


Fig. 4. Number of steps for stabilization

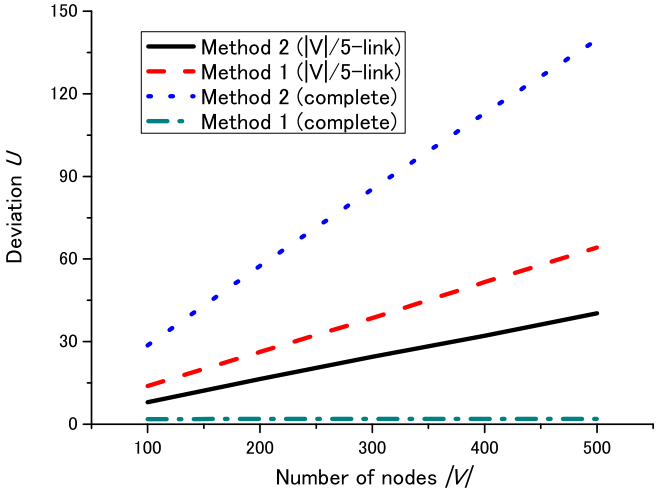


Fig. 5. Estimation of number of bidders

where $e_i(B)$ and $a_i(B)$ are the estimation of B and the actual result of B, respectively. We collect the value of U in each step and take an average of them. Notice that U represents a deviation from the actual number of bidders, and its range is $0 \leq U \leq |V|$.

Figure 5 shows the difference between the methods for two graphs. Method 2 is effective in a random k-link graph, where $k = |V|/5$, rather than a complete graph. On the other hand, method 1 is effective in a complete graph rather than a random k-link graph. Since the error rate is about 5% for the k-link graph, Method 2 could be useful.

6 Conclusion

In this paper we considered a new network model for the price stabilization. The model shows that the self-stabilization has a wide application to various areas. First, we presented a protocol model in which every bidding price is between the max price difference. Then we provided the condition that price stabilization occurs in our model. Next, we investigated how to determine the bidding price by using Bayesian-Nash equilibrium. Finally, we presented simulation results. In summary, our network model reveals the following facts.

- The price stabilization occurs in our model if bids have the same order as values and any contract price lies between buyer's price and seller's price.
- Dense networks are easy to reach an equilibrium because there are multiple paths for spreading the prices. Thus, the equilibrium prices in such networks tend to be low.
- For the best bidding, the Bayesian-Nash solution needs the number of bidders B. Regarding the estimation of B, our method 2 would be useful for a random k-link graph, while our method 1 is useful for a complete graph.

Our goal is to construct a good multiagent protocol which enables us to simulate a realistic social system. Then, we could analyze and estimate several economic phenomena. Our future work includes investigating an asynchronous system and developing other protocols.

Acknowledgments. This work was partially supported by Grant-in-Aid for Scientific Research ((C)20 510139) of the Ministry of Education, Science, Sports, and Culture of Japan.

References

1. Beauquier, J., Herault, T., Schiller, E.: Easy Stabilization with an Agent. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 35–50. Springer, Heidelberg (2001)
2. Blume, L.E., Easley, D., Kleinberg, J., Tardos, E.: Trading Networks with Price-Setting Agents. Games and Economic Behavior 67, 36–50 (2009)
3. Challet, D., Marsili, M., Zhang, Y.-C.: Minority Games — Interacting Agents in Financial Markets. Oxford University Press, New York (2005)
4. Dolev, S., Kat, R.I., Schiller, E.M.: When Consensus Meets Self-stabilization. Journal of Computer and System Sciences 76(8), 884–900 (2010)
5. Dolev, S., Schiller, E.M., Spirakis, P.G., Tsigas, P.: Strategies for Repeated Games with Subsystem Takeovers Implementable by Deterministic and Self-stabilizing Automata. In: Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems (Autonomics 2008), pp. 23–25. ICST, Brussels (2008)
6. Dolev, S., Schiller, E.M., Welch, J.L.: Random Walk for Self-stabilizing Group Communication in Ad Hoc Networks. IEEE Transactions on Mobile Computing 5(7), 893–905 (2006)

7. Dasgupta, A., Ghosh, S., Tixeuil, S.: Selfish Stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 231–243. Springer, Heidelberg (2006)
8. Dolev, S.: Self-stabilization. The MIT Press, Cambridge (2000)
9. Dolev, S., Israeli, A., Moran, S.: Analyzing Expected Time by Scheduler-Luck Games. IEEE Transactions on Software Engineering 21(5), 429–439 (1995)
10. Even-Dar, E., Kearns, M., Suri, S.: A Network Formation Game for Bipartite Exchange Economies. In: Proceedings of the 18th ACM-SIAM Simposium on Discrete Algorithms (SODA 2007), pp. 697–706. ACM, SIAM, New York, Philadelphia (2007)
11. Ghosh, S.: Agents, Distributed Algorithms, and Stabilization. In: Du, D.-Z., Eades, P., Castro, V.E., Lin, X., Sharma, A. (eds.) COCOON 2000. LNCS, vol. 1858, pp. 242–251. Springer, Heidelberg (2000)
12. Gouda, M.G., Acharya, H.B.: Nash Equilibria in Stabilizing Systems. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 311–324. Springer, Heidelberg (2009)
13. Herman, T., Masuzawa, T.: Self-stabilizing Agent Traversal. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 152–166. Springer, Heidelberg (2001)
14. Kakade, S.M., Kearns, M., Ortiz, L.E., Pemantle, R., Suri, S.: Economic Properties of Social Networks. In: Proceedings of the Neural Information Processing Systems (NIPS 2004), pp. 633–640. The MIT Press, Cambridge (2004)
15. Kiniwa, J., Kikuta, K.: Analysis of an Intentional Fault Which Is Undetectable by Local Checks under an Unfair Scheduler. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 443–457. Springer, Heidelberg (2009)
16. Kiniwa, J., Kikuta, K.: A Network Model for Price Stabilization. In: Proceedings of the 3rd International Conference on Agents and Artificial Intelligence (ICAART 2011), pp. 394–397. SciTePress, Portugal (2011)
17. Krishna, V.: Auction Theory. Academic Press, Orlando (2002)
18. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco (1996)
19. Myerson, R.B.: Game Theory: Analysis of Conflict. Harvard University Press, Cambridge (1991)
20. Raberto, M., Cincotti, S., Dose, C., Focardi, S.M., Marchesi, M.: Price Formation in an Artificial Market: Limit Order Book Versus Matching of Supply and Demand. In: Thomas, L., Stefan, R., Eleni, S. (eds.) Nonlinear Dynamics and Heterogeneous Interacting Agents. LNEMS, vol. 550, pp. 305–315. Springer, Heidelberg (2005)
21. Stiglitz, J.E.: Principles of Micro-economics. W.W.Norton & Company, New York (1993)
22. Varian, H.R.: Microeconomic Analysis. W.W.Norton & Company, New York (1992)

Appendix

Suppose there are B bidders for an auction. Agent i wins the auction whenever he submits the highest bid b. Let Y be the highest of $B-1$ values except for agent i. Then, agent i wins whenever $S(Y) < b$ or equivalently, whenever $Y < S^{-1}(b)$. Let $G(v)$ be the distribution of Y and let g be its density. Since we assume that

each agent's value is independently and identically distributed on some interval $[\alpha, \beta]$, the expected payoff of agent i is

$$G(S^{-1}(b))(v-b) + (1 - G(S^{-1}(b))) \cdot 0.$$

We would like to maximize the expected payoff $G(S^{-1}(b))(v-b)$. Differentiating the payoff with respect to v gives

$$\frac{g(S^{-1}(b))}{S'(S^{-1}(b))}(v-b) - G(S^{-1}(b)) = 0. \tag{3}$$

At a symmetric equilibrium, $b = S(v)$ holds. Thus, (3) yields

$$G(v)S'(v) + g(v)S(v) = vg(v)$$

or equivalently,

$$\frac{d}{dv}(G(v)S(v)) = vg(v)$$

Since $S(\alpha) = 0$, we have

$$S(v) = \frac{1}{G(v)} \int_{\alpha}^{v} yg(y)dy.$$

Dynamic Regular Registers in Systems with Churn*

Andreas Klappenecker, Hyunyoung Lee**, and Jennifer L. Welch

Department of Computer Science and Engineering
Texas A&M University
{klappi,hlee,welch}@cse.tamu.edu

Abstract. Distributed systems with churn, or dynamic distributed systems, allow the processes to join and leave the system at will. In this paper, we present a new consistency condition for shared read-write registers which is based on multi-writer regularity, but allows for the likelihood of the register to lose its state with some probability; we call this a *dynamic regular register*. We then describe an algorithm for implementing a dynamic regular register using copies of the register distributed among the processes. When a process joins the system, it attempts to obtain an up-to-date copy of the data from other processes. Copies of the register are updated by broadcasting information. To model the dynamicity of the system with churn, we use a continuous-time birth-death process which is a special case of continuous-time Markov processes. Then, we analyze the probability and the time duration that the dynamic regular register system keeps its state, given the joining rate and the leaving rate of the processes.

Keywords: Dynamic Regular Register, Dynamic Systems, Churn, Register, Multi-Writer Regularity, Markov Process.

1 Introduction

Distributed systems with churn, or dynamic distributed systems, are increasingly deployed as a result of ever-growing wireless communication infrastructure and mobile computing entities, such as robots, vehicles, laptops, and hand-held computers. In a dynamic system, processes can begin and stop participating in the computation as they wish, i.e., the membership of the active processes in the system dynamically changes. In this paper, we specify a new consistency condition for a *dynamic regular register* which is appropriate for sharing data in a dynamic system.

Consider, for example, a mobile application that needs to exchange information among the present participants via shared objects. One example is search-and-rescue type applications, in which location information is shared among the

* This research was supported in part by NSF grant CCF 1018500 and NSF grant 0964696.

** Corresponding author.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 296–310, 2011.

processes to subdivide them into disjoint search areas. Another example is dynamic sensor applications, where the sensor data need to be shared among the participating sensor nodes. Whoever is participating at that moment should be able to update the information. Each process keeps a copy of the shared object. When a process wants to participate in the dynamic mobile application, it first needs to join the system. Upon successfully joining the system, the process is now considered *active* and can participate in the application by invoking operations on the shared objects. Here, an important, challenging problem is to keep the shared data consistent among the dynamically changing members without relying on a centralized server. Furthermore, the system should be able to warn the user of the application when there is a chance that the state of the shared object may be lost.

Contributions. We first specify a new type of shared register system with a new consistency condition, called *dynamic regular register* that allows for the likelihood of the register to lose its state with some probability. Then, we model the churn of the dynamic regular register system using a continuous-time birth-death process which is a special case of continuous-time Markov processes. We then analyze the probability that the system may lose the state of the dynamic regular register at some point in time in the future and how long the system would be able to keep the state of the register, given the churn parameters of the system, such as the joining rate and the leaving rate of the processes. To the best of our knowledge, this is the first work to specify a *dynamic regular register* (or, more generally, *dynamic register*) and to analyze and predict the probabilistic guarantee for maintaining the state of such registers in a system with churn.

Multi-writer Regular Registers and Synchrony Assumption. Because read/write registers provide the fundamental semantics of querying and updating shared data, we focus on a shared read/write register. Motivated by, for example, a data sharing application in mobile ad hoc networks, we assume a multi-writer multi-reader register. Furthermore, by considering such applications running on smart phones or high-end hand-held computers, we assume a synchronous message passing system where the processes have access to the synchronized clocks of the devices and the message transmission bounds are known.

Regularity has been increasingly studied because concurrent operations are dealt with in a more relaxed way than with linearizability (or atomicity). In the case of a single writer, a register is defined to be *regular* if every read returns the value written either by an overlapping write or by the most recent write that precedes the start of the read [13]. Recently, implementation of single-writer/multi-reader regular registers in dynamic systems was studied in [4, 5]. In this paper, we also focus on regularity, however we consider multi-writer/multi-reader regular registers. The multi-writer regularity condition we adopt in this paper is a multi-writer version of Lamport's regularity condition [13] that is tailored to synchronous systems (more details are in Sect. 3); see [20] for a study of multi-writer versions of regularity for asynchronous systems. Here, our assumption of synchronous systems allows us to use the process clock time combined with the

process ID as the timestamp of an operation invocation. Throughout the paper, we will sometimes denote the multi-writer regularity condition we use as "mw-regularity." Also, we will sometimes use the simpler form "dynamic register" or simply "the system" to denote the *dynamic regular register* system.

2 Related Work

Dynamic distributed systems with unbounded number of processes have been studied in numerous papers, for example, [1, 16]. The first theoretical study on a distributed protocol that involves infinitely many processes can be found in [8]. The position paper [3] proposes issues and difficulties in defining a dynamic distributed system considering a simple data aggregation problem. Anceaume et al. [2] propose a framework for defining and comparing the self-organizing properties of dynamic systems. All these works provide rigorous theoretical understanding of dynamic distributed systems.

Most closely related to our work is the implementation of regular registers in dynamic distributed systems in [4, 5]. The similarities of our work to those previous works are the focus on regular registers and the synchrony assumption with reliable broadcast and the timely delivery assumption – in our work, we call it "reliable δ-broadcast" as explained in the next section. The differences are that, in [4, 5], an analysis is given under a discrete-time execution scenario, assuming that there are at least a certain number of active processes at all times. There, two discrete-time functions $\lambda(t)$ and $\mu(t)$ are employed. The function $\lambda(t)$ returns the number of processes that invoke the join operation at time t and $\mu(t)$ the number of processes that leave the system at time t. Then, the main result is to give an upper bound on $\mu(t)$ to guarantee the conditions for regular registers in the system. In our work we specify a new consistency condition for *dynamic regular register* and analyze the probabilistic guarantee and the prediction of the duration that the dynamic regular register system will keep the state given a joining rate and a leaving rate, assuming a more realistic continuous-time execution model.

There have been other shared register implementations for mobile ad hoc networks, e.g. [7, 9, 15, 19], that also rely on a reliable deterministic broadcast primitive, which is similar to our reliable δ-broadcast assumption. The Geo-Quorums algorithm in [7] adapts ideas from RAMBO [9, 15] to implement an atomic register using geographically-based quorums.. The geo-register abstraction in [19] is somewhat analogous to the dynamic membership abstraction of our dynamic regular register, but, in the former, a given geographic region serves as the boundary for process participation, whereas, in the latter, there is no such physical boundary for process participation as long as the process is interested in taking part in the dynamic register system. These papers concentrate on worst-case analyses, while our paper does a probabilistic analysis.

It is not surprising that there is a rich literature that studies the dynamic nature of peer-to-peer (P2P) systems and networks. However, the focus is mostly on the dynamics of the topology of the P2P networks; see for example, [11,

12, 14]. Various churn models in dynamic P2P systems are proposed in [11]. In [12], the authors propose a recovery scheme for dynamic P2P systems so as to remain fully functional (i.e., maintains desirable properties such as low peer degree and low network diameter) in spite of a powerful adversary which continuously adds and removes peers. The scheme is based on constructing an efficient distributed hash table which is resilient to churn. In [14], the resilience of dynamic, k-regular (i.e., each node maintains k neighbors) P2P networks is studied using a simple node-failure model based on user lifetimes, using a continuous-time Markov analysis. In [17], an incentive-based scheme is proposed that encourages nodes to help others in order to improve the weights they need to continue receiving downloads. Such incentive-based strategy can be employed in a future direction of our work in this paper in that when the expected lifetime approaches zero, the incentivizing strategy may delay some processes leaving the system so as to prolong the lifetime of the dynamic register.

3 The Dynamic Regular Register System Model

We consider a dynamic system that allows the processes to join and leave the system at will. After successfully joining, the process becomes *active*, that is, *participates* in the computation. We are interested in sharing information among the participating processes by querying and updating a shared register. The *state* of the shared register consists of its value, initially defined to be $\perp$.

We assume a synchronous message passing system in which each process has access to a synchronized clock (for example, of the hand-held device on which the process is running), and there is a fixed, known upper bound on the message delay. We assume a reliable δ-broadcast where the dissemination of a broadcast message takes at most δ time, i.e., if a message m is *broadcast* by a process p at time t and p does not leave the system by $t + \delta$, m is *delivered* unmodified, exactly once, to all the processes that are in the system throughout the interval $[t, t + \delta]$. We also assume a reliable δ'-point-to-point communication where a message *sent* by a process to a process p at time t is *received* unmodified, exactly once, by p by time $t+\delta'$ if p does not leave the system by $t+\delta'$. We assume that $\delta \geq \delta'$ and both δ and δ' are known to the processes. For simplicity, we assume that local computation takes zero time. Furthermore, we assume that there are no process failures – either benign or malicious.

We specify four operations for the dynamic register system: join, read, write and leave. By invoking the join operation, a process gets the current state of the register and becomes active. A join returns DONE. A read() operation returns the value of the register. A write(v) operation updates the state of the register with the value v and returns ACK. There is no constraint on how many processes can read or write the register, although a process is allowed to have at most one operation pending at a time. We specify the leave operation by process i as the time when process i ceases to be active in the system. Thus, each of the three operations – join, read and write – has its invocation time t and response time t', $t \leq t'$, however, the leave operation simply occurs at a certain point in time.

We define an *execution* on the register in a dynamic system as an infinite sequence of operation invocations and responses on the register by all the processes in the system; the real time of occurrence is associated with each invocation and response.

Ideally, a read should return "the most recent value" written to the register. However, there are a couple of wrinkles that need to be smoothed out in looking more closely at this ideal. First, because a read can overlap with a write and writes can overlap with each other, we need to be more precise about what values we allow to be returned by a read. Given an execution, we define the "legal" values for a read as follows:

Definition 1. *For read r, let w be the write, among all writes that end before r begins, whose invocation time is the latest, breaking ties with process ids. Then the set of* legal *values for r consists of the value written by w and the values written by any writes that overlap r.*

Requiring every read to return a legal value provides a consistency condition that is a multi-writer version of Lamport's regularity condition [13] and is tailored to synchronous systems; see [20] for a study of multi-writer versions of regularity for asynchronous systems.

The second wrinkle is the possible absence of any writes that can provide legal values for a read. Because of the dynamic nature of the system, during one part of the execution there can be many active processes, whereas during another part of the execution there may be no active processes (called a *dormant period*).

The latter (dormant period) is a case when the state of the shared data may be lost, that is, if a newly joined process right after the dormant period invokes a read operation and there are no overlapping writes, it will return the initial value $\perp$ which is not a legal value. However, even after the dormant period, if the newly joined process first invokes a write operation or it invokes a read operation and there is an overlapping write, then it is possible for reads to return legal values.

We define an *era* as a maximal length subsequence of an execution in which the number of active processes is positive. In other words, an era begins at the first time after a dormant period when at least one process becomes active and ends at the first time the number of active processes becomes 0. Thus, an execution may consist of one era or can be a series of more than one eras.

Furthermore, we assume the following conditions on the executions:

[EX1] In every interval between the end of a write and the beginning of another write in which no write is pending, the number of reads that are contained in this interval (as well as the number of reads before the first write) is bounded by a constant κ.

[EX2] Each process executes one read operation right after becoming active.

[EX3] Each process executes at least one write operation.

Now we specify a *dynamic regular register* as a shared multi-writer, multi-reader register in which legal values are guaranteed to be returned with some probability based on the churn of the system (more details are below).

[DR1] Every operation invocation in every execution has a matching response.
[DR2] (Existence of Era) Every execution consists of one or more eras, in which the expected length of the era, as measured in real-time, is positive.
[DR3] (Probabilistic MW-Regularity) Suppose that the execution satisfies [EX1], [EX2] and [EX3]. Then there exists ϵ in (0,1) such that every read returns a legal value with probability at least $1 - \epsilon$. If a read returns a value that is not legal, then that value must be the initial value $\perp$.

We say that a read is "successful" if it returns a legal value, and "unsuccessful" if it returns $\perp$.

Roughly speaking, [DR2] specifies maximal time intervals – called eras – during which there is at least one active process. The condition [DR3] ensures that during an era, a read can return the legal value of the register with probability greater than 0.

4 The Algorithms

As specified in the previous section, there are four basic operations for a dynamic regular register R: join and leave to manage the membership and read and write to share information using the shared register. Upon joining R successfully, the process becomes "active" and can "participate" in R by invoking the register operations read and write and responding to inquiry messages of other joining processes.

The leave event is performed implicitly, for example, if the user simply closes the application that was using R or turns off her hand-held device, the system R may not distinguish the difference between the intentional act of leaving and a clean crash (of a process without a pending operation). Thus, we do not need an algorithm for the leave event.

Below, we describe the algorithms for join, read and write, which are modified versions of those in [5]. The two main differences are: (i) the construction of timestamps to suit the multi-writer register, and (ii) a simplified join operation. Every process p_i in R maintains the following local variables:

1. val_i: the value of the local copy of the register. Initially $\perp$.
2. ts_i: a timestamp that is in the form of a pair (the process clock time : the process ID); this timestamp is accompanied to the value val_i whenever p_i invokes a write operation.
3. $active_i$: a boolean variable indicating whether p_i has successfully joined R (and so is now active) or not. It is set to `true` upon successfully completing the joining operation. Initially `false`.

4. $recvd_i$: a set of received messages as response to p_i's broadcast message INQUIRE. Initially $\emptyset$.
5. $reply_i$: a set of process ID's, to whom p_i needs to send REPLY messages once p_i becomes active. Initially $\emptyset$.

The first two variables (val_i, ts_i) are used to maintain the state of the local copy of the register. The remaining three variables are used only in the join operation.

The Join Operation. When a process wants to actively participate in updating and accessing the state of the register, the process needs to join the system. Joining the system requires every process to get up-to-date state of the register by broadcasting an INQUIRY message. Before broadcasting an INQUIRY message, the process waits δ time so that a possibly concurrent write that is broadcast shortly before the beginning of this join operation can be delivered to all active processes in the system by the reliable δ-broadcast assumption. Then, after broadcasting the INQUIRY message, the process waits 2δ time so that it receives the replies to its INQUIRY from all active processes in the system. While waiting, the process updates the state of its local copy of the register with the value accompanied by the largest timestamp in lexicographic order and becomes "active" by setting the local variable *active*. Once a process becomes active, it can send REPLY messages to the INQUIRYs of other processes, and invoke read or write operations on the register. The pseudocode is given as Algorithm 1.

Algorithm 1. Join Operation and Event Handling for Process p_i

```
1: val_i := ⊥, ts_i := −1, active_i := false, recvd_i := ∅, replyTo_i := ∅
2: wait(δ)
3: broadcast(INQUIRY_i)
4: wait(2δ)
5: active_i := true
6: for all j ∈ replyTo_i do
7:    send (REPLY ⟨val_i, ts_i⟩) to p_j
8: end for
9: return DONE
10:
11: when (REPLY ⟨v, ts⟩) is received:
12: if ts_i < ts then
13:    (val_i, ts_i) := (v, ts)
14: end if
15:
16: when (INQUIRY_j) is delivered:
17: if active_i then
18:    send (REPLY ⟨val_i, ts_i⟩) to p_j
19: else
20:    replyTo_i := replyTo_i ∪ j
21: end if
```

The Read and Write Operations. When a process p_i invokes a read, val_i is returned immediately, thus the time needed for a read operation is zero. When a process p_i invokes a write(v), a new value for ts_i is constructed and (WRITE $\langle v, ts_i \rangle$) message is broadcast, and then it waits δ time so that, by the reliable δ-broadcast assumption, the write message is delivered to other processes. Thus, a write operation takes δ time to return. The pseudocode is given as Algorithm 2.

Algorithm 2. Read and Write Operations for Process p_i

```
1: when read() is invoked: return val_i
2:
3: when write(v) is invoked:
4: ts_i := (clock time : i)
5: val_i := v
6: broadcast(WRITE ⟨v, ts_i⟩)
7: wait(δ)
8: return ACK
9:
10: when (WRITE ⟨v, ts⟩) is delivered:
11: if ts_i < ts then
12:     (val_i, ts_i) := (v, ts)
13: end if
```

Remark. If a process p_i invokes the join at time t, then p_i becomes active at time $t + 3\delta$. Recall that p_i invokes a read r at time $t + 3\delta$. Assume there are no writes that overlap r. Let w be the write with the largest timestamp in lexicographic order among all the writes that return during the time interval $[t, t + 3\delta)$. If r and w belong to the same era, then we claim that r returns the value written by w. Indeed, if w returns in $[t, t + \delta)$, then p_i receives that value through broadcast(INQUIRY$_i$) at line 3 of Algorithm 1. If w returns in $[t + \delta, t + 3\delta)$, then p_i receives the value due to lines 10–13 of Algorithm 2, which proves our claim.

5 The Analysis

In this section, we analyze the behavior of a dynamic register R. We freely use the theory of continuous-time Markov chains in this analysis, see for example [6, 10, 21, 22] for more background.

Let us consider an arbitrary arrival stream of processes p_i for $i \in \{1, 2, 3, \dots\}$. We denote by t_i the arrival time of the process p_i, that is, the time when the process p_i issues a join command. Suppose that s_i denotes the time that the process p_i spent in the system before leaving. In other words, the process p_i is in the system from the arrival time t_i to the departure time $t_i + s_i$.

Let $N_i(t)$ denote the indicator variable for the ith process in the system, that is $N_i(t) = 1$ for all times t in the range $t_i \le t < t_i + s_i$, and $N_i(t) = 0$ otherwise. Even though a process might join and leave the system and later join once again,

for simplicity, we count it each time as a new process. This is why the support of the indicator variable $N_i(t)$ is an interval.

Let $A_i(t)$ denote the indicator variable that process p_i is *active* at time t, that is, $A_i(t) = 1$ if process p_i is active at time t and $A_i(t) = 0$ otherwise. Recall that a join operation takes 3δ time units before the process becomes active. Therefore, $A_i(t) = 1$ if and only if t is contained in the interval $[t_i + 3\delta, t_i + s_i)$.

Let $N(t)$ and $A(t)$ respectively denote the *number of processes* and the *number of active processes* in the system at time t. Then

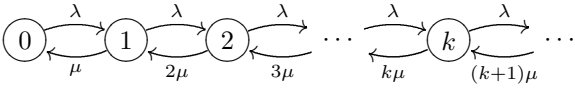
$$N(t) = \sum_{i=1}^{\infty} N_i(t) \qquad \text{and} \qquad A(t) = \sum_{i=1}^{\infty} A_i(t).$$

We model the dynamic register system R with the help of a Markov chain. Let us suppose that the arrival of the processes p_i is governed by a Poisson process with arrival rate λ. In other words, $1/\lambda$ is the mean time between arrivals. More precisely, for a sufficiently small positive real number h, we have

i) Pr[exactly one process arrives in $[t, t+h)$] $= \lambda h + o(h)$,
ii) Pr[no process arrives in $[t, t+h)$] $= 1 - \lambda h + o(h)$,
iii) Pr[more than one process arrive in $[t, t+h)$] $= o(h)$.

Once a process has successfully joined the dynamic register system R, it is assumed that it is active for a non-zero amount of time and then it leaves R. We assume that this time duration of a process being active can be modeled by an exponential distribution with parameter μ. We refer to the parameter μ as the *leaving rate*.

Therefore, the Markov chain modeling the number of active processes in the dynamic register system R can be described by the state transitions.



If we express this in the form of an infinitesimal generator matrix Q, we obtain

$$Q = \begin{pmatrix} -\lambda & \lambda & & & & \\ \mu & -(\lambda+\mu) & \lambda & & & \\ & 2\mu & -(\lambda+2\mu) & \lambda & & \\ & \ddots & \ddots & \ddots & & \\ & & & k\mu & -(\lambda+k\mu) & \lambda \\ & & & & \ddots & \ddots & \ddots \end{pmatrix} \tag{1}$$

The transition matrix $P(t)$ is given by $P(t) = e^{tQ}$. The probability that the system will transition from state i to state j within time t is given by $P_{ij}(t) := (P(t))_{ij}$.

Theorem 1. *Suppose that a dynamic register system has arrival rate λ and leaving rate μ. Then the fraction of the time that a joining process finds the*

system in the dormant period, without active processes, converges in probability to $e^{-\lambda/\mu}$.

Proof. The limiting probabilities of a continuous Markov chain with infinitesimal generator matrix P given by (1) exist, are independent of the initial value m, and are given by

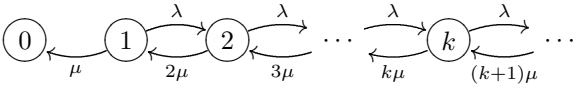
$$\lim_{t\to\infty} P_{mn}(t) = e^{-\lambda/\mu}\left(\frac{\lambda}{\mu}\right)^n \Big/ n!,$$

see for instance [6, page 58]. Let $C_{mn}(t)$ denote the time spent by this Markov process in the state n starting from state m during the interval $(0, t]$. Then the fraction of the time that the system has n active processes is determined by the limiting probability; specifically, we have

$$\lim_{t\to\infty}\left(\left|\frac{C_{mn}(t)}{t} - e^{-\lambda/\mu}\left(\frac{\lambda}{\mu}\right)^n \Big/ n!\right| > \epsilon\right) = 0$$

due to the general result [6, Theorem 4.1.3]. Substituting $n = 0$ in the last equation yields the claim. □

Recall that the *era* of a dynamic register is defined as the first time the number of active processes becomes 0 when starting with $m \geq 1$ active processes. The era of a dynamic register can also be modeled by a continuous-time Markov chain, now with state transition diagram



Thus, the Markov chain modeling one era has an absorbing state 0. If we examine the behavior of this Markov chain only at the transition times, ignoring the waiting time in each state, then we obtain an associated discrete-time Markov chain, called the embedded Markov chain. In this case, the embedded Markov chain has the transition probability matrix (see [21, Section 5.1])

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots \\ d_1 & 0 & u_1 & 0 & \cdots \\ 0 & d_1 & 0 & u_1 & \cdots \\ \vdots & \vdots & & & \end{pmatrix}$$

with $d_n = n\mu/(\lambda + n\mu)$ and $u_n = \lambda/(\lambda + n\mu)$ for all $n \geq 1$.

Theorem 2. *Let R be a dynamic regular register system. Suppose that the arrival of the processes in the system R can be modeled by a Poisson process with arrival rate λ, and the time each process stays active while using R is exponentially distributed with rate μ. Then the mean duration of an era starting with m active processes and ending with 0 active processes is given by*

$$\sum_{n=1}^{\infty} \frac{\lambda^{n-1}}{n!\mu^n} + \sum_{k=1}^{m-1} \frac{k!\mu^k}{\lambda^k} \sum_{j=k+1}^{\infty} \frac{\lambda^{j-1}}{j!\mu^j}$$

In particular, the mean duration of an era is at least $(e^{\lambda/\mu} - 1)/\lambda$.

Proof. Without loss of generality, let us assume that the beginning of an era starts at time $t = 0$, so that there are $A(0) = m$ active processes. Let

$$Z_m = \inf\{t > 0 : A(t) = 0\}$$

denote the time to the end of this era. Our goal is to calculate the mean duration $W_m = \mathrm{E}[Z_m]$ of this era. In other words, W_m is the mean time one has to wait until zero processes are active.

Let us consider the states following the first transition. The mean waiting time in state $m \geq 1$ is $1/(\lambda + m\mu)$. The system will transition from m to $m+1$ active processes with probability $\lambda/(\lambda + m\mu)$, and from m to $m-1$ active processes with probability $m\mu/(\lambda + m\mu)$. Therefore, we obtain the recurrence relation

$$W_m = \frac{1}{\lambda + m\mu} + \frac{\lambda}{\lambda + m\mu} W_{m+1} + \frac{m\mu}{\lambda + m\mu} W_{m-1} \tag{2}$$

for all $m \geq 1$.

Let us denote by $D_m = W_m - W_{m+1}$ the difference in mean time to absorption when starting with a different number of active processes. The differences D_m have the advantage that one can rewrite the recurrence relation (2) in the more pleasing form

$$D_m = \frac{1}{\lambda} + \frac{m\mu}{\lambda} D_{m-1},$$

as one can readily check by substituting the definition of D_m into the latter expression. If we iterate these expressions, then we obtain

$$\begin{aligned}
D_1 &= \frac{1}{\lambda} + \frac{\mu}{\lambda} D_0, \\
D_2 &= \frac{1}{\lambda} + \frac{2\mu}{\lambda} D_1 = \frac{1}{\lambda} + \frac{2\mu}{\lambda^2} + \frac{(2\mu)\mu}{\lambda^2} D_0, \\
D_3 &= \frac{1}{\lambda} + \frac{3\mu}{\lambda^2} + \frac{(3\mu)(2\mu)}{\lambda^3} + \frac{(3\mu)(2\mu)\mu}{\lambda^3} D_0, \\
&\vdots \\
D_m &= \sum_{i=1}^{m} \frac{1}{\lambda} \prod_{j=i+1}^{m} \frac{i\mu}{\lambda} + \left(\prod_{j=1}^{m} \frac{j\mu}{\lambda} \right) D_0.
\end{aligned}$$

The latter expression can be simplified to

$$D_m = \sum_{i=1}^{m} \frac{1}{\lambda} \frac{c_m}{c_i} + c_m D_0, \tag{3}$$

where the coefficient c_i is given by $c_i = i!\mu^i/\lambda^i$ for all i in the range $1 \leq i \leq m$. Since $D_0 = W_0 - W_1 = -W_1$, it follows from equation (3) that

$$\frac{1}{c_m} D_m = \frac{1}{c_m}(W_m - W_{m+1}) = \sum_{i=1}^{m} \frac{1}{\lambda c_i} - W_1.$$

One can show that $\lim_{m\to\infty} \frac{1}{c_m}(W_m - W_{m+1}) = 0$ holds. Hence, letting $m \to \infty$, we get

$$W_1 = \sum_{n=1}^{\infty} \frac{1}{\lambda c_n} = \sum_{n=1}^{\infty} \frac{\lambda^{n-1}}{n!\mu^n} = (e^{\lambda/\mu} - 1)/\lambda \tag{4}$$

By substituting equations (4) and (3) into the equality

$$W_m = W_1 - \sum_{k=1}^{m-1} D_k,$$

we obtain the claim. □

Due to the stochastic nature of a dynamic register, one might worry that a process joining at time t will not be able to read the current register content, because all active processes might have left before the joining process becomes active and reads the register value at time $t + 3\delta$. The next theorem calculates the probability of this event.

Theorem 3. *Suppose that a dynamic register has n active processes at time t. Suppose that the time before an active process leaves the dynamic register is exponentially distributed with parameter μ. Then the probability that the current era ends before time $t + 3\delta$ is given by $(1 - e^{-3\delta\mu})^n$.*

Proof. Suppose that $p_1, \ldots, p_n$ are the active processes in the system at time t. Let $Y_1, \ldots, Y_n$ be the random variables respectively describing the leaving time of process $p_1, \ldots, p_n$. Then the probability that the era ends before time $t + 3\delta$ is given by

$$\begin{aligned}
&\Pr[\max\{Y_1, Y_2, \ldots, Y_n\} < t + 3\delta \mid Y_1 \geq t, \ldots, Y_n \geq t] \\
&\quad = \Pr[Y_1 < t + 3\delta, \ldots, Y_n < t + 3\delta \mid Y_1 \geq t, \ldots, Y_n \geq t] \\
&\quad = \Pr[Y_1 < t + 3\delta \mid Y_1 \geq t]^n = \Pr[Y_1 < 3\delta]^n = (1 - e^{-3\delta\mu})^n.
\end{aligned}$$

Here we have used the definition of a maximum and the fact that exponentially distributed random variables are memoryless. □

Let us illustrate this theorem with a small example.

Example 1. Suppose that the active processes remain in the dynamic register system on average for $1/\mu = 5$ minutes, and let $\delta = 56$ ms. Then

$$\begin{aligned}
&\Pr[\text{era ends before } t + 3\delta | 1 \text{ active process at time } t] < 0.0005599 \\
&\Pr[\text{era ends before } t + 3\delta | 2 \text{ active process at time } t] < 3.2 \times 10^{-7} \\
&\Pr[\text{era ends before } t + 3\delta | 8 \text{ active process at time } t] < 10^{-26}
\end{aligned}$$

Thus, if there are eight or more active processes, then the join operation will be as reliable as current CMOS gates, which have an error rate of 10^{-25}.

We define the "lifetime" of a process to be the time during which the process is active in the system.

Theorem 4. *Suppose a dynamic register system has arrival rate λ and leaving rate μ. Then the probability that an era spans just the lifetime of a single process and there are no further active processes throughout this era is given by $\mu/(\mu+\lambda)$.*

Proof. The arrivals of active processes are governed by a Poisson process with arrival rate λ. Let T_1 (resp., T_2) denote the time the first (resp., second) process in this era becomes active. The interarrival times between processes becoming active are exponentially distributed with mean $1/\lambda$. The duration L_1 during which the first process remains active is exponentially distributed with rate μ. The probability that the first era ends before the second process becomes active is given by

$$\Pr[\text{first era ends before second process becomes active}] = \Pr[L_1 < T_2 - T_1].$$

We have $\Pr[L_1 < T_2 - T_1] = \mu/(\mu + \lambda)$, see [18, Section 5.2.3]. This is precisely the probability that the era spans just a single process. □

After the first write, all subsequent reads in the same era are successful. If the initial read of a process (cf. [EX2]) is unsuccessful, the process does not need further (redundant) reads until the first write operation in the era occurs, since the write is broadcast in Algorithm 2. Thus, we assume, without loss of generality, that a process makes at most one unsuccessful read.

Theorem 5. *Suppose that a dynamic register has arrival rate λ and leaving rate μ, and the three conditions [EX1], [EX2] and [EX3] are satisfied. Then Algorithms 1 and 2 implement the dynamic regular register.*

Proof. If a join operation is invoked at time t, then it returns DONE at time $t + 3\delta$. A read operation returns the value of the register immediately. If a write operation is invoked at time t, then it returns ACK at time $t + \delta$. Therefore, the condition [DR1] is satisfied.

Since $e^{\lambda/\mu} > 1$ for all arrival rates λ and leaving rates μ, the duration of an era is at least $(e^{\lambda/\mu} - 1)/\lambda > 0$ by Theorem 2. Therefore, condition [DR2] holds.

Recall that a read is unsuccessful if it returns the initial value of the register; otherwise, the read is successful. An unsuccessful read can only happen (i) if not preceded by a write in the same era and (ii) overlapping writes did not yet update the local variable of the reading process. After the first successful read operation, all subsequent read operations of the era are successful.

Let us denote by S the random variable describing the time of the beginning of an era. Let T denote the random variable describing the time that elapses until the first write is issued within this era, that is, $S + T$ denotes the time of the first write within this era. Let X denote the number of unsuccessful reads during $(S, S+T]$. The probability of an unsuccessful read is given by ϵ. The reads of the arriving processes are independent, as they only depend on the arrival of the process. Therefore, $\epsilon^\kappa \leq \Pr[X = n]$ for all n in the range $0 \leq n \leq \kappa$.

The probability $\Pr[X = n]$ coincides with the probability that n processes arrive during $(S, S + T]$ conditioned on the fact that not more than κ processes

arrive during this interval. Let us denote by $\{N(t) \mid t \geq 0\}$ the counting process underlying the arrival of active processes. We have

$$\epsilon^\kappa \leq \Pr[X = n] = \Pr[N(S+T) - N(S) = n | N(S) = 1 \text{ and } N(S+T) \leq \kappa]$$

for all n in the range $0 \leq n \leq \kappa$. We omit calculating the latter probability due to lack of space, but simply note that it follows from Theorem 4 that $0 < \Pr[X = 0] < 1$. Therefore,

$$\epsilon^\kappa \leq \max_{0 \leq n \leq \kappa} \Pr[X = n] =: \alpha < 1$$

It follows that the probability ϵ of an unsuccessful read is bounded by $\epsilon < \alpha^{1/k}$, hence $\epsilon < 1$. Therefore, we can conclude that [DR3] holds. □

6 Conclusions

In this paper we defined a new consistency condition for the *dynamic regular register*, gave a set of simple algorithms that implement the dynamic regular register, and analyzed the expected lifetime of the dynamic register until the state of the register may be lost. We also analyzed the probability of the dormant period (with no active process in the system) happening when the join operation takes 3δ time, and illustrated this probability with a small example with a realistic value for $\delta = 56ms$. The example calculation tells us an interesting observation that as long as there are some small number of active processes at any point in time, the dynamic register system will be able to sustain its state with high probability, even when the churn is rather high (average active time of a process being only 5 minutes).

References

1. Aguilera, M.K.: A pleasant stroll through the land of infinitely many creatures. SIGACT News Distributed Computing Column 35, 36–59 (2004)
2. Anceaume, E., Défago, X., Potop-Butucaru, M., Roy, M.: A framework for proving the self-organization of dynamic systems. CoRR abs/1011.2312 (2010)
3. Baldoni, R., Bertier, M., Raynal, M., Tucci-Piergiovanni, S.: Looking for a definition of dynamic distributed systems. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 1–14. Springer, Heidelberg (2007)
4. Baldoni, R., Bonomi, S., Kermarrec, A.M., Raynal, M.: Implementing a register in a dynamic distributed system. In: The 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), pp. 639–647 (2009)
5. Baldoni, R., Bonomi, S., Raynal, M.: Regular register: An implementation in a churn prone environment. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 15–29. Springer, Heidelberg (2010)
6. Bhat, U.N.: An Introduction to Queuing Theory. Birkhäuser, Basel (2008)
7. Dolev, S., Gilbert, S., Lynch, N.A., Shvartsman, A.A., Welch, J.L.: GeoQuorums: Implementing atomic memory in mobile ad hoc networks. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 306–320. Springer, Heidelberg (2003)

8. Gafni, E., Koutsoupias, E.: On uniform protocols. Tech. rep. (1998), http://www.cs.ucla.edu/~eli/eli.html
9. Gilbert, S., Lynch, N.A., Shvartsman, A.A.: Rambo: A robust, reconfigurable atomic memory service for dynamic networks. Distributed Computing 23(4), 225–272 (2010)
10. Kijima, M.: Markov Processes for Stochastic Modeling. Chapman & Hall, Boca Raton (1997)
11. Ko, S.Y., Hoque, I., Gupta, I.: Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In: IEEE Symposium on Reliable Distributed Systems (SRDS 2008), pp. 259–268 (2008)
12. Kuhn, F., Schmid, S., Smit, J., Wattenhofer, R.: A blueprint for constructing peer-to-peer systems robust to dynamic worst-case joins and leaves. In: 14th IEEE International Workshop on Quality of Service (IWQoS 2006), pp. 12–19 (2006)
13. Lamport, L.: On interprocess communication, Part I: Models, Part II: Algorithms. Distributed Computing 1(2), 77–101 (1986)
14. Leonard, D., Yao, Z., Rai, V., Loguinov, D.: On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. IEEE/ACM Trans. Netw. 15, 644–656 (2007)
15. Lynch, N.A., Shvartsman, A.A.: Rambo: A reconfigurable atomic memory service for dynamic networks. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 173–190. Springer, Heidelberg (2002)
16. Merritt, M., Taubenfeld, G.: Computing with infinitely many processes under assumptions on concurrency and participation. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 164–178. Springer, Heidelberg (2000)
17. Neely, M.J., Golubchik, L.: Utility optimization for dynamic peer-to-peer networks with tit-for-tat constraints. In: 30th IEEE International Conference on Computer Communications, IEEE INFOCOM (2011)
18. Ross, S.M.: Introduction to Probability Models, 7th edn. Academic Press, London (2000)
19. Roy, M., Bonnet, F., Querzoni, L., Bonomi, S., Killijian, M.O., Powell, D.: Geo-registers: An abstraction for spatial-based distributed computing. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 534–537. Springer, Heidelberg (2008)
20. Shao, C., Welch, J.L., Pierce, E., Lee, H.: Multiwriter consistency conditions for shared memory registers. SIAM J. on Computing 40, 28–62 (2011)
21. Taylor, H.M., Karlin, S.: An Introduction to Stochastic Modeling, 3rd edn. Academic Press, London (1998)
22. Wolf, R.W.: Stochastic Modeling and the Theory of Queues. Prentice-Hall, Englewood Cliffs (1989)

Space-Efficient Fault-Containment in Dynamic Networks*

Sven Köhler and Volker Turau

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

Abstract. Bounding the impact of transient small-scale faults by self-stabilizing protocols has been pursued with independent objectives: Optimizing the system's reaction upon topological changes (e.g. super-stabilization), and reducing system recovery time from memory corruptions (e.g. fault-containment). Even though transformations adding either super-stabilization or fault-containment to existing protocols exist, none of them preserves the other. This paper makes a first attempt to combine both objectives. We provide a transformation adding fault-containment to silent self-stabilizing protocols while simultaneously preserving the property of self-stabilization and the protocol's behavior in face of topological changes. In particular, the protocol's response to a topology change remains unchanged even if a memory corruption occurs in parallel to the topology change. The presented transformation increases the memory footprint only by a factor of 4 and adds $\mathcal{O}(1)$ bits per edge. All previously known transformations for fault-containing self-stabilization increase the memory footprint by a factor of $2m/n$.

1 Introduction

In the absence of faults, self-stabilizing [4] protocols converge to fault-free configurations in finite time, without any external intervention, and regardless of the initial configuration. Afterwards, they remain fault-free until the next fault. In result, self-stabilizing protocols recover from transient faults of any scale or nature. Transient faults may be memory corruptions but also topology changes. Hence, self-stabilizing protocols are suitable for dynamic systems, provided that topology changes don't occur too frequently. However, in self-stabilizing research, protocols are often only optimized with respect to their worst-case stabilization time, but not with respect to their response to small-scale transient faults. This issue has been addressed by super-stabilization [5] and fault-containment [6].

Super-stabilizing protocols are self-stabilizing, i.e. they tolerate any of transient fault. In addition they provide certain guarantees during convergence after a topology change, provided that the configuration prior to the topology change was legitimate. For example, Blin et al. [1] give a super-stabilizing protocol, that guarantees a loop-free construction of a spanning tree after a topology change. This is especially desirable if the tree

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

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 311–325, 2011.

is used for routing purposes. Likewise, self-stabilization has been enhanced with fault-containment to obtain protocols recovering from memory corruptions of a single node in very short time in addition to recovering from transient faults of larger scale or different nature (e.g. topology changes) [6,9]. Protocols are known, for which the corruption of a single bit of a legitimate configuration can lead to the disruption of the local states of 50% of the system's nodes (examples are given in [6,9]). Due to limited knowledge, nodes in the neighborhood of the corrupted node cannot identify faulty information and allow this information to spread to their state and then to farther away nodes. This process is called contamination. Transformations adding fault-containment to existing silent self-stabilizing protocols [6,9] exist. They maintain backups of each node's local state in the neighborhood. Using these backups, a node can recover the value of its local state prior to the memory corruption. To circumvent contamination, nodes wait until all neighbors have completed their recovery attempt. These transformations store backups on every neighbor of a node. Hence, the average number of backups stored per node is $2m/n$. To save memory, lowering the number of backups is desirable.

Dolev and Herman [5] assume that the distributed system is equipped with a mechanism to detect topology changes. However, we argue that the variables used for this purpose must be regarded as subject to memory corruptions when designing fault-containing protocols. It is an open question, whether it is feasible to reliably detect topology changes in spite of memory corruptions. All known transformations for fault-containment do not solve this problem. They use a specific technique to recover from an insufficient number of backups which breaks in face of topology changes. Instead, they should gracefully handle the case in which backups become unreachable by a topology change.

Our Contribution. This paper describes a transformation to add fault-containment to any silent self-stabilizing protocol. The transformed protocol is still self-stabilizing but also recovers from the corruption of a single node's local state in constant time. The transformation preserves the behaviour of the untransformed protocol with respect to topology changes, no matter whether an additional memory corruption happens in parallel to the topology change. At the same time, the number of backups distributed among the neighbors of each node is limited to a constant. The transformation focuses on topology changes that add or remove a single edge. More severe topology changes are discussed in Section 4. The given transformation shows, that reliably detecting topology changes in spite of memory corruption is feasible. However, the implementation is complex and there are many special cases to take care of.

1.1 Related Work

This section focuses on work related to super-stabilization and resilience against topological faults in general. A comprehensive overview of work related to resilience against memory corruptions can be found in [9].

Dolev and Herman [5] introduce the concept of super-stabilization. Formally, a self-stabilizing protocol is called super-stabilizing with respect to class Λ of topology changes if all configurations following a legitimate configuration satisfy a passage predicate even in case of a single class Λ topology change. The passage predicate usually

specifies some desired safety property. Topology changes include edge removal and edge additions. Node crashes or recoveries are modelled as removal and addition of all adjacent edges. To keep track of topology changes, Dolev and Herman allow the protocols to define so-called interrupt statements. These are executed immediately after a topology change and prior to other regular moves. However, this mechanism is assumed to be intrinsic to the distributed system. It is unclear whether a protocol that uses such interrupt routines can be transformed into a protocol that does not. Super-stabilizing protocols not relying on interrupt routines exist, e.g. the super-stabilizing spanning tree protocol given in [5]. Another example is the protocol for loop-free spanning tree construction by Blin et al. [1].

Herman [7] and Katayama et al. [8] consider super-stabilization (that is maintaining a safety property after a transient fault) in the context of memory corruption of a single node. The given protocols are not designed to be super-stabilizing with respect to topology changes. Since the safety property can easily be violated by a memory corruption, the authors allow the protocols to violate it once. Despite its similarity to fault-containment, the focus seems to be on satisfying the safety property, while the focus of fault-containment is fast repair of corruptions without any additional guarantees.

Datta et al. describe a self-stabilizing protocol for leader-election in dynamic networks [2] offering distinct features in response to transient topological faults: If possible, it re-elects a node that was a leader prior to the topology change. Furthermore, during recovery from a topology change, no node changes its choice of leader more than once. The topology changes can be of arbitrary scale, provided that they happen in a legitimate configuration. However, no topology changes may occur during stabilization. This subject is discussed by Derhab and Badache [3]. Their leader-election protocol is self-stabilizing even if topological changes occur frequently during stabilization.

Pectu and Faltings published a self-stabilization protocol for multiagent combinatorial optimization [11]. They describe two extensions: one to make the protocol fault-containing and one to make it super-stabilizing with respect to topology changes. However, it is unclear whether both extensions can be combined, and whether they remain functional in case a memory corruption happens in parallel with a topology change.

1.2 Model of Computation

A distributed system is described by an undirected graph (V, E) where V is the set of *nodes* and $E \subseteq V \times V$ is the set of *edges* (also called *topology*). Let $n = |V|$, $m = |E|$, and Δ denote the maximal degree of the graph. Pairs of nodes connected by an edge are called neighbors. The set of neighbors of node v is denoted by $N(v)$ and $N[v] := N(v) \cup \{v\}$. Each node $v \in V$ executes a number of protocols. If node v executes protocol X, then the tuple (v, X) is called an *instance* of X. Node v designates a set of variables to protocol X constituting the *local state* of instance (v, X). The local states of all instances on node v constitute the *local state* of node v. The local states of all nodes constitute the *configuration* of the system. Each protocol consists of a set of *rules* of the form $guard \longrightarrow statement$ where $guard$ is a Boolean predicate. Nodes communicate using locally shared memory; guards and statements may reference all variables within $N[v]$, no matter which protocol they belong to. However, statements must only modify those variables of node v which are designated to the instance the statement belongs to.

A rule is called *enabled*, if its guard evaluates to true. An instance (resp. node) is called *enabled*, if any of its rules (resp. instances) is enabled. The execution of statements is controlled by a scheduler, which operates in *steps*. This paper focuses on the *central scheduler*. In the i-th step, it non-deterministically selects an enabled node $s_i \in V$ to make a *move*. No fairness assumptions are made. During its move, the node inspects all instances, one by one, in arbitrary order. If an instance is enabled, the node executes the statements of one enabled rule of that instance. Using [10] the transformation can be shown to work under the distributed scheduler. However, due to space limitations, this is omitted.

An *execution* $e = \langle c_0, c_1, c_2, \ldots \rangle$ is a sequence of configurations, where c_0 is called the *initial configuration* and c_i is the configuration after the i-th step. Topology changes are assumed to happen in between steps. The dynamic topology of the system is described by the sequence $\langle E_0, E_1, E_2, \ldots \rangle$, where each E_i is a set of edges. The set E_{i-1} describes the topology before and during the i-th step. E_0 is also called the *initial topology*. In other words, if the current configuration is c_{i-1}, the current topology is E_{i-1}, and the node s_i makes a move, then this yields c_i. Time is measured in *rounds*. The first round of e and the corresponding topology sequence is finished if every node enabled in c_0 either had the chance to make a move or has been disabled due to a move of a neighboring node or a topology change. All further rounds are derived recursively by applying the definition of the first round to the suffix of e which starts at the last configuration of the first round.

Let $\mathcal{L}$ denote a Boolean predicate that decides whether a configuration is *legitimate* with respect to the corresponding topology. A distributed system is called *self-stabilizing* if, in the absence of topology changes and memory corruptions, any execution reaches a legitimate configuration after a finite number of steps (*convergence*) and once a legitimate configuration is reached all subsequent configurations are also legitimate (*closure*). A *silent* self-stabilization protocol guarantees that all nodes become disabled eventually. Let P be a protocol that is self-stabilizing with respect to the Boolean predicate $\mathcal{L}_P$. Without loss of generality, it is assumed that P uses only a single variable $v.p$ on node v. The transformation adds additional variables to each node's local state. Those variables are called *secondary*. The variable $v.p$ of the original protocol P is called *primary*. By the distinction of primary and secondary variables, any configuration is split up into an ordered pair of a *primary configuration* and a *secondary configuration*. The primary configuration is called *legitimate* with respect to the corresponding topology, if $\mathcal{L}_P$ is satisfied.

A single topology change is defined to be the removal or addition of any number of edges, or any combination thereof. Let Λ be an arbitrary class of topology changes. The tuple (E, c) is called (Λ, k)-*faulty*, if a topology E' and a configuration c' exist, such that c' is legitimate with respect to E' and either E differs from E' in a single topology change of class Λ, c differs from c' in the variables of k nodes, or both. Informally, it is said that c is (Λ, k)-faulty. Note that in following, this paper focuses on the class Λ which describes the removal or addition of a single edge and $k = 1$, i.e. the corruption of the local state of a single node. The transformed protocol is called $(\Lambda, 1)$-*fault-containing* if for each connected component, after $\mathcal{O}(1)$ rounds, starting in a $(\Lambda, 1)$-faulty configuration, the transformed protocol (1) is able to reverse the

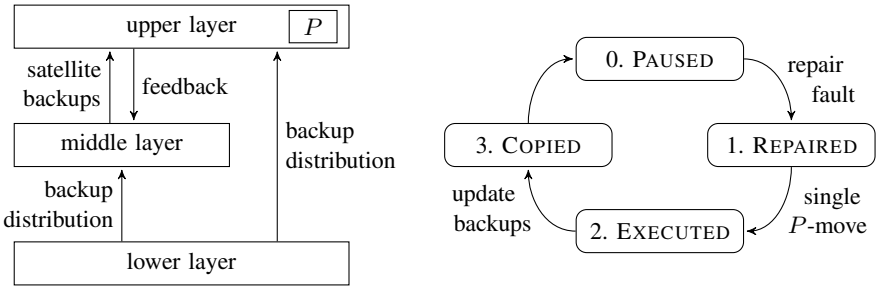


Fig. 1. General architecture (left); States-machine of cells in the upper layer (right)

memory corruption such that it shows the same reaction as the untransformed protocol P in response to the topology change or (2) reaches a legitimate primary configuration if no topology change has happened. The *containment time* denotes the number of rounds needed until (1) or (2) holds. The number of nodes per connected component that change their primary state during the containment time is called *contamination number*. The *fault-gap* is the number of rounds it takes until the transformed protocol reaches a legitimate configuration when started in a $(\Lambda, 1)$-faulty configuration. A notable exception from (1) holds, if connected components of two or fewer nodes existed prior to the topology change. In general, it is impossible to determine the values of the local states prior to a memory corruption within such components. In such cases, the behaviour of the transformed protocol after a topology change matches the behaviour of protocol P in response to the topology change happening in a legitimate configuration different from the one prior to the memory corruption.

The algorithms in this paper evaluate guards of protocol P which is to be transformed. For that matter, we define the Boolean predicate $G_P(v)$, which checks whether P is currently enabled for node v. However, two variants $G_P(v : x, \bot)$ and $G_P(v : x, u : y)$ of this predicate are defined in Section 2.3 to resolve certain special cases. Furthermore, it is assumed that each node executes a fixed and a dynamic set of protocols. These sets determine which instances exist. The dynamic set allows it to *spawn* and *delete* instances at runtime. The function $K(v)$ returns the set of protocols that node v is currently executing, e.g. $X \in K(v)$ if and only if (v, X) has been spawned and was not deleted yet. Memory corruption of the dynamic set can (1) delete any number of instances or (2) spawn additional instances with arbitrary variable values. In addition to that, a memory corruption may perturb all variables of any instance (static or dynamic).

2 The Transformation

Figure 1 shows the three layer architecture of the transformation described in this paper. The upper layer implements fault detection and repair. In case no faults are detected or all faults have been repaired, the upper layer starts executing protocol P, the protocol to be transformed. Per node, the upper layer keeps backups of each node's primary state on three adjacent nodes. The backups allow to detect and repair faults in most cases. However, the main challenges arise from nodes with less than three neighbors

and backups becoming unavailable due to topology changes. For such nodes, the middle layer provides up to 2 additional backups. The lower layer decides to which nodes the backups are distributed. However, the lower layer is not in the focus of this paper and is seen as a black box. It is assumed to be a silent self-stabilizing protocol which provides its output in form of two functions described in Sections 2.1 and 2.2.

Roughly speaking, the transformation attempts to restore the corrupted primary states to the values prior to the memory corruption. If that is successful, the transformed protocol inherits the behaviour of protocol P concerning topology changes. Note that at least two backups are needed to decide whether a single memory corruption has affected the primary state of node v or one of the backups. This cannot be satisfied in any case. The worst case is that v is part of a connected component of only one or two nodes after an edge removal. In the following, such components are called *minor components*. The resolution of this case is described in Section 2.3. If v is not part of a minor component after the edge removal, it can be ensured that two backups are reachable within the 2-hop neighborhood. Certainly, this is satisfied in the trivial case, that 3 backups exist within the 1-hop neighborhood of v before the edge removal. In case that $N(v) = \{a, b\}$, backups are kept on a, b, and on one neighbor of each a and b. If $N(v) = \{a\}$, then backups are kept on node a and two nodes adjacent to a. The backups at 2-hop distance are called *satellite backups*. The middle layer is responsible for creating, updating, and deleting satellite backups. Note, that the topology may not permit to distribute the backups as described (e.g. if $N(v) = \{a\}$ but a has only one neighbor). However, in case of such a topology it suffices to store backups on the available nodes. We ask the reader to verify, that after removing an arbitrary single edge, at least 2 backups are reachable within the 2-hop distance of node v or, if that is not the case, node v has become part of a minor component. It is worth mentioning, that the addition of an edge which connects a former minor component to another component of the system is equally challenging. Within a minor component, one cannot establish the necessary redundancy to reverse a memory corruption. Hence, other means of repair have been implemented. They are described in Section 2.3.

2.1 Upper Layer

To implement the upper layer, a technique similar to [9] is used. The technique is based on the notion of cells. Cell C_v, $v \in V$ is constituted by the so-called *center instance* (v, Q) and all so-called *responding instances* (u, R_v), $u \in N(v)$. Individual responding instances may temporarily not exist. Note that protocol R_v is parameterized. In fact, all instances (v, R_u) with $v \in V$ and $u \in N(v)$ may exist. Protocol Q and R_v implement a query-response mechanism, the so-called *dialog*, between the center and all responding instances. The repair mechanism is implemented in form of two procedures $action_Q$ and $action_{R_v}$ which are called by Q and R_v. Due to the lack of space, the implementation of protocols Q and R_v is omitted. The rules of Q and R_v can be found in [9] and are informally described in this section. Note that in [9], a static topology is assumed and no explicit rules for adjusting to topology changes are given. A naive approach would result in self-stabilizing behaviour, but would not show the fault-containing behaviour desired. This section describes a dialog implementation which is more resilient

against topology changes and provides an enhanced repair mechanism by refining the predicates used by the guards of Q and R_v and the procedures $action_Q$ and $action_{R_v}$.

Two cells C_v and C_u are *neighbors*, if v and u are neighbors. Each cell is a distributed implementation of the state-machine shown in Figure 1. The dialog mechanism allows for implementing each transition of the state-machine as one invocation of $action_{R_v}(u)$ for each responding instance (u, R_v) followed by one invocation of $action_Q(v)$. This way, $action_{R_v}$ can provide information about v's 2-hop neighborhood which is then used by the fault repair implemented in $action_Q$. The given dialog-implementation satisfies the important requirement, that each individual cell starts with the first transition from PAUSED to REPAIRED when starting in a $(\Lambda, 1)$-faulty configuration. Also, the dialog of C_v can be delayed temporarily depending on the state of neighboring cells. This mechanism allows to execute any fault repair strictly prior to any moves of protocol P. Premature execution of P would lead to contamination, since P is assumed to be non-fault-containing.

In the following, the four states of the state-machine are called positions, in order to avoid confusion with the notion of local states. (v, Q) maintains a variable $v.s$ storing the current position within the state-machine. In order to change the current position, (v, Q) first sets the variable $v.q$ to the desired position. The pair $(v.s, v.q)$ is said to be a *query* for a transition from $v.s$ to $v.q$ if $v.q \neq v.s$. Otherwise, the pair said to be a *pause* at position $v.s$. Each responding instance (u, R_v) maintains a so-called *response-variable* $u.r_v$ which is set to the value of $v.q$ in order to acknowledge a query by (v, Q). Along with setting $u.r_v$, $action_{R_v}(u)$ is called. If all responding instances acknowledge, then C_v is called *dialog-acknowledged* and (v, Q) finishes the current transition by setting $v.s := v.q$ and invoking $action_Q(v)$. Note, that queries for a transition not shown in Figure 1 and pauses at a position other than PAUSED are invalid. So if $v.s \neq$ PAUSED, then (v, Q) must ask for another transition straight away. While the dialog progresses, cell C_v remains in a *dialog-consistent* state, that is $(v.s, v.q)$ is either a valid query or a valid pause and each response-variables is equal to either $v.s$ (acknowledgement of the previous query) or $v.q$ (acknowledgement of the current query). A cell is called *dialog-paused*, if $(v.s, v.q)$ is a valid pause and if all response variables are equal to PAUSED as well. Any non-dialog-consistent cell C_v performs a reset to position PAUSED: First, (v, Q) sets both $v.s$ and $v.q$ to PAUSED. Then all responding instances follow this call for a reset and set their response-variable to PAUSED as well. After the reset, the cell is dialog-paused and hence dialog-consistent. To make the dialog of a cell more resilient against topology changes, it is allowed to continue even if not all of v's neighbors execute an instance of R_v. However, cells without any response-instances are said to be *incomplete* and are considered non-dialog-consistent. Hence, they perform a reset as described above. If the current position is $v.s =$ PAUSED, then C_v wait for all missing instances of R_v to spawn. These concepts are formalized by the following predicates:

$$
\begin{aligned}
\mathit{incomplete}(v) &\equiv N(v) \neq \emptyset \wedge R(v) = \emptyset \\
\mathit{validQuery}(v) &\equiv v.q = (v.s + 1) \bmod 4 \\
\mathit{dialogConsistent}(v) &\equiv (v.s = v.q = \text{PAUSED} \ \vee \ \mathit{validQuery}(v)) \wedge \\
&\quad \neg \mathit{incomplete}(v) \ \wedge \ \forall u \in R(v) : u.r_v \in \{v.s, v.q\}
\end{aligned}
$$

$$\begin{aligned} \mathit{dialogPaused}(v) \equiv\ & v.s = v.q = \textsc{Paused} \ \wedge\ \neg \mathit{incomplete}(v) \wedge \\ & \forall u \in R(v) : u.r_v = \textsc{Paused} \\ \mathit{dialogAcknowledged}(v) \equiv\ & \mathit{validQuery}(v) \ \wedge\ \neg \mathit{incomplete}(v) \wedge \\ & (\forall u \in R(v) : u.r_v = v.q) \ \wedge\ (v.s = \textsc{Paused} \Rightarrow R(v) = N(v)) \end{aligned}$$

where $R(v) = \{u \in N(v) \mid R_v \in K(u)\}$ denotes the set of all neighbors of v that execute an instance of R_v. Incomplete cells occur in particular if a topology change adds an edge that connects a single node to an existing graph component or if the only instance of R_v within C_v is deleted by a memory corruption. The dialog reset ensures that cells execute the first transition of state-machine before any other, as desired. We assume, that a node is enabled if it does not execute all expected response instances and that these are spawned as soon as the node is selected by the scheduler, but only if the corresponding cell is at position PAUSED. Similarly, nodes delete all instances (v, R_u) with $u \notin N(v)$. When spawned, the local state of each instance (u, R_v) is initialized such that $u.r_v = \textsc{Paused}$ and $u.c_v = \bot$.

Each responding-instance (u, R_v) maintains a variable $u.c_v$ which either stores a backup of $v.p$ or is equal to $\bot$. The latter indicates that currently no backup is stored. Certainly, memory corruptions can also delete a backup by setting $u.c_v$ to $\bot$. Furthermore, it is assumed that a value of $\bot$ occupies only $\mathcal{O}(1)$ bits in the local state of u. The lower layer provides the desired backups distribution in form of a (deterministic) function $bd(v)$. It returns a set of at most 3 neighbors of v which are the ones to store the backups. Whenever C_v is dialog-paused and (v, Q) detects a mismatch between $bd(v)$ and the current backup distribution, a new cycle of the state-machine is started by setting $v.q := \textsc{Repaired}$. During the subsequent transition from EXECUTED to COPIED, the given implementation of action_Q and action_{R_v} updates the variable $v.cptr$ with the current value of $bd(v)$ and all responding-instances delete or update their backups, depending on whether $v.cptr$ points at them. Note that for notational convenience, any reference to a non existing variable $u.c_v$ (i.e. $R_v \notin K(u)$) is assumed to yield $\bot$. Furthermore, a cycle is started, if v is enabled with respect to P. The following start-condition formalizes the above:

$$\begin{aligned} \mathit{startCond}_Q(v) \equiv\ & G_P(v) \ \vee\ v.cptr \neq bd(v) \vee \\ & (\exists u \in N(v) : u \in v.cptr \wedge u.c_v \neq v.p) \vee \\ & (\exists u \in N(v) : u \notin v.cptr \wedge u.c_v \neq \bot) \end{aligned}$$

As mentioned earlier, the dialog of cell C_u is delayed under certain conditions in order to keep node u from executing P prematurely. A responding instance (v, R_u) simply delays its acknowledgement if $u.q = \textsc{Executed}$ and as long as an inconsistency in C_v is detected. For the purpose of the latter, we define a predicate $\mathit{repaired}(v)$. If this predicate is satisfied, then C_v will not attempt any further repair of $v.p$. Otherwise, a repair of $v.p$ by C_v is still pending. A backup with a value equal to $v.p$ is called *confirmation*. Clearly, $\mathit{repaired}(v)$ should be true if C_v is *copy-consistent* which means that at least one backup exists and all existing backups are confirmations. However, at positions REPAIRED and EXECUTED, C_v is expected to be non-copy-consistent. In this

Procedure $action_Q(v)$

Nodes: v is the current node

if $v.q =$ REPAIRED **then**
- $v.o := oldNID(v)$
- **if** $|N(v)| = 1$ **and** $\exists u \in R(v) : u.d_v =$ SINGLE **then** $nodePairReset_Q(v, u)$
- **else if** $\exists u \in R(v) : u.c_v = v.p \vee u.d_v =$ KEEP **then** keep value of $v.p$
- **else if** $\exists u, w \in R(v) : u \neq w \wedge u.c_v = w.c_v \wedge u.c_v \neq \perp$ **then** $v.p := u.c_v$
- **else if** $\exists u \in R(v) : u.c_v \neq \perp \wedge u.d_v =$ UPDATE **then** $v.p := u.c_v$
- **else if** $|N(v)| \geq 2$ **and** $\exists u \in R(v) : u.id = v.o \wedge$ $u.d_v \in \{$SINGLE, MINOR$\}$ **then** $nodePairReset_Q(v, u)$
- **else if** $|N(v)| = 1$ **and** $\exists u \in R(v) : u.d_v =$ MINOR **then** $nodePairReset_Q(v, u)$
- **else** compute new value of $v.p$ such that $\neg G_P(v : v.p, \perp)$

if $v.q =$ EXECUTED **then**
- $v.cptr := bd(v)$
- **if** $|N(v)| = 1$ **and** $\exists u \in N(v) : u.d_v =$ SINGLE **then** keep value of $v.p$
- **else if** $G_P(v)$ **then** execute a move of P

Procedure $action_{R_v}(u)$

Nodes: u is the current node, v is the center node

if $v.q =$ REPAIRED **then**
- **if** $|N(u)| = 1$ **then** $u.d_v :=$ SINGLE
- **else if** $\exists w \in N(u) : w \neq u \wedge w.sb_{u,v} = v.p$ **then** $u.d_v :=$ KEEP
- **else if** $u.c_v \neq \perp$ **and** $\exists w \in N(u) : w \neq u \wedge w.sb_{u,v} = u.c_v$ **then** $u.d_v :=$ UPDATE
- **else if** $noBackups(u)$ **and** $repaired(u)$ **and** $v.id = u.o$ **then** $u.d_v :=$ MINOR
- **else if** $(\neg noBackups(u)$ **or** $\neg repaired(u))$ **and** $v.id = oldNID(u)$ **then** $u.d_v :=$ MINOR
- **else** $u.d_v :=$ NONE

if $v.q =$ COPIED **then**
- **if** $u \in v.cptr$ **then** create or update variable $u.c_v := v.p$
- **else** delete variable $u.c_v$

Procedure $nodePairReset_Q(v, u)$

Nodes: v is the current node, u is a neighbor of v

if $enabled(v : v.p, u : u.p)$ **then**
- **if** $\exists x : \neg enabled(v : x, u : u.p)$ **then** $v.p := x$
- **else if** $\nexists y : \neg enabled(v : v.p, u : y)$ **then**
  - compute new value of $v.p$ such that $\exists y : \neg enabled(v : v.p, u : y)$

case, $repaired(v)$ indicates whether C_v will be copy-consistent once it has completed the transition to COPIED.

$$\begin{aligned} copyConsistent(v) \equiv\ & (\forall u \in R(v) : u.c_v = \bot \vee u.c_v = v.p) \wedge \\ & (N(v) \neq \emptyset \Rightarrow \exists u \in R(v) : u.c_v = v.p) \\ repaired(v) \equiv\ & copyConsistent(v) \ \vee \ (dialogConsistent(v) \wedge \\ & (v.s = \text{REPAIRED} \ \vee \ (v.s = \text{EXECUTED} \wedge \\ & \qquad (N(v) \neq \emptyset \Rightarrow v.cptr \cap R(v) \neq \emptyset))) \wedge \\ & (\forall u \in R(v) : (u \in v.cptr \wedge u.r_v = \text{COPIED}) \Rightarrow u.c_v = v.p) \wedge \\ & (\forall u \in R(v) : (u \notin v.cptr \wedge u.r_v = \text{COPIED}) \Rightarrow u.c_v = \bot)) \end{aligned}$$

Like in our previous work, $repaired(v)$ is designed to be a stable predicate, i.e. once it is true it remains true under the execution of the transformed protocol. For this to be the case, $bd(v)$ must return at least one neighbor of v at all times. If necessary, $bd(v)$ may (deterministically) choose an arbitrary neighbor of v.

Recall how the repair-mechanism is implemented by the transition from PAUSED to REPAIRED: first, $action_{R_v}(u)$ is invoked for all $u \in N(v)$. Subsequently, $action_Q(v)$ is invoked. Clearly, the value of $v.p$ must not be changed if a confirmation is found. Confirmations within the 1-hop neighborhood are detected by $action_Q$ itself. Confirming satellite backups are detected by $action_{R_v}(u)$ which sets the so-called *decision-variable* $u.d_v$ to KEEP. If sufficiently many backups are reachable and $v.p$ has been corrupted, then two backups with an equal value can be found. Again, such a pair of backups is either detected by $action_Q$ itself or, if it includes a satellite backup, it is detected by $action_{R_v}(u)$ which sets $u.d_v$ to UPDATE. In these cases, $v.p$ is updated using the backups. It is worth mentioning, that a confirmation might actually be a *forged backup*, i.e. a backup which did not exist before the memory corruption. Forged confirmations actually have a beneficial effect and there is no reason to detect them.

If none of that succeeds, then it can be concluded that either (1) the topology change was an edge removal and node v is now part of a minor component or (2) the topology change added an edge and node v was part of a minor component prior to the topology change. This is discussed in Section 2.3.

2.2 Middle Layer

The middle layer is responsible for maintaining the satellite backups. We present an implementation based on the cell technique which has already been used in the upper layer. To avoid name clashes with the upper layer, names and variables are annotated by a (M) superscript. Like the cells of the upper layer, each cell $C_v^{(M)}$, $v \in V$ consists of the instances $(v, Q^{(M)})$ and $(u, R_v^{(M)})$ which implement a query-response dialog. However, the corresponding state-machine is much simpler and consists of two transitions between two states. Also, the corresponding procedures $action_Q^{(M)}$ and $action_{R_v}^{(M)}$ differ. Recall that the middle layer must not delete, create, or update any satellite backups as long as a repair attempt by the upper layer is still pending. Again, the technique of delaying acknowledgements is used to achieve this.

Each node $u \in V$ maintains a variable $u.sc$. It is expected to hold $u.sc = 0$ if $|N(u)| \geq 3$, $u.sc = 1$ if $|N(u)| = 2$, and $u.sc = 2$ if $|N(u)| = 1$. It denotes, how many satellite backups u expects to be reachable via each of its neighbors. It is assumed that u is enabled as long as $u.sc$ does not have the expected value and that u adjusts the value as soon as it is selected by the scheduler. At the end of the first transition of the state-machine, the center-instance of $C_v^{(M)}$ updates the variable $v.sbptr$. For each $u \in N(v)$ with a non-zero $u.sc$, $v.sbptr$ contains the pointers to those neighbors of v which are to store the backups of u's primary state. The lower layer provides the desired distribution of the satellite backups in form of the function $sbd(v, u)$ which returns a set of neighbors of v, excluding u. The function may access $u.sc$ to determine how many satellite backups are to be created. During the second transition of the state-machine, the responding-instances then create, update, or delete the satellite backups. The satellite backup of $u.p$ held by $(w, R_v^{(M)})$ is stored in in the variable $w.sb_{v,u}$. If $(w, R_v^{(M)})$ does not hold a satellite backup of $u.p$, then it is said that $w.sb_{v,u} = \perp$. Note that cell $C_v^{(M)}$ does not manage the satellite backups of node v's primary state. Instead, it maintains the satellite backups for all $u \in N(v)$.

A responding instance $(w, R_v^{(M)})$ delays its acknowledgements as long as *repaired* (w) is false. When starting in a $(\Lambda, 1)$-faulty configuration, this prevents the middle layer from prematurely modifying any satellite backups. Cell $C_v^{(M)}$ starts a new cycle of the state-machine if for some neighbor $u \in N(v)$ a mismatch between $u.sc$, $sbd(v, u)$ and the current distribution of the satellite backups is detected or simply if a satellite backup is out of date.

2.3 Detecting New Edges Near Minor Components

During the design of this transformation, it turned out (probably counter intuitional) that the collapse of an edge is rather easy to compensate compared to the addition of an edge. The core problem arises from nodes that have been part of a minor component prior to the addition of an edge. For other nodes, at least two backups are reachable and can be used to determine the original values of the primary states prior to a memory corruption. In particular, if a node has been part of a minor component, then the memory corruption may delete the only backup in existence, or even worse it may add a forged backup within the 2-hop neighborhood of the node.

In case that no confirming backup is found, the implementation tries to detect the edge added by the topology change. The implemented detection is successful if the minor component consist of two nodes. The result is reported to both nodes and they reset their local states such that they would be disabled with respect to P if the new edge did not exist. The detection of minor components consisting of a single node prior to the topology change is implemented as a fallback solution: if all other tests fail, then it is assumed that the node had no neighbors prior to the topology change. The node performs a reset of its primary state, such that the node would be disabled with respect to P if there were no adjacent edges. For that purpose we define predicate $G_P(v : x, \perp)$, which evaluates $G_P(v)$ in a virtual topology and configuration with no edges adjacent to v and in which the primary state of v is equal to x. Similarly, predicate $G_P(v : x, u : y)$ evaluates $G_P(v)$ in a virtual topology and configuration in which u is the only neighbor of v and in which the primary states of v and u are equal to x and y respectively.

Let u and v denote two nodes that formed a minor component before the topology change. Assume that the topology change has added new edges adjacent to v. The case that nodes u and v were not affected by the memory corruption can be ignored. In this case, $u.c_v$ is a confirmation of $v.p$ and $v.c_u$ is a confirmation of $u.p$ and so neither v nor u change their primary state during repair. So assume that either u or v was affected by a memory corruption. Then v uses the following logic to identify node u among its neighbors: If v sees exactly one backup of its own primary state within the 1-hop neighborhood, then it must be $u.c_v$. It doesn't matter whether the backup was corrupted or not. Its mere existence identifies node u. If v sees no backup within its 1-hop neighborhood, then the memory corruption must have affected node u, but not the local state of v. In this case, node v stores a backup of exactly one of its neighbors, namely node u.

A more detailed discussion is needed, to show that the tests performed by node v identify the former minor component as long as required, that is until it holds for both cells v and u, that the cell is repaired or there is a confirmation of its primary state. Recall that responding nodes may delay their acknowledgments during the transition REPAIRED $\rightarrow$ EXECUTED. Due to this synchronization mechanism between neighboring cells, cell v cannot update its backups unless cell u is repaired. Hence, the parts of the tests that are based on backups within cell v are guaranteed to work as long as needed. The second part of the detection is partly based on (the non-existence of) backups of cells neighboring to v. Let (v, w) be an edge added by the topology change. As soon as cell v is repaired, cell w can update the variable $v.c_w$. The detection ceases to work: Node u is not the only node anymore, for which v stores a backup. Even worse, cell w can start a new cycle, during which w can execute a move of P. After this, the situation is completely symmetrical from the point of view of node v. Hence, before cell v becomes repaired, $action_Q(v)$ stores the result of the detection in the variable $v.o$ for later use. Procedure $action_{R_u}(v)$ uses the memorized value when appropriate.

The following function implements the detection as described above and returns the Id of the neighbor that presumably was part of the minor component prior to the topology change. The quantifier $\exists!$ denotes "there exists exactly one". In case the detection is unsuccessful, the value $\perp$ is returned.

$$noBackups(v) \equiv \forall u \in N(v) : u.c_v = \perp$$

$$oldNID(v) := \begin{cases} u.id \text{ if } \exists! u \in N(v) : u.c_v \neq \perp \\ w.id \text{ if } noBackups(v) \ \wedge \ \exists! w \in N(v) : v.c_w \neq \perp \\ \perp \quad \text{otherwise} \end{cases}$$

Once a former minor component of two nodes has been detected, both nodes try to perform a reset. They do that by calling procedure $nodePairReset_Q$. It first tries to determine which of the two nodes needs to change its primary state to disable protocol P on both nodes. If only one node's memory was corrupted, then it is always sufficient that only one node changes its primary state. $nodePairReset_Q$ correctly detects that, using the following predicate:

$$enabled(v : x, u : y) \equiv G_P(v : x, u : y) \vee G_P(u : y, v : x)$$

However, the same procedure is also used, to speed up stabilization of a connected component that contains only two nodes. In this case the node which calls $nodePairReset_Q$ first, detects whether both primary states need to be resetted. If that is the case, then it adjusts its own primary state such that a value exists, with which its neighbor (that calls $nodePairReset_Q$ second) can disable both nodes with respect to P. Otherwise, only one node adjusts its primary state. Note that in order to work under the distributed scheduler, $nodePairReset_Q$ needs to be modified such that it performs symmetry breaking based on node identifiers.

3 Analysis

This section gives an overview over the space and time complexity of the transformation. Since the exact memory consumption per node depends on the backup distribution computed by the lower layer, we give an upper bound on the average space consumed per node. Consider that according to the model defined Section 1.2, the three layers run in parallel in the sense that one round covers one round of each individual layer. A configuration is legitimate if all three layers have terminated. In particular, the termination of the upper layer guarantees that P has terminated as well and the primary configuration is legitimate.

Let T_L^{stab} (T_P^{stab} resp.) denote the stabilization-time of the lower layer (protocol P resp.). Furthermore, let T_L^{fgap} denote the fault-gap of the lower layer for a $(\Lambda, 1)$-faulty initial configuration. Similarly, T_P^{fgap} denotes the fault-gap of protocol P for a $(\Lambda, 0)$-faulty initial configurations (in [5] it is called super-stabilization time). The following Theorem summarizes the main result of this paper:

Theorem 1. *The transformed protocol is self-stabilizing with a stabilization time of* $\mathcal{O}(1) + \max\{9T_P^{stab}, T_L^{stab}\}$ *rounds. Furthermore, it is* $(\Lambda, 1)$*-fault-containing with a fault-gap of* $\mathcal{O}(1) + \max\{9T_P^{fgap}, T_L^{fgap}\}$*, containment-time* $\mathcal{O}(1)$*, and contamination-number* 2*. The average size of the space required per node is* $\mathcal{O}(m/n + S_P + S_L)$*, where* $S_P + S_L$ *denotes the average size of the space required per node by the lower layer and protocol* P*. Implementations of a lower layer exist, such that both* T_L^{stab} *and* T_L^{fgap} *are constant.*

Furthermore, the transformation provides $(\emptyset, 1)$-fault-containment in the sense of [6,9], i.e. a memory corruption but no topology change occurs. The fault-gap is then lowered to $\mathcal{O}(1) + \hat{T}_L^{fgap}$ where $\hat{T}_L^{fgap}$ denotes the fault-gap of the lower layer for $(\emptyset, 1)$-faulty initial configurations. Simple implementations of the lower layer which randomly select neighbors are feasible and provide constant T_L^{stab}, T_L^{fgap}, and $\hat{T}_L^{fgap}$.

Due to the lack of space, only a short sketch of the proofs is given. First, consider the fault-gap and stabilization time: In parallel with the stabilization of the lower layer, the cells of the upper layer execute rounds of protocol P. Hence, the lower layer and the primary states stabilize in parallel. The redesign of the upper layer preserved all major properties of our previous design. In particular, all cells of the upper layer become dialog-consistent and repaired after a constant number of rounds. Every 9 rounds, the upper layer has finished one round of P. After the lower layer has terminated, the cells

of the upper layer may have to move the backups to their final destinations. When both layers have terminated, the middle layer can finalize the distribution of the satellite backups. These final adjustments take a constant number of rounds.

To show that the composition of the three layers terminates within a finite number of steps, we first show that the upper layer terminates in a finite number of steps under the assumption that the variables of the lower layer do not change. Similarly, we are able to show that the middle layer terminates in a finite number of steps under the assumption that the variables of neither the lower nor the upper layer change. In conclusion, the composition of all three layers terminates after a finite number of steps under the assumption that the lower layer terminates after a finite number of steps. The proof that the upper layer terminates follows the same strategy than the proof given in [9]. The important stepping stones are that $dialogConsistent(v)$ and $repaired(v)$ are stable predicates, and that each cell becomes dialog-consistent and repaired within a finite number of moves of the cell. The proof concludes with its main argument that the number of modifications of a primary state (either by repair or by a move of P) and hence the number of cycles per cell is bounded by a function of n and the worst-case stabilization time (in steps) of protocol P. Similar arguments hold for the middle layer since it reuses the cell technique.

The variables of protocols R_v and $R_v^{(M)}$, with the exception of backups, consume a few bits per edge. Per node, at most three copy-variables and two satellite backups are created. The variables of protocols Q and $Q^{(M)}$ consume a constant number of bits, with the exception of the set of pointers stored by $Q^{(M)}$. Per node, there exist at most two entries in this set. Thus, the average space requirement per node is $\mathcal{O}(m/n + S_P + S_L)$.

4 Final Remarks

Once a legitimate configuration has been reached, the transformed protocol recovers from a memory corruption within a constant number of rounds, even if a single edge has been added or removed simultaneously. However, we believe that this also holds for the crash and the recovery of a node. When a node crashes, then for each of its neighbors sufficiently many backups remain reachable or the neighbor is part of a minor component. Also the addition of multiple edges adjacent to the same node was taken into account during the design of the transformation. In particular the detection of former minor components remains functional. Furthermore, the transformation can handle the removal of more than one edge, if more backups per node are created. Backups on all nodes within 2-hop distance of each node are sufficient to handle the removal of any number of edges. However, topology changes that add and remove edges at the same time remain problematic in some cases.

The given transformation is also resilient against certain topology changes that happen during stabilization. The transformed protocol then shows the same behaviour as the original protocol P. For edge additions, this is always the case. For edge removals it must hold that all affected cells are repaired prior to the topology change and that those cells are still repaired after the topology change. The likelihood that this condition is met increases, if the lower layer selects multiple responding-nodes per cell for backup

storage during stabilization. It takes cells at most one full cycle, that is a constant number of rounds, to adapt to the new topology. Then the transformed protocol is ready to handle another topology change.

We are not aware of methods to further reduce the space overhead without increasing the time-complexity of the transformation. We conjecture that three backups do not suffice to guarantee the bounds described in Theorem 1. However, it seems feasible to reduce the average $\mathcal{O}(m/n)$ bits per node for storing the dialog-variables to $\mathcal{O}(\log \Delta)$ bits while increasing the time-complexity of the transformation by a factor of $\mathcal{O}(\Delta)$.

Advanced implementations of the lower layer are considered an open problem. While the lower layer can be used to compute backup distributions that meet specific design goals of a distributed system, a general goal of the lower layer is to avoid high concentrations of backups on individual nodes, i.e. the lower layer should produce uniform backup distributions. That is considered to be future work.

References

1. Blin, L., Potop-Butucaru, M., Rovedakis, S., Tixeuil, S.: Loop-free super-stabilizing spanning tree construction. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 50–64. Springer, Heidelberg (2010)
2. Datta, A.K., Larmore, L., Piniganti, H.: Self-stabilizing leader election in dynamic networks. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 35–49. Springer, Heidelberg (2010)
3. Derhab, A., Badache, N.: A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. IEEE Trans. on Parallel and Distributed Systems 19, 926–939 (2008)
4. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
5. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago Journal of Theoretical Computer Science 1997(4) (1997)
6. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing 20(1), 53–73 (2007)
7. Herman, T.: Superstabilizing mutual exclusion. Distributed Computing 13, 1–17 (2000)
8. Katayama, Y., Ueda, E., Fujiwara, H., Masuzawa, T.: A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. Journal of Parallel and Distributed Computing 62(5), 865–884 (2002)
9. 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)
10. Köhler, S., Turau, V.: A new technique for proving self-stabilization under the distributed scheduler. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 65–79. Springer, Heidelberg (2010)
11. Petcu, A., Faltings, B.: Superstabilizing, fault-containing distributed combinatorial optimization. In: Proceedings of the 20th National Conference on Artificial Intelligence, vol. 1, pp. 449–454. AAAI Press, Menlo Park (2005)

The OCRC Fuel Cell Lab Safety System: A Self-Stabilizing Safety-Critical System

William Leal[1], Micah McCreery[2], and Daniel Faria[3]

[1] The Ohio State University
[2] Battelle
[3] Accenture

Abstract. We describe the practical application of self-stabilization to a safety-critical system. The Ohio Coal Research Center (OCRC) at Ohio University has a fuel-cell laboratory that uses explosive and poisonous gases. The lab is located in and uses the ventilation system of a large campus building that houses offices, other labs, and classrooms. The OCRC fuel cell lab safety system seeks to protect lab and other building personnel in the event of a gas leak. We present the system and the use of self-stabilization to ensure that, in the presence of actual or potential hazards, the lab converges to as safe a state as possible. It is responds to environmental conditions such as gas leaks and is tolerant to faults that affect the system's sensors and actuators.

1 Introduction

The United States is rich in coal, but since much of it is high in sulphur, it is difficult to generate electricity without using expensive techniques to trap the sulphur emissions. The Ohio Coal Research Center (OCRC) at Ohio University has undertaken a project to use gasified high-sulphur coal for fuel cells to generate electricity without combustion while effectively scrubbing the sulphur from the exhaust stream. These are 150–200% more efficient than current methods of converting coal to electricity, promising improved efficiency with lower grade coal. This contrasts with existing fuel cells that use gasified coal but require that the sulphur be removed before use.

To work on the problem, OCRC has established a fuel cell research laboratory on the campus of Ohio University. Fuel cells are fabricated and used in the lab for experimental testing. A fuel cell experiment, which can run for several hundred hours, is provided with a custom mixture of gases that simulate a particular kind of gasified coal. While some of the gases are benign, others, including hydrogen, hydrogen sulfide and carbon monoxide, are explosive and/or toxic. Protecting people and property from these hazards in the event of leak or equipment failure is critical.

Given that venting to the outside were practical, it would suffice to provide a sufficient exhaust airflow along with gas level monitors. However, a decision by the University administration dictated that exhaust from the lab be vented into

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 326–340, 2011.

the building exhaust system, which also serves classrooms, faculty offices, and departmental and college administration offices. Hence hazardous gases from the lab could enter the building exhaust. As a demonstration, we simulated a failure of the building exhaust and observed positive airflow via the exhaust vents into areas outside the lab, showing that that hazardous gases, if present, could be pushed into other building spaces.

In response, we developed a self-stabilizing safety monitoring and control system for the lab. We developed fault models for sensors, actuators and the network, and used these to analyze the effect of faults on safety. The system has been deployed for over six years, with a generally positive experience. Recent results of experimental fuel cell work conducted in the lab are given in [7,1]. In [5], we described the safety system for professionals in the coal industry. A study of the safety system is in [2] and an expanded version of this paper is in [4].

Our approach is that of safe stabilization [3]. After the environment has stopped changing, we want to stabilize in such a way that in any program pre-/post-state pair, the post-state is not less safe than the pre-state. However, this is not always possible, as we discuss in Section 4.

In the rest of this paper, we describe the instrumented lab and discuss the use of self-stabilization to manage it.

2 The OCRC Fuel Cell Lab

A schematic of the lab is shown in Fig. 1. See the Appendix for photos. This gives the physical placement of the various cabinets, ventilation systems, and alarms. Sensors for airflow are located near the fans, sensors for gas are located in the gas and fuel cell test stand cabinets and in the ceiling.

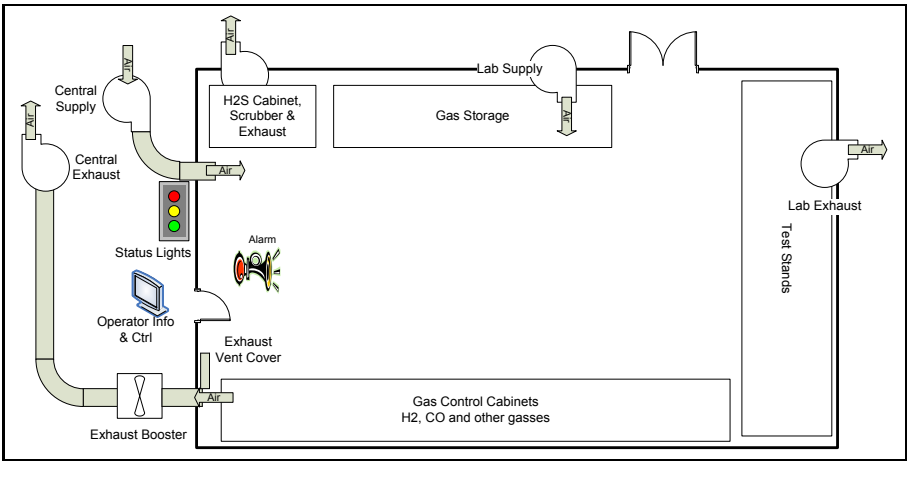


Fig. 1. Lab Layout

There are several ventilation systems present. On the left are central supply and exhaust. Under normal conditions, these provide the main ventilation for the lab. Because of the extreme toxicity of hydrogen sulfide, a special scrubber and exhaust system located in the hydrogen sulfide cabinet runs continuously and vents outside. In the event of central system ventilation failure, local lab supply and exhaust fans can be enabled. These are intended for temporary backup use only and hence are enabled only when necessary.

The interface to the central exhaust includes a booster fan to push lab air into the exhaust, along with a vent gate. If the central exhaust fails, the vent gate can be closed to keep lab air from being pushed into other parts of the building.

There are visual and audible status indicators. The visual indicator is a green-yellow-red light that indicates safe, possibly dangerous and dangerous lab status, respectively. The audible status indicator is a horn that sounds periodically when not safe, with a shorter period indicating dangerous.

There is an operator information and control panel that consists of a touch-screen display showing the lab status, including detailed sensor/actuator status, and that permits an operator to log in for administrative control.

3 Safety System Design

Overall, the goal of the safety system is to prevent harm to people or property while still permitting fuel cell experiments to run. Specifically, it was designed to satisfy the following safety objectives. (a) Control the system to the safest possible state; when possible, without making it less safe while doing so. (b) Maintain a safety margin by alerting detected equipment failures so repairs can be effected before other failures or environment conditions further compromise safety.

In addition, it was designed with the following operational objectives. (a) Allow operators to put the system into test mode in which faults can be injected without affecting fuel cell experiments. (b) Allow an administrator to change key behaviors of the system, such as threshold levels for sensors or responses to ventilation conditions without modifying the programs. (c) Be secure to unauthorized access.

In designing the system we had the choice between focusing on state or events. In aerospace, nuclear and and other safety-related engineering fields, event-oriented fault tree analysis is commonly used; see [6] for example. This approach focuses on undesired effects in subsystems with a view to understanding their causes as well as likelihood of occurring. In our case, the causes of faults are simple (equipment fails in some way) and are easy for operators to identify. We're more interested in the fact of a fault and in available options to control it to a safer state, so a state-based approach is more suitable.

The system is generally untimed as this simplifies reasoning. However, as a practical system it must take time into account. Once a fan has turned on, for example, we need to wait until it has had time to actually start. If we take

a sensor reading too soon, we will falsely detect that the fan is off and hence assume a fault. In the implementation we handle this by atomically changing a fan's status to Unknown and starting a timer. When the timer expires, the fan's status is changed to that of the sensor readings. The same holds for the vent gate. Similarly, when detecting gas concentrations, we require that a concentration such as High be maintained for a specified period of time before the concentration is reported; this avoids altering on transient concentrations that are safely evacuated. The use of time is isolated to the components that need it so that the system as a whole can be regarded as untimed. For simplicity of presentation in this paper we suppress time by assuming atomicity.

Normally, the decision to address a potential safety concern is made on the basis of likelihood of occurrence, severity and cost. Not having reliability data, especially for older equipment, we treated each potential fault as equally likely, so many of the sensors are duplicated. Under the assumption that only one fails, this eliminates false negatives (missing a hazard) but does admit false positives (wrongly detecting a hazard). We considered triplicating sensors to avoid false positives, but in the end, project managers decided that the cost of replication did not justify the potential inconvenience.

3.1 Safety System Major Components

We took a hierarchical, component-oriented approach to system design. This reduced complexity, made it easier to isolate undesirable behavior, and simplified verification by allowing a compositional approach. The architecture is shown in Fig. 2. Arrows indicate the flow of information. For example, Lab Control provides information to Notification Control but not vice versa. Across the bottom of the figure are the various sensor components that monitor air flow, gas concentration, and the like. Above those are the control components that assess the overall status and determine responses (Lab Control), enact changes in air flow (Ventilation Control), notify lab staff (Notification Control) and so forth.

The system is distributed across a local area network. The box labeled "NI Compact FieldPoint" in the figure shows the components in that node. Notification Control is on a separate PC, as is Operator Information & Control. As shown in the figure, the original plan integrated test stands, partly for safety but more for experiment management. However, funding and schedule limitations precluded implementation, so we do not consider them further.

Inputs to the system are handled by sensor components that discretize the values (giving gas concentrations, for example, in the range {None, Trace, Low, Med, High}) and detect faults (such as sensor disconnection). The control components seek to maintain safety by adjusting ventilation, controlling experiments and notifying personnel of actual and potential safety issues. For example, if the environment risk severity level is high enough due to gas concentrations, the Lab Control component will try to isolate the lab from the rest of the building while also notifying lab personnel.

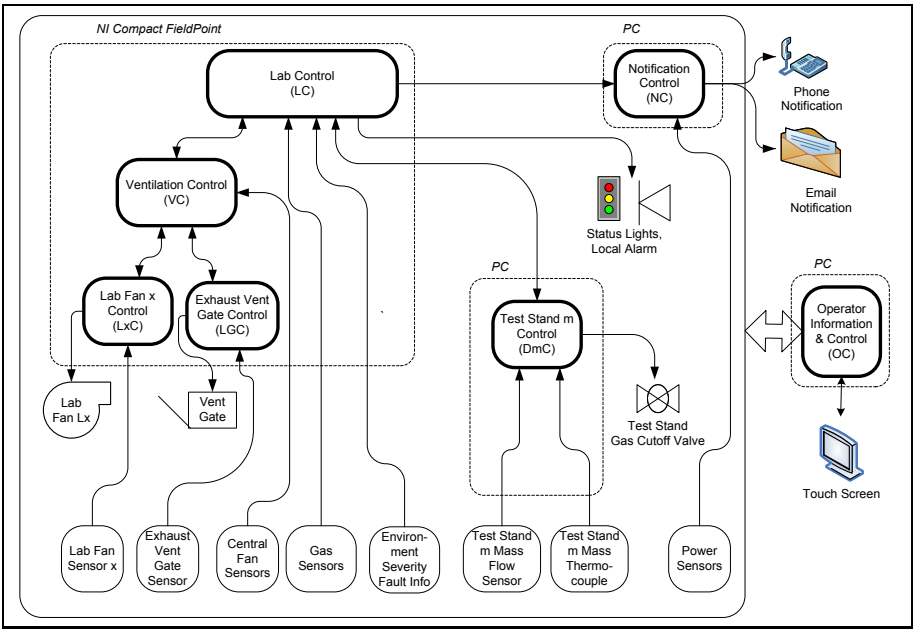


Fig. 2. Safety System Architecture

In the following sections we characterize the components and give guarded command programs for key items. We assume interleaving semantics with fairness.

Sensor Components. Sensor components produce both a sensor reading and a severity code based on fault detection. These contribute to an overall system severity level. For example, in a duplicated pair of sensors, if one is disconnected (or failed), the fault severity is 1 (low, indicating that attention is required but personnel safety is not compromised). If the remaining sensor indicates a high concentration of gas, then the gas severity is 4 (highest, indicating that the building should be evacuated immediately).

We give detail for the hydrogen gas sensor component as representative of the other sensor components. Fault actions that can occur are to change sensor readings (sensor inaccuracy), or to disconnect a sensor. A connected sensor's output is reported as a discrete value in the total order {None, Trace, Low, Med, High} according to a definition of gas concentrations. As noted above, acceptable transients of gas concentration can occur (when changing a gas bottle, for example), so values of Trace or higher must be maintained for a specified period of time before they are reported.

Gas sensors are duplicated to increase reliability. A voting component reports a composite value from discretized sensors and assesses fault severity as shown in the component actions for the hydrogen component, HmR, Fig. 3 . For example,

```
In: HmR0 (hydrogen sensor 0), HmR1 (hydrogen sensor 1);
Out: HmR.Status, HmR.Severity
1  (∀i::HmRi.Status≠Disc) ∧ HmR0.Status=HmR1.Status →
          HmR.Status, HmR.Severity := HmR0.Status, 0;
2  (∀i::HmRi.Status≠Disc) ∧ HmR0.Status≠HmR1.Status →
          HmR.Status, HmR.Severity :=
          max(HmR0.Status,HmR0.Status), 2;
3  (∃i::HmRi.Status=Disc) ∧ (∃j::HmRj.Status≠Disc) →
          HmR.Status, HmR.Severity := HmRj.Status, 1;
4  ∀i::HmRi.Status=Disc →
          HmR.Status, HmR.Severity := Disc, 2;
```

Fig. 3. Hydrogen Sensor Component (HmR)

action 3 handles the case that one sensor is disconnected. In this case, gas level status is that of the connected sensor and a fault severity of 2, indicating a moderate emergency level. When sensors disagree (action 2), the higher value is selected as that is more hazardous. Here and elsewhere, when there is a fault involving duplicated sensors, the status selected is the most hazardous. This can result in false positives, which are inconvenient, but avoid false negatives, which could be deadly.

Other equipment sensors are handled similarly, producing discrete values along with fault detection. These sensors are duplicated. For network monitoring, each node exchanges heartbeats with neighbors to detect disconnection.

```
In: FmR (hydrogen sulfide sensor), XmR (carbon monoxide sensor),
HmR (hydrogen sensor), VC (ventilation control), LxC (lab fan x
[supply, exhaust, booster]), LGC (vent gate), TR (network monitor)
GasStatus() ≡      max({FmR.Status} ∪ {XmR.Status} ∪ {HmR.Status})
Severity_Gas() ≡   0, if GasStatus() = None
                   1, if GasStatus() = Trace
                   2, if GasStatus() = Low
                   3, if GasStatus() = Med
                   4, if GasStatus() = High
Severity_GasFault ≡ max({FmR.Severity} ∪ {XmR.Severity} ∪
                   {HmR.Severity}
Severity_VC() ≡    0, if VentSafety() = Safe
                   1, if VentSafety() = NotLocal
                   2, if VentSafety() = UnsafeLab
                   4, if VentSafety() = UnsafeStocker
Severity_LxC() ≡   0, if ∀x::LxC.Fail = None
                   1, if ∃x::LxC.Fail ∈ {FailOn, FailOff}
Severity_LGC() ≡   0, if LGC.Fail = None
                   1, if LGC.Fail ∈ {FailOpen, FailClose}
Severity_TR() ≡    0, if TR.Status = Connected
                   3, if TR.Status = Disc
Severity() ≡       max of all Severity functions above
```

Fig. 4. Lab Control (LC) Functions

```
In: LC function inputs. Out: LC.VMode, LC.Emergency, LC.LocalAlert
//Set mode for VC
1  Severity_Gas() ≤ 1 →    LC.VMode := Auto;
2  Severity_Gas() > 1) →   LC.VMode := Local;
//Set emergency level, local alert and experiment status
3  Severity() = 0 →        LC.Emergency, LC.LocalAlert
                           := None, Green;
4  Severity() = 1 →        LC.Emergency, LC.LocalAlert := Low, Green;
5  Severity() = 2 →        LC.Emergency, LC.LocalAlert :=
                           Moderate, Yellow;
6  Severity() = 3 →        LC.Emergency, LC.LocalAlert
                           := Urgent, Red;
7  Severity() = 4 →        LC.Emergency, LC.LocalAlert
                           := Evacuate, Red;
```

Fig. 5. Lab Control (LC)

Lab Control (LC). The lab control component evaluates the environmental severity implied by sensors and fault detection and seeks to maintain safety by adjusting ventilation and by altering personnel as shown in Figs. 4 and 5. For ventilation, there are two modes available (actions 1-2). In Auto mode, the building's exhaust should be used if available and otherwise the local exhaust; in Local mode, the lab should be isolated, using local exhaust. This value is input to Ventilation Control, VC. Alerting (actions 3-7) assesses the severity and sets an emergency level for notification in {None, Low, Moderate, Urgent, Evacuate}, sets the color of a light outside the lab in {Green, Yellow, Red}, sounding an alarm if not Green, and sets experiment status to Run or Stop. These values are input to Notification Control, NC, and to the status lights and local alarm.

Ventilation Control (VC). This component adjusts ventilation according to the assessment of Lab Control; see Figs. 6 and 7. In all cases, the component tries to use the central building supply, reverting to local supply if necessary. For Auto mode, if there is a problem with central supply, it will use the local supply fan. If the central exhaust fails, it will go to Local mode. In Local mode, whether specified by Lab Control or required by central exhaust failure, it will isolate the lab with the vent gate and use the local exhaust fan.

In actions 3 and 4, VC checks whether the mode should be Local or Central, using the function CkMode(). For example, while we might want Local mode (LC.VMode = Local), it might be that the vent gate has failed on (see LocalCkMode()); if supply, exhaust and booster are OK, then Central configuration is the safest.

Note that VentSafety() depends in part on ventilation configuration, so actions of VC can change the value of VentSafety() and hence the value of Severity(), potentially causing LC to make a change of ventilation mode. VentSafety() has a value of Safe if actuation is correct; NotLocal when local is required but actuation failure makes it impossible, but ventilation is adequate; UnsafeLab if improper ventilation but the lab is isolated from the main building; and UnsafeCentral if improper ventilation and the lab is not isolated.

```
In: KSR (central supply fan), LSC (lab supply fan), KER (central
exhaust fan), LEC (lab exhaust fan), LGC (vent gate), LBC (booster
fan), LC (lab control)
                                        Control Functions
 //check supply is OK
 SpOK() ≡          true, if (KSR.Status = On ∨ LSC.Ctrbl ∈ {Ctrbl, FailOn})
                   false, otherwise
 //check exhaust OK
 ExOK() ≡          true, if KER.Status = On ∧ LEC.Ctrbl ∈ {Ctrbl, FailOff}
                   false, otherwise
                                        Mode functions
 //check possible mode if local desired
 LocalCkMode() ≡   Central, if LGC.Ctrbl = FailOpen ∧ SpOK() ∧ ExOK() ∧
                     LBC.Ctrbl ∈ {Ctrbl, FailOn}
                   Local, otherwise
 //check possible mode if auto desired
 AutoCkMode() ≡    Central, if LGC.Ctrbl ∈ {Ctrbl, FailOpen} ∧ SpOK() ∧ ExOK() ∧
                     LBC.Ctrbl ∈ {Ctrbl, FailOn}
                   Local, otherwise
 //check possible mode
 CkMode() ≡        LocalCkMode(), if LC.VMode = Local
                   AutoCkMode(), if LC.VMode = Auto
                                        Safety Functions
 //supply safety
 SpSafe() ≡        Safe, if KSR.Status = On ∨ LSC.Status = On
                   Unsafe, otherwise
 //exhaust safety
 ExSafe() ≡        KSafe, if (LGC.Status = Open ∧ LEC.Status = Off ∧ KER.Status = On ∧
                     LBC.Status = On)
                   LSafe, if (LGC.Status = Close ∧ LEC.Status = On)
                   Unsafe, otherwise
 //overall ventilation safety
 VentSafety() ≡    Safe, if (SpSafe() = Safe ∧ ((LC.VMode = Auto ∧ ExSafe() ≠ Unsafe) ∨
                     (LC.VMode = Local ∧ ExSafe() = LSafe)))
                   NotLocal, elseif (LC.VMode = Local ∧ SpSafe() =
                     Safe ∧ ExSafe() = KSafe)
                   UnsafeLab, elseif (LGC.Status = Close)
                   else UnsafeCentral;
```

Fig. 6. Ventilation Control (VC) Functions

```
In: VC function inputs. Out: VC.LECtrl (set lab exhaust fan),
VC.LBCtrl (set booster fan), VC.LGCtrl (set vent gate), VC.LSCtrl
(set local supply fan).
 //Set local supply off if central supply on & vice versa
 1   KSR.Status = Off →     VC.LSCtrl = On;
 2   KSR.Status = On →      VC.LSCtrl = Off;
 //Set exhausCt / booster fan & vent gate per possible mode
 3   CkMode() = Local →     VC.LECtrl, VC.LBCtrl, VC.LGCtrl
                              := On, Off, Close;
 4   CkMode() = Central →   VC.LECtrl, VC.LBCtrl, VC.LGCtrl
                              :=  Off, On, Open;
```

Fig. 7. Ventilation Control (VC)

Note also the interaction between Severity() and VentSafety(). If Severity() ≤ 1, LC will set LC.VMode to Auto; otherwise it will set it to Local, since an isolated lab is safer for higher severity levels. Suppose that Local configuration cannot be achieved due to a fault but Central configuration can be achieved. If Severity() ≤ 1, Central mode will result in VentSafety() = Safe; but if Severity() > 1, it will result in VentSafety() = NotLocal, which is less safe.

```
In: VC.LGCtrl, LGR.Status (lab gate sensor). Out: LGC.Fail,
LGC.Ctrbl, LGC.Status. Local: LGC.Fail
1  VC.LGCtrl = LGR.Status ∧ LGC.Fail = None →
     LGC.Fail, LGC.Ctrbl, LGC.Status := None, Ctrbl, VC.LGCtrl;
2  (VC.LGCtrl = LGR.Status = Open ∧ LGC.Fail ≠ None) ∨
   (VC.LGCtrl = Close ∧ LGR.Status = Open) →
     LGC.Fail, LGC.Ctrbl, LGC.Status := FailOpen, FailOpen, Open;
3  (VC.LGCtrl = LGR.Status = Close ∧ LGC.Fail ≠ None) ∨
   (VC.LGCtrl = Open ∧ LGR.Status = Closed) →
     LGC.Fail, LGC.Ctrbl, LGC.Status
     := FailClose, FailClose, Closed;
```

Fig. 8. Exhaust Vent Gate Control (LGC)

Lab Fans (LxC) and Exhaust Vent Gate (LGC) Controls The lab fans are exhaust (LEC), supply (LEC) and booster (LBC). When VC wants to isolate the lab in Local configuration, it tries to close the vent gate and turn the lab exhaust fan on; this seals the lab from the rest of the building. Actuation failure may prevent this from happening. A control component such as that for the exhaust vent gate LGC (Fig. 8) detects these faults. In action 1, if the VC desired control VC.LGCtrl (Open or Close) is the same as that actuated then there is no failure, it is controllable and the status is as desired. In action 2 if VC desires Close but the status is Open, then the gate has failed open so it is no longer controllable but FailOpen.

The component includes a latch, LGC.Fail. Suppose LGC.Ctrbl is FailOpen and subsequently it is controlled to Open. In this case, LGC.Fail will ensure that LGC.Ctrbl remains FailOpen rather than changing to Ctrbl. An operator must check and clear the fault once the equipment has been fixed. Fault detection results are passed back to Lab Control for severity assessment. The lab fan components are similar to the gate component.

Notification Control (NC) and Operation Information & Control (OC). Notification Control takes input from Lab Control and as appropriate sends emails and places phone calls.

Operator Information & Control displays lab status information, including gas and fan status. There are multiple access levels with different privileges, controlled by fingerprint scanning combined with username/password. Operators use it to start/stop the system and enter a test mode to simulate faults.

4 Safety System Stabilization

We begin with some definitions. A computation is an infinite sequence of system state. A variable in the system terminates when a computation suffix stutters the same variable value. A component or a function terminates when its variables have terminated; the system as a whole terminates when all variables have terminated. In a terminal state, guards can be enabled but no changes to the state take place.

For stabilization, our major concern is ensuring that the system reaches a state that is as safe as possible for personnel in the lab and the surrounding building. This includes notifications. In Fig. 2 we see that information flow is mostly one-way; an inspection of the programs assures us that if Ventilation Control and Lab Control stabilize, notification will be correct. Hence we focus on stabilization of ventilation, which has the potential for a cycle.

Our major concern is stabilization of Ventilation Control, VC. Under the normal assumption that the environment has stopped changing, we want VC to terminate in a state in which ventilation is as safe as possible. To define this we use the function VentSafety() from Fig. 6 which yields a total order {Safe, NotLocal, UnsafeLab and UnsafeCentral} from most safe to to least safe. We characterize the function is as follows, where "elseif" means previous conditions do not hold.

- Safe, if the supply is safe (central or local supply is on, or both) and one of the following is true
  - Desired ventilation mode is Auto and central exhaust is safe (gate is open, central exhaust is on, booster is on, lab exhaust is off)
  - Desired ventilation mode is Local and lab exhaust is safe (gate is closed, local exhaust is on and central exhaust is off)
- NotLocal, elseif desired ventilation mode is Local, supply is safe and central exhaust is safe
- UnsafeLab, elseif vent gate is closed
- else UnsafeCentral

Safe means that the system has achieved the desired ventilation control. NotLocal means that though desired ventilation is Local, the system is using central ventilation. UnsafeLab means ventilation is unacceptable (a supply or exhaust problem) but the vent gate is closed so that the lab is isolated from the rest of the building. UnsafeCentral means that ventilation is unacceptable and the vent gate is open, so hazardous gasses if present could be admitted into the building. A state is the "safest controllable state" if, given the value of VentSafety() in that state, it is not possible to control to another state in which VentSafety() is safer.

We begin with a lemma that shows that LC terminates and use that to show that VC stabilizes to the safest controllable state. Note that stabilization is from an arbitrary state, so transient state corruption is tolerated.

Lemma 1. *LC terminates.*

Proof. Assuming the environment has stopped changing, $\text{Severity}_{\text{Gas}}()$ will not change, so LC.VMode will terminate. Begin with a state in which LC.VMode has terminated. An action of VC may attempt to actuate a fan or the gate and thereby detect a fault, possibly raising the value of Severity(). Since the environment has stopped changing, Severity() will not decrease and eventually there will be no change to Severity() due to detected faults. VentSafety() depends only on LC.VMode and equipment status variables, so it terminates.

There is no cycle involving the variables of Severity() with LC.Emergency and LC.LocalAlert, so Severity() terminates and hence LC terminates. □

Theorem 2. *VC stabilizes to the safest controllable state.*

Proof. We begin by fixing a value of LC.VMode and show that VC stabilizes by terminating in the safest state. The stabilization can take up to two steps: first VC tries to set ventilation as desired; but a new fault can be detected if, for instance, VC tries to open the vent gate but finds that it does not do so. In that case, VC will choose another ventilation configuration and terminate.

First we note that since the environment does not change, component KSR (central supply fan sensor) terminates with a value of On or Off and so VC.LSCtrl terminates. Similarly KER (central exhaust fan sensor) terminates. From the lemma we know that LC terminates.

Given a computation, begin in a state in which these components have terminated. We next show that VC.LECtrl, VC.BCtrl and VC.GCtrl terminate. These are assigned by VC actions 3 and 4, and will change only if CKMode() changes. CKMode() depends on LSC.Ctrbl, LEC.Ctrbl, LGC.Ctrbl, KSR.Status and KER.Status. The latter two variables have terminated so we consider the first three, beginning with LGC. Recall that for simplicity we assume that actuation of the gate and detection of the status are atomic. Suppose VC action 3 is enabled. If the gate is controllable (LGC.Ctrbl = Ctrbl) and there is no fault then the gate will be closed and value on which CKMode() depends will not change. If the gate is FailOpen or FailClose there will again be no change in the value. However, if LGC.Ctrbl = Ctrbl, there could be an undetected fault such as a gate or sensor failure. In this case, if the gate status is open, the attempt to close it will trigger detection the fault and LGC.Ctrbl will be set to FailOpen. Once LGC.Ctrbl is set to other than Ctrbl there will be no further change. A similar reasoning holds for action 4 and for the LSC.Ctrbl and LEC.Ctrbl variables. Since the variables on which CkMode() depends terminate, these variables terminate; and further, LSC, LEC and LGC terminate.

Now we consider VentSafety() in VC's terminal state. Suppose VentSafety() is NotLocal but that, for the sake of contradiction, it is possible to control the ventilation to Safe. VentSafety() = NotLocal means LC.Mode = Local, SpSafe() = Safe and ExSafe() = KSafe. To control to VentSafety() = Safe, we need to control to ExSafe() = LSafe. If LocalCkMode() = Local then VC action 3 is enabled and should result in ExSafe() = LSafe; if it does not, there is an actuation fault, violating our assumption that we can control ventilation to Safe. If LocalCkMode() = Central then Lgc.Ctrbl = FailOpen, again violating the assumption. By similar reasoning on other values of VentSafety() we conclude that it terminates in the safest state for a fixed value of LC.VMode.

Hence we conclude that VC stabilizes to the safest controllable state. □

Now we consider whether the system stabilization safely: does the stabilization process take VentSafety() from a safer to less safe value? In the fault-free case, VC stabilizes in a single step, by switching between Local and Central configurations. For example, suppose that ventilation is in Central configurtion, and suppose

that a gas concentration has changed Severity() to 2 so that VentSafety() is NotLocal. In this case VC will control to Local configuration and in the next state, VentSafety() will be Safe, so we meet the condition for safe stabilization. The same holds for all other cases except two, in which safe stabilization does not hold.

If VC is Safe in Central configuration, LC may specify Local mode depending on Severity(), bringing the system to NotLocal. In, response VC will try to control to Local configuration. If previously undetected faults prevent this, the resultant state can be UnsafeLab or UnsafeCentral depending on whether the vent gate closed or not, passing from a safer to a less-safe state. VC will revert to Central configuration and the terminal state will be NotLocal. There is a similar scenario when VC is Safe in Local configuration and LC specifies Auto mode: the attempt to switch to Central configuration fails, resulting is less safety until it switches back to Local configuration. This is unavoidable since actuation faults can be detected only when there is an attempt to actuate. Hence we have the following result.

Theorem 3. *VC stabilization is safe with the exceptions noted.*

5 Safety System Implementation and Evaluation

5.1 Implementation

The safety system was implemented using National Instruments' LabView software along with associated sensor and actuator interface hardware, chosen for quality, support, integration and industry acceptance.

Most of the components are represented by a finite state machine that defines inputs and outputs in comma-separated tables, and likewise the parameters for sensor thresholds and timers. These are input directly. A system administrator can modify a table whose behavior will be enacted when the system restarts. In fact, some time after the system was deployed, phosphene gas was substituted for hydrogen sulphide in the syngas mixture. The administrators replaced the appropriate sensors and modified the associated tables, accomplishing the modification without changing program code.

The main controller is a National Instruments Compact Fieldpoint (CFP), selected as the most reliable among alternatives. As shown in Fig. 2, notification control and operator control are on separate PCs. The computers are connected via a private Ethernet network. The notification PC also has a separate Ethernet interface with Internet connectivity for email notification. We used Voiceguide for telephone notification.

The major implementation issue turned out to be sensing whether a fan is on or off. Air flow meters seemed natural, but the ones we tested required precise positioning in the airstream that would be difficult to maintain. We chose air pressure meters—manometers—instead, but variations in the airflow dynamics

in the ventilation system made it difficult to determine reliable thresholds that would let us know whether a fan was on or off. We took measurements over a 24 hour period, adjusting the dynamics in various ways, and were able to establish initial thresholds.

5.2 Evaluation

After completing the system, the choice of a state-oriented stabilization approach seems natural. Failures, when they occur, are pretty obvious and the main concern is responding so as to maximize safety. The system stabilizes from an arbitrary state so it tolerates transient state corruption. The component-based approach corresponds naturally to the physical components and the logic is self-contained and easy to compose. Overall, stabilization is a natural fit for this kind of system.

The system has been and continues to be tested in several ways.

- From time to time it is put into test mode and failures are simulated using the Operator Information & Control component. These all succeed.
- Sensors have to be calibrated periodically, accomplished by applying test gas to the sensor nose. This results (correctly) in detection of trace levels of gas and, since calibration is one sensor at a time, in sensor disagreement.
- Personnel in the lab itself must wear a personal gas monitor that measures hydrogen sulfide, hydrogen and carbon monoxide. These readings have always been consistent with the safety system.
- When connecting and disconnecting test stands, small amounts of gas can be released. This is detected by the system.

On one occasion, by chance, a technician misaligned a gas fitting, resulting in a release of gas. This was detected by both the personal monitor and the safety system.

The decision not to triplicate sensors appears to be sound. There has been no case to date of a false positive caused by a faulty sensor value.

We have found modeling errors where the system specification was flawed. The main issue centers around the building exhaust. The system correctly determines whether the central exhaust fan is on or off. However, belts can wear, resulting in a decrease in airflow, and at the threshold, the system can cycle between Central and Local configurations, sending out email notifications when it does so. Had we been aware of this issue, we could have modeled fan sensing by including a hysteresis zone between on and off, requiring it to pass through the zone before changing state. This should have the effect of reducing or eliminating the cycling.

A minor issue is that email notification is too sensitive, sometimes sending out a stream of similar but not identical emails.

These modeling errors could be handled by changing the system. But on the whole, after years of operation, the conclusion of lab personnel is that the system has worked well, fulfilling its original mandate.

References

1. Cooper, M., DeSilva, C., Bayless, D.: Comparison of LSV/YSZ and LSV/GDC SOFC anode performance in coal syngas containing H2S. Journal of the Electrochemical Society 157(11), B1713–B1718 (2010)
2. Faria, D.: Verification and validation of a safety system for a fuel-cell research facility: a case study. Master's thesis, Ohio University (2007)
3. Ghosh, S., Bejan, A.: A framework of safe stabilization. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 129–140. Springer, Heidelberg (2003)
4. Leal, W., McCreery, M., Faria, D.: The OCRC fuel cell lab safety system: A self-stabilizing safety-critical system. Technical Report OSU-CISRC-5/11-TR17, Department of Computer Science and Engineering, The Ohio State University (2011)
5. Leal, W., Xiao, D., de Faria, D.C., Kremer, G., Switzer, S., McCreery, M.: Design of a safety monitoring and control system for a fuel cell laboratory. In: Proceedings of the 31st Clearwater Coal Conference, CCC (2006)
6. NASA. Fault tree handbook with aerospace applications (2002), http://www.hq.nasa.gov/office/codeq/doctree/fthb.pdf
7. Silva, C.D., Kaseman, B., Bayless, D.: Silver (Ag) as anode and cathode current collectors in high temperature planar solid oxide fuel cells. International Journal of Hydrogen Energy 36, 779–786 (2011)

Appendix: Lab Photos

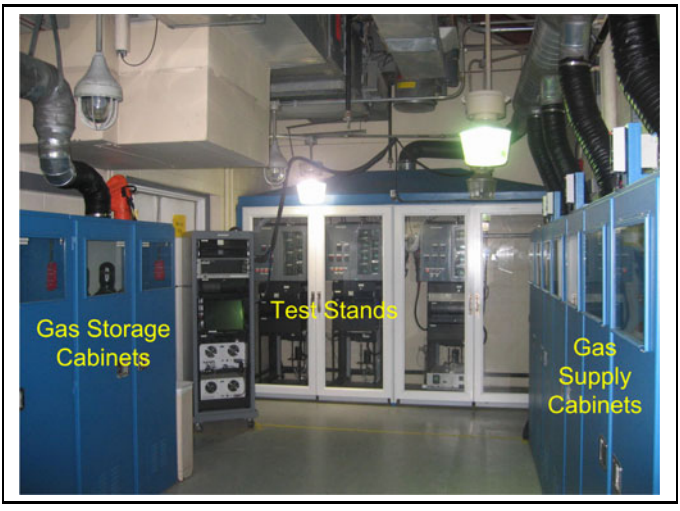


Fig. 9. The OCRC Fuel Cell Lab

Fig. 10. Operator Information and Control Touch-Screen Panel

Relations Linking Failure Detectors Associated with k-Set Agreement in Message-Passing Systems

Achour Mostéfaoui[2], Michel Raynal[1,2], and Julien Stainer[2]

[1] IUF
[2] IRISA
Université de Rennes 1, France
{achour,raynal,Julien.Stainer}@irisa.fr

Abstract. The k-set agreement problem is a coordination problem where each process is assumed to propose a value and each process that does not crash has to decide a value such that each decided value is a proposed value and at most k different values are decided. While it can always be solved in synchronous systems, k-set agreement has no solution in asynchronous send/receive message-passing systems where up to $t \geq k$ processes may crash.

A failure detector is a distributed oracle that provides processes with additional information related to failed processes and can consequently be used to enrich the computability power of asynchronous send/receive message-passing systems. Several failure detectors have been proposed to circumvent the impossibility of k-set agreement in pure asynchronous send/receive message-passing systems. Considering three of them (namely, the generalized quorum failure detector Σ_k, the generalized loneliness failure detector $\mathcal{L}_k$ and the generalized eventual leader failure detector Ω_k) the paper investigates their computability power and the relations that link them. It has three mains contributions: (a) it shows that the failure detector Ω_k and the eventual version of $\mathcal{L}_k$ have the same computational power; (b) it shows that $\mathcal{L}_k$ is realistic if and only if $k \geq n/2$; and (c) it gives an exact characterization of the difference between $\mathcal{L}_k$ (that is too strong for k-set agreement) and Σ_k (that is too weak for k-set agreement).

Keywords: Asynchronous message-passing system, Distributed computability, Equivalence, Eventual leader, Failure detector, Fault-tolerance, Impossibility, Quorum, Realistic failure detector, Reduction, k-Set agreement, Theory.

1 Introduction

On failure detectors. Let us observe that in asynchronous systems where the only means for processes to communicate is using send/receive message-passing, no process is able to know if another process has crashed or is only very slow. The concept of a failure detector originates from this simple observation. A failure detector is a device (distributed oracle) whose aim is to enrich a distributed system by providing alive processes with information on failed processes [5]. Several classes of failure detectors can be defined according to the type of information on failures they provide to processes (see [15] for an introduction to failure detectors).

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 341–355, 2011.

The k-set agreement problem This problem, that has been introduced by S. Chaudhuri [7], is a coordination problem that generalizes the consensus problem. It can be defined as follows [7,14]. Each process proposes a value and every non-faulty process has to decide a value (termination) in such a way that any decided value is a proposed value (validity) and no more than k different values are decided (agreement). The problem parameter k defines the coordination degree: $k = 1$ corresponds to its most constrained instance (consensus) while $k = n - 1$ corresponds to its weakest non-trivial instance (called set agreement).

Let t be the model parameter that denotes the upper bound on the number of processes that may crash in a run, $1 \leq t < n$. If $t < k$, k-set agreement can be trivially solved in both synchronous and asynchronous systems: k predetermined processes broadcast the values they propose and a process decides the first proposed value it receives. Hence, the interesting setting is when $k \geq t$, i.e., when the number of values that can be decided is smaller or equal to the maximal number of processes that may crash in a run.

Algorithms that solve the k-set agreement problem in synchronous message-passing systems when $k \leq t$ are presented in [17]. These algorithms are based on a sequence of synchronous communication rounds. It is shown in the three books previously referenced that $\lfloor \frac{t}{k} \rfloor + 1$ rounds are necessary and sufficient to solve k-set agreement. (This lower bound is still valid in more severe failure models such as general omission failures [17].)

For crash-prone asynchronous systems (be the communication medium a read/write shared memory or a send/receive message-passing network) the situation is different, namely, the k-set agreement problem has no solution when $t \geq k$ [4,11,19].

The cases $k = 1$ and $k = n - 1$ in message-passing systems When $k = 1$, as already indicated k-set agreement boils down to consensus, and it is known that the failure detector denoted Ω is the weakest to solve consensus in asynchronous message-passing systems where $t < n/2$ [6]. Ω ensures that there is an unknown but finite time after which all the processes have the same non-faulty leader (before that time, there is an anarchy period during which each process can have an arbitrarily changing leader). This lower bound result is generalized in [9] where the failure detector Σ is introduced and it is shown that the pair $\langle \Sigma, \Omega \rangle$ is the weakest failure detector to solve consensus in message-passing systems when $t < n$. This means that Σ is the minimal additional power (as far as information on failures is concerned) required to overcome the barrier $t < n/2$ and attain $t \leq n - 1$. Actually, the power provided by Σ is the minimal one required to implement a shared register in a message-passing system [3,9]. Σ provides each process with a quorum (set of process identities) such that the values of any two quorums (each taken at any time) intersect, and there is a finite time after which any quorum includes only correct processes. Fundamentally, Σ prevents partitioning. A failure detector $\langle \Sigma, \Omega \rangle$ outputs a pair of values, one for Σ and one for Ω.

The *Loneliness* failure detector (denoted $\mathcal{L}$) has been proposed in [10] where it is proved that it is the weakest failure detector for solving $(n - 1)$-set agreement in the asynchronous message-passing model. Such a failure detector provides each process p with a boolean (that p can only read) such that the boolean of at least one process remains always false and, if all but one process crash, the boolean of the remaining

process becomes and remains true forever. It is important to notice that, albeit surprisingly, the weakest failure detector for $(n-1)$-set agreement is not the same in the read/write shared memory model (where it is $\overline{\Omega}_{n-1}$) and the send/receive message-passing model (where it is $\mathcal{L}$).

Unfortunately, the weakest failure detector for k-set agreement when $1 < k < n-1$ in message-passing asynchronous crash-prone systems is not yet known. The interested reader will find an introductory survey on failure detectors suited to k-set agreement in [18].

Content of the paper. Among the failure detectors for k-set agreement, $1 \leq k \leq n-1$, that have been proposed in the past few years, this paper investigates the relations on three of them, namely the ones denoted $\mathcal{L}_k$, Ω_k and Σ_k (their precise definitions are given in Section 3). The failure detector Ω_k, introduced in [13], is a generalization of Ω (Ω_1 is Ω).

The failure detector $\mathcal{L}_k$, introduced in [1], is a generalization of $\mathcal{L}$. It allows the k-set agreement problem to be solved in message-passing despite asynchrony and any number of process crashes. Unfortunately, $\mathcal{L}_k$ has been proved to be (a little bit) too strong to solve k-set agreement. Hence, the question "How much stronger is it?"

The failure detector Σ_k, introduced in [2], is a generalization of Σ. It is shown in [2] that Σ_k is necessary (but unfortunately not sufficient) to solve k-set agreement. Hence, the question "How much weaker is it?" It is also shown in [2] that Σ_{n-1} and $\mathcal{L}_{n-1}$ are equivalent (they provide processes with the same computational power).

Answering the two previous questions seems difficult as it would provide us with key elements to obtain the weakest failure detector for asynchronous message-passing k-set agreement. Hence, we consider a more modest question in this paper whose answer will help us better understand and pave the way to the discovery of the weakest failure detector for message-passing k-set agreement. The question is "Which is the property that has to be added to Σ_k in order to obtain exactly $\mathcal{L}_k$?" To be more explicit let X_k be this property. Answering this question means solving the equation $\langle \Sigma_k, X_k \rangle \simeq \mathcal{L}_k$ where X_k is the unknown and "$\simeq$" means "have the same computational power". This paper has three contributions.

- It first focuses on the implementability of $\mathcal{L}_k$ in a synchronous system. Let us remember that a failure detector is *realistic* if it can be implemented in a synchronous system [8]. The paper shows that $k \geq n/2$ is a necessary and sufficient requirement for $\mathcal{L}_k$ to be realistic.
- It then answers the previous question by giving X_k.
- It finally shows that the "eventual" version of $\mathcal{L}_k$, which we denote $\Diamond\mathcal{L}_k$, is nothing else than Ω_k. ("Eventual" means that the properties defining $\mathcal{L}_k$ are required to be satisfied only after some finite time).

Roadmap. The paper is made up of 7 sections. Section 2 presents the base computation model and the k-set agreement problem. Section 3 introduces the failure detectors in which we are interested, i.e., Σ_k, $\mathcal{L}_k$ and Ω_k. The next three sections are the contributions of the paper: Section 4 is on the realism of $\mathcal{L}_k$, Section 5 solves the equation $\langle \Sigma_k, X_k \rangle \simeq \mathcal{L}k$, while Section 6 shows that $\Diamond\mathcal{L}_k$ and Ω_k are equivalent. Finally, Section 7 concludes the paper.

2 Base Computation Model and k-Set Agreement

2.1 Computation Model

Process model. The system consists of a set of n sequential processes denoted p_1, ..., p_n. $\Pi = \{1, \ldots, n\}$ is the set of process identities. Each process executes a sequence of (internal or communication) atomic steps. A process executes its code until it possibly crashes (if it ever crashes). After it has crashed, a process executes no more step. A process that crashes in a run is said *faulty* in that run, otherwise it is *correct*. Given a run, $\mathcal{C}$ and $\mathcal{F}$ denote the set of processes that are correct and the set of processes that are faulty, respectively. Up to $t = n - 1$ processes may crash in a run, hence, $1 \leq |\mathcal{C}| \leq n$.

Communication model. The processes communicate by executing atomic communication steps which are the sending or the reception of a message. Every pair of processes is connected by a bidirectional channel. The channels are failure-free (no creation, alteration, duplication or loss of messages) and asynchronous (albeit the time taken by a message to travel from its sender to its receiver is finite, there is no bound on transfer delays). The notation "broadcast MSG_TYPE(m)" is used as a (non-atomic) shortcut for "**for each** $j \in \Pi$ **do** send MSG_TYPE(m) to p_j **end for**".

Underlying time model The underlying *time model* is the set $\mathbb{N}$ of natural integers. As we are in an asynchronous system, this time notion is not accessible to processes (hence, the model is sometimes called time-free model). It can only be used from an external observer point of view to state or prove properties. Time instants are denoted τ, τ', etc.

Notation. The previous asynchronous crash-prone message-passing system model is denoted $\mathcal{AMP}[\emptyset]$. $\mathcal{AMP}$ stands for "$\mathcal{A}$synchronous $\mathcal{M}$essage-$\mathcal{P}$assing"; $\emptyset$ means the "base" system (not enriched with a failure detector).

2.2 The k-Set Agreement Problem

As already indicated, the k-set agreement problem ($1 \leq k \leq n$) has been introduced by Soma Chaudhuri [7]. It generalizes the consensus problem (that corresponds to $k = 1$). It is defined as follows. Each process proposes a value and has to decide a value in such a way that the following properties are satisfied:

- Termination. Every correct process decides a value.
- Validity. A decided value is a proposed value.
- Agreement. At most k different values are decided.

3 The Failure Detectors Ω_k, Σ_k and $\mathcal{L}_k$

This section presents the three failure detectors we are interested in. (People interested in the underlying assumptions and algorithms to implement failure detectors in asynchronous systems can consult Chapter 7 of [16].)

The system model $\mathcal{AMP}[\emptyset]$ enriched with any of these failure detectors A is denoted $\mathcal{AMP}[A]$. A failure detector provides each alive process p_i with a read-only local variable, say xxx_i. The value of xxx_i at time τ is denoted xxx_i^τ.

3.1 The *Eventual Leadership* Failure Detector Ω_k

The failure detector Ω_k has been introduced in [13]. Its local output at p_i is a set denoted $leaders_i$. Ω_k is defined by the two following properties (where $LD \subseteq \Pi$).

- Validity. $\forall i, \forall \tau$: $(leaders_i^\tau \subset \Pi) \ \wedge \ (|leaders_i^\tau| = k)$.
- Eventual leadership. $\exists\, LD, \exists\, \tau$: $LD \cap \mathcal{C} \neq \emptyset \wedge \forall \tau' \geq \tau, \forall i \in \mathcal{C} : \ leaders_i^{\tau'} = LD$.

Validity means that the values of $leaders_i$ are k process identities. Eventual leadership states that, after some unknown but finite time, all correct processes have the same set of leaders and at least one of these leaders is a correct process. Before all processes are provided with the same set of leaders, there is possibly an unknown but finite anarchy period during which the sets $leaders_i$ have arbitrary values.

The failure detector from which Ω_k originates is Ω (which is the same as Ω_1). As already noticed, Ω has been introduced in [6] where it is shown to be the weakest failure detector to solve consensus in message-passing systems with a majority of correct processes. It has later been shown in [9] that $\langle \Sigma, \Omega \rangle$ is the weakest failure detector to solve consensus for $t = n - 1$.

3.2 The *Quorum* Failure Detector Σ_k

The failure detector Σ_k has been introduced in [2]. It is a generalization of the failure detector Σ (called quorum failure detector) introduced in [9] where it is shown to be the weakest failure detector to implement a register in $\mathcal{AMP}[\emptyset]$ (Σ_1 is Σ).

The local output at p_i of Σ_k is a set qr_i (called quorum) that satisfies the following properties.

- Liveness. $\exists\, \tau$: $\forall\, i \in \mathcal{C}, \forall\, \tau' \geq \tau$: $qr_i^{\tau'} \subseteq \mathcal{C}$.
- Intersection. Let $\mathcal{I} = \{id_x\}_{1 \leq x \leq k+1} \subset \Pi$ be a multiset (or bag) of $k+1$ process identities. Let $\mathcal{T} = \{\tau_x\}_{1 \leq x \leq k+1}$ be a multiset of $k+1$ time instants. $\forall\, \mathcal{I}, \forall\, \mathcal{T}$: $\exists\, i, j \in [1..k+1] : (i \neq j) \wedge (qr_{id_i}^{\tau_i} \cap qr_{id_j}^{\tau_j} \neq \emptyset)$.

The liveness property states that the quorum of a correct process eventually includes only correct processes. The intersection property states that any set of $k+1$ quorums, whose values are taken at any times, contains two intersecting quorums. This means that the intersection property prevents processes to partition in more than k subsets. It is shown in [2] that Σ_k is necessary when one wants to solve k-set agreement in $\mathcal{AMP}[\emptyset]$.

3.3 The *Loneliness* Failure Detector $\mathcal{L}_k$

The $\mathcal{L}_k$ family introduced in [1] is a generalization of the failure detector $\mathcal{L}$ proposed in [10] where it is shown that $\mathcal{L}$ is the weakest failure detector for $(n-1)$-set agreement in asynchronous message-passing systems. $\mathcal{L}_{n-1}$ is $\mathcal{L}$.

The local output at p_i of $\mathcal{L}_k$ is a boolean variable $alone_i$ that satisfies the following two properties (after a process p_i has crashed, we have $alone_i = false$ by definition).

- Stability. $\exists\, K \subset \Pi : (|K| = n - k) \ \wedge \ (\forall\, i \in K, \ \forall\, \tau : \ alone_i^\tau = false)$.
- Loneliness. $(|\mathcal{C}| \leq n - k) \ \Rightarrow \ (\exists\, i \in \mathcal{C}, \exists \tau : \ \forall\, \tau' \geq \tau : \ alone_i^{\tau'} = true)$.

Stability states that the boolean variables of at most k processes can take the value $true$, while the loneliness property states that, if at least k processes crash, then there is a finite time after which there is a correct process p_i whose boolean variable $alone_i$ remains forever equal to $true$. An algorithm solving the k-set agreement problem in $\mathcal{AMP}[\mathcal{L}_k]$ is described in [11,18]. It is also shown in [11] that, for $1 < k < n-1$, $\mathcal{L}_k$ is not the weakest failure detector for k-set agreement in $\mathcal{AMP}[\emptyset]$.

4 On the Implementability of $\mathcal{L}_k$ in a Synchronous System

This section addresses the realism of $\mathcal{L}_k$, i.e., its implementability in a synchronous distributed system.

Synchronous distributed system. Synchrony is an abstraction that encapsulates (and hides) specific timing assumptions. A synchronous system provides the processes with a global clock r called *round number* which entails the progress of the computation. The processes progress simultaneously from round to round. During a round, a process sends a message to all processes, receives messages from other processes and executes local computation. The fundamental property associated with a synchronous system is that a message sent by a process during a round r is received by the other processes during the very same round r. (More information on synchronous systems can be found in the book [17] that is entirely devoted to such systems.)

Let $\mathcal{SMP}$ denotes a $\mathcal{S}$ynchronous $\mathcal{M}$essage-$\mathcal{P}$assing system made up of n processes, in which up to $t = n-1$ processes may crash.

Building $\mathcal{L}_k$ in $\mathcal{SMP}$ when $k \geq n/2$. Algorithm 1 presents the code of such a construction for a process p_i. Initially, $alone_i$ is false. Then, during each round r, p_i broadcasts a message ALIVE(i) to inform the other processes that it was alive at the beginning of round r (line 04). Then, p_i receives round r messages and updates rec_ids_i accordingly (line 05). Finally, if rec_ids_i contains at most $n-k$ process identities, p_i sets $alone_i$ to $true$ (line 06).

```
(01) init alone_i ← false;
(02) when r = 1, 2, ... do
(03) begin synchronous round
(04)    broadcast ALIVE(i);
(05)    rec_ids_i ← { j such that ALIVE(j) received during the current round };
(06)    if (|rec_ids_i| ≤ n − k) then alone_i ← true end if
(07) end synchronous round.
```

Algorithm 1. Building $\mathcal{L}_k$ in $\mathcal{SMP}$ when $k \geq n/2$ (code for p_i)

Lemma 1. *Let $k \geq n/2$. Algorithm 1 builds a failure detector $\mathcal{L}_k$ in $\mathcal{SMP}$.*

Proof. Proof of the stability property. Let r be the first round during which a process p_i sets $alone_i$ to $true$. This means that at least k processes have crashed before sending

their round r message to p_i. Consequently, at most $n-k$ processes will ever set their boolean $alone_i$ to $true$. As $k \geq n/2 \Rightarrow n-k \leq k$, it follows that at least k processes will never set their boolean variables to $true$ which proves the property.

Proof of the loneliness property. If k or more processes crash during a run, for each correct process p_i, there is a round r such that p_i receives at most $n-k$ messages at every round $r' \geq r$, from which it follows that, from round r, $alone_i$ remains forever equal to $true$. $\square_{Lemma\ 1}$

Lemma 2. *It is not possible to build $\mathcal{L}_k$ in $\mathcal{SMP}$ when $k < n/2$.*

Proof. Assuming by contradiction that there is a synchronous algorithm $\mathcal{A}$ that builds $\mathcal{L}_k$ (with $k < n/2$) in $\mathcal{SMP}$, let us consider a run r_{k+1} of $\mathcal{A}$ in which the processes $p_1, p_2, ..., p_k$ have initially crashed. It follows from the loneliness property (guaranteed by $\mathcal{A}$) that there is a process (say p_{k+1}) whose boolean $alone_{k+1}$ becomes eventually $true$ and remains afterward forever equal to $true$.

Let us now consider a second run r_{k+2} of $\mathcal{A}$ that is identical to r_{k+1} until $alone_{k+1}$ becomes and remains forever equal to $true$ and such that, after that round, p_{k+1} crashes. As previously, it follows from the loneliness property (guaranteed by $\mathcal{A}$) that there is a process (say p_{k+2}) whose boolean $alone_{k+2}$ becomes and remains forever equal to $true$. It is possible to continue to design similar runs r_{k+3}, etc., until r_n (in each r_x, $k+1 \leq x \leq n$, $x-1$ processes crash).

Hence, in the run r_n, $n-k$ processes p_j have set their boolean variable $alone_j$ to $true$. As $k < n/2 \Rightarrow n-k > k$, this contradicts the fact that the algorithm $\mathcal{A}$ guarantees the stability property of $\mathcal{L}_k$, namely, at most k processes never set their boolean variable to $true$. A contradiction from which the lemma follows. $\square_{Lemma\ 2}$

The next theorem follows from Lemma 1 and Lemma 2.

Theorem 1. *$k \geq n/2$ is a necessary and sufficient condition for $\mathcal{L}_k$ to be realistic.*

Let us notice that the failures detectors $\mathcal{P}$ (perfect failure detector) [5], Σ, Σ_k, Ω, Ω_k are realistic.

5 Relating $\mathcal{L}_k$ and Σ_k

This section determines the property that has to be added to Σ_k (which is necessary to solve k-set agreement) in order to obtain exactly $\mathcal{L}_k$. Let X_k be this property. Hence, this section solves the equation $\langle \Sigma_k, X_k \rangle \simeq \mathcal{L}_k$ where X_k is the unknown and $A \simeq B$ means "$\mathcal{AMP}[A]$ and $\mathcal{AMP}[B]$ have the same computational power".

5.1 The Property X_k

X_k is designed to work with Σ_k. It is defined by the following property where qr_i is the output of Σ_k at p_i.

– Loneliness. $(|\mathcal{C}| \leq n-k) \Rightarrow (\exists\, i \in \mathcal{C}, \exists \tau : \forall\, \tau' \geq \tau : qr_i^{\tau'} = \{i\})$.

Hence, X_k requires that, when at least k processes are faulty, there a time instant at which the quorum of a correct process forever contains only itself.

The next theorem follows directly from the lemmas 3 and 4 that are proved in the next two sections.

Theorem 2. *$\mathcal{AMP}[\mathcal{L}_k]$ and $\mathcal{AMP}[\langle \Sigma_k, X_k\rangle]$ have the same computational power.*

5.2 Building $\mathcal{L}_k$ in $\mathcal{AMP}[\langle \Sigma_k, X_k\rangle]$

Algorithm 2 presents a very simple algorithm that builds $\mathcal{L}_k$ in $\mathcal{AMP}[\langle \Sigma_k, X_k\rangle]$. The output $alone_i$ of each process p_i is initialized to $false$ and is set to $true$ if the predicate $qr_i = \{i\}$ becomes satisfied at least once.

init $alone_i \leftarrow false$;
when $qr_i = \{i\}$: $alone_i \leftarrow true$.

Algorithm 2. Building $\mathcal{L}_k$ in $\mathcal{AMP}[\langle \Sigma_k, X_k\rangle]$

Lemma 3. *Algorithm 2 builds $\mathcal{L}_k$ in $\mathcal{AMP}[\langle \Sigma_k, X_k\rangle]$.*

Due to page limitations, the reader will find the proof in [12]. (It is worth noticing that this proof does not use the liveness property of Σ_k.)

5.3 Building $\langle \Sigma_k, X_k\rangle$ in $\mathcal{AMP}[\mathcal{L}_k]$

Underlying principle and description of the algorithm Algorithm 3 relies on the observation of the predicate $qr_i = \{i\}$ that it makes stable (once satisfied, it remains forever satisfied).

The variable qr_i of each process is initialized to Π in order not to compromise the intersection property. Then, if $alone_i$ becomes true, p_i sets qr_i to $\{i\}$ (line 03) and, from then on, qr_i will keep that value forever and p_i repeatedly informs of it the other processes by broadcasting a message ALONE(i).

When it receives ALONE(j) from another process (line 04), p_i learns that qr_j is keeping forever the value $\{j\}$. If $qr_i \neq \{i\}$, p_i updates accordingly qr_i to $\{i, j\}$ in order to preserve the intersection property of Σ_k (line 05).

Independently of its other statements, p_i repeatedly informs the other processes that it is alive (task $T1$ where messages ALIVE(i) are broadcast forever). This triggers coordination among the processes even if no boolean $alone_i$ becomes equal to $true$. Its aim is to ensure the liveness property of Σ_k.

The task $T2$ is associated with the processing of the ALIVE(j) received by p_i. When it has received such messages from $(n - k)$ distinct processes, if qr_i has not stabilized on $\{i\}$, p_i resets qr_i to the set containing i and the processes from which alive messages have been received.

Lemma 4. *Algorithm 3 builds $\langle \Sigma_k, X_k\rangle$ in $\mathcal{AMP}[\mathcal{L}_k]$.*

(01) **init**: $qr_i \leftarrow \Pi$; start $T1$, $T2$.

(02) **when** $alone_i$ becomes equal to $true$:
(03) $qr_i \leftarrow \{i\}$; **repeat forever** broadcast ALONE(i) **end repeat**.

(04) **when** ALONE(j) with $j \neq i$ **is received**:
(05) **if** $(qr_i \neq \{i\})$ **then** $qr_i \leftarrow \{i, j\}$ **end if**.

(06) **task** $T1$: **repeat forever** broadcast ALIVE(i) **end repeat**.

(07) **task** $T2$:
(08) **repeat forever**
(09) **wait until** (new ALIVE(j) messages with $j \neq i$ received from $n - k$ processes);
(10) **let** $proc_i$ = the set of $n - k$ processes from which messages have been received;
(11) **if** $(qr_i \neq \{i\})$ **then** $qr_i \leftarrow \{i\} \cup proc_i$ **end if**
(12) **end repeat**.

Algorithm 3. Building $\langle \Sigma_k, X_k \rangle$ in $\mathcal{AMP}[\mathcal{L}_k]$

Proof. Proof of the liveness property of Σ_k and the loneliness property of X_k.
Let A be the set of correct processes p_i whose boolean variable $alone_i$ takes the value $true$. Due to line 02, each process p_i of A sets qr_i to $\{i\}$ and, due to lines 05 and 11, it follows that qr_i remains forever equal to $\{i\}$, We consider two cases.

- Case 1: There are k or more faulty processes ($|\mathcal{C}| \leq n - k$).
  As $|\mathcal{C}| \leq n - k$, it follows from the loneliness property of $\mathcal{L}_k$ that $A \neq \emptyset$ which establishes the loneliness property of X_k.
  It follows from line 03 that each process p_i of A broadcasts forever the message ALONE(i) and from line 05 that, after all the messages ALONE() sent by faulty processes have been received, no process p_j will add the identity of a faulty process to its quorum qr_j at line 05. Moreover, after k processes have crashed, the task $T2$ of each correct process p_i not in A eventually remains blocked forever at line 09 and p_i will never add faulty processes to qr_i at line 11. Consequently, there is a time after which no faulty process will ever be added to the quorum qr_i of a correct process from which we conclude that there is a time after which the quorum qr_i of any correct process p_i contains only its identity or its identity plus the identity of another process of A. This concludes the proof of the liveness property of Σ_k when $|\mathcal{C}| \leq n - k$.
  Remark. Let us observe that eventually any correct process p_i of A is such that $qr_i = \{i\}$ while any process p_i not in A is such that $qr_i = \{i, j\}$ where p_j is a process of A that can change with time. In the paragraph that follows the proof, the set A is called *kernel*.
- Case 2: There are less than k faulty processes ($|\mathcal{C}| > n - k$).
  As $|\mathcal{C}| > n-k$, there are at least $(n-k+1)$ correct processes that forever broadcast messages ALIVE() (line 06) and consequently no process will ever block forever at

line 09. Hence, every correct process p_i that does not belong to A will infinitely often updates qr_i to sets of $(n-k+1)$ correct processes (line 11). Moreover, after it has received the last message ALONE() sent by a faulty process, p_i no longer includes a faulty process in its quorum qr_i at line 05. It follows that there is a time after which the quorum of a correct process contains only correct processes which proves the liveness property of Σ_k. The loneliness property of X_k follows trivially from $\neg(|\mathcal{C}| \leq n-k)$.

Proof of the intersection property of Σ_k.
Let us assume by contradiction that there is a set of quorum values $\{q_j\}_{1\leq j\leq k+1}$ computed by the algorithm that are such that $\forall j_1 \neq j_2 : q_{j_1} \cap q_{j_2} = \emptyset$. Let P be the set of identities of the processes from which these $k+1$ quorum values have been obtained. As, for any process p_i, we always have $i \in qr_i$ (lines 01, 03, 05 and 11), it follows that $|P| = k+1$.

Let us assume without loss of generality that $P = \{1, 2, \ldots, k+1\}$ and q_j has been computed by p_j. The value of each q_j has been computed at line 03, 05 or 11 (it cannot be the initial value $qr_j = \Pi$ because that quorum value would intersect any other quorum and our contradiction assumption would not be satisfied).

Let $B \subseteq P$ the set of processes whose quorum values have been computed at line 03 or 05. Hence, each of these quorum values contains a process p_x whose boolean variable $alone_x$ has taken the value $true$. It then follows from the stability property of $\mathcal{L}_k$ that at most k processes p_x can have $alone_x = true$, from which we conclude that $0 \leq |B| \leq k$. It follows from $B \subseteq P$, $|P| = k+1$ and $0 \leq |B| \leq k$ that $P \setminus B \neq \emptyset$.

Let $i \in P \setminus B$. Due to the definition of B, p_i has computed q_i at line 11 and, consequently, we have $|q_i| = n-k+1$. As $\forall j_1, j_2 \in P : (j_1 \neq j_2) \Rightarrow (q_{j_1} \cap q_{j_2} = \emptyset)$ and $\forall i \in P : i \in q_i$, we have $q_i \cap (P \setminus \{i\}) = \emptyset$. It follows that $|q_i| \leq |\Pi| - |P \setminus \{i\}| = n-k$ which contradicts $|q_i| = n-k+1$ and concludes the proof of the intersection property of Σ_k. $\square_{Lemma\ 4}$

A property of $\langle \Sigma_k, X_k \rangle$ The loneliness property of X_k is $(|\mathcal{C}| \leq n-k) \Rightarrow (\exists\, i \in \mathcal{C}, \exists \tau : \forall \tau' \geq \tau : qr_i^{\tau'} = \{i\})$. Actually, Algorithm 3 ensures a stronger property, that we call *strong loneliness*, defined as follows.

- Strong loneliness. $(|\mathcal{C}| \leq n-k) \Rightarrow \big(\exists \tau : \forall \tau' \geq \tau, \forall\, i \in \mathcal{C} : (\exists\, j \in qr_i^{\tau'} : \forall \tau'' \geq \tau : qr_j^{\tau''} = \{j\})\big)$.

When $|\mathcal{C}| \leq n-k$, let us call *kernel* the set of processes p_i that after some time have forever $qr_i = \{i\}$ (this is the set called A in the proof of the liveness property of Σ_k and the loneliness property of X_k in Lemma 4). As it requires that, after some time, each correct process p_i either belongs to the kernel or has a quorum containing a process of the kernel (which can change with time) it is easy to see that strong loneliness property implies loneliness defined by X_k.

As far as the other direction is concerned we have the following. As algorithm 3 ensures strong loneliness, it follows from Lemma 3 and Lemma 4 that this stronger property is implicitly contained in $\langle \Sigma_k, X_k \rangle$.

6 Relating $\mathcal{L}_k$ and Ω_k

This section shows that a simple and pretty natural weakening of the stability of $\mathcal{L}_k$ gives rise to a new failure detector which is equivalent to Ω_k. This establishes a strong relation linking $\mathcal{L}_k$ and Ω_k.

6.1 The Failure Detector $\Diamond\mathcal{L}_k$

Weakening the stability property of $\mathcal{L}_k$ from perpetual to eventual gives rise to the failure detector $\Diamond\mathcal{L}_k$. Hence, $\Diamond\mathcal{L}_k$ is defined by the following properties.

- Eventual stability. $\exists\tau, \exists K \subset \Pi : (|K| = n-k) \wedge (\forall\, i \in K, \forall\, \tau' \geq \tau : alone_i^{\tau'} = false)$.
- Loneliness. $(|\mathcal{C}| \leq n-k) \;\Rightarrow\; (\exists\, i \in \mathcal{C}, \exists\tau :\; \forall\, \tau' \geq \tau :\; alone_i^{\tau'} = true)$.

Hence, when compared to $\mathcal{L}_k$, $\Diamond\mathcal{L}_k$ ensures the boolean variables of at least $n-k$ processes remains forever equal to $false$ only after an unknown but finite time.

The next two sections show that $\Diamond\mathcal{L}_k$ and Ω_k are equivalent in $\mathcal{AMP}[\emptyset]$, i.e., $\Diamond\mathcal{L}_k$ can be built in $\mathcal{AMP}[\Omega_k]$ and Ω_k can be built in $\mathcal{AMP}[\Diamond\mathcal{L}_k]$. The next theorem follows directly from the lemmas 5 and 7 that are proved in the next two sections.

Theorem 3. *$\mathcal{AMP}[\Diamond\mathcal{L}_k]$ and $\mathcal{AMP}[\Omega_k]$ have the same computational power.*

6.2 Building $\Diamond\mathcal{L}_k$ in $\mathcal{AMP}[\Omega_k]$

Algorithm 4 is a very simple construction of $\Diamond\mathcal{L}_k$ in $\mathcal{AMP}[\Omega_k]$. The boolean $alone_i$ of each process p_i is initialized to $false$. Then, a process p_i repeatedly updates it to $true$ or $false$ according to the fact that its identity appears or does not appear in the local output $leaders_i$ currently provided by its underlying failure detector Ω_k.

init $alone_i \leftarrow false$;
repeat forever $alone_i \leftarrow (i \in leaders_i)$ **end repeat**.

Algorithm 4. Building $\Diamond\mathcal{L}_k$ in $\mathcal{AMP}[\Omega_k]$ (code for p_i)

Lemma 5. *Algorithm 4 builds $\Diamond\mathcal{L}_k$ in $\mathcal{AMP}[\Omega_k]$.*

Proof. Let τ be a finite time after which all faulty processes have crashed and there is a set LD of k process identities such that $LD \cap \mathcal{C} \neq \emptyset, \forall\, i \in \mathcal{C}, \forall\, \tau' \geq \tau$: $leaders_i^{\tau'} = LD$. (Due to the eventual leadership property of Ω_k, τ and LD do exist.) It follows from the algorithm that, after τ, each process p_i with $i \in LD \cap \mathcal{C}$ executes forever $alone_i \leftarrow true$ (Observation O1) while each process p_i with $i \in \mathcal{C} \setminus LD$ executes forever $alone_i \leftarrow false$ (Observation O2). Let $\tau' \geq \tau$ be a time instant at which each correct process p_i has updated at least once its boolean variable $alone_i$.

Proof of the eventual stability property of $\Diamond\mathcal{L}_k$. Let $K = \Pi \setminus LD$ (hence $|K| = n-k$). For the faulty processes of K, let us remember that, by definition, the boolean variable $alone_x$ of a crashed process p_x is equal to $false$. The correct processes p_i of K are such that $i \in \mathcal{C} \setminus LD$ and it follows from observation O2 that the boolean

variables of all correct processes in $\mathcal{C} \setminus LD$ remain forever equal to $false$ after τ'. Hence, we have $\exists \tau'' \geq \tau', \exists\, K : (|K| = n-k) \wedge (\forall\, i \in K,\ \forall\, \tau''' \geq \tau'' :\ alone_i^{\tau'''} = false)$, which proves the eventual stability property of $\diamond\mathcal{L}_k$.

Proof of the loneliness property of $\diamond\mathcal{L}_k$. Let us observe that, due to Ω_k, $\mathcal{C} \cap LD \neq \emptyset$, hence $\exists i \in \mathcal{C} \cap LD$. It then follows from observation O1 that, after τ', p_i forever executes $alone_i \leftarrow true$ which concludes the proof. $\square_{Lemma\ 5}$

6.3 Building Ω_k in $\mathcal{AMP}[\diamond\mathcal{L}_k]$

Underlying principle. The idea of the algorithm is the following. We know from $\diamond\mathcal{L}_k$ that there is a finite time after which there is a set X of at least $n-k$ processes p_i that will no longer have their boolean $alone_i$ equal to $true$. The algorithm strives to capture the complementary set Y of X in order to have Y in the final output LD of Ω_k. Let us observe that we have the following when Y has been captured. If k or more processes are faulty, it follows from the loneliness property of $\diamond\mathcal{L}_k$ that there is a correct process p_i whose boolean becomes and remains true forever, hence we have $i \in Y$ and we can take any LD such that $Y \subseteq LD$. If less than k processes are faulty, it is relatively easy for the correct processes to agree on any set of k processes (as any such set includes a correct process).

A list of all subsets of size k. Let us consider all possible subsets of k processes. Moreover, let us order all of them according to lexicographical ordering when considering each subset as a sorted array. Hence, the set $\{1, 2, \ldots, k-1, k\}$ is the first, $\{1, 2, \ldots, k-1, k+1\}$ the second, etc., until the set $\{n-k+1, \ldots, n-1, n\}$ that is the last one. Let L be this sorted list of all subsets of size k. Moreover, let L' be the list L where $\{1, 2, \ldots, k-1, k\}$ is defined as the successor of the last subset $\{n-k+1, \ldots, n-1, n\}$.
Finally, let $\mathsf{next\_ld}(sbst)$ be the function that returns the subset that follows $sbst$ in L'.

Local variables. To attain their goal, the processes execute asynchronous rounds. The local variable r_i contains the current round number of p_i. The current local output computed by p_i to implement Ω_k is kept in the local variable $leaders_i$.

Each process p_i manages a set called $next\_set_i$. This set contains all the pairs $(r, leaders)$ received by p_i. The aim of this set is to allow the processes to proceed in the *very same order* from the pair $(1, \{1, 2, \ldots, k-1, k\})$, to the pair $(1, \{1, 2, \ldots, k-1, k+1\})$, etc., until the pair $(1, \{n-k+1, \ldots, n-1, n\})$ during the first round, then from $(2, \{1, 2, \ldots, k-1, k\})$ until $(2, \{n-k+1, \ldots, n-1, n\})$ during the second round, etc., until they stop during a round r on the very same pair $(r, leaders)$ and define accordingly $LD = leaders$.

Behavior of a process p_i. Algorithm 5 describes the behavior of a process p_i. Let us first observe that, if no correct process p_i ever receives a message $\text{ALONE}(j)$, we can conclude from the eventual stability property of $\diamond\mathcal{L}_k$ that less than k processes are faulty. Then no message is ever exchanged among the correct processes p_i and they all agree on $leaders_i = LD = \{1, 2, \ldots, k\}$ (that contains at least one correct process) and Ω_k is trivially implemented. Let us now consider the general case. Each time it

```
(01) init leaders_i ← {1, 2, ..., k}; r_i ← 1; next_set_i ← ∅;
(02) repeat forever if (alone_i) then broadcast ALONE(i) end if end repeat.

(03) when ALONE(j) is received: if (j ∉ leaders_i) then broadcast NEXT(r_i, leaders_i) end if.

(04) when NEXT(r, leaders) is received for the first time:
(05)   broadcast NEXT(r, leaders);
(06)   next_set_i ← next_set_i ∪ {(r, leaders)};
(07)   while ((r_i, leaders_i) ∈ next_set_i) do
(08)     leaders_i ← next_ℓd(leaders_i);
(09)     if (leaders_i = {1, 2, ..., k}) then r_i ← r_i + 1 end if
(10)   end while.
```

Algorithm 5. Building Ω_k in $\mathcal{AMP}[\diamond\mathcal{L}_k]$ (code for p_i)

reads $true$ from $alone_i$, process p_i broadcasts a message ALONE(i). When it receives a message ALONE(j) such that p_j is not currently in $leaders_i$, process p_i broadcasts NEXT$(r_i, leaders_i)$ (line 03). This message is to indicate to the others processes that its current set $leaders_i$ seems not to be the final one and consequently p_i demands the others processes to try the next candidate leader set obtained from the list L'.

When p_i receives a message NEXT$(r, leaders)$ for the first time, it forwards it to all in case the sender crashed during its broadcast (line 05) and saves it in $next\_set_i$ (line 06). Then if the current pair $(r_i, leaders_i)$ belongs to $next\_set_i$ (line 07), p_i progresses to the next candidate set of the list L' (line 08). If the new value of $leaders_i$ is $\{1, \ldots, k\}$, all the elements of L have been tried during round r_i and consequently p_i progresses to the next round (line 09) to try again the elements of L with the aim to stop on one of them.

Let $(r1, leaders1) < (r2, leaders2) \stackrel{def}{=} (r1 < r2) \vee \big((r1 = r2) \wedge (leaders1 < leaders2)\big)$ (where $leaders1 < leaders2$ is the order of the sorted list L).

Lemma 6. *Let p_i be a correct process. Let $r_i = r$ and $leaders_i = \ell d$. Process p_i has received and broadcast all the messages* NEXT$(r', \ell d')$ *such that $(r', \ell d') < (r, \ell d)$.*

Proof. Due to lines 08-09, when modified, the new value of the pair $(r_i, leaders_i)$ is the direct successor pair, according to the order defined on these pairs. If follows from this observation and the initial value of the pair $(r_i, leaders_i)$ (namely, $(1, \{1, \ldots, k\})$) that, when $(r_i, leaders_i) = (r, \ell d)$, the pair $(r_i, leaders_i)$ has taken all the pair values $(r', \ell d')$ such that $(1, \{1, \ldots, k\}) \leq (r', \ell d') < (r, \ell d)$.

Let us now observe that the pair $(r_i, leaders_i)$ progresses to its next value only when its current value belongs to the set $next\_set_i$ (line 07). Moreover, a pair $(r', \ell d')$ is added to $next\_set_i$ only when a message NEXT$(r', \ell d')$ is received by p_i for the first time, and this message is then systematically forwarded to all (lines 04-07).

It follows that, when $(r_i, leaders_i) = (r, \ell d)$, p_i has received and broadcast all the messages NEXT$(r', \ell d')$ such that $(1, \{1, \ldots, k\}) \leq (r', \ell d') < (r, \ell d)$. $\square_{Lemma\ 6}$

Lemma 7. *Algorithm 5 builds Ω_k in $\mathcal{AMP}[\diamond\mathcal{L}_k]$.*

Proof. Let us first observe that the values taken by any set $leaders_i$ are the subsets of size k defined in the list L. The validity property of Ω_k follows trivially.

Let Π_1 denote the set of processes that broadcast an infinity of messages ALONE(). It follows from the eventual stability property of $\diamond\mathcal{L}_k$ that there is a finite time τ_1 after which at least $n-k$ boolean variables $alone_i$ remain forever equal to $false$, hence $|\Pi_1| \leq k$. Moreover, there is a time $\tau_2 \geq \tau_1$ from which (a) all received messages ALONE() are from processes of Π_1 and (b) no process broadcasts at line 03 a message NEXT(r, ld) with $\Pi_1 \subset ld$ (this follows from the predicate used in the **if** statement of line 03).

Let $(r0, \ell d0)$ be the greatest pair such that $\Pi_1 \subseteq \ell d0$ and there is a message NEXT$(r0, \ell d0)$ that entailed an update of $leaders_i$ at some correct process p_i (line 08). If there is no such pair, let $(r0, \ell d0) = (0, \alpha)$ where $\mathsf{next\_ld}(\alpha) = \{1, \ldots, k\}$. Let $(r1, \ell d1)$ be the smallest pair such that $(r0, \ell d0) < (r1, \ell d1)$ and $\Pi_1 \subset \ell d1$.

It follows from the definition of $(r0, \ell d0)$ and Lemma 6 that all NEXT$(r, \ell d)$ messages such that $(r, \ell d) \leq (r0, \ell d0)$ have been sent by correct processes from which we conclude that each correct process p_i eventually updates $leaders_i$ to $\mathsf{next\_ld}(\ell d0)$.

If $\Pi_1 \not\subset leaders_i$ at some correct process p_i, it follows from the definition of Π_1 that p_i eventually receives a message ALONE(j) sent by p_j such that $j \in \Pi_1 \setminus leaders_i$. When it receives such a message (line 03), process p_i broadcasts NEXT$(-, leaders_i)$. It follows that, for each pair $(r, \ell d)$ such that $(r0, \ell d0) < (r, \ell d) < (r1, \ell d1)$, each correct process broadcasts (at line 03 or line 05) a message NEXT$(r, \ell d)$. Moreover, once all these messages have been received, the correct processes p_i eventually agree on the same set $leaders_i = \ell d1$ and no longer broadcast NEXT$(-,-)$ messages.

If there are k or more faulty processes, the loneliness property of $\diamond\mathcal{L}_k$ implies that $\Pi_1 \cap \mathcal{C} \neq \emptyset$ and consequently $\ell d1$ contains at least one correct process. If there are less than k faulty processes, as $|\ell d1| = k$, $\ell d1$ contains at least one correct process, which concludes the proof of the lemma. $\square_{Lemma\ 7}$

7 Conclusion

This paper has investigated the computability power and explored the relations linking three failure detectors that have been proposed to solve the k-set agreement problem in asynchronous crash-prone message-passing systems. (The k-set agreement problem is a coordination problem that generalizes the consensus problem.) These three failure detectors are the generalized quorum failure detector Σ_k, the generalized loneliness failure detector $\mathcal{L}_k$ and the generalized eventual leader failure detector Ω_k.

The paper has (a) shown that the failure detector Ω_k and the eventual version of $\mathcal{L}_k$ have the same computational power; (b) shown that $\mathcal{L}_k$ is realistic if and only if $k \geq n/2$; and (c) given an exact characterization of the difference between $\mathcal{L}_k$ and Σ_k. Hence, this paper provides us with a better understanding of these failure detectors in the quest for the weakest failure detector to solve the k-set agreement problem in asynchronous message-passing crash-prone systems.

Acknowledgments. The authors want to thank F. Bonnet, M. Biely, P. Robinson and U. Schmid for constructive comments.

References

1. Biely, M., Robinson, P., Schmid, U.: Weak Synchrony Models and Failure Detectors for Message Passing (k-)Set Agreement. In: Proc. 11th Int'l Conference on Principles of Distributed Systems (OPODIS 2009). LNCS, vol. 5923, pp. 285–299. Springer, Heidelberg (2009)
2. Bonnet, F., Raynal, M.: On the Road to the Weakest Failure Detector for k-Set Agreement in Message-passing Systems. To appear in Theoretical Computer Science (2010)
3. Bonnet, F., Raynal, M.: A Simple Proof of the Necessity of the Failure Detector Σ to Implement an Atomic Register in Asynchronous Message-passing Systems. Information Processing Letters 110(4), 153–157 (2010)
4. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on Theory of Computation (STOC 1993), San Diego (CA), pp. 91–100 (1993)
5. Chandra, T., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
6. Chandra, T., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus. Journal of the ACM 43(4), 685–722 (1996)
7. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105, 132–158 (1993)
8. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A Realistic Look At Failure Detectors. In: Proc. 42th Int'l IEEE/IFIP Conference on Dependable Systems and Networks (DSN 2002), pp. 345–353. IEEE Press, Los Alamitos (2002)
9. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Tight Failure Detection Bounds on Atomic Object Implementations. Journal of the ACM, 57(4) Article 22 (2010)
10. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Tielmann, A.: The Weakest Failure Detector for Message Passing Set-Agreement. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 109–120. Springer, Heidelberg (2008)
11. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
12. Mostéfaoui, A., Raynal, M., Stainer, J.: Relations Linking Failure Detectors Associated with k-Set Agreement in Message-passing Systems. Technical Report #1973, 13 pages, IRISA, Université de Rennes 1 (F) (April 2011)
13. Neiger, G.: Failure Detectors and the Wait-free Hierarchy. In: 14th ACM Symposium on Principles of Distributed Computing (PODC 1995), pp. 100–109. ACM Press, New York (1995)
14. Raynal, M.: Set Agreement. In: Encyclopedia of Algorithms, pp. 829–831. Springer, Heidelberg (2008)
15. Raynal, M.: Failure Detectors for Asynchronous Distributed Systems: an Introduction. Wiley Encyclopdia of Computer Science and Engineering 2, 1181–1191 (2009)
16. Raynal, M.: Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems, 251 pages. Morgan & Claypool Pub. (2010) ISBN 978-1-60845-293-4
17. Raynal, M.: Fault-Tolerant Agreement in Synchronous Message-Passing Systems, 165 pages. Morgan & Claypool Publishers (2010) ISBN 978-1-60845-525-6
18. Raynal, M.: Failure Detectors to solve Asynchronous k-set Agreement: a Glimpse of Recent Results. The Bulletin of EATCS 103, 75–95 (2011)
19. 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)

Corona: A Stabilizing Deterministic Message-Passing Skip List

Rizal Mohd Nor[1], Mikhail Nesterenko[1], and Christian Scheideler[2,⋆]

[1] Department of Computer Science, Kent State University, Kent, OH, USA
[2] Department of Computer Science University of Paderborn, Paderborn, Germany

Abstract. We present Corona, a deterministic self-stabilizing algorithm for skip list construction in structured overlay networks. Corona operates in the low-atomicity message-passing asynchronous system model. Corona requires constant process memory space for its operation and, therefore, scales well. We prove the general necessary conditions limiting the initial states from which a self-stabilizing structured overlay network in message-passing system can be constructed. The conditions require that initial state information has to form a weakly connected graph and it should only contain identifiers that are present in the system. We formally describe Corona and rigorously prove that it stabilizes from an arbitrary initial state subject to the necessary conditions. We extend Corona to construct a skip graph.

1 Introduction

In a peer-to-peer overlay network, each process can communicate with any other peer process over the underlying network as long as the process is aware of the peer's identifier. These identifier records form the network topology. Peer-to-peer networks are effective for distributed information storage, group communication and large scale computations. The amount of research literature on this subject is extensive [2,3,4,6,13,16,22,23,25].

The skip list [20] is a popular peer-to-peer topology as it allows efficient search and quick topology updates. Specifically, both identifier search as well as process deletion or addition in a skip list take $O(\log n)$ steps, where n is the number of nodes. A skip list may be either randomized or deterministic. While the randomized version may be simpler to implement, the deterministic one provides firm search and topology update bounds as well as greater assurance against failures, malicious behavior and unfavorable topology changes.

A skip list may not be sufficiently robust against node crashes. Indeed, a single node failure may disconnect the skip list. Neither is a skip list particularly suitable for concurrent searches. The standard measures of robustness and concurrency are expansion and congestion [4]. The expansion and congestion of

⋆ Supported in part by DFG awards SCHE 1592/1-1 and SFB 901 (On-the-Fly Computing).

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 356–370, 2011.

the skip list are $O(1/n)$ and $\Omega(n)$ respectively. A skip list extension, the skip graph [3], significantly improves these metrics.

Peer-to-peer systems may include millions of nodes. At such scale, fault-tolerance and topology maintenance become a major concern. Self-stabilization [11] may be a particularly suitable failure recovery approach for peer-to-peer systems [1,19] as it is oblivious to the exact nature of the fault. As soon as the influence of the fault stops, regardless of the state in which this fault leaves the system, its self-stabilization is guaranteed to return it to a correct state.

Due to the large initial state space, self-stabilization programs require careful correctness proofs. If practical low atomicity communication models, such as the message-passing system, are considered such proofs may become difficult both to construct and to verify. Furthermore, a large initial state space may lead to excessive process memory demands during stabilization, especially during initial linearization: topological sorting of the processes [19].

Our Contribution. In this paper we present Corona: a self-stabilizing deterministic skip list construction algorithm in message-passing systems. To the best of our knowledge Corona is the first such algorithm.

Before describing Corona, we prove two necessary conditions for the existence of a self-stabilizing solution to any overlay network problem. The conditions limit the possible initial states in two ways: the state information must form a weakly connected graph, and the states should not include identifiers that are not present in the system. Subject to these restrictions, we rigorously prove Corona to correctly stabilize from an arbitrary initial state.

Instead of struggling to counteract the large state space of message passing systems, we are able to use the low-atomicity model to our advantage: the channels are employed as extra identifier storage space. This allows us to keep the Corona design relatively straightforward and to linearize processes using process memory that is independent of the system size. We extend Corona to build skip graphs and to accommodate topology updates.

Related Literature. There is a large body of literature on how to efficiently maintain peer-to-peer networks. Most of the results focus on preserving the overlay network in the legal set of states. Relatively few studies address the self-stabilization of such networks. Due to the topology being part of system state, the majority of classic self-stabilizing techniques are not applicable to peer-to-peer networks.

Let us survey the publications in self-stabilization of peer-to-peer networks. A few papers address simple topologies. The Iterative Successor Pointer Rewiring Protocol [10] and the Ring Network [24] organize the nodes in a sorted ring. Onus et al. [18] linearize a network into a sorted linked list. However, they use a simplified synchronized communication model for their algorithm.

There are several studies of more sophisticated structures. Hérault et al. [14] describe a self-stabilizing spanning tree algorithm. Caron et al. [8] present a Snap-Stabilizing Prefix Tree for Peer-to-Peer systems while Banchi et al. [7]

show stabilizing peer-to-peer spatial filters. However, none of these structures approach the congestion and expansion of a skip graph. Clouser et al. [9] propose a deterministic self-stabilizing skip list for shared register communication model. Gall et al. [12] discuss models that capture the parallel time complexity of locally self-stabilizing networks that avoids bottlenecks and contention. Jacob et al. [21] generalize insights gained from graph linearization to two dimensions and present a self-stabilizing construction for Delaunay graphs. In another paper, Jacob et al. [15] present a self-stabilizing, randomized variant of the skip graph and show that it can recover its network topology from any weakly connected state in $\mathcal{O}(\log^2 n)$ communication rounds with high probability in a simple, synchronized message passing model. In [5] the authors present a general framework for the self-stabilizing construction of any overlay network. However, the algorithm requires the knowledge of the 2-hop neighborhood for each node and involves the construction of a clique. In that way, failures at the structure of the overlay network can easily be detected and repaired.

2 Model, Notation and Definitions

Peer-to-Peer Networks. A peer-to-peer overlay network program consists of a set N of n processes with unique identifiers. A process can communicate with any other process as long as it has a record of its identifier. The communication is by passing messages through channels.

Peer-to-peer networks often require ordering the processes in a sequence according to their identifiers. Two processes a and b are *consequent*, denoted $\mathbf{cnsq}(a, b)$, if $(\forall c : c \in N : (c < a) \vee (b < c))$. That is, two consequent processes do not have an identifier between them. For the sake of completeness, we assume that $-\infty$ is consequent with the smallest id process in the system. Similarly, the largest id process is consequent with $+\infty$.

Graph terminology helps us in reasoning about peer-to-peer networks. A *link* is a pair of identifiers (a, b) defined as follows: either message $message(b)$ carrying identifier b is in the incoming channel of process a, or process a stores identifier b in its local memory. See Figure 2 for illustration. Note that a thus defined link is directed. In referring to such a directed link (a, b), we always state the predecessor process a first and the successor process b second. The *length* of a link (a, b) is the number of processes c such that $a < c < b$. Note that the length of (a, b) is zero if $\mathbf{cnsq}(a, b)$ is true. The length of $(-\infty, a)$ is zero if a is the smallest id in the system, it is n otherwise. Similarly, the length of $(b, +\infty)$ is zero if b is maximum and n otherwise. The *process connectivity* graph CP is the graph formed by the links of the identifiers stored by the processes. A *channel connectivity* multigraph CC includes both locally stored and message-based links. Self-loop links are not considered. By this definition, CP is a subgraph of CC. Note that besides the processes, CC and CP may contain two nodes $+\infty$ and $-\infty$ and the corresponding links to them. Graph CP captures current network connectivity information the set of processes possesses. CC reflects the connectivity data that is stored implicitly in the messages in communication channels. Again, refer to Figure 2 for an example of both graph types.

Computation Model. Each process contains a set of variables and actions. A *channel* C is a special kind of variable whose values are sets of messages. We assume that the only information a message carries is process identifiers. We further assume that a message carries exactly one identifier. The identifiers are defined. That is, a message cannot carry ∞. Channel message capacity is unbounded. Messages cannot be lost. The order of message receipts does not have to match the order of transmission. That is, the channels are not FIFO. Due to this, we treat all messages sent to a particular process as belonging to a single incoming channel.

An action has the form $\langle guard \rangle \longrightarrow \langle command \rangle$. *guard* is either a predicate over the contents of the incoming channel or **true**. In the latter case the predicate and corresponding action are *timeout*. *command* is a sequence of statements assigning new values to the variables of the process or sending messages to other processes.

Program state is an assignment of a value to every variable of each process and messages to each channel. A program state may be arbitrary, the messages and process variables may contain identifiers that are not present in the network. An identifier is *existing* if it is present in the network. An action is *enabled* in some state if its guard is **true** in this state. It is *disabled* in this state otherwise. A timeout action is always enabled. We consider programs with timeout actions, hence, in every state there is at least one enabled action.

A *computation* is an infinite fair sequence of states such that for each state s_i, the next state s_{i+1} is obtained by executing the command of an action that is enabled in s_i. This disallows the overlap of action execution. That is, action execution is *atomic*. We assume two kinds of fairness of computation: weak fairness of action execution and fair message receipt. *Weak fairness* of action execution means that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. *Fair message receipt* means that if the computation contains a state where there is a message in a channel, the computation also contains a later state where this message is not present in the channel.

We focus on programs that do not manipulate the internals of process identifiers. Specifically, a program is *compare-store-send* if the only operations that it does with process identifiers is comparing them, storing them in local process memory and sending them in a message. That is, operations on identifiers such as addition, radix computation, hashing, etc. are not used. In a compare-store-send program, if a process does not store an identifier in its local memory, the process may learn this identifier only by receiving it in a message. A compare-store-send program cannot introduce new identifiers to the network, it can only operate on the ids that are already there. If a computation of a compare-store-send program starts from a state where every identifier is existing, each state of this computation contains only existing identifiers.

A state *conforms* to a predicate if this predicate is **true** in this state; otherwise the state *violates* the predicate. By this definition, every state conforms to predicate **true** and none conforms to **false**. Let A and B be predicates over

program states. Predicate A is closed with respect to the program actions if every state of the computation that starts in a state conforming to A also conforms to A. Predicate A converges to B if both A and B are closed and any computation starting from a state conforming to A contains a state conforming to B.

Problems. The *overlay network problem* maps each set of identifiers to a set of acceptable process connectivity graphs. For example, for every set of processes, the *linearization problem* specifies exactly one graph where each process is linked with its consequent processes.

Linearized overlay networks simplify process search. When discussing a linearized network, processes with identifiers greater than p are to the *right* of p, while processes with identifiers smaller than p are to the *left* of p. That is, we consider processes arranged in the increased order of identifiers from left to right. See Figure 2 for an illustration.

The process search time in a simple linearized network is proportional to its size. This may not be acceptable in large-scale networks. Shortcut links are added to accelerate navigation. In a deterministic *skip list*, these links are created recursively by levels. The zero (bottom) level is the linearized list of processes. In a *k-l* skip list, a node a has a link to node b at level i if a and b are between k and l hops away at level $i-1$. For example, in a *1-2* skip list, a and b are linked at level i if they are no more than three and no less than two hops away at level $i-1$. Refer to Figure 4 for an example of a *1-2* skip list.

In the *k-l skip list construction* problem, a set of processes is mapped to the set of possible skip lists. Note that in a linearization problem the set of identifiers uniquely determines the connectivity graph. In case of *k-l* skip list construction, depending on which processes participate at each level, the same list of identifiers may form several possible skip lists. Hence, the skip list construction problem specifies multiple acceptable CP graphs for a single set of processes.

We define the two problem properties below to aid us in formally stating the necessary conditions for the existence of a solution. An overlay network problem is *single component* if it maps every set of processes to a weakly connected process connectivity graph. Intuitively, a single component network overlay problem prohibits a program from separating the network into multiple components. The linearization and skip list construction problem are single component.

An overlay network problem $\mathcal{PG}$ is *disconnecting* if there is at least one set of processes S such that for every channel connectivity graph CP to which $\mathcal{PG}$ maps S, there is a cut set CS such that $|CS| < n-1$ which disconnects S. Note that such a cut set exists for any graph except for a completely connected one. Essentially, a disconnecting network overlay problem requires that in at least one case the desired channel connectivity graph is not completely connected. Again, both the linearization and skip list construction problem are disconnecting.

Problem Solutions. A program $\mathcal{PG}$ *satisfies* or *solves* a problem $\mathcal{PR}$ from a predicate P if, for every set S, every computation of $\mathcal{PG}$ that starts in a state conforming to P contains a suffix with the following property. The channel connectivity graph CP is the same in every state of this suffix and this CP is

one of the graphs to which $\mathcal{PR}$ maps S. That is, starting from the initial state in P, the solution has to implement at least one of the required CPs.

Program stabilization is *graph-identical* if every computation of a stabilizing program contains a suffix where CC contains the same links as CP. Such program generates CC links that are already present in CP. If a process of such program receives a message, this message carries an identifier that the recipient process already stores and the process ignores the message.

A program is *unconditionally stabilizing* (or just *stabilizing*) if it solves the problem from $P \equiv \mathbf{true}$. That is, every computation of a stabilizing program, regardless of the initial state, contains a correct suffix. Unconditional stabilization may be too strong for a program to possess. A program is *conditionally stabilizing* if $P \not\equiv \mathbf{true}$. That is, such program stabilizes from a limited set P of states.

We define two special cases of conditionally stabilizing programs. A program is *weakly channel-connectivity stabilizing* if it stabilizes only from the initial states where the channel-connectivity graph is weakly connected. A program is *existing identifier stabilizing* if it stabilizes only from states where every identifier is existing.

3 Necessary Conditions

The necessary conditions stated in this section show that common overlay network topology specifications prohibit the existence of unconditionally stabilizing solutions. The necessary conditions are that initially the channel connectivity graphs need to be connected and non-existing identifiers are not present.

The proofs for these conditions rely on the lemma below. Intuitively, the lemma states that for the processes to form a connected topology they have to be at least weakly connected initially.

Lemma 1. *If a computation of a compare-store-send program starts in a state where the channel connectivity graph CC is disconnected, the graph is disconnected in every state of this computation.*

Proof: Let us consider, without loss of generality, a program state where the connectivity graph contains two components C_1 and C_2. Assume the opposite: the computation starting from this state contains states where the two components of CC are connected. Let us consider the first such state s_1. In this state there must be two process $a \in C_1$ and $b \in C_2$ that are neighbors. Assume the link is from a to b. That is, $(a, b) \in CC$.

Since s_i is the first connected state, this link does not belong to CC in the preceding state s_{i-1}. Since the program is compare-store-send, the new link can not appear in the process memory, it must be due to a message sent to a by another process c in state s_{i-1}. A message to a carrying b can only be sent by a process c that has links to both a and b in s_{i-1}.

Since $(c, a) \in CC$, c belongs to the same component C_1 as a in s_{i-1}, and since $(c, b) \in CC$, c belongs to the same component C_2 as b in s_{i-1}. This means that

C_1 and C_2 are weakly connected in a state s_{i-1} that precedes s_i. However, we assumed that s_i is the first state where the two components are connected. This contradiction proves the lemma. □

Theorem 1. *If a compare-store-send self-stabilizing program is a solution to a single-component overlay network problem, this program must be weakly channel-connectivity stabilizing.*

Proof: Assume the opposite. That is, there is a self-stabilizing program $\mathcal{PG}$ that solves a single-component overlay network problem $\mathcal{PR}$ and it is not weakly channel-connectivity stabilizing.

Since $\mathcal{PG}$ is a solution to $\mathcal{PR}$, for each set S, every computation of $\mathcal{PG}$ contains a suffix with the prescribed CP. Since $\mathcal{PG}$ is not necessarily weakly channel-connectivity stabilizing, this holds true for computations starting from a state where CC is disconnected. Program $\mathcal{PG}$ is a compare-store-send program. According to Lemma 1, if its computation starts from a state where CC is disconnected, it is disconnected in every state of this computation. Since CP is a subgraph of CC, it has to be disconnected in every state of this computation as well. However, $\mathcal{PR}$ is single-component. Since $\mathcal{PR}$ is single component, it maps every set of processes S to a weakly connected process CP. This means that, contrary to our initial assumption, $\mathcal{PR}$ is not a solution to $\mathcal{PG}$. Hence the theorem. □

Theorem 2. *If a graph-identical compare-store-send program is a stabilizing solution to a single-component disconnecting overlay network problem, this program must be existing identifier stabilizing.*

Proof: Assume the opposite. Let $\mathcal{PG}$ be a compare-store-send program that is a graph-identical self-stabilizing solution to a single-component disconnecting overlay network problem $\mathcal{PR}$. Since $\mathcal{PR}$ is disconnecting, there is a set of processes S such that for every connectivity graph, there is a cut set that disconnects this graph.

Consider a computation σ of $\mathcal{PG}$ with set S. Let CP be the process connectivity graph to which this computation converges. Let CS be the cut set that separates S into two subsets S_1 and S_2. Since $\mathcal{PG}$ is graph-identical, σ contains a suffix where, in every state, CC has the same links as CP. Let s_1 be the first state of this suffix.

We examine a set of processes $S_1 \cup S_2$ and construct a state of the program for this set as follows. The state of every process in $S_1 \cup S_2$ and its incoming channel is the same as in the initial state of σ. In addition, the incoming channels of each process a belonging to $S_1 \cup S_2$ in this state contain the messages that are sent to a by processes in CS. From this state, we execute the actions of $\mathcal{PG}$ for processes $S_1 \cup S_2$ in the same sequence as in σ. The presence of messages from processes in CP allows us to do that. After this procedure we arrive at a state s_2. We then execute the actions of $\mathcal{PG}$ in arbitrary fair manner. Thus constructed sequence is a computation of $\mathcal{PG}$.

Note that each process of $S_1 \cup S_2$ has the same state in s_1 and s_2. Since CS was a cut set of CP in s_1, there are no links between processes of S_1 and S_2 in either s_1 or s_2. This means that CP is disconnected in s_2. Graph CC has the same links as CP in s_1. This means that CC is disconnected in s_2 as well. According to Lemma 1, both CC and CP are disconnected in every state of this computation past s_2.

However, $\mathcal{PG}$ is supposed to be a solution to $\mathcal{PR}$. Problem $\mathcal{PR}$ is single component. This means our constructed computation has to contain a suffix where CP is weakly connected in every state. This contradiction proves the theorem. □

4 Linearization

Problem Statement. In the linearization problem, each set of processes is mapped to the following process connectivity graph CP. Each process p in CP contains exactly two outgoing links: $p.r$ and $p.l$. The links conform to the following predicate LP:

$$(\forall a, b \in N : a < b : \mathbf{cnsq}(a, b) \Leftrightarrow ((a.r = b) \wedge (b.l = a)))$$

The predicate states that two processes are neighbors if and only if they are consequent.

l-Corona Description. Each process p maintains two variables r and l as required by the problem specification. The range of each variable are the process identifiers respectively to the left and to the right of p. That is, r can only store identifiers that are greater than p, while l – less than p. The value of each variable may be undefined. In this case it is equal to respectively $-\infty$ and $+\infty$. If non-existent identifiers are not present in the initial state of the program computation, the l variable of the smallest id process and the r variable of the largest id process are always set to $-\infty$ and $+\infty$ respectively.

Each process p of l-Corona contains two actions: a receive-action and a timeout action. The receive action is enabled when there is a message in the incoming channel $p.C$. The operation of the action depends on the id carried by the message. If id is greater than p, it is compared to r. If id is less than r, then p discovered a closer right neighbor. Process p then forwards the old right neighbor identifier to the new process and reassigns its variable r. However, if the received id is no less than r, then the current right neighbor of p is no further away than id. In this case p sends id for process r to process. If r is not initialized, it is assigned the received id. The identifier that is smaller than p is handled similarly. The timeout action sends the process identifier to its left and right neighbors. An example computation of l-Corona is shown in Figure 2.

Correctness Proof. Due to the lack of space the actual proofs in this section are relegated to the technical report [17].

Observe that due to the operation of the algorithm, in case $a < b$, link (a, b) can only be replaced by a link (a, c) such that $a < c < b$. Likewise, link (b, a)

```
process p
variables
    r,   // right identifier, greater than p
    l    // left identifier, less than p
actions
    message(id) ∈ p.C ⟶
        receive message(id)
        if id > p then
            if id < r then
                if r < +∞ then
                    send message(r) to id
                r := id
            else
                send message(id) to r
        if id < p then
            if id > l then
                if l > −∞ then
                    send message(l) to id
                l := id
            else
                send message(id) to l
    true ⟶
        if r < +∞ then send message(p) to r
        if l > −∞ then send message(p) to l
```

Fig. 1. Linearization component of Corona (l-Corona)

can only be replaced by (b, c) such that $a < c < b$. That is, a link in CP can only be shortened. An example of CP link shortening is shown in Figure 2: the link (b, d) is shortened to (b, c) in transition from 2(a) to 2(b). Note that every process in CP contains exactly two outgoing links. One is pointing to the left, the other — to the right.

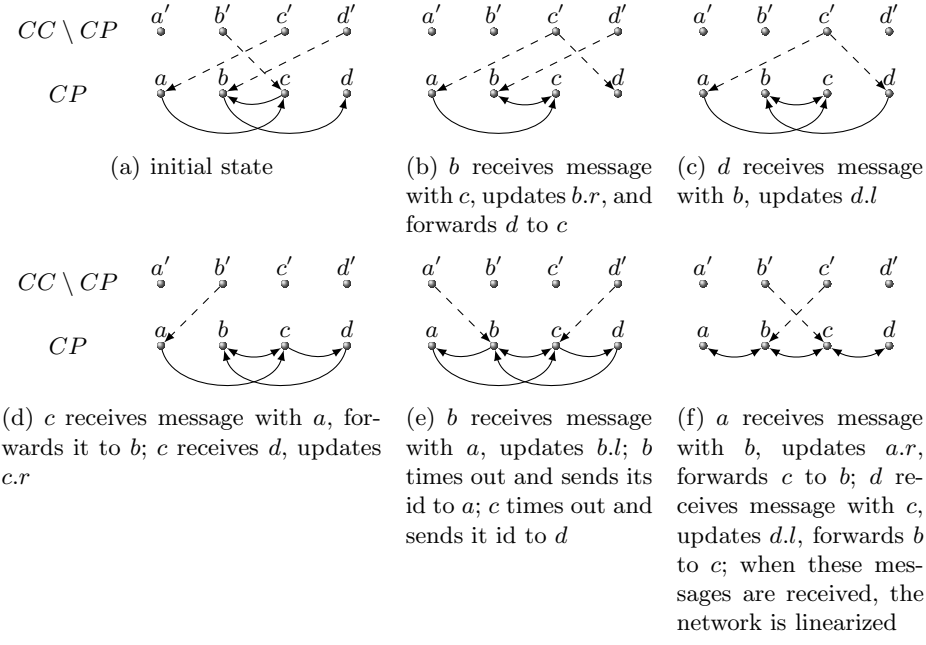
Similarly, in case $a < b$, a link $(a, b) \in CC \setminus CP$ can be replaced only by a link (c, b) such that $a < c < b$. In the other direction, a link $(b, a) \in CC \setminus CP$ can be replaced only by a link (c, a) such that $a < c < b$. Again, the link in CC can only be shortened. For example, link $(c, a) \in CC \setminus CP$ in Figure 2 is shortened to (b, a) in transition from 2(c) to 2(d). Note that unlike CP, a process may contain more than two outgoing links in $CC \setminus CP$. And, while some links are shortened, longer ones may be added by timeout actions.

Lemma 2. *If a computation of l-Corona starts from a state where CC contains a path from process a to b, then in every state of this computation, there is a path from a to b as well.*

Lemma 3. *If a computation of l-Corona starts in a state where for some process a there are two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ such that $a < c < b$, then this computation contains a state where there is a link $(a, d) \in CP$ where $d \leq c$.*

Similarly, if the two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ are such that $b < c < a$, then this computation contains a state where there is a link $(a, d) \in CP$ where $d \geq c$.

Intuitively, Lemma 3 states that if there is a link in the incoming channel of a process that is shorter than what the process already stores, then, the process' links will eventually be shortened. The proof is by simple examination of the algorithm.



(a) initial state

(b) b receives message with c, updates $b.r$, and forwards d to c

(c) d receives message with b, updates $d.l$

(d) c receives message with a, forwards it to b; c receives d, updates $c.r$

(e) b receives message with a, updates $b.l$; b times out and sends its id to a; c times out and sends it id to d

(f) a receives message with b, updates $a.r$, forwards c to b; d receives message with c, updates $d.l$, forwards b to c; when these messages are received, the network is linearized

Fig. 2. Example computation of l-Corona. To simplify the picture each process is represented by two nodes. The primed nodes are the process' incoming channel. Solid lines denote identifiers stored in l and r of each process. Dashed lines are identifiers in the incoming channel.

Lemma 4. *If a computation of l-Corona starts in a state where for some process a there is an edge $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ such that $a < b < c$, then the computation contains a state where there is a link $(d, c) \in CP$, where $d \leq b$.*

Similarly, if the two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ are such that $c < b < a$, then this computation contains a state where there is a link $(d, c) \in CP$, where $d \geq b$.

Intuitively, the above lemma states that if there is a longer link in the channel, it will be shortened by forwarding the *id* to its closer successor.

Lemma 5. *If a computation of l-Corona starts in a state where for some processes a, b, and c such that $a < c < b$ (or $a > c > b$), there are edges $(a, b) \in CP$ and $(c, a) \in CC$, then the computation contains a state where either some edge in CP is shorter than in the initial state or $(a, c) \in CP$.*

Lemma 6. *If a computation starts in a state where there is a link $(a, b) \in CP$, then the computation contains a state where some link in CP is shorter than in the initial state or there is a link $(b, a) \in CP$.*

Lemma 7. *If the computation is such that if $(a, b) \in CP$ then $(b, a) \in CP$ in every state of the computation, then this computation contains a suffix where $((a, b) \in CP) \Rightarrow ((a, b) \in CC)$*

Lemma 7 states that if CP does not change in a computation then eventually, the links in CP contain all the links of CC.

Lemma 8. *Let CP is strongly connected in some state of the system. Let also for every pair of processes a and b in this state, if $(a, b) \in CP$ then also $(b, a) \in CP$. In this case, this state satisfies LP.*

Theorem 3. *Program l-Corona is a weakly channel-connectivity existing identifier stabilizing solution to the linearization problem.*

5 Skip List Stabilization

Problem Statement. The problem maps each set of processes to a set of valid *1-2* skip lists. In each skip list the bottom level is linearized and for each level $i > 0$, the following predicate SL holds: any two processes a and b are neighbors at level i if the distance between a and b at level $i - 1$ is no less than 2 and no more than 3 hops.

s-Corona Description. Each level of s-Corona has two sub-levels: *status decision* sublevel — sd-Corona, and *neighbor linking* sublevel sn-Corona.

sd-Corona of level i uses neighborhood information of level $i - 1$ to determine the *status* of a process at level i. Depending on whether the process participates at level i, the process status is either **up** or **down**. If a process is **down** at level i it is **down** at all levels above i. On the basis of this information sn-Corona links p with its left and right neighbor at level i. sn-Corona of level i does not influence the operation of sd-Corona at level i. If process p is **up**, sn-Corona inspects $i - 1$ neighbors three hops away from p to determine the nearest **up** neighbor and connects it to p. To ensure overall CC connectivity preservation sn-Corona sends itself the link to the previous neighbor at level 0 for l-Corona to handle. The stabilizing implementation of sn-Corona is relatively straightforward. We, therefore, do not present it and focus on sd-Corona instead.

sd-Corona Description. sd-Corona operates similarly at each level. At every level it maintains a set of variables that belong to only this level. At level i, process p of sd-Corona makes use of the identities $p.(i-1).l$ and $p.(i-1).r$ of its respective left and right neighbors at level $i - 1$. sd-Corona at level i does not change these identities. Therefore, they are assumed constant for the operation of sd-Corona at this level.

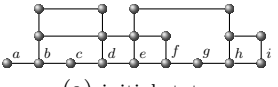
At level i, process p of sd-Corona maintains two status variables: $p.i.st$ and $p.i.str$. The values for both are **up** and **down**. Variable $p.i.st$ stores the status of p itself. Variable $p.i.str$ keeps the status of the right neighbor of p. The status of the rightmost and leftmost process at level i are fixed as **up** and **down** respectively and are considered constant.

The idea of sd-Corona is to ensure that no two consequent neighbors are **up** and no three of them are **down**. To break symmetry in deciding who of the neighbors should change status, the decision of the right neighbor is favored.

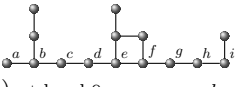
```
process p
constants
    p.(i − 1).r, p.(i − 1).l    // identifiers of right and left neighbors at level i − 1
variables
    p.i.st,     // own status at level i, either up or down
                // constant and set to up for process with highest id
                // constant and set to down for process with lowest id
    p.i.str     // status of right neighbor
actions
    message(status) ∈ p.C from p.(i − 1).r ⟶
            receive message(status),
            p.i.str := status,
            if (p.i.st = up) ∧ (p.i.str = up) then
                p.i.st := down

    message(status) ∈ p.C from p.(i − 1).l ⟶
            receive message(status),
            if (status = down) ∧ (p.i.st = down) ∧ (p.i.str = down) then
                p.i.st := up

    true ⟶
            if p.(i − 1).r < +∞ then send message(p.i.st) to p.(i − 1).r,
            if p.(i − 1).l > −∞ then send message(p.i.st) to p.(i − 1).l
```

Fig. 3. Status decision component of skip list part of Corona (sd-Corona)

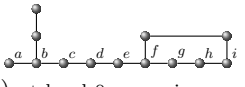
sd-Corona has three guards. The timeout guard sends the status of p to its neighbors. The two receive guards process messages from the left and right neighbors of p. If p receives a status value from its right neighbor, it updates $p.i.str$ and its own status. If both p and its right neighbor are **up** then p changes its status to **down**. If p receives a message from its left neighbor and discovers that its neighbors and itself are **down**, it changes its own status to **up**. The operation of s-Corona is illustrated in Figure 4.



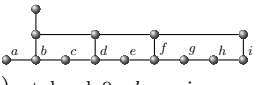
(a) initial state



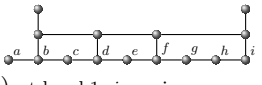
(b) at level 0, processes d and h receive messages that their right neighbors are **up**, they change their statuses to **down**



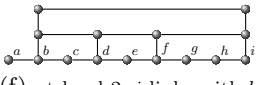
(c) at level 0, e receives message from f that its status is **up** and changes its own status to **down**; f and i are linked at level 1



(d) at level 0, d receives messages that both c and e are **down** and changes its status to **up**, links with neighbors at level 1



(e) at level 1, i receives message from f that its status is **down**, updates its own status to **up**



(f) at level 2, i links with b

Fig. 4. Example computation of s-Corona. For simplicity, neighbor links are always assumed bidirectional.

Correctness Proof. Due to the lack of space the actual proofs in this section are relegated to the technical report [17].

Lemma 9. *If process a at level i of sd-Corona changes its status st only a finite number of times in the computation, then this computation contains a suffix where every message in the outgoing channel of a carries the same value as a.i.st and b.i.str = a.i.st for the left neighbor b of a.*

Proposition 1. *If, in some computation, none of the processes at some level i change their status, then this computation also contains a suffix where for each process a, a.i.r and a.i.l point to the nearest up process at this level and do not change.*

Lemma 10. *If in some computation none of the processes at some level $i-1$ change their right and left neighbors, then this computation also contains a suffix where none of the processes at level i change their status.*

Lemma 11. *In each computation of s-Corona, every process p changes its status and its left and right neighbors only finitely many times.*

Theorem 4. *s-Corona is a weakly channel-connectivity existing identifiers stabilizing solution to the 1-2 skip list construction problem.*

6 Skip Graph

In closing we would like to describe the extension of Corona to skip-graph. The skip list may not be robust or convenient for concurrent searches. Indeed, a failure of a single top-level node may disconnect the system. A *k-l skip graph* [3], the processes at level $i-1$ that do not participate at level i, form an alternative list at level i. The process continues recursively both at the main as well as at the alternative list. That is, each list splits into several at each level. This way, most nodes have links at all levels of the skip graph. This property makes skip graphs more robust and better suited for concurrent searchers than skip lists.

Corona can be extended to construct a skip-graph. For that, Corona has to run two instances of sn-corona at each level i. The main instance operates as before, while the alternative instance constructs an alternative list out of the nodes that do not participate in the main list. Note that in the *1-2* skip list, one alternative list can always be constructed. An instance of sd-Corona at level $i+1$ runs each of the lists. The process of splitting into main and alternative list continues iteratively on each thus formed list. No changes are required in either l-Corona or sd-Corona.

References

1. Alima, L.O., Haridi, S., Ghodsi, A., El-Ansary, S., Brand, P.: Position paper: Self-.properties in distributed k-ary structured overlay networks. In: Proceedings of SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems (May 2004)

2. Andersen, D., Balakrishnan, H., Kaashoek, F., Morris, R.: Resilient overlay networks. In: SOSP 2001: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, pp. 131–145. ACM, New York (2001)
3. Aspnes, J., Shah, G.: Skip graphs. ACM Transactions on Algorithms 3(4), 37:1–37:25 (2007)
4. Awerbuch, B., Scheideler, C.: The hyperring: a low-congestion deterministic data structure for distributed environments. In: SODA 2004: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 318–327. Society for Industrial and Applied Mathematics, Philadelphia (2004)
5. Berns, A., Ghosh, S., Pemmaraju, S.V.: Brief announcement: a framework for building self-stabilizing overlay networks. In: Proc. of the 29th ACM Symp. on Principles of Distributed Computing (PODC), pp. 398–399 (2010)
6. Bhargava, A., Kothapalli, K., Riley, C., Scheideler, C., Thober, M.: Pagoda: a dynamic overlay network for routing, data management, and multicasting. In: SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 170–179. ACM, New York (2004)
7. Bianchi, S., Datta, A., Felber, P., Gradinariu, M.: Stabilizing peer-to-peer spatial filters. In: ICDCS 2007: Proceedings of the 27th International Conference on Distributed Computing Systems, p. 27. IEEE Computer Society Press, Washington, DC, USA (2007)
8. Caron, E., Desprez, F., Petit, F., Tedeschi, C.: Snap-stabilizing prefix tree for peer-to-peer systems. Parallel Processing Letters 20(1), 15–30 (2010)
9. Clouser, T., Nesterenko, M., Scheideler, C.: Tiara: A Self-stabilizing Deterministic Skip List. In: Kulkarni, S.S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 124–140. Springer, Heidelberg (2008)
10. Cramer, C., Fuhrmann, T.: Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe (2005)
11. Dijkstra, E.W.: Self-stabilization in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
12. Gall, D., Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: Time complexity of distributed topological self-stabilization: The case of graph linearization, pp. 294–305 (2010)
13. Harvey, N.J.A., Jones, M.B., Saroiu, S., Theimer, M., Wolman, A.: Skipnet: a scalable overlay network with practical locality properties. In: USITS 2003: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems, p. 9. USENIX Association, Berkeley (2003)
14. Hérault, T., Lemarinier, P., Peres, O., Pilard, L., Beauquier, J.: Brief Announcement: Self-stabilizing Spanning Tree Algorithm for Large Scale Systems. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 574–575. Springer, Heidelberg (2006)
15. Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: Proc. of the 28th ACM Symp. on Principles of Distributed Computing (PODC), pp. 131–140 (2009)
16. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: a scalable and dynamic emulation of the butterfly. In: PODC 2002: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, pp. 183–192. ACM, New York (2002)
17. Nor, R., Nesterenko, M., Scheideler, C.: Corona: A stabilizing deterministic message-passing skip list. Technical Report TR-KSU-2011-01, CS Dept., Kent State University (May 2011)

18. Onus, M., Richa, A., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia (2007)
19. Onus, M., Richa, A., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: ALENEX 2007: Proceedings of the Workshop on Algorithm Engineering and Experiments. SIAM, Philadelphia (2007)
20. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM 33(6), 668–676 (1990)
21. Scheideler, C., Jacob, R., Ritscher, S., Schmid, S.: A self-stabilizing and local delaunay graph construction. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 771–780. Springer, Heidelberg (2009)
22. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: SIGCOMM 2001: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 161–172. ACM, New York (2001)
23. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Liu, H. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
24. Shaker, A., Reeves, D.S.: Self-stabilizing structured ring topology P2P systems. In: Proc. 5th IEEE International Conference on Peer-to-Peer Computing, pp. 39–46 (2005)
25. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for Internet applications. IEEE / ACM Transactions on Networking 11(1), 17–32 (2003)

Using Zero Knowledge to Share a Little Knowledge: Bootstrapping Trust in Device Networks

Ingy Ramzy and Anish Arora

Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio, 43210
{youssef,anish}@cse.ohio-state.edu

Abstract. In device networks, trust must often be established in the field despite limited a priori knowledge of the network and the possibility of adversaries in the network environment. This paper presents a solution to the problem of bootstrapping trust that is minimal in the sense that it circumvents ongoing maintenance of security material. Specifically, security material is communicated to members of a device group just once by using zero knowledge identification in a new and efficient way, whereby devices in the group may henceforth securely verify each other as well as initialize mutual keys for confidentiality without needing to update that security material over time. In its basic form, the solution uses a base station to communicate the security material for group membership verification. The solution allows for scaling by letting the base station hierarchically delegate the task of bootstrapping to subordinate trusted nodes.

Keywords: Device Networks, Trust, Bootstrap, Zero-Knowledge, Diffie-Hellman, Secure Group Membership, Key Distribution.

1 Introduction

System security today relies essentially on the use of shared keys. Key management however incurs significant human involvement. For the case of device networks—whose numbers, size, and day-to-day influence is growing rapidly—it is impractical to sustain the status quo. Solutions that automate and/or reduce key management are therefore important.

An extreme position is to explore solutions that use no shared keys. Along these lines, recent work, including some of ours [18,3], has considered the alternative of eschewing shared keys via physical layer security. The central idea is to realize a seminal result from information theory [22], which has been underexploited in practice to date, via a number of physical primitives for realizing security properties such as confidentiality, authentication, etc. without relying on shared keys.

Nonetheless, consideration of bootstrapping these physical primitives for security reveals an interesting chicken-and-egg problem: Even though physical primitives themselves do not use shared keys, bootstrapping —as well as maintaining and restoring— security material for enabling physical primitives itself assumes that inter-node trust relationships have been established beforehand. For example, instantiation of confidentiality, say using dialog codes [3], assumes a mechanism by which the sender and the

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 371–385, 2011.

receiver can authenticate each other and agree to launch cooperative jamming. Shared keys may be used for bootstrapping confidential communications. Alternatively, we could use physical signatures/fingerprints as the security material for authentication, if we wished to avoid shared keys. But in that case, the instantiation of physical signatures in turn assumes there is a mechanism for coordination between nodes that trust each other whereby the signature is learned. In sum, bootstrapping of shared-key-free physical primitives itself assumes shared keys!

For what sorts of trust relationships, then, should we bootstrap a device with shared keys? One is a device's trust in the network that it itself is to be a part of. And two, its trust in one or more devices in the network with which it is to communicate as part of some common application. While the former can be bootstrapped before devices are deployed, the latter is typically bootstrapped after devices are deployed, as which particular devices will end up communicating with each other is often not known a priori. In any case, both of these sorts can be abstracted as trust in some group of devices. And the shared keys established for the former can be used to establish shared keys for the latter.

Our Contributions. In this paper, we present a solution to the problem of trust bootstrapping in device networks. As our goal includes reducing the key management overhead, we adopt the position that all of the shared key material available a priori for trust establishment is in some sense minimal. In particular, for the first sort of trust, nothing is needed a priori, since this can be performed pre-deployment. For the latter sort of trust, we show that it suffices for each device to share pre-deployment only two keys with a well known device, namely the base station. These two keys are persistent; i.e., they do not need to be updated.

The security of using shared keys typically degrades over time. The idea central to our eschewing the need to update these two pre-shared keys is to use them in a zero knowledge fashion. More specifically, we use them with a *zero knowledge identification scheme, to serve as the basis for communicating security material and thereby to enable the sharing of secrets*. Our solution is distinguished by the following properties:

- In its basic form, it uses a base station to communicate to group members the relevant security material (i.e., group permissions). A straightforward extension of it allows for scaling by letting the base station hierarchically delegate the task of bootstrapping to subordinate trusted nodes.
- It is collusion resistant in the sense that compromising any device j does not compromise the network's ability to bootstrap devices other than j with new group permission security material.
- It uses standard cryptographic constructs, so it is flexible enough to allow plug in of different realizations of these constructs, including ones which are more efficient or are more secure.
- It uses these constructs in a non standard, efficient way. That is, it uses zero knowledge (ZK) identification schemes to communicate group permission secrets, and to construct Diffie-Hellman keys (DH) without exchanging key parts. The secret lies in the order and the way of verifying the ZK proof, which involves the use of the two

pre-shared keys. To the best of our knowledge, ZK schemes have not been used in this way before. Efficiency follows from eschewing the need for many rounds in the ZK scheme as well as reducing the overhead of DH.
– It bootstraps trust in levels, by developing a trust hierarchy wherein trust can propagate top down.

Organization of the Paper. In Section 2, we describe the system and adversary model, and present the problem of bootstrapping trust. We then present our solution in terms of a trust hierarchy and corresponding protocols, in Section 3. In Section 4, we discuss implementation issues, performance costs and distinctive properties. We contrast our work with related work in Section 5. Finally, in Section 6, we make concluding remarks and discuss future work.

2 System Model and the Trust Bootstrap Problem

Network Model. Given is a set of device nodes and a base station (BS). Nodes can communicate with one another and with BS; communication with BS may involve one or more hops. The network is a programmable "fabric" in the sense that multiple applications (aka, groups) may coexist. Associated with each application is a subset of network nodes, which are to execute that application. We assume that application groups are not known a priori, i.e., before the network is deployed.

Attacker Model. Our attack model includes the Dolev-Yao attacker: the attacker can overhear, intercept, and synthesize any message but is limited by the constraints of the cryptographic methods used. In addition, our attack model includes the compromise of any node. When a node is compromised, its state and all of its secrets and programs are available to the attacker. The attacker may use the material and programs of its own, for instance, to collude with other compromised nodes and/or impersonate other nodes. During a protocol execution, the attacker may execute multiple instances of the protocol scheduled in an arbitrary way. We assume however that BS cannot be compromised.

Trust Bootstrap Problem. Given is a subset of nodes, G, in the network W. Required is a protocol to initialize security materials (e.g., shared keys) in each node in G so that *only nodes in G can successfully test each other's membership in G*. That is, a node j can prove that j is in G to a node i only if j is in G and, conversely, i can verify that j is in G only if i is a member of G, thus, trusted nodes in G can authenticate communications from other trusted nodes in G.

Solution Considerations. Standard approaches to establish a secure group include (i) using pre-shared keys or keying materials, (ii) group nodes exchanging information with their immediate neighbors, or (iii) group nodes exchanging information with computationally robust nodes, for example, BS. The number of pre-shared keys per device typically depends on the size of the network, so scalability can be a problem with pre-shared keys. The idea of random key distribution yields indeterministic solutions in terms of sufficing for groups, which makes it unsuitable for our case since node groups

are not known a priori. The approach of exchanging information with immediate neighbors is inappropriate given the system model, since nodes can not trust any other node in the fabric a priori. The approach of using BS is feasible, in part because the BS is always trusted. The use of BS however risks becoming a bottleneck and a single point of failure. Ideally, one would like to design alternatives where BS can delegate its task to nodes it trusts so as to reduce the overhead of potentially multi-hop communications as well as to obtain robustness to BS failure.

3 Zero Knowledge Approach for Bootstrapping Trust

Our bootstrapping approach makes use of pre-shared keys in the following minimal sense. Every node shares a symmetric key with BS just once. BS uses this key to deliver group permission secrets to the node. The node in turn can use these permissions to establish trust with other nodes in the groups. If a pre-shared key is used to directly encrypt information, its security degrades over time and so it should not be used indefinitely. For the security of the key to be preserved forever, our approach builds upon that of Zero Knowledge (ZK).

Table 1. Notation

i, j	Nodes in W
BS	Base station
r, n	Randoms
S, v	Main secrets, unique and private to each node
g	Group permission secret, local and private to members of G
$k(m)$	Encryption of m using a symmetric key k
α	Group generator
x	Large prime that is global and public
t	Time stamp
f, T	Special purpose functions
k_a	Diffie-Hellman key part, constructed using the secret exponent a
$k_{a,b}$	Diffie-Hellman key, constructed using the secret exponents a and b

Recall the standard form of the ZK identification schema [8,5], cf. Figure 1: In $ZK[r,n,S]$, a prover P sends a commit, $f(r)$ where f is a one way function, and then based on receiving the challenge n from the verifier V, sends a response $f(r,n,S)$ to V in order to prove to V that P knows S. This process does not reveal any knowledge including S beyond the validity of the proof to any node other than P.

Using ZK lets us guarantee that the symmetric key shared between BS and each node does not degrade over time. More specifically, our solution extends the standard ZK identification schema to also securely communicate group permission secrets to nodes in G. Moreover, our solution is designed such that BS is not aware of group/pairwise keys established within the groups, but rather just the permission secrets that enable nodes to authenticate their membership in the group.

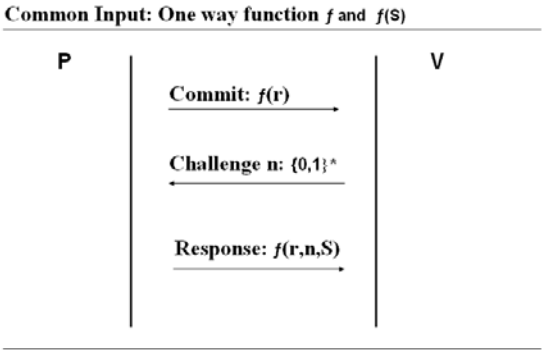


Fig. 1. Zero-Knowledge Identification Schema: $ZK[r, n, S]$

3.1 Trust Levels: The Architecture

Our solution bootstraps trust in levels, resulting in a trust hierarchy. At the top of the hierarchy is BS, which represents the device network. Lower levels of the hierarchy correspond to trust in members of groups and/or subgroups in the superset of groups **G** in W. Corresponding to the levels in the hierarchy are different types of secrets, namely: main, permission, group, and pairwise, defined as follows.

Main Secrets. Each node j has two unique secrets, v and S, that j shares with the BS. These are persistent secrets that can be used to bootstrap j any number of times for membership in a number of groups of W.

Permission Secrets. Each group G, $G \in \mathbf{G}$, has a single unique secret g shared by all nodes $\in G$, which is used solely for authenticating membership in G. The group permission secret g is not used directly for encrypting any exchanges, hence it can be used infinitely in G as long as nodes of G are not compromised.

Group Secrets. Each group G, $G \in \mathbf{G}$, has zero or more secrets, $\hat{S}$, that are shared by all or some nodes $j \in G$, and used say to authorize service requests, provide services, relay communications, receive results, or instantiate physical primitives. Nodes may use these secrets explicitly in encryption, hence they may need to update them from time to time.

Pairwise Secrets. Each node pair, $i, j \in G$, has zero or more pairwise keys. Again, nodes may use these secrets explicitly in encryption, hence they may need to update them from time to time.

First Level of Trust (TL1). For each node j, j and BS can mutually authenticate.

- TL1 is established pre-deployment by configuring j with the main secrets v and S shared by the BS.
- S is used in a ZK way only, s.t. while j is capable of extracting knowledge from the ZK proof using S, for all nodes other than j, the proof is zero knowledge. Also v is never used directly in any exchanges.

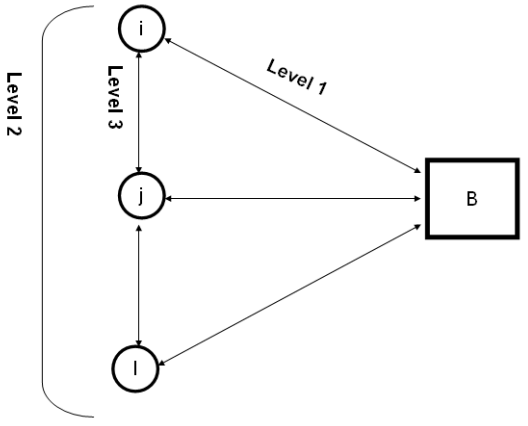


Fig. 2. Levels of Trust

– BS can use the main secrets to deliver permission secrets to j for joining a group $G \in \mathbf{G}$.

Second Level of Trust (TL2). For each node j, nodes in $G \in \mathbf{G}$ can authenticate j iff j is in G, based on TL1:

– TL2 is established in the field, based on knowing the permission secret g.
– g can be updated if some node of G is (suspected of being) compromised.
– BS maintains the permission secrets of each defined group $G \in \mathbf{G}$.

Third Level of Trust (TL3). For each node pair $i, j \in G$, i and j can establish group secrets and/or pairwise secrets, based on TL2.

– TL3 is established in the field.
– Group secrets, $\hat{S}$, can be communicated as part of the authentication of i by j, or alternatively using pairwise keys.
– Pairwise keys can be used to update g as needed.

3.2 Setup and Secrets Generation

– ZK and DH parameters: Depending on the particular ZK and DH protocol realizations chosen (i.e., ECC vs standard), relevant parameters are generated. The parameters are global and generated pre-deployment.
– S: A node's symmetric secret, pre-configured and shared with BS. The security properties of S are dependent on the particular ZK scheme realization that is selected.
– v: A symmetric randomly chosen secret, $1 \leq v \leq x-1$, that is pre-configured in the node and shared with BS.
– g: Let x be the DH large prime. The permission secret g, $1 \leq g \leq x-1$, is generated by BS for any new group G.
– k_{r_j}: For each node j, a static DH key part is generated pre-deployment using some random secret exponent, r_j. This is used only in TL3 to reduce the overhead of DH.

3.3 Trust Protocols

TL1 is established pre-deployment, hence every node j in W is capable of verifying BS exchanges. Nonetheless, any node i, $i \neq j$, may be able to verify a BS exchange to j but would not be able to extract any knowledge from it.

A Protocol Using TL1 to Establish TL2. The following protocol uses the secrets S and v established in TL1 to establish TL2 by communicating the permission secret g of a group $G \in \mathbf{G}$ to node j.

$$r' = T(r) \tag{1}$$

TL2 Protocol

$$BS \longrightarrow j \ : \ ZK[r,n,S] \ ; \ k_{r',v}(g,j,n,t)$$

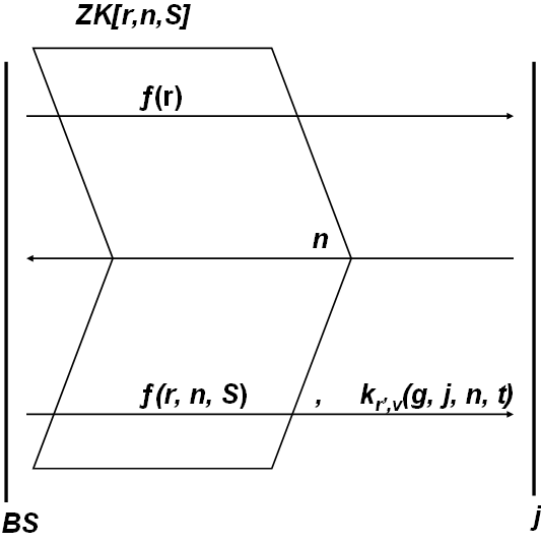


Fig. 3. TL2 Protocol

Let T be some transformation mapping a random number from one range to another while preserving the randomness property. To bootstrap a node j, the protocol uses a standard ZK identification session, $ZK[r,n,S]$ between BS and j. The ZK commit in terms of the random r along with the challenge n and main secret S formulates the BS response. As shown in Figure 3, BS sends along with the ZK response, an encrypted message, $k_{r',v}(g,j,n,t)$, to j containing the permission secret g along with some other parameters to tie the message to the proof. The encryption key $k_{r',v}$ is a DH key, formulated using r' and v as the secret exponents. Note that none of the DH key parts are exchanged in public. r', the value of the ZK random r mapped onto the DH ring using T, along with the precomputed DH key part using v, will never be known except to j.

j uses its main secret S to extract r during the proof verification process. Note that S is known only to j and BS, thus for all nodes other than j, the proof is zero knowledge. j then uses r', along with its local precomputed key part in terms of v, to construct the key $k_{r',v}$ for decrypting $k_{r',v}(g,j,n,t)$.

Property 1. *The value of r' is random in the DH range:* $[1, x-1]$.

Theorem 1. *The zero knowledge property of $ZK[r,n,S]$ is preserved by Protocol TL2.*

Proof Outline. Assume the existence of an efficient oracle O that, given an encrypted message, returns the key used for encryption. An adversary $\hat{A}$ could query O for $k_{r',v}$ (g,j,n,t) and be given $k_{r',v}$. Note that the key's exponent, $r'.v$, would be some random in the DH ring, since both r' and v are randoms. Finding $r'.v$ implies solving the Discrete Log problem, which is assumed to be intractable. We hence have a contradiction that proves the hypothesis.

Theorem 2. *It is computationally hard to deduce the secret v of any node j regardless of how many times Protocol TL2 is executed.*
The proof of Theorem 2 closely follows that of Theorem 1.

A Protocol Using TL2 to Establish TL3. TL3 lets a node in G prove its membership to any other member node. This allows members of G to exchange group and pairwise secrets as need be.

TL3 Protocol

$$i \longrightarrow j \ : \ k_{r_i} \ , \ k_{r_i,g}(i,j,t,\hat{S})$$

Node i sends a message encrypted with a DH key $k_{r_i,g}$ constructed using g and its private random r_i corresponding to its precomputed static DH key part k_{r_i}. Note that the DH key part in terms of g is never communicated in the clear, hence, only G's nodes are capable of composing the key $k_{r_i,g}$. This simple one-way exchange serves for two-way membership authentication due to the ability of both i and j to successfully construct such a key. Note that we do not require that nodes store the DH key parts of other nodes, so, i has to send its precomputed key part along with the membership proof. $\hat{S}$, a group secret communicated from i to j can be used in a number of ways. Note also that based on successful membership authentication, i and j can establish a pairwise key using the precomputed DH key parts.

4 Implementation and Analysis

In this section, we propose possible realizations of Protocol TL2 using different ZK identification schemes that cater to devices which are resource constrained or which demand a high level of security. We then develop a simple cost model for estimating the overheads of Protocols TL2 and TL3. Finally, we discuss the distinctive properties of our solution, towards arguing that the potential merits of our approach justify the cost overheads, which per se are not high.

4.1 Protocols Realizations

We consider three realizations for Protocol TL2. One uses the Guillou-Quisquater (GQ) ZK identification scheme [9], which is based on the hardness of the RSA problem, the other two use the Elliptic Curve (ECC) versions of Schnorr (SC) [21,19] and Okamoto's (OK) ZK [4,15] identification schemes, which are based on the discrete log assumption. The base versions of GQ-ZK and SC-ZK are known for their efficiency, while OK-ZK scheme is known for its security features. Device and deployment constraints would dictate which of these to use in particular contexts.

GQ TL2 Protocol. GQ-ZK requires some parameters that are global to the network: w, the RSA modulus, and e, a prime RSA exponent. Local parameters include the node's main secret $S \in {Z_w}^*$ that is shared with BS. α is the chosen public DH generator. Let T be some transformation scaling randoms in the range $[1, w-1]$ to the range $[1, x-1]$ while preserving the randomness property. Note that numerous efficient implementations exist for similar transformations, also based on how both ranges compare, the transformation can be as simple as taking the modulus, i.e. mod x. In our protocol, BS acts as a prover while a node j, which is to be bootstrapped to join a group G, is the verifier. In the protocol, detailed below, BS selects a random $r \in {Z_w}^*$ and sends a commitment to j which replies back with the challenge n. Finally BS sends its response to the challenge along with an encrypted message. The key used for encryption can be viewed as another challenge because it is not known to the node (i.e. not pre-shared), nor is it based on any key parts communicated in the clear. This key is constructed in terms of r' and v, where r' is the mapping of r onto the DH group using T. Hence, the only way j could construct this key is to use its secret S to extract the random from the BS proof. Let R be the commit, C be the response, and $D \equiv S^e (mod w)$. The GQ-ZK acceptance condition then is $C^e \equiv RD^n (mod\ w)$. In our protocol, acceptance is reduced to checking if $R \equiv r^e (mod\ w)$, which is attributed to j's ability to learn r from the proof.

Protocol for TL2 establishment: GQ version

$$BS \longrightarrow j \ : \ r^e \ (mod\ w)$$

$$j \longrightarrow BS \ : \ n$$

$$BS \longrightarrow j \ : \ rS^n \ (mod\ w),\ k_{r',v}(g, j, n, t)$$

ECC-SC TL2 Protocol. ECC-SC global parameters include: q, which specifies the finite field, a, b, which define an elliptic curve, P, a point on the curve of order x, and the cofactor h. Let $D = -S.P$, where S is a main secret of j. The SC-ZK standard verification would be: if $C.P + n.D = R$ accept else reject, where C is the response, R is the commit and n is the challenge. On the other hand, our protocol uses S to extract r and verify that $R = r.P$. j then uses its secret $v.P_1$ and performs a single multiplication by r to derive the challenge key $k_{r,v}$. Notice that in this protocol version, ECC-DH is used, where P_1 is a public high order base point of the aforementioned chosen curve. If

we choose the order of P_1 to be the same as P, then Property 2 would be a necessary condition for correctness, since we require anonymity of the key parts composing $k_{r,v}$. Note that in this case, no transformations would be needed for r. A safer option though would be to choose a different order for P_1, then a scaling transformation, similar to the one used in the GQ TL2 Protocol, would be needed to map r from the range defined by the order of P to that of P_1.

Property 2. *For DH generators P_1 and P of order x, given $r.P$ for some index r, it is hard to find $r.P_1$ without knowledge of r.*

$$k_{r,v} = r.v.P_1 \tag{2}$$

Protocol for TL2 establishment: ECC-SC version

$BS \longrightarrow j : r.P$

$j \longrightarrow BS : n$

$BS \longrightarrow j : Sn + r \ (mod\ x),\ k_{r,v}(g, j, n, t)$

ECC-OK TL2 Protocol. For OK-ZK, the public global parameters are q, which specifies the finite field, a, b, which define an elliptic curve, P_1 and P_2, points on the curve of order x, and the cofactor h. In this case, our protocol uses a pair (S_1, S_2) as a main secret. Let $D = -S_1.P_1 - S_2.P_2$, a node j accepts the proof iff $C_1.P_1 + C_2.P_2 + n.D = R$, where C_1 and C_2 are the ZK responses of BS, R is the commit and n is the challenge. Here, we have two randoms r_1, r_2, so one choice would be to use a function f that maps the two randoms into a single random. This single random is then used by the node to construct the DH key, $k_{r',v}$, for decrypting the message $k_{r',v}(g, j, n, t)$. Note that it is safe to choose order x for the DH base point P_3, since even if a trivial isomorphic mapping from P_1 and P_2 onto the group generated by P_3 exists, then based on the DH assumption, given $r_1.P_3$ and $r_2.P_3$ it would be hard to compute $r'.P_3$, assuming f is not a trivial addition for example. Note that it is also hard to extract $r_1.P_1$ and $r_2.P_2$ from the commit in the first place.

$$r' = f(r_1, r_2) \tag{3}$$

$$k_{r',v} = r'.v.P_3 \tag{4}$$

Protocol for TL2 establishment: ECC-OK version

$BS \longrightarrow j : r_1.P_1 + r_2.P_2$

$j \longrightarrow BS : n$

$BS \longrightarrow j : S_1 n + r_1 \ (mod\ x),\ S_2 n + r_2 \ (mod\ x),\ k_{r',v}(g, j, n, t)$

Security Features of the Realizations. OK-ZK is provably secure against impersonation under active and passive attacks. GQ-ZK and SC-ZK schemes are known for their efficiency, and also to be secure against impersonation under passive attacks, assuming honest verifiers. In addition, in [6], the authors extend the results of GQ-ZK to include security against active attacks based on the assumed security of RSA under one more inversion. The authors also provide such a proof for SC-ZK based on a corresponding discrete-log related assumption, they further extend their proof to establish security against impersonation under concurrent attack for both schemes. In our protocols, we only change the order in which the node verifies the proof, such that using S, the node can extract the random and verify it against the commitment. Next, the node uses the extracted random to construct the challenge key for decrypting the group permission message. Note that no key parts are sent in the clear, and the challenge key is constructed using the ZK random and the main secret v.

4.2 Performance Evaluation

Our trust hierarchy uses ZK identification, DH key agreement protocol and symmetric encryption/decryption. In evaluating the performance of protocols TL2 and TL3, if one were to consider a baseline for comparison which used only symmetric keys explicitly, one would have to model the cost of operations which are performed on an ongoing basis to update these keys. Rather than do so, we focus on estimating the overheads of ZK and DH as they are used by the protocols. We show how the choice of the cryptographic constructs and the pre-calculation of some values can decrease the expected overhead.

We define a cost function Ψ for Protocol TL2. The cost of establishing TL2 for a node j in group G is given by the following equation.

$$\Psi_j(G) = \delta_{ZK} + \delta_{DH} + \delta_{sym} \tag{5}$$

Where $\delta_{ZK}, \delta_{DH}, \delta_{sym}$ define the costs of ZK identification scheme, DH key construction, and symmetric encryption/decryption, respectively. The cost of establishing TL2 for a group G of n nodes is given by:

$$\Psi(G) = \Pi_j{}^n \Psi_j(G) \tag{6}$$

Next, we define a cost function ω for Protocol TL3. The cost of establishing TL3 for a node j relative to a group G of n nodes is given by:

$$\omega_j(G) = \delta_{DH} + \delta_{sym} \tag{7}$$

Assuming ECC-ZK and ECC-DH, we can express δ_{ZK} and δ_{DH} in terms of the number of scalar point multiplications. Let Γ be the cost of performing a single scalar point multiplication. Using static pre-computed DH key parts brings down δ_{DH} to a single ECC point multiplication ($\delta_{DH} = \Gamma$). In ZK schemes, we are interested in the verifier load share only, which is the node to be bootstrapped. The verification of SC-ZK requires two point multiplications ($\delta_{ZK} = 2\Gamma$), while OK-ZK requires four ($\delta_{ZK} = 4\Gamma$). This totals to an overhead of three point multiplications for ECC-SC TL2 ($\Psi_j(G) = 3\Gamma + \delta_{sym}$), versus five for ECC-OK TL2 ($\Psi_j(G) = 5\Gamma + \delta_{sym}$). TL3 has an overhead of a single point multiplication ($\omega_j(G) = \Gamma + \delta_{sym}$). Note that for TL2, a node can have some

pre-computed $S.n$ values, where S is one of the node's main secrets, and n is the ZK challenge, hence bringing down the overhead to two point multiplications for ECC-SC TL2 ($\Psi_j(G) = 2\Gamma + \delta_{sym}$), versus three for ECC-OK TL2 ($\Psi_j(G) = 3\Gamma + \delta_{sym}$).

Several optimized implementations exist for point multiplications [20]. For example, an elliptic curve implementation over a 192-bit prime field for MICAz motes [11] yielded a full scalar multiplication in 0.71 sec (5.20 Â· 10^6 cycles) when the base point is fixed and known a priori. Note that we are not restricting the protocols to specific ZK schemes, instead, the realizations serve to demonstrate the flexibility of adapting different constructs to the basic schema.

4.3 Features and Merits

Lastly, we discuss some of the distinguishing features of our bootstrap approach and the related protocols.

ZK Made More Efficient. ZK protocols have lighter computational requirements than public key protocols since ZK uses an iterative process involving lighter transactions, thereby achieving its result with one to two orders of magnitude less computing power. A typical implementation might require 20–30 modular multiplications that can be optimized to 10–20 with pre-calculation. This is much faster than RSA [2]. In our protocols, more specifically in Protocol TL2, we use ZK in a way that makes a single round sufficient.

Theorem 3. *One ZK round is sufficient for TL2 establishment.*

Proof. For example, assume GQ-ZK. Let R be the commit defined in terms of the random r, C be the response, and $D \equiv S^e \, (mod \; w)$. The GQ-ZK acceptance condition is $C^e \equiv RD^n \, (mod \; w)$. An adversary $\hat{A}$ trying to impersonate BS to some node j can guess the challenge and prepare a commit: RD^{-n} using its guess. $\hat{A}$ would then send r as the response. If $\hat{A}$'s guess was correct, then j would successfully verify the ZK proof, and the extracted value would be: $r.S^{-n}$. Nonetheless, $\hat{A}$ would fail to construct the challenge key that j would derive. This is because, for $\hat{A}$, this key uses the extracted value $r.S^{-n}$, which involves the unknown secret S, along with v, which is also unknown.

DH with No Public Exchanges. Our protocols do not transfer key parts explicitly, yet the recipient node is capable of building a one time session key with BS to extract the permission secret at no additional overheads except for the extra bits of information sent and the single point multiplication (for ECC-DH). So even if an attacker launches a timing attack on the exchanges, the computing time cannot be related to particular values, since no key parts are sent in the clear. In TL3, we relax our requirements slightly by assuming that one of the key parts is known, nonetheless, the other key part involving g is never communicated in the clear, again elevating the security of the scheme.

Collusion Resistance. We define collusion resistance as follows: For all nodes j in W, the compromise of j does not affect the network's ability to bootstrap any other node i.

For example, if BS shares the same key with all nodes in W, then the compromise of a single node directly affects the network's ability to bootstrap any other node, since the compromised node can easily learn the communicated secrets. On the other hand,

if nodes share unique secrets with BS, but symmetric key or public key primitives are used, then if those keys were not periodically updated, they can degrade over time, and a compromised node which gets actively involved in intercepting the messages sent to other nodes could successfully compromise the nodes' secrets.

Theorem 4. *The bootstrap approach is collusion resistant.*

Proof. For each node j, j shares unique main secrets with BS, that are used in a zero knowledge way. Based on the assumption that keys used in ZK schemes do not degrade over time, along with Theorem 1 and Theorem 2, we can conclude that no nodes other than j can learn any of j's main secrets during the bootstrap of j, so these secrets can be used indefinitely to bootstrap j with new group secrets.

Scalability. Our bootstrap approach is made scalable by allowing BS to delegate trust to a selected set of nodes in group G, which in turn act as a trusted base (TB) for that group. The delegation of trust can be achieved by communicating the permission secret g to a selected set of nodes forming the TB of G, following which, BS computes and maintains the key part k_g in terms of g while permanently deleting g. BS can then use k_g to grant permissions to G that are verifiable only by TB of G. Once a node's membership has been verified by a TB node, selected security materials are communicated to this node, consecutively, the node becomes a member of the subgroup of G holding those secrets. It should be noted that if G spans the whole network, the trusted base of G can be viewed as a set of base stations, each of which are capable of bootstrapping any node j in the network. This hierarchical delegation of trust enables the scalability of the proposed bootstrap approach while emphasizing the independence of the trust levels.

5 Related Work

There is rich literature on ZK identification schemes. ZK research in the context of resource constrained networks has focused on optimizing the implementation of these schemes. An illustrative example [1] is a modified version of the Guillou-Quisquater (GQ) identification scheme that is used in conjunction with the ÎŒTESLA protocol [16] to authenticate the base station using a group of nodes. Another example [10] refines the Feige-Fiat-Shamir ZK scheme to reduce the number of challenge-response rounds so as to speed up the authentication process; the refinement uses bursts of parallelism while maintaining serial execution of Feige-Fiat-Shamir in order to preserve the ZK property. Efficient hardware implementation of the ZK identification schemes have been presented in the context of authenticating RFID-tags [21,4]. To the best of our knowledge, however, none of the existing works in the ZK literature on device networks have considered the use of ZK for communicating security material, as we have for the purpose of communicating group permissions as part of the bootstrapping process.

A variety of efforts have attempted to examine the use of public key cryptography in resource constrained networks [17,14,13], and many have focused on the special case of ECC [20,12]. There are a few effort related to optimizing DH implementations for device networks [11,12]. To the best of our knowledge, again, we are not aware of previous work that makes use of DH keys in conjunction with the ZK identification scheme such that key agreement can be accomplished without explicit exchange of key

parts. The approach of assuming pre-shared key parts between individual nodes and BS has been used previously [7], to update the mutual keys by regularly broadcasting fresh key parts, but this lacks authentication. Moreover, unlike our protocol, the updates are sent in the clear, which is vulnerable to timing attacks, especially if the same key part is to be used by all nodes in the network.

6 Conclusions and Future Work

We have argued that bootstrapping device networks motivates new constraints for initializing trust. The model we have proposed assumes that just a couple of secrets are pre-shared between the base station and each node in the field. Our solutions avoid the need to update these secrets over time by using them in a zero knowledge way to grant group permissions to nodes. This is achieved by extending the standard ZK identification schema so that the ZK proof can be used to communicate secrets to a group node, while preserving the zero knowledge property with respect to non-group nodes. The extension is a form of DH key agreement that instead of exchanging key parts, relies solely on the ZK proof and a single pre-shared secret. We proposed a protocol suite for the evolution of trust in levels which could lead up to the instantiation of physical primitives, thereby yielding a system where key management is largely automated and rarely invoked.

We are investigating whether a single pre-shared secret can suffice. For the case where a group member is suspected of being compromised, we are investigating ongoing management of group secrets that provides a lightweight way of revoking trust such that the revoked node cannot determine that it is being removed from a group. A potential approach would be to use this node to inform group members of the trust revocation and to propagate new permission secrets to them, without it knowing that it is acting as a mediator between BS and the intended group. Thus, our method could maintain symmetry between adding and revoking trust. Finally, we are investigating the incorporation of trust models in our protocols so that a dynamic trust metric is maintained for each node and its ability to gain various secrets depends on this metric.

References

1. Anshul, D., Roy, S.: A zero-knowledge-proof identification scheme for base nodes in wireless sensor networks. In: ACM Symposium on Applied Computing, pp. 319–323 (2005)
2. Aronssona, H.A.: Zero knowledge protocols and small systems, http://www.tml.hut.fi/Opinnot/Tik-110.501/1995/zeroknowledge.html
3. Arora, A., Sang, L.: Dialog codes for secure wireless communications. In: IPSN, pp. 13–24 (2009)
4. Batina, L., Guajardo, J., Kerins, T., Mentens, N., Tuyls, P., Verbauwhede, I.: Public-key cryptography for rfid-tags. In: PerCom Workshops, pp. 217–222 (2007)
5. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 390–420. Springer, Heidelberg (1993)
6. Bellare, M., Namprempre, C., Neven, G.: Security proofs for identity-based identification and signature schemes. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 268–286. Springer, Heidelberg (2004)

7. Chung, A., Roedig, U.: DHB-KEY: An efficient key distribution scheme for wireless sensor networks. In: Proceedings of the 4th IEEE International Workshop on Wireless and Sensor Networks Security, WSNS 2008 (2008)
8. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing 18(208) (1989)
9. Guillou, L.C., Quisquater, J.: A Paradoxical identity-based signature scheme resulting from zero-knowledge. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 216–231. Springer, Heidelberg (1990)
10. Kizza, J.M.: Feige-fiat-shamir zkp scheme revisited. International Journal of Computing and ICT Research 4(1), 9–19 (2010)
11. Lederer, C., Mader, R., Koschuch, M., Großschädl, J., Szekely, A., Tillich, S.: Energy-efficient implementation of ECDH key exchange for wireless sensor networks. In: Markowitch, O., Bilas, A., Hoepman, J.-H., Mitchell, C.J., Quisquater, J.-J. (eds.) WISTP 2009. LNCS, vol. 5746, pp. 112–127. Springer, Heidelberg (2009)
12. Liu, A., Ning, P.: Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In: IPSN, pp. 245–256 (2008)
13. Malan, D.J., Welsh, M., Smith, M.D.: Implementing public-key infrastructure for sensor networks. TOSN 4(4) (2008)
14. Munivel, E., Ajit, G.M.: Efficient public key infrastructure implementation in wireless sensor networks. In: ICWCSC, pp. 1–6 (2010)
15. Okamoto, T.: Provably secure and practical identification schemes and corresponding signature schemes. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 31–53. Springer, Heidelberg (1993)
16. Perrig, A., Canetti, R.R., Tygar, J.D., Song, D.: The tesla broadcast authentication protocol. In: CryptoBytes, pp. 2–13 (2002)
17. Roman, R., Alcaraz, C.: Applicability of public key infrastructures in wireless sensor networks. In: López, J., Samarati, P., Ferrer, J.L. (eds.) EuroPKI 2007. LNCS, vol. 4582, pp. 313–320. Springer, Heidelberg (2007)
18. Sang, L., Arora, A.: A shared-secret free security infrastructure for wireless networks. To appear in ACM Transactions on Autonomous and Adaptive Systems
19. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: CRYPTO (1990)
20. Szczechowiak, P., Oliveira, L.B., Scott, M., Collier, M., Dahab, R.: Nanoecc: Testing the limits of elliptic curve cryptography in sensor networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 305–320. Springer, Heidelberg (2008)
21. Tuyls, P., Batina, L.: RFID-tags for anti-counterfeiting. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 115–131. Springer, Heidelberg (2006)
22. Wyner, A.D.: The wire-tap channel. Bell Syst. Tech. 1355–1387 (1975)

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

1 Introduction

Replication and consistency are essential features of any large distributed system, such as the WWW, peer-to-peer, or cloud computing platforms. The standard "strong consistency" approach serialises updates in a global total order [10]. This constitutes a performance and scalability bottleneck. Furthermore, strong consistency conflicts with availability and partition-tolerance [8].

When network delays are large or partitioning is an issue, as in delay-tolerant networks, disconnected operation, cloud computing, or P2P systems, *eventual consistency* promises better availability and performance [17,21]. An update executes at some replica, without synchronisation; later, it is sent to the other

* This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208). Marek Zawirski is supported in part by his Google Europe Fellowship in Distributed Computing 2010. Nuno Preguiça is partially supported by CITI (PEst-OE/EEI/UI0527/2011). Carlos Baquero is partially supported by FCT project Castor (PTDC/EIA-EIA/104022/2008).

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 386–400, 2011.

replicas. All updates eventually take effect at all replicas, asynchronously and possibly in different orders. Concurrent updates may conflict; conflict arbitration may require a consensus and a roll-back.[1]

This weaker consistency is considered acceptable for some classes of applications. However, conflict resolution is hard. The literature offers little guidance on designing a correct optimistic system. Ad-hoc approaches are brittle and error-prone; witness for instance the concurrency anomalies of the Amazon Shopping Cart [3].

We propose a simple, theoretically-sound approach to eventual consistency. Our system model, Strong Eventual Consistency or SEC, avoids the complexity of conflict resolution and of roll-back. *Conflict-freedom* ensures safety and liveness despite any number of failures. It leverages simple mathematical properties that ensure absence of conflict, i.e., monotonicity in a semi-lattice and/or commutativity. A trivial example is a replicated counter, which (assuming no overflow) converges because its increment and decrement operations commute. In our *conflict-free replicated data types* (CRDTs), an update does not require synchronisation, and CRDT replicas provably converge to a correct common state. CRDTs remain responsive, available and scalable despite high network latency, faults, or disconnection.

Non-trivial CRDTs are known to exist: for instance, we previously published Treedoc, a sequence CRDT for co-operative text editing [14]. Our aim here is to expand our knowledge of the principles and practice of CRDTs. We claim the following contributions for this paper:

- A solution to the CAP problem, Strong Eventual Consistency (SEC).
- Formal definitions of Strong Eventual Consistency (SEC) and of CRDTs.
- Two sufficient conditions for SEC.
- A strong equivalence between the two conditions.
- We show that SEC is incomparable to sequential consistency.
- Description of basic CRDTs, including integer vectors and counters.
- More advanced CRDTs, including sets and graphs.

We refer the interested reader to a separate technical report [18] for further detail and for a comprehensive portfolio of CRDT designs.

2 System Model

We consider a system of processes interconnected by an asynchronous network. The network can partition and recover. We assume a finite set $\Pi = \{p_0, \ldots, p_{n-1}\}$ of non-byzantine processes. Processes in Π may crash silently; a crashed process may remain crashed forever, or may recover with its memory intact. A non-crashed process is said *correct*.

[1] A conflict is a combination of concurrent updates, which may be individually correct, but that, taken together, would violate some invariant.

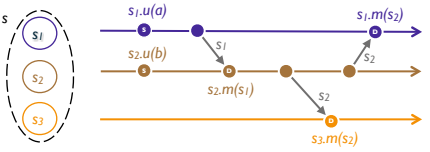


Fig. 1. State-based replication

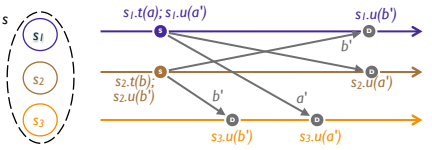


Fig. 2. Operation-based replication

2.1 State-Based Object

In this section we specify replicated objects in the so-called *state-based* style. The intuition is illustrated in Figure 1. Executing an update modifies the state of a single replica. Every replica occasionally sends its local state to some other replica, which *merges* the state thus received into its own state. In this way, every update eventually reaches every replica, either directly or indirectly.

With no loss of generality, we consider a single object with one replica at each process. An object is a tuple (S, s^0, q, u, m). The replica at process p_i has state $s_i \in S$, called its *payload*; the initial state is s^0. A client of the object may read the state of the object via *query* method q and modify it via *update* method u. Method m serves to *merge* the state from a remote replica. A method (whether q, u or m) executes at a single replica.

Systems that deliver every update to every replica eventually in a fault-tolerant manner are well-known in the literature, for instance gossip or anti-entropy approaches [5,13]. For simplicity, we will assume hereafter a fully connected communication graph, where every arc is a fair-lossy channel. Infinitely often, the replica at p_i sends (if it is correct) its current state to p_j; replica p_j (if it is correct) *merges* the received state into its local state by executing method m.

A method whose precondition is satisfied is said *enabled*. We assume that an enabled method executes as soon as it is invoked. Method executions at some replica are numbered sequentially from 1. The k^{th} method execution at replica i will be noted $f_i^k(a)$, where f is either q, u or m, and a denotes the arguments. We note $K_i(f)$ the ordinal of execution f at replica i, i.e., $K_i(f_j^k(a)) = k$ for $i = j$, and is undefined otherwise. (Abusing notation somewhat, we may drop subscripts, superscripts and/or arguments when there is no ambiguity.)

The states of a replica are numbered sequentially incrementing with each method execution. Thus, replica i has initial state $s_i^0 = s^0$. Before its k^{th} execution of a method it has state s_i^{k-1}, and s_i^k afterwards. We note the transition $s_i^{k-1} \bullet f_i^k(a) = s_i^k$.

We define state equivalence $s \equiv s'$ if all queries return the same result for s and s'. A query has no side-effects, i.e., $(s \bullet q) \equiv s$.

Definition 1 (Causal History (state-based)). *We define the object's* causal history $C = [c_1, \ldots, c_n]$ *(where c_i goes through a sequence of states $c_i^0, \ldots, c_i^k, \ldots$) as follows. Initially, $c_i^0 = \varnothing$, for all i. If the k^{th} method execution at i is:* (i) *a query q: the causal history does not change, i.e., $c_i^k = c_i^{k-1}$;* (ii) *an*

update (noted $u_i^k(a)$): it is added to the causal history, i.e., $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$; (iii) *a merge $m_i^k(s_{i'}^{k'})$, then the local and remote histories are unioned together: $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$.*

We say that an update is *delivered* at some replica when it is included in the causal history at that replica. An update u *happened-before* u' iff u is delivered when u' executes: $u \to u' \stackrel{\text{def}}{=} u \in c_j^{k-1}$, where u' executes at replica p_j and $K_j(u') = k$. Updates are *concurrent* if neither happened-before the other: $u \parallel u' \stackrel{\text{def}}{=} u \not\to u' \wedge u' \not\to u$. Note that the causal history is a formal reasoning device, which is normally not needed in a concrete implementation.

Given our communication assumptions, we can conclude that, in a state-based object, every update is eventually delivered to all replicas. However, this is not sufficient to ensure that replicas converge. For instance, if the merge method m is a no-op, an update executed at some replica has no effect on other replicas, and they will never converge.

2.2 Strong Eventual Consistency

Informally, eventual consistency means that replicas eventually reach the same final value if clients stop submitting updates. We capture this intuition as follows:

Definition 2 (Eventual Consistency (EC))
Eventual delivery: *An update delivered at some correct replica is eventually delivered to all correct replicas: $\forall i, j : f \in c_i \Rightarrow \Diamond f \in c_j$.*
Convergence: *Correct replicas that have delivered the same updates eventually reach equivalent state: $\forall i, j : \Box c_i = c_j \Rightarrow \Diamond\Box s_i \equiv s_j$.*
Termination: *All method executions terminate.*

Several EC systems will execute an update immediately, only to discover later that it conflicts with another, and to roll back to resolve this conflict [20]. This constitutes a waste of resources, and in general requires a consensus to ensure that all replicas arbitrate conflicts in the same way. To avoid this, we require a stronger condition:

Definition 3 (Strong eventual consistency (SEC)). *An object is Strongly Eventually Consistent if it is Eventually Consistent and:*
Strong Convergence: *Correct replicas that have delivered the same updates have equivalent state: $\forall i, j : c_i = c_j \Rightarrow s_i \equiv s_j$.*

2.3 State-Based Convergent Replicated Data Type (CvRDT)

We now propose a sufficient condition for strong convergence in state-based objects. A join semilattice (or just semilattice hereafter) is a partial order $\leq$ equipped with a *least upper bound* (LUB) $\sqcup$ for all pairs: $m = x \sqcup y$ is a Least Upper Bound of $\{x, y\}$ under $\leq$ iff $\forall m', x \leq m' \wedge y \leq m' \Rightarrow x \leq m \wedge y \leq m \wedge m \leq m'$. It follows that $\sqcup$ is: commutative: $x \sqcup y = y \sqcup x$; idempotent: $x \sqcup x = x$; and associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$.

Definition 4 (Monotonic semilattice object). *A state-based object, equipped with partial order* $\leq$*, noted* $(S, \leq, s^0, q, u, m)$*, that has the following properties, is called a monotonic semi-lattice:* (i) *Set* S *of payload values forms a semilattice ordered by* $\leq$*.* (ii) *Merging state* s *with remote state* s' *computes the LUB of the two states, i.e.,* $s \bullet m(s') = s \sqcup s'$*.* (iii) *State is monotonically non-decreasing across updates, i.e.,* $s \leq s \bullet u$*.*

Theorem 1 (Convergent Replicated Data Type (CvRDT)). *Assuming eventual delivery and termination, any state-based object that satisfies the monotonic semilattice property is SEC.*

For lack of space, we omit the proof that is presented in a companion technical report [19]. A CvRDT converges towards the LUB of the most recent updates. We require that $x \leq y \wedge y \leq x \Rightarrow x \equiv y$.

2.4 Op-Based Commutative Replicated Data Type (CmRDT)

Alternatively to the state-based style, a replicated object may be specified in the *operation-based* (or op-based) style. An op-based object is a tuple (S, s^0, q, t, u, P), where S, s^0 and q have the same meaning as above (respectively state domain, initial state and query method). An op-based object has no merge method; instead an update is split into a pair (t, u), where t is a side-effect-free *prepare-update* method and u is an *effect-update* method. The prepare-update executes at the single replica where the operation is invoked (its *source).* At the source, prepare-update method t is followed immediately by effect-update method u, i.e., $f_i^{k-1} = t \Rightarrow f_i^k = u$. (If this were not true, there would be no causality between successive updates.)

The effect-update method executes at all replicas (said *downstream*). The source replica delivers the effect-update to downstream replicas using a communication protocol specified by the delivery relation P, explained below.

We use the same notations for states and causal history as above, except that now f can refer to any of q, t or u. Both queries and prepare-update methods are side-effect-free, i.e., $s \bullet q \equiv s \bullet t \equiv s$.

Definition 5 (Causal History (op-based)). *An object's* causal history $C = \{c_1, \ldots, c_n\}$ *is defined as follows. Initially,* $c_i^0 = \varnothing$*, for all* i*. If the* k^{th} *method execution at* i *is:* (i) *a query* q *or a prepare-update* t*, the causal history does not change, i.e.,* $c_i^k = c_i^{k-1}$*;* (ii) *an effect-update* $u_i^k(a)$*, then* $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$*.*

An update is said delivered at a replica when the update is included in the replica's causal history. Update (t, u) happened-before (t', u') iff the former is delivered when the latter executes: $(t, u) \rightarrow (t', u') \Leftrightarrow u \in c_j^{k-1}$, where t' executes at p_j and $k = K_j(t')$. The definition of concurrent updates remains as above.

We assume an underlying reliable causally-ordered broadcast communication protocol, i.e., one that delivers every message to every recipient exactly once

and in an order consistent with happened-before. Such protocols are a standard feature of distributed systems; they do not require consensus and they deliver to all correct processes as long as any network partition eventually recovers (as we assumed earlier). It follows that two updates that are related by happened-before execute at all replicas in the same sequential order: $(t, u) \rightarrow (t', u') \Rightarrow \forall i, K_i(u) < K_i(u')$. However, concurrent updates may be delivered in any order.

Definition 6 (Commutativity). *Updates (t, u) and (t', u') commute, iff for any reachable replica state s where both u and u' are enabled, u (resp. u') remains enabled in state $s \bullet u'$ (resp. $s \bullet u$), and $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$.*

Clearly, a sufficient condition for convergence of an op-based object is that all its concurrent operations commute. An object satisfying this condition is called a Commutative Replicated Data Type (CmRDT).

P is a delivery precondition, i.e., effect-update method u is enabled only if the precondition is satisfied. We interpret this temporally, i.e., delivery of u at replica i may delayed, until $P(s_i, u)$ is true. Therefore, for liveness, we now have the added obligation to prove that delivery is eventually enabled. Therefore we restrict our scope to preconditions for which causally-ordered broadcast is sufficient to ensure P.

Theorem 2 (Commutative Replicated Data Type (CmRDT)). *Assuming causal delivery of updates and method termination, any op-based object that satisfies the commutativity property for all concurrent updates, and whose delivery precondition is satisfied by causal delivery, is SEC.*

The proof is presented in [19].

3 Some Results

3.1 Fault-Tolerance and the CAP Theorem

The CAP theorem states that it is impossible to simultaneously ensure strong consistency (C), availability (A) and tolerate network partition (P) [8]. As, network faults unavoidably occur in a large-scale environment, a real system must sacrifice either consistency or availability. Availability is often the top priority in practice [3]: does this mean giving up all consistency guarantees?

No: SEC provides a solution. A SEC replica is always available for both reads and writes, independently of network conditions. Any communicating subset of replicas of a SEC object eventually converges, even if partitioned from the rest of the network. SEC is weaker than strong consistency but nonetheless provides the well-defined guarantee of strong eventual convergence.

SEC provides an extreme form of fault tolerance, as a SEC object tolerates up to $n - 1$ simultaneous crashes. Remarkably, SEC does not require to solve consensus.

3.2 CvRDTs and CmRDTs are Equivalent

Operation-Based Emulation of a State-Based Object

Theorem 3 (CmRDT emulation). *Any SEC state-based object can be emulated by a SEC op-based object of a corresponding interface.*

Proof. Given a CvRDT represented by tuple $(S, \leq, s^0, q, u, m)$, we emulate it by a CmRDT object (S, s^0, q, t, u', P), which we specify hereby.

State and query of CvRDT can be directly stored and processed by emulating CmRDT using the same definitions. A prepare-update $t(a)$ has the same interface (accepts the same domain of arguments and returns the same domain of value) as an update $u(a)$. It records the result of applying update $u(a)$ on a copy of current replica state s: $s' = s \bullet u(a)$; return value of $u(a)$ is passed to the client. Recorded state s' is used as an argument of an actual effect-update $u'(s')$, which is delivered to all replicas by the underlying protocol of CmRDT. Precondition P is unrestricted and enables delivery at any time. Effect-update $u'(s')$ merges received state using original CvRDT method: $s \bullet u'(s') \stackrel{\text{def}}{=} s \bullet m(s')$.

Since merge always commutes, then updates $u'(s')$ commute and since the communication is reliable, we have a CmRDT with strong eventual consistency, which propagates all updates of emulated CvRDT.

State-Based Emulation of an Operation-Based Object. State-based emulation of an operation-based object essentially formalises the mechanics of an epidemic reliable causal broadcast.

Theorem 4 (CvRDT emulation). *Any SEC op-based object can be emulated by a SEC state-based object of a corresponding interface.*

Proof. Given a CmRDT represented by tuple (S, s^0, q, t, u, P), we emulate it by a CvRDT object $((S \times U \times U), \leq, (s^0, \varnothing, \varnothing), q', u', m)$, which we specify hereby.

Without loss of generality, we assume that each invocation u_i^k is unique across replicas and set U denotes all possible updates. CvRDT's state is then defined as a triple (s_m, M, D), where s_m is a state of emulated CmRDT, M and D are two add-only sets of, respectively, known and delivered updates. A relation $\leq$ is defined as following: $(s_m, M, D) \leq (s'_m, M', D') \stackrel{\text{def}}{=} M \subseteq M' \wedge D \subseteq D'$.

A query $q'(a)$ has the same interface as $q(a)$; we define it as a trivial delegation to $q(a)$ on the CmRDT, $s_m \bullet q(a)$. An update $u'(a)$ has the same interface as prepare-update $t(a)$. It first delegates the invocation to prepare-update $t(a)$ of the CmRDT that in turn triggers effect-update $u(a)$, which becomes a locally known update. Finally, $u'(a)$ uses a recursive function d to process updates:

$$d(s_m, M, D) \stackrel{\text{def}}{=} \begin{cases} d(s_m \bullet u(a), M, D \cup \{u(a)\}) & \text{if } \exists u(a) \in M \setminus D : P(s_m, u(a)) \\ (s_m, M, D) & \text{otherwise} \end{cases}$$

Hence, $u'(a)$ is defined as: $(s_m, M, D) \bullet u'(a) \stackrel{\text{def}}{=} d(s_m \bullet t(a), M \cup \{u(a)\}, D)$.

Finally, merge m takes a union of known messages and processes available updates: $(s_m, M, D) \bullet m(s'_m, M', D') \stackrel{\text{def}}{=} d(s_m, M \cup M', D)$.

Since the emulation ensures that messages are delivered exactly once to each replica's embedded object, in the appropriate order, and since the CvRDT conforms to SEC criteria, the embedded CmRDT instance is also SEC.

Note that the emulating object forms a monotonic semilattice over domain $S \times U \times U$. Calling or delivering an operation adds it to the relevant message set, and therefore advances the state in the partial order. The merge method m is defined to take the union of the M sets and (possibly) updating D, and is thus a LUB operation. This construction is similar to Wuu and Bernstein's log covered in Section 4.2.

3.3 SEC is Incomparable to Sequential Consistency

A state-based replica executes a sequence of query, update, and merge methods. In addition to its sequential behaviour, a CRDT specifies concurrent behaviours that must satisfy the strong convergence property. As we show now, this permits executions that would be impossible in a sequentially-consistent system.

Consider a Set CRDT S with operations $add(e)$ and $remove(e)$. Immediately after $add(e)$, the state will satisfy $e \in S$; after $remove(e)$ the state satisfies $e \notin S$. In a sequential execution, the last update wins, e.g., after $remove(e) \rightarrow add(e)$ the state satisfies $e \in S$. Concurrent adds or removes of different elements are independent, e.g., after $add(e) \parallel remove(e')$ the state satisfies $e \in S \wedge e' \notin S$.

There is a choice of alternative semantics for concurrent updates of the same element. When concurrently adding and removing the same element, the add could win, or the remove could win, or the update of the replica with the highest IP address could win, or the state might be reset to a distinguished state $\perp$, and so on. All these alternatives satisfy the strong convergence condition, and any of them may be reasonable for some application.

Let us consider the add-wins alternative: after $add(e) \parallel remove(e)$ the state satisfies $e \in S$. Now consider the following scenario. Replica p_0 executes the sequence $add(e); remove(e')$. Concurrently, replica p_1 executes $add(e'); remove(e)$. Then, replica p_3 merges the state from p_0 and p_1. According to the concurrent specification, the final state at p_3 satisfies $e \in S \wedge e' \in S$. Such a state would never occur in a sequentially-consistent execution, in which either $remove(e)$ or $remove(e')$ must be last. Thus, there is a SEC object that is not sequentially consistent.

Now consider the converse. In the absence of crashes, a sequentially-consistent object is SEC. Indeed, sequential consistency is defined by a single order of operations, after which all replicas must terminate with the same state. However, in the general case, sequential consistency requires consensus, which cannot be solved in the presence of $n - 1$ crashes. Therefore, SEC is incomparable with sequential consistency.

4 Example CRDTs

We now recall some basic CRDTs that are known in the existing literature, which we will later compose to build higher-level objects. We will use state- or op-based specifications as most convenient. Generally, we find the state-based style more compact and easier to reason about formally, whereas the op-based style is often convenient for implementation.

4.1 Integer Vectors and Counters

Consider the state-oriented specification of a vector-of-integers object: $(\mathbb{N}^n, [0,\ldots,0], \leq^n, [0,\ldots,0], value, inc, \max^n)$. Vectors $v, v' \in \mathbb{N}^n$ are (partially) ordered by $v \leq^n v' \Leftrightarrow \forall j \in [0..n-1], v[i] \leq v'[i]$. A query invocation $value()$ returns a copy of the local payload. An update $inc(i)$ increments the payload entry at index i, that is, $s \bullet inc(i) = [s'[0],\ldots,s'[n-1]]$ where $s'[j] = s[j] + 1$ if $i = j$ and $s'[j] = s[j]$ otherwise. Merging two vectors takes the per-index maximum, i.e., $s \bullet \max^n(s') = [\max(s[0], s'[0]),\ldots,\max(s[n-1], s'[n-1])]$. We omit the proof that it is a CRDT.

If each process p_i is restricted to incrementing its own index $inc(i)$, this is the well-known vector clock [11].

An increment-only integer counter is very similar; the only difference being that query invocation $value()$ of a vector in state v returns $|v| \stackrel{\text{def}}{=} \sum_j v[j]$. We construct an integer counter that can be both incremented and decremented, by basically associating two increment-only counters I and D, where incrementing increments I and decrementing increments D, whereas $value()$ returns $|I| - |D|$. The ordering method $\leq$ is defined as $(I, D) \leq (I', D') \stackrel{\text{def}}{=} I \leq^n I' \wedge D \leq^n D'$.

4.2 U-Set, Map and Log

Another simple CRDT construct is an add-only set object $(S, \subseteq, \varnothing, value, add(e), \cup)$. The payload is any set; sets are ordered by inclusion. A query $value()$ returns a copy of the local payload. Update $add(e)$ adds element e to the set, i.e., $s \bullet add(e) = s \cup \{e\}$. It is well-known that sets ordered by $\subseteq$ form a semi-lattice with $\cup$ as the LUB operator. It is clearly monotonic by the definition of add. Therefore, the add-only set is a CRDT.

Wuu and Bernstein build further CRDTs by combination of these basic components [23]. They propose a set with both add and $remove$ operations by associating two add-only sets A and R; adding an element adds it to A, removing it adds it to R; query $value()$ returns the set difference $A \setminus R$. (R is often called the tombstone set. A client is allowed to remove only an element that is currently in A). Note that they assume that every element is unique and added only once; we call their construct U-Set [18]. Wuu and Bernstein derive their Dictionary data type from U-Set in the obvious way.

A Log is a replicated object, whose payload contains a set (initially empty) of (event, timestamp) pairs. It assumes that each process maintains a vector clock

in the usual manner [11]. When an event e occurs at process i, the process invokes update $add(e)$; the update method updates the vector clock (say, to state v) and adds the pair (e, v) to the set. The timestamp ensures that each entry is unique. The merge method takes the union of the local and a remote set.

To avoid unbounded growth, Wuu and Bernstein propose a distributed garbage collection algorithm that discards unneeded entries. In order to tolerate $n-1$ crashes, only an entry that has been delivered to all processes may be discarded. If vector clock entry $v_i[j] = k$, this implies that process i has delivered all k first events of process p_j. Each replica maintains in its payload a copy of all remote vector clocks; for each remote site, the merge procedure keeps the largest version. Then, a replica may discard a log entry as soon as its timestamp is less than all the remote vector clocks. This algorithm does not require a consensus, but it is live only if no process is crashed. However, this may be acceptable, since the liveness of garbage collection does not impact the correctness of the main algorithm.

This algorithm may be adapted to other data types, for instance to discarding the A and R entries of a removed element in the U-Set.

5 Directed Graph CRDT

Now let us examine how one would design a more complex data type: a Directed Graph CRDT. Graphs are an important general-purpose data structure. Some important applications and algorithms work on graphs, e.g., shortest-path or web page-rank.

5.1 Thought Experiment

To motivate our graph design, consider the "thought experiment" of designing a web search engine. The search engine uses a directed graph representing the web structure. This graph may be used, among other things, to compute page rank. Such an application processes large amounts of data and performs many updates. For efficiency and scalability, processing should be asynchronous; for responsiveness, processing should be incremental, as fast as each page is crawled. Processing should not require any synchronisation, e.g., transactions. A CRDT could be ideal.

We start with a Set CRDT containing some initial URLs to be crawled. A number of crawler processes run in parallel; each one removes some URL from the set and downloads it. (It might happen that the same page is downloaded twice but this does not impact correctness.)

When a crawler finds a new page, it executes the corresponding *addVertex*. For every page, it parses the links that it contains, comparing it with the page's previous version, if any, and executes the corresponding *addArc* and *removeArc* invocations. Finally, the URLs of the linked pages are added to the set to be crawled. Note that *addArc* must work even if the page at the tail of the arc has not yet been found (it might not even exist), but such an arc is not functional; a lookup of the corresponding arc should fail. Similarly if a node has been removed,

payload set V, A -- *sets of pairs* { (element e, unique-tag w), ... }
  initial $\varnothing, \varnothing$ -- *V: vertices; A: arcs*
query *lookup* (vertex v) : boolean b
  let $b = (\exists w : (v, w) \in V)$
query *lookup* (arc (v', v'')) : boolean b
  let $b = (lookup(v') \wedge lookup(v'') \wedge (\exists w : ((v', v''), w) \in A)$
update *addVertex* (vertex v)
  prepare $(v) : w$
    let $w = unique()$ -- $unique()$ *returns a unique value*
  effect (v, w)
    $V := V \cup \{(v, w)\}$ -- v + *unique tag*
update *removeVertex* (vertex v)
  prepare $(v) : R$
    pre $lookup(v)$ -- *precondition*
    pre $\nexists v' : lookup((v, v'))$ -- *v is not the head of an existing arc*
    let $R = \{(v, w) | \exists w : (v, w) \in V\}$ -- *Collect all unique pairs in V containing v*
  effect (R)
    $V := V \setminus R$
update *addArc* (vertex v', vertex v'')
  prepare $(v', v'') : w$
    pre $lookup(v')$ -- *head node must exist*
    let $w = unique()$ -- $unique()$ *returns a unique value*
  effect (v', v'', w)
    $A := A \cup \{((v', v''), w)\}$ -- (v', v'') + *unique tag*
update *removeArc* (vertex v', vertex v'')
  prepare $(v', v'') : R$
    pre $lookup((v', v''))$ -- $arc(v', v'')$ *exists*
    let $R = \{((v', v''), w) | \exists w : ((v', v''), w) \in A\}$
  effect (R) -- *Collect all unique pairs in A containing arc (v', v'')*
    $A := A \setminus R$

Fig. 3. Directed Graph Specification (op-based)

all arcs incident to the node disappear. In this way, the behaviour of our CRDT will be consistent with that of web pages, which are allowed to contain non-functional URLs. Once the linked page is created, the link become relevant, e.g., for navigation and for page-rank computation.

In the web application, the graph is very large; sending the state between replicas and merging would be very costly. Therefore, we choose an op-based approach.

5.2 Design Alternatives for Arc Removal

A directed graph is a pair of sets (V, A), called vertices and arcs respectively, such that $A \subseteq V \times V$. Updates must maintain the invariant that the head and

tail vertices of an arc both exist. Therefore, adding an arc to A has the precondition that its two vertices are in V; conversely, a vertex may be removed only if it supports no arc; these are preconditions to prepare-update. Furthermore, the system must ensure that concurrent $addArc(v', v'') \parallel removeVertex(v')$ do not violate the invariant. Several alternatives may be considered: *(i)* Give precedence to $removeVertex(v')$: all edges to or from v' are removed as a side effect. This is easy to implement, by hiding any arc that includes a removed vertex. *(ii)* Give precedence to $addArc(v', v'')$: if either v' or v'' has been removed, it is restored. This requires recreating nodes that have being explicitly deleted. *(iii)* $removeVertex(v')$ is delayed until all concurrent *addArc* operations have executed. This requires synchronisation which violates the goals of asynchrony and fault tolerance. There is no perfect choice. Hereafter, we choose Option *(i)* because it is adequate in our application scenario.

5.3 Graph Specification

Figure 3 shows our specification for a Directed-Graph CRDT. We prove that this object is indeed a CmRDT in [19].

This CRDT maintains two sets internally, one for the vertices and one for the arcs. To add a vertex v, the prepare-update method creates a unique identifier, w, and the effect-update method adds the pair (v, w) to the set of vertices. With this approach, each vertex has an unique internal identifier. If the same vertex is added twice, the two additions will be distinguished by their two unique identifiers. A lookup will mask the duplicates.

To remove vertex v, the prepare-update computes the set R of pairs that contain v, i.e., all copies known in the source replica; the effect-update method removes this same set R from the set of vertices in all replicas. As operations are delivered in causal order, when the effect-update method executes in some replica, for each pair in R, the correspondent *addVertex* operations has already executed. Thus, unlike the state-based solution of Section 4.2, a set need not keep tombstones.

If the same vertex is removed and added concurrently, the *addVertex* wins, as the new unique identifier is not included in the set computed by the remove's prepare-update. This approach is consistent with a sequential execution, as the a vertex can removed only if it is observed. The same approach is used for arcs.

To remove a vertex, the source replica checks that the vertex is observed, and also that it is not the head of any existing arc. Conversely, to add an arc, its head node must exist, but there is no check for existence of the tail. The lookup method will mask the existence of such an arc. However, if the tail is added later, then the arc becomes visible. Similarly, concurrent updates may remove a vertex that is the head of an arc. However, the lookup method will mask such an arc.

6 Comparison with Previous Work

Eventual consistency has been an active topic of research in highly-available, large-scale asynchronous systems [17,21]. Contrary to much previous work [3, for instance], we take a formal approach grounded in the theory of commutativity and semilattices.

The state-based approach was invented for register-like objects, where the only update operation is assignment. It is in wide use in file systems such as NFS, AFS or Coda, and in key-value stores such as Dynamo [3] and Riak. Op-based approaches are used when the cost of transferring state is too high, e.g., databases, and when operation semantics are important, e.g., cooperative systems such as Bayou [13] or IceCube [15].

Although the CRDT concept was identified only recently, related designs have been published before. Johnson et. al. invented the LWW-Register [9]. They propose a database of registers that can be created, updated and deleted, using the last-writer-wins (LWW) rule to arbitrate between concurrent changes. LWW ensures a total order of operations, at the cost of losing concurrent updates.

Concurrent editing uses the related concept of Operational Transformation (OT), due to Ellis and Gibbs [7]. To ensure responsiveness, a local operation executes immediately. Operations are not designed to commute; however, a replica receiving an update transforms it against previously-executed concurrent updates to achieve a similar result. OT algorithms for a decentralised architecture have been proposed; Oster et al. show that most of them are incorrect [12]. We believe that designing for commutativity from the start is cleaner and simpler.

The foundations of CvRDTs were introduced by Baquero and Moura [1]. We extend their work with CmRDTs and with a number of new results. The CRDT concept was invented by Shapiro and Preguiça on their work on Treedoc, a Sequence CRDT for concurrent editing [14]. Logoot is another Sequence CRDT that supports an *undo* mechanism based on a CRDT Counter [22].

Roh et al. [16] independently developed the related concept of Replicated Abstract Data Type. They generalise LWW to a partial order of updates, which they leverage to build several LWW-style classes.

Burckhardt and Leijen propose the Concurrent Revisions programming model for shared abstract data types, in which a forked revision runs in isolation until it joins again. Join is based on a three-way merge function [2]. They show that simple sequential merge functions exist for ADTs built upon Abelian groups. We have also demonstrated the relation between CRDTs and sequential consistency in a similar, but more loosely-coupled, replication model.

Ducourthial et al. study algebraic structures with specific properties in order to solve self-stabilisation problems [6]. They propose the so-called r-operator for "silent" tasks [4]. Strong convergence can be seen as as a silent task, given a limited number of disturbing updates. However, there are differences between the two approaches. Whereas a self-stabilising system must tolerate arbitrary memory corruption, a shared mutable object should change state durably only by executing update operations. Furthermore, whereas CvRDT states constitute a monotonic semi-lattice, the r-operator requries a total order.

7 Conclusion

We presented the concept of a CRDT, a replicated data type for which some simple mathematical properties guarantee eventual consistency. In the state-based style, the successive states of an object should form a monotonic semilattice, with merge computing a least upper bound. In the op-based style, concurrent operations should commute. Assuming only that the communication subsystem ensures eventual delivery (in causal order for op-based objects), CRDTs are guaranteed to converge towards a common, correct state, without requiring any synchronisation.

We presented some simple CRDT examples, such as sets, and detailed how to create a directed Graph CRDT, which might be used in a large-scale web search engine. Our data types have a clean and deterministic semantics in the presence of concurrent updates.

Eventual consistency is a critical technique in many large-scale distributed systems, including delay-tolerant networks, sensor networks, peer-to-peer networks, collaborative computing, cloud computing, and so on. However, work on eventual consistency was mostly ad-hoc so far. Although some of our CRDTs were known before in the literature or in the folklore, this is the first work to engage in a systematic study. We believe this is required if eventual consistency is to gain a solid theoretical and practical foundation.

Future work is both theoretical and practical. On the theory side, this will include understanding the class of computations that can be accomplished by CRDTs, the complexity classes of CRDTs, the classes of invariants that can be supported by a CRDT, the relations between CRDTs and concepts such as self-stabilisation and aggregation, and so on. On the practical side, we plan to implement the data types specified herein as a library, to use them in practical applications, and to evaluate their performance analytically and experimentally. Another direction is to support infrequent, non-critical synchronous operations, such as committing a state or performing a global reset. We will also look into stronger global invariants, possibly using probabilistic or heuristic techniques.

References

1. Baquero, C., Moura, F.: Specification of convergent abstract data types for autonomous mobile computing. Tech. rep., Departamento de Informática, Universidade do Minho (October 1997)
2. Burckhardt, S., Leijen, D.: Semantics of concurrent revisions. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 116–135. Springer, Heidelberg (2011)
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles (SOSP). Operating Systems Review, vol. 41, pp. 205–220 (October 2007)
4. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with R-operators revisited. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, pp. 68–80. Springer, Heidelberg (2005)

5. Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J.: Epidemic algorithms for replicated database maintenance. In: Symp. on Principles of Dist. Comp. (PODC), pp. 1–12 (August 1987); also appears Op. Sys. Review 22(1), 8–32 (1988)
6. Ducourthial, B.: R-semi-groups: a generic approach for designing stabilizing silent tasks. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 281–295. Springer, Heidelberg (2007)
7. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: Int. Conf. on the Mgt. of Data (SIGMOD), pp. 399–407 (1989)
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. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (January 1976)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
11. Mattern, F.: Virtual time and global states of distributed systems. In: Int. W. on Parallel and Distributed Algorithms, pp. 215–226 (1989)
12. Oster, G., Urso, P., Molli, P., Imine, A.: Proving correctness of transformation functions in collaborative editing systems. Rapport de recherche RR-5795, LORIA – INRIA Lorraine (December 2005)
13. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Symp. on Op. Sys. Principles (SOSP), pp. 288–301 (October 1997)
14. Preguiça, N., Marquès, J.M., Shapiro, M., Leţia, M.: A commutative replicated data type for cooperative editing. In: Int. Conf. on Distributed Comp. Sys (ICDCS), pp. 395–403 (June 2009)
15. Preguiça, N., Shapiro, M., Matheson, C.: Semantics-based reconciliation for collaborative and mobile environments. In: Chung, S., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 38–55. Springer, Heidelberg (2003)
16. Roh, H.G., Jeon, M., Kim, J.S., Lee, J.: Replicated abstract data types: Building blocks for collaborative applications. Journal of Parallel and Dist. Comp., pp. 354–368 (March 2011)
17. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005)
18. Shapiro, M.: Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche 7506, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (January 2011)
19. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free Replicated Data Types. Rapport de recherche 7686, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (July 2011)
20. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: 15th Symp. on Op. Sys. Principles (SOSP), pp. 172–182 (December 1995)
21. Vogels, W.: Eventually consistent. ACM Queue 6(6), 14–19 (2008)
22. Weiss, S., Urso, P., Molli, P.: Logoot-undo: Distributed collaborative editing system on P2P networks. IEEE Trans. on Parallel and Dist. Sys (TPDS) 21, 1162–1174 (2010)
23. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC), pp. 233–242 (August 1984)

Analysis of DSR Protocol in Event-B

Dominique Méry and Neeraj Kumar Singh

Université Henri Poincaré Nancy 1
LORIA, BP 239, 54506 Vandoeuvre lès Nancy, France
{mery,singhnne}@loria.fr

Abstract. This paper presents an incremental formal development of the Dynamic Source Routing (DSR) protocol in Event-B. DSR is a reactive routing protocol, which finds a route for a destination on demand, whenever communication is needed. Route discovery is an important task of any routing algorithm and formal specification of it, itself is a challenging problem. The specification is performed in a stepwise manner composing more advanced routing components between the abstract specification and topology. It is verified through a series of refinements. The specification includes safety properties as set of invariants, and liveness properties that characterize when the system reaches stable states. We establish these properties by proof of invariants, event refinement and deadlock freedom. The consequence of this incremental approach helps to achieve a high degree of automatic proof. Our approach can be useful for formalizing and developing other kinds of reactive routing protocols (i.e. AODV etc.).

Keywords: Abstract model, Event-B, Event-driven approach, Proof-based development, Refinement, Ad hoc Network.

1 Introduction

Formal models have a valuable role to play in validating requirements and designs for distributed systems. In a mobile ad hoc networks, nodes move arbitrarily and change the network topology. Frequently changing topology presents a fundamental challenge for routing protocols. This paper presents a case study on the Dynamic Source Routing (DSR) protocol [1]. Reactive routing protocol is generally not dependent on exchanges of periodic route information and route calculations. Instead, whenever a route is needed the node has to perform a route discovery before it can send any packet to a destination node. Our approach is here to specify and formally develop the DSR protocol. We use an incremental development of the DSR protocol with stepwise refinements in Event-B [2,3]. Event-B [2,3] is a formal modeling language, which supports refinement based formal development. We proceed by constructing the proof-based series of models, where the initial model specifies the system requirements and final model describes the resulting system.

It is a significant case study in specifying and developing the real routing protocol algorithms. In routing protocols each host works as a router and constructs a graph representing the network topology. In this graph, vertices and edges represent routing nodes and direct connection between nodes, respectively. Each node uses this graph to find the optimized routing table and determines the correct route from source node

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 401–415, 2011.

to destination node. The main challenging task in route discovery is to find the exact distribution of nodes in a dynamic network and routing updates after changing network topology.

To specify the correct desired properties of protocol at abstract level and in carrying out the development and proofs in subsequent refinement models, is a challenging problem. This challenging problem comes from the fact that the protocol should function in dynamically changing environment. The main characteristics of an ad hoc network is dynamic behavior of the network: nodes can be added and deleted in a dynamic manner. The topology information in all the reactive protocol is only transmitted by nodes on-demand such as a node wishes to transmit the data packets to a node to which it has no route, it will generate a route request message that will be flooded in a limited way to other nodes. A route is considered found when the route request message reaches either at a destination itself, or at an intermediate node with a valid route entry for the destination node.

One of the key aspect of our development is to verify $stability$ of the system. $Stability$ of the system is a most important property of this chaotic networks which implies correct local view of the current system. Intuitively, in stable states, all nodes have the maximum knowledge of the environment that can be acquired by route discovery and communication. This notion of system stability is an instance of the general notion of a *stable system property* [4].

The models of DSR protocol must be validated to ensure that they meet the requirements. Our abstract specification includes events of basic communication protocol. The nature of the refinement that we verify using Rodin [2] proof tools are safety refinement. Thus the behavior of final resulting system is preserved by abstract model as well as in correctly refined models. Proof-based development methods [3] integrate formal proof techniques in the development of software systems. The main idea is to start modeling with an abstract model and details are gradually added to the abstract model to produce a sequence of concrete events. The relationship between two successive models is known as *refinement* [3]. The current work intends to explore problems related to the modeling of distributed systems where an environment is changing dynamically. Moreover, the stepwise development of the DSR protocol model helps to discover the exact behavior of basic communication protocol and route discovery protocol in dynamic environment.

The outline of remaining paper organizes as follows. Section 2 describes the modeling framework, which outline some general idea of modeling, that we found useful in this work. In section 3, we describe an informal description of the DSR protocol. Requirements and assumptions are described in section 4. Section 5 explores the formal development of the DSR protocol using stepwise refinement. Finally, section 6 presents discussion and conclusion of the work.

2 The Modeling Framework

We will summarize the concepts of the Event-B modeling language developed by Abrial [5,3] and will indicate the links with the tool called RODIN [2]. Considering the Event-B modeling language, we notice that the language can express *safety* properties, which are either *invariants* or *theorems* in a machine corresponding to the system. Recall that two main structures are available in Event-B:

- Contexts express static information about the model.
- Machines express dynamic information about the model, invariants, safety properties, and events.

An Event-B model defines either a context or a machine. A machine organizes events modifying state variables and it uses static information defined in a context. These basic structure mechanisms are extended by the refinement mechanism which provides a mechanism for relating an abstract model and a concrete model by adding new events or by adding new variables. This mechanism allows us to develop gradually Event-B models and to validate each decision step using the proof tool. The refinement relationship should be expressed as follows: a model M is refined by a model P, when P is executing M. The final concrete model is close to the behavior of real system that executes events using real source code. We give details now on the definition of events, refinement and guidelines for developing complex system models.

2.1 Modeling Actions over States

The event-driven approach [5,3] is based on the B notation. It extends the methodological scope of basic concepts to take into account the idea of *formal models*. Briefly, a formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*, where an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. In the following, we summarize definitions and principles of formal models and explain how they can be managed by tools [2].

Generalized substitutions are borrowed from the B notation. They provide a means to express changes to state variable values. In its general form, an event has three main parts, namely a list of local parameters, a guard and a relation over values denotes pre values of variables and post values of variables. The most common event representation is (ANY t WHERE $G(t,x)$ THEN $x : |(R(x,x',t))$ END). The *before–after* predicate $BA(e)(x,x')$, associated with each event, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the *execution* of event *e*. The form is semantically equivalent to $\exists t \cdot (G(t,x) \ \wedge \ R(x,x',t))$.

Table 1. Event-B proof obligations

PROOF OBLIGATIONS
– (INV 1) $Init(x) \ \Rightarrow \ I(x)$
– (INV 2) $I(x) \ \wedge \ BA(e)(x,x') \ \Rightarrow \ I(x')$
– (FIS) $I(x) \ \wedge \ \mathsf{grd}(e)(x) \ \Rightarrow \exists y.BA(e)(x,y)$

Proof obligations (INV 1 and INV 2) are produced by the Rodin tool [2] from events to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definition of the before–after predicate $BA(e)(x,x')$ of each event e (see Table-2). Note that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false. Whenever this

is the case, the event is said to be *disabled*. The proof obligation FIS expresses the feasibility of the event e with respect to the invariant I.

2.2 Model Refinement

The refinement of a formal model allows us to enrich the model via a *step-by-step* approach and is the foundation of our correct-by-construction approach [6]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event to a corresponding concrete version, and by adding new events. The abstract (x) and concrete (y) state variables are linked by a *gluing invariant* $J(x, y)$. A number of proof obligations make sure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines $skip$, (3) no new event takes control for ever, and (4) relative deadlock freedom is preserved. Details of the formulation of these proofs follows.

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. If $BA(e)(x, x')$ and $BA(f)(y, y')$ are the abstract and concrete before–after predicates of the same event, e and f respectively, we have to prove the following statement, corresponding to proof obligation (1):

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ BA(f)(y,y') \ \Rightarrow \ \exists x' \cdot (BA(e)(x,x') \ \wedge \ J(x',y'))}$$

Now, proof obligation (2) states that $BA(f)(y, y')$ must refine $skip$ ($x' = x$), generating the following simple statement to prove (2).

$$\boxed{I(x) \ \wedge \ J(x,y) \ \wedge \ BA(f)(y,y') \ \Rightarrow \ J(x,y')}$$

In refining a model, an existing event can be refined by strengthening the guard and/or the before–after predicate (effectively reducing the degree of nondeterminism), or a new event can be added to refine the *skip* event. The feasibility condition is crucial to avoiding possible states that have no successor, such as division by zero. Furthermore, this refinement guarantees that the set of traces of the refined model contains (up to stuttering) the traces of the resulting model. The refinement of an event e by an event f means that the event f simulates the event e.

The Event-B modeling language is supported by the Rodin platform [2] and has been introduced in publications [3,5], where the many case studies and discussions about the language itself and the foundations of the Event-B approach. The language of *generalized substitutions* is very rich, enabling the expression of any relation between states in a set-theoretical context. The expressive power of the language leads to a requirement for help in writing relational specifications, which is why we should provide guidelines for assisting the development of Event-B models.

3 Informal Description of DSR Protocol

The DSR protocol is a simple and efficient routing protocol designed specifically to use in multi-hop wireless ad hoc networks of mobile nodes. It allows the network to be completely self-organizing and self-configuring, without the need for any existing network infrastructure or administration. In source routing techniques, a sender determines the complete sequence of nodes through which it forwards the data packet. The sender explicitly lists this route in the packets header, identifying each forwarding 'hop' by the address of the next node to which transmits the data packet on its way to the destination node. The sender then transmits the packet over its wireless network interface to the first hop identified in the source route. When a host receives a packet, if this host is not the destination of the packet, it simply transmits the packet to the next hop identified in the source route in the packet header. Once the packet reaches its destination, the packet is delivered to the host. The protocol presented here is explicitly designed for use in the wireless environment of an ad hoc network. There are no periodic router advertisements in the protocol. Instead, when a node needs a route to another node, it dynamically determines one based on a local routing table or a route cached information and on the results of a route discovery protocol [1]. DSR consists of two mechanisms: $route\ discovery$ and $route\ maintenance$.

Route Discovery: Whenever a source needs to communicate to a destination and does not have a route in its routing table, it broadcasts a route request (RREQ) message to find a route. Each neighbor receives the RREQ and (if it has not already processed the same request earlier) appends its own address to the address list in the RREQ and rebroadcasts the packet. This process continues until either the maximum hop counter is exceeded (and RREQ is discarded) or the destination is reached. In the latter case, the destination receives the RREQ, appends its address and generates a route reply packet (RREP) back towards the source using the reverse of the accumulated route [1].

Route Maintenance: Route maintenance is used to manage (cache, expire, switch among) previously discovered routes. Each node along the route, when transmitting the packet to the next hop, is responsible for detecting next connected link. When the retransmission and acknowledgement mechanism detects that the link is broken, the detecting node returns a route error packet (RERRP) to the source of the packet. The node will then search its route cache to find if there is an alternative route to the destination of this packet. If there is one, the node will change the source route in the packet header and send it using this new route. When a route error packet (RERRP) is received or overheard, the link in error is removed from the local route cache, and all routes which contain this hop must be truncated at that point [1]. The source can then attempt to use any other route to the destination that is already in its route cache, or can invoke route discovery again to find a new route.

4 Requirements and Assumptions

The protocol must work in an environment where the status of links may change at any time. If the environment changes sufficiently rapidly, then links reported as down may actually be up and vice versa. Hence the local routing table may bear little relationship

to the actual network topology. To tackle this problem, we focus on the limiting, and most important, case of the algorithm's behavior: its behavior when the environment is sufficiently quiescent. In this case, we expect that the local routing table will eventually "stabilize" to states of the actual global topology. According to the basic graph theory, any graph can be decomposed into a collection of strongly-connected components. Our main system requirements are:

System Requirement 1: Data packet must be transmitted successfully from source node to destination node in a dynamic ad hoc network.

System Requirement 2: If the environment is inactive for a sufficiently long time then communication stabilizes and each node has the correct view of the links between all nodes in its connected subnetwork.

System Requirement 3: Route discovery protocol must discover a new route from the connected network where the status of links may change at any time.

Before developing the formal model of DSR protocol, we have some assumptions as follows:

- There are finite numbers of nodes or hosts.
- There are directed, one-way links between some pairs of distinct nodes. Links may come up and go down at any time.
- Nodes are communicating by broadcasting where node (x) sends a message to other node (y) when they are directly connected.
- When a link goes down, any message sent on it and not yet received are lost. This reflects that communication is asynchronous. There is a delay between message transmission and reception, and messages can be lost during this time interval.
- The hosts do not continuously move so rapidly as to make the flooding of every packet.

5 Formal Development

DSR protocol development is expressed in an abstract and general way. We describe the incremental development of DSR protocol in two phases as basic communication protocol and route discovery protocol. We develop the six models related by refinements. The initial model formalizes our system requirements and environmental assumptions, whereas the subsequent models introduce design decisions for the resulting system.

Initial Model : To specify basic communication protocol of data packet sending, receiving, losing, and network topology changes using some initial events (*sending, receiving, losing, remove_link* and *add_link*).

Refinement 1 : Introducing store and forward architecture for data packets passing from source node to destination node.
Refinement 2 : Introducing local routing table.
Refinement 3 : Introducing route discovery protocol to discover a new route.
Refinement 4 : Provides more detail information about route discovery protocol.
Refinement 5 : Introducing sequence numbers for tracking fresh route request packets information.

5.1 The Context and Initial Model

We define a carrier set ND of network nodes. It is finite and is represented by an axiom $(axm1)$. The network is supported by a directed graph g built on ND, is defined by an axiom $(axm2)$. An axiom $(axm3)$ specifies that there is no self loop connection in the network means any node is not directly connected to itself. Axioms $(axm4$ and $axm5)$ represent that the total functions map from a carrier set Msg to a set of nodes ND. The constants $source$ and $target$ are two necessary fields of a data packet for presenting source and destination references. An additional constant $(closure)$ is defined by axioms$(axm6 - axm9)$ that formalizes the transitive closure of binary relations between a set of nodes (ND). Note that ";" denotes forward relational composition.

$axm1 : finite(ND)$	$axm6 : closure \in (ND \leftrightarrow ND) \rightarrow (ND \leftrightarrow ND)$
$axm2 : g \subseteq ND \times ND$	$axm7 : \forall r \cdot r \subseteq closure(r)$
$axm3 : id(ND) \cap g = \varnothing$	$axm8 : \forall r \cdot closure(r); r \subseteq closure(r)$
$axm4 : source \in Msg \rightarrow ND$	$axm9 : \forall r, s \cdot r \subseteq s \wedge s; r \subseteq s \Rightarrow closure(r) \subseteq s$
$axm5 : target \in Msg \rightarrow ND$	

In abstract model, we formalize behavior of the basic communication protocol and dynamic environment where links may go ups and down at any time. New variables $sent$, got and $lost$ $(inv1 - inv3)$ are introduced to represent the set of sending data packets by any source node, successfully received data packets by any destination node and lost data packets due to network failure, respectively. A variable $ALinks$ (i.e active link) represents a set of links that currently up and keeps up-to-date information about all adding and removing links in the network. An invariant $(inv5)$ represents a safety property and states that all the received and lost data packets are subset of the sending data packets.

The sets got and $lost$ are disjoint $(inv6)$ since a data packet cannot be simultaneously both received and lost. We include events modeling, atomic transfer of the data packets between moving nodes, successfully receiving of the data packets by destination node, losing of the data packets due to network failure and dynamic changing in network topology.

$inv1 : sent \subseteq Msg$
$inv2 : got \subseteq Msg$
$inv3 : lost \subseteq Msg$
$inv4 : ALinks \in ND \leftrightarrow ND$
$inv5 : got \cup lost \subseteq sent$
$inv6 : got \cap lost = \varnothing$

EVENT sending
ANY s,t,data_msg
WHERE
grd1 : $data\_msg \in Msg$
grd2 : $data\_msg \notin sent$
grd3 : $s \in ND \wedge t \in ND \wedge s \neq t$
grd4 : $source(data\_msg) = s$
grd5 : $target(data\_msg) = t$
THEN
act1 : $sent := sent \cup \{data\_msg\}$
END

There are five significant events in our abstract model. An event $sending$ represents the sending of a data packet $(data\_msg)$ from a source node (s) to a destination node (t). Guards of this event state that a new data packet $(data\_msg)$ is sending from the source node (s) to the destination node (t) and both source and destination are different nodes. An event $receiving$ represents for successful receiving of the data packet $(data\_msg)$ by the destination node (t). A guard $(grd1)$ of $receiving$ states that the sending data packet $(data\_msg)$ is a member of $sent$ and the data packet is not received by either got or $lost$ variables. The data packet $(data\_msg)$ has correct references of the source node (s) and the destination node (t) is represented by a guard $(grd2)$.

EVENT receiving
ANY s,t,data_msg
WHERE
grd1 : $data\_msg \in sent \setminus (got \cup lost)$
grd2 : $source(data\_msg) = s \wedge$
$target(data\_msg) = t$
THEN
act1 : $got := got \cup \{data\_msg\}$
END

EVENT losing
ANY s,t,data_msg
WHERE
grd1 : $data\_msg \in sent \setminus (got \cup lost)$
grd2 : $source(data\_msg) = s \wedge$
$target(data\_msg) = t$
grd3 : $s \mapsto t \notin closure(ALinks)$
THEN
act1 : $lost := lost \cup \{data\_msg\}$
END

An event $losing$ represents loss of data packets due to network failure or suddenly powered off of any node or moving of node to new location, and disconnected from the network. Guards state that the sending data packet $(data\_msg)$ is not received by either got or $lost$ variables and there is not any valid connected route from the source node (s) to the destination node (t). Guard $grd3$ states that a data packet never gets the destination (t) node when path is broken.

EVENT add_link
ANY x,y
WHERE
grd1 : $x \mapsto y \notin ALinks$
grd2 : $x \neq y$
THEN
act1 : $ALinks := ALinks \cup \{x \mapsto y\}$
END

EVENT remove_link
ANY x,y
WHERE
grd1 : $x \mapsto y \in ALinks$
grd2 : $x \neq y$
THEN
act1 : $ALinks := ALinks \setminus \{x \mapsto y\}$
END

There is no more fixed infrastructure in wireless ad hoc network and every node in the network works as router and all nodes move from one place to other place without giving any information, so network link information always changes. For modeling this dynamic behavior in the system we have proposed the two events add_link and $remove\_link$. Some new arbitrary links come up and some old links are removed from the network. New links are added to the set of $ALinks$ and old link are removed from the set $ALinks$ (if it is not existing). This event always keeping up-to-date information of the ad hoc network.

5.2 First Refinement : Store and Forward Architecture

In the abstract model, we have presented that the data packets have been transferred in an atomic step from the source node to the destination node. But in real protocol the data packet is transferred hop by hop from the source node (s) to the destination node (t). So our goal is to model the store and forward architecture, where all nodes are not directly connected, and a data packet must pass through a number of intermediate nodes before reaching to the destination node. We introduce a new variable $gstore$ as binary relation between ND and Msg is represented by invariant $(inv1)$.

$inv1 : gstore \in ND \leftrightarrow Msg$
$inv2 : \forall i \cdot i \in ND \wedge i \in dom(gstore) \Rightarrow (got \cup lost) \cap gstore[\{i\}] = \varnothing$
$inv3 : ran(gstore) \cup (got \cup lost) = sent$
$inv4 : \forall i \cdot i \in ND \Rightarrow gstore[\{i\}] \subseteq sent$
$inv5 : \forall m \cdot m \in Msg \wedge m \notin sent \Rightarrow \begin{pmatrix} m \notin got \quad \wedge m \notin lost \quad \wedge \\ (\forall i \cdot i \in ND \Rightarrow i \mapsto m \notin gstore) \end{pmatrix}$
$inv6 : \forall m, i, j \cdot i \mapsto m \in gstore \wedge j \mapsto m \in gstore \Rightarrow i = j$

In the network, any data packet is stored by either $got \cup lost$ or in local variable $gstore$ by any node is represented by invariant $(inv2)$. Invariant $(inv3)$ represents a set of total distributed data packets $(ran(gstore) \cup (got \cup lost))$ in the network is equal to the sending data packets $(sent)$. Each sending data packet is belonging from the set of sending data packets $(sent)$ is given in invariant $(inv4)$. Next invariant $(inv5)$ states that a new data packet is not a member of the network distributed data packets if it is not member of the sending data packets $(sent)$. Same data packet is not mapped by two different nodes in relation $(gstore)$ is represented by last invariant $(inv6)$, means a node cannot store contradictory information about the same data packet.

A new event $forward$ introduces in this refinement, is used to transfer the data packets between two connected neighbouring nodes in the route. First two guards represent that a new sending data packet is not received by $got \cup lost$, and intermediate nodes x and y are directly connected. Third and fourth guards state that a destination node is t of a data packet $(data\_msg)$, and an intermediate node x is not the destination node (t). Last two guards represent that the data packet $(data\_msg)$ is stored at the node x, not at the node y. In this refinement, we introduce some new guards and actions in events $sending, receiving$ and $losing$.[1]

EVENT forward
ANY t,x,y,data_msg
WHERE
grd1 : $data\_msg \in sent \setminus (got \cup lost)$
grd2 : $x \mapsto y \in ALinks$
grd3 : $target(data\_msg) = t$
grd4 : $x \neq target(data\_msg)$
grd5 : $x \mapsto data\_msg \in gstore$
grd6 : $y \mapsto data\_msg \notin gstore$
THEN
act1 : $gstore := (gstore \setminus \{x \mapsto data\_msg\}) \cup \{y \mapsto data\_msg\}$
END

EVENT sending
$\oplus$ $grd7 : s \mapsto data\_msg \notin gstore$
$\oplus$ $act2 : gstore := gstore \cup \{s \mapsto data\_msg\}$
EVENT receiving
$\oplus$ $grd4 : t \mapsto data\_msg \in gstore$
$\oplus$ $act2 : gstore := gstore \setminus \{t \mapsto data\_msg\}$
EVENT losing
$\oplus$ $grd4 : x \mapsto data\_msg \in gstore$
$\oplus$ $act2 : gstore := gstore \setminus \{x \mapsto data\_msg\}$

Note that, together with the events $sending$, $receiving$, $losing$, $remove\_link$, add_link and $forward$ from initial model and all defined invariants establish **System Requirement 1**.

5.3 Second Refinement : Routing Update

In this refinement, we introduce a routing table or a route cache for updating the route information from the dynamic changing network. In the DSR protocol any node updates the local routing table, when a node wants to send data packets to any destination node and a route is not available in a local routing table. We define a new variable $alinks$ as $alinks \in ND \rightarrow (ND \leftrightarrow ND)$, and it represents that the routing information is stored by each node. The local routing table $(alinks)$ always keeps some stale links information due to continue changing of nodes location in the ad hoc network. We introduce a new event $update\_routing\_table$ for updating the routing table. First two guards of this event represent that the path is not existing between a source node (s) to a destination node (t). A set of links, which generates a route from the source node (s) to any other node (x) is represented as a strongly connected graph by last three guards $(grd3 - grd5)$. An action

[1] $\oplus$: To add a new guard and an action. , $\ominus$: To remove an old guard and an action

$(act1)$ of $update\_routing\_table$ states that the set of nodes E updates their local routing table using a variable $(routeSet)$. We introduce local routing table variable $alinks$ and some new guards in all other events of basic communication protocol.

EVENT update_routing_table
ANY s,t,E,routeSet
WHERE
grd1 : $s \in ND \wedge t \in ND$
grd2 : $s \mapsto t \notin closure(alinks(s))$
grd3 : $routeSet \in ND \leftrightarrow ND$
grd4 : $E \subseteq \{x | x \in ND \wedge s \mapsto x \in closure(ALinks)\}$
grd5 : $routeSet \subseteq closure(E \lhd ALinks)$
THEN
act1 : $alinks := alinks \Leftarrow (\lambda n \cdot n \in E | alinks(n) \cup routeSet)$
END

EVENT sending
$\oplus$ $grd8 : s \mapsto t \in closure(alinks(s))$
$\oplus$ $act2 : gstore := gstore \cup \{s \mapsto data\_msg\}$
EVENT losing
$\ominus$ $grd3 : s \mapsto t \notin closure(ALinks)$
$\oplus$ $grd5 : s \mapsto t \notin closure(alinks(x))$
EVENT forwarding
$\oplus$ $grd7 : y \mapsto t \in closure(alinks(x))$

One of the key aspects of our development strategy is to specify a so-called *observer event* [4]. This event $(stabilize)$ has no effect on this system state itself as its action is **skip**. Rather, its guard is used to define the notion of a *stable state* of the system.

EVENT stabilize
ANY
WHERE
grd1 : $\forall x, y \cdot x \mapsto y \in ALinks \Leftrightarrow x \mapsto y \in alinks(y)$
grd2 : $\forall n, m \cdot m \mapsto n \in closure(ALinks) \Rightarrow (\forall k \cdot (k \mapsto m \in alinks(n) \Leftrightarrow k \mapsto m \in alinks(m)))$
THEN
skip
END

First guard of event $stabilize$ represents that every node y knows the correct status of all connected links, i.e., y has detected all environment changes with respect to connected links. The next guard represents that if there is a path from a node m to n, then n has the same (up) information as m for all connected links to m. Hence, the observer event fires in those states where nodes know the correct status of their neighbors and this status has already been propagated through the network along all links. Intuitively, in stable states, all nodes have the maximum knowledge of the environment that can be acquired by route discovery and communication. We say that the system is in stable state when observer event $(stabilize)$ can fire.[2] A central property that we proved is as follows:

Theorem 1 (Stability implies correct local view). *If the system is stable, then for any strongly-connected component G in the network and any node n in G, n has the correct view of the status (up) of all links in G.*

We formulate this theorem in Event-B as follows, where $guardStablize$ refers to the guards of the observe event $(stabilize)$.

$$guardStablize \Rightarrow (\forall G \cdot (\forall f, l \cdot f \in G \wedge l \in G \wedge f \neq l \Rightarrow f \mapsto l \in closure(ALinks)) \Rightarrow (\forall n \cdot n \in G \Rightarrow G \lhd alinks(n) \rhd G = G \lhd ALinks \rhd G))$$

Here, a set of nodes G defines a strongly-connected component of the graph whose edge relation defines by $ALinks$, when for every distinct pair of node f and l in G, then $f \mapsto l \in closure(ALinks)$. The operators $\lhd$ and $\rhd$ respectively restrict the domain

[2] This notion of system stability is an instance of the general notion of a *stable system property* [7,4],which is a property P is true of any reachable state s then P is true of all states reachable from s.

and the range of relation to a set. The theorem itself constitutes part of the proof of **System Requirements 2**. Namely, in a stable state, each node has the correct view of all links in its strongly-connected components.

5.4 Third Refinement : Route Discovery Protocol

The route discovery protocol is an important and complex refinement of this model. We define two carrier sets rrq and rrp of route request packets and route reply packets, respectively. Two extra constants $source\_rrq$, and $target\_rrq$ represent the total function maps a set of route request packets rrq to a set of nodes ND, for storing the source and destination references in each route request packet. A new constant ($source\_rrp$) represents the total function maps a set of route reply packets rrp to a set of nodes ND for initializing the source node for each route reply packet. Two new variables $bcast\_rrq$ and $network\_rrp$ are defined as a subset of route request packets (rrq) and route reply packets (rrp), respectively.

$axm1 : source\_rrq \in rrq \rightarrow ND$
$axm2 : target\_rrq \in rrq \rightarrow ND$
$axm3 : source\_rrp \in rrp \rightarrow ND$
$inv1 : bcast\_rrq \subseteq rrq$
$inv2 : network\_rrp \subseteq rrp$

The route request packet identifies the node, referred to as the destination node of the route discovery, for which route is requested. If the route discovery is successful then the source node receives a route reply packet listing a sequence of network hops through which it may reach to the destination node.

Two new events $broadcast\_rrq$ and $received\_rrq$ are introduced in this refinement of the route discovery protocol. The event $broadcast\_rrq$ broadcasts a route request packet for discovering a route to any destination node. First two guards ($grd1 - grd2$) of this event represent that the route is not existing between the source node (s) to the destination node (t). Next guard presents type of rrq_pkt. Last three guards ($grd4 - grd6$) state that each new route request packet rrq_pkt have references of the source node (s) and the destination node (t), then the route request packet rrq_pkt is broadcasted by initial node for discovering a new route.

EVENT broadcast_rrq
ANY s, t, rrq_pkt
WHERE
grd1 : $s \in ND \wedge t \in ND$
grd2 : $s \mapsto t \notin closure(alinks(s))$
grd3 : $rrq\_pkt \in rrq$
grd4 : $rrq\_pkt \notin bcast\_rrq$
grd5 : $source\_rrq(rrq\_pkt) = s$
grd6 : $target\_rrq(rrq\_pkt) = t$
THEN
act1 : $bcast\_rrq := bcast\_rrq \cup \{rrq\_pkt\}$
END

EVENT received_rrq
ANY t, rrq_pkt, rrp_pkt
WHERE
grd1 : $t \in ND$
grd2 : $rrq\_pkt \in bcast\_rrq$
grd3 : $target\_rrq(rrq\_pkt) = t$
grd4 : $source\_rrq(rrq\_pkt) \neq t$
grd5 : $rrp\_pkt \notin network\_rrp$
THEN
act1 : $network\_rrp := network\_rrp \cup \{rrp\_pkt\}$
act2 : $bcast\_rrq := bcast\_rrq \setminus \{rrq\_pkt\}$
END

A new event $received\_rrq$ returns a route reply packet rrp_pkt to the initial node with discovered route information from the network. Guards ($grd1 - grd4$) of this event state that the broadcasted route request packet (rrq_pkt) is received by the destination node (t) and the source node (s) of the route request packet is not same as the destination node (t). Last guard states that the returning route reply packet (rrp_pkt) is not already received by the route requesting node. Actions of this event state that the destination node returns the route reply packet (rrp_pkt) to the initial node and remove the route request packet (rrq_pkt) from the network.

Note that, together with the events $broadcast\_rrq$, $received\_rrq$ and $update\_routing\_table$ from initial model and all invariants establish **System Requirement 3**.

5.5 Fourth Refinement : Continue Route Discovery Protocol

The route discovery protocol discovers route in the several steps. An address of the original initiator of the request and the target of the request, each route request packet contains a route record, where it is accumulated a record of the sequence of hops taken by the route request packet as it is propagated through the ad hoc network during this route discovery. A new variable ($route\_record\_rrq$) is declared to store the link information at the time of propagation of a route request packet from one node to other node. If the route request receiver node is not the target node then it add the link information to the route record ($route\_record\_rrq$) of the route request packet (rrq_pkt) and again broadcasts it. Similarly, other new variable ($route\_record\_rrp$) used to store the a link information which is collected from the route request packet, when a destination node returns a route reply packet to the initial node. Two more invariants $(inv3, inv4)$ are introduced as safety properties, which represent that the sequence of accumulated node information and the route record information is a subset of all the connected nodes to the source node and a subset of connected links from the source node to all other nodes.

$$
\begin{array}{l}
inv1 : route\_record\_rrq \in rrq \rightarrow (ND \leftrightarrow ND) \\
inv2 : route\_record\_rrp \in rrp \rightarrow (ND \leftrightarrow ND) \\
inv3 : \forall rp, al, s \cdot s \in ND \wedge rp \in rrp \wedge al \subseteq ND \times ND \wedge al \in dom(closure) \Rightarrow \\
\qquad dom(route\_record\_rrp(rp)) \subseteq \{x \cdot s \mapsto x \in closure(al) | x\} \\
int4 : \forall al, E, rp \cdot E \subseteq ND \wedge rp \in rrp \wedge al \subseteq ND \times ND \wedge al \in dom(closure) \Rightarrow \\
\qquad route\_record\_rrp(rp) \subseteq closure(E \lhd al)
\end{array}
$$

EVENT forward_broadcast
ANY x, y, rrq_pkt
WHERE
grd1 : $x \in ND \wedge y \in ND$
grd2 : $x \mapsto y \in alinks(x) \wedge x \mapsto y \in alinks(y)$
grd3 : $rrq\_pkt \in bcast\_rrq$
grd4 : $source\_rrq(rrq\_pkt) \neq y$
grd5 : $target\_rrq(rrq\_pkt) \neq y$
THEN
act1 : $route\_record\_rrq(rrq\_pkt) := route\_record\_rrq(rrq\_pkt) \cup \{x \mapsto y\}$
act2 : $bcast\_rrq := bcast\_rrq \cup \{rrq\_pkt\}$
END

EVENT update_routing_table
$\ominus$ $grd4 : E \subseteq \{x | x \in ND \wedge s \mapsto x \in closure(ALinks)\}$
$\ominus$ $grd5 : routeSet \subseteq closure(E \lhd ALinks)$
$\oplus$ $grd3 : E = dom(route\_record\_rrp(rrp\_pkt))$
$\oplus$ $grd4 : routeSet = route\_record\_rrp(rrp\_pkt)$
EVENT received_rrq
$\oplus$ $grd6 : s \mapsto t \in closure(route\_record\_rrq(rrq\_pkt))$
$\oplus$ $grd7 : route\_record\_rrp(rrp\_pkt) = route\_record\_rrq(rrq\_pkt)$

A new event $forward\_broadcast$ introduces for broadcasting a route request packet to neighboring nodes, when any node is not the destination node for a route discovery process. First two guards state that node x is directly connected with node y and this information is stored by a local routing table of nodes x and y. Next guard ($grd3$) states that a route request packet (rrq_pkt) is already broadcasted and last two guards ($grd4, grd5$) state that node y is not either source or destination nodes of the route request packet. Two actions of this event, add a new link information ($x \mapsto y$) as a route record of the route request packet rrq_pkt, and again broadcasts it continue for route discovery process. This process is repeated many times, until the destination node does not receive the route request packet. In this refinement, we introduce some new guards and remove some old guards from events $update\_routing\_table$ and $received\_rrq$.

5.6 Fifth Refinement : Sequence Number

In this last refinement, we introduce a constant $seqNo$ as $seqNo \in rrq \rightarrow \mathbb{N}1$ for representing a sequence number stored in each route request packet. The sequence number is set by the initiator from a locally-maintained sequence number. In order to detect duplicate route requests received packets, each node in the ad hoc network maintains a list of the route request packet that it has recently received on any route request. The route request thus propagates through the ad hoc network until it reaches the destination node, which then replies to the initiator. The original route request packet is received only by those nodes within wireless transmission range of the initiating node, and each of these nodes propagates the request if it is not the target and if the request does not appear to this host to be redundant. Discarding the request as well as recently seen request packet because the address of node is already listed in the route record guarantees that no single copy of the request can propagate around a loop [1]. A new variable $store\_rrq$ is declared as $store\_rrq \in ND \rightarrow \mathbb{P}(rrq)$, which represents a recently seen request table by each nodes. The recently seen request table keeps all visited route request packets information.

A new event $forward\_broadcast\_skip$ is used to discard the route request packet when the request packet is already stored by the recently seen request table ($store\_rrq$). This event is refinement of the event $forward\_broadcast$. Guard ($grd5$) of this event states that route request packet (rrq_pkt) is already received by a node y and it is stored by the recently seen request table ($store\_rrq$). Last guard states that the sequence number ($seqNo$) of the received route request packet is already stored by the recently seen request table ($store\_rrq$) of a node y.

EVENT forward_broadcast_skip
ANY x, y, rrq_pkt
WHERE
grd1 : $x \in ND \land y \in ND \land x \mapsto y \in alinks(x)$
grd2 : $rrq\_pkt \in bcast\_rrq$
grd3 : $source\_rrq(rrq\_pkt) \neq y$
grd4 : $target\_rrq(rrq\_pkt) \neq y$
grd5 : $rrq\_pkt \in store\_rrq(y)$
grd6 : $seqNo(rrq\_pkt) \in \{p \cdot p \in store\_rrq(y) | seqNo(p)\}$
THEN
skip
END

EVENT broadcast_rrq
$\oplus$ $act2 : store\_rrq(s) := store\_rrq(s) \cup \{rrq\_pkt\}$
EVENT forward_broadcast
$\oplus$ $grd6 : rrq\_pkt \notin store\_rrq(y)$
$\oplus$ $grd7 : seqNo(rrq\_pkt) \notin \{p \cdot p \in store\_rrq(y) | seqNo(p)\}$
$\oplus$ $act3 : store\_rrq(y) := store\_rrq(y) \cup \{rrq\_pkt\}$

When route request initiator node (s) broadcasts the route request packet (rrq_pkt), the route request packet is stored in the recently seen request table ($store\_rrq$), which is represented by an extra action in the event $broadcast\_rrq$. Two extra guards $(grd6, grd7)$ and an action $(act3)$ are introduced in the event $forward\_broadcast$. The guards state that a new request packet (rrq_pkt) is received by a node y and sequence number ($seqNo$) of route request packet is different from the recently seen request table ($store\_rrq$). The action $(act3)$ states that the intermediate node y stores the route request packet (rrq_pkt) by the recently seen request table ($store\_rrq$).

5.7 Proof Statistics

Table-2 is expressing the proof statistics of the formal development of DSR protocol in the Rodin tool. These statistics measure the size of the model, the proof obligations

generated and discharged by the Rodin platform, and those interactively proved. The complete development of the DSR protocol results in 104(100%) proof obligations, in which 83(80%) are proved completely automatically by Rodin tool. The remaining 20(20%) proof obligations are proved interactively by Rodin tool. In the model, many proof obligations are generated in first refinement due to introduction of store and forward architecture for a data packet passing in the dynamic network.

In order to guarantee the correctness of these behaviors, we have established various invariants in stepwise refinement. The stepwise refinement of the DSR protocol helps to achieve a high degree of automatic proof.

Table 2. Proof statistics

Model	Total number of POs	Automatic Proof	Interactive Proof
Abstract Model	16	16(100%)	0(0%)
First Refinement	37	20(55%)	17(45%)
Second Refinement	15	13(91%)	2(9%)
Third Refinement	5	5(100%)	0(0%)
Fourth Refinement	19	17(89%)	2(11%)
Fifth Refinement	12	12(100%)	0(0%)
Total	104	83(80%)	21(20%)

6 Discussion and Conclusion

Discussion: We have found some works on using model checkers and theorem provers to verify properties of routing protocol by O.Wibling et al. [8] and relatively few case studies (e.g., [4,9]) using formal methods to develop different kinds of protocols. Yang et al. [10] have presented both safety and liveness properties of the DSR protocol. The proofs have been mechanically checked using theorem proving tool Isabelle/HOL. Another paper [11] presents a validation model for the DSR protocol using SDL and concludes that *Route Request table* correctly updated after receiving RREP.

This paper contributes to incremental formal development of the DSR protocol using proof-based refinement. The specification is performed in a stepwise manner composing more advanced routing components between the abstract specification and topology. An incremental development helps to verify consistency and correctness of the system. This formal model is designed according to the requirements of the DSR protocol, and provides main characteristics of ad hoc network in form of dynamic networks: nodes can be added and deleted in a dynamic manner. We have introduced several invariants as *safety* properties to verify the system and introduce *liveness* properties that characterize when the system reaches stable states. All these invariants are useful to generate the test cases from formal models, which can be used for testing like route discovery, route updating and response time etcetera.

Conclusion: We have presented a case study for formalizing and reasoning about the DSR protocol in Event-B. Formal development of the DSR protocol is presented in two phases as basic communication protocol and route discovery protocol. In basic communication protocol, we consider the data packets are passing from source node to destination node in changing network. The route discovery protocol is used to find the route from initial node to a destination node. We formalize several different developments, each highlighting different aspects of the problem, making different assumptions and establishing different properties. We consider the case of dynamic environment and express properties for holding the stable states. We have explained our approach for developing DSR protocol using refinement, which allow us to achieve a very high degree of automatic proof. The powerful support is provided by the Rodin tool. Rodin proof

is used to generate the proof obligations and to discharge those obligations automatically and interactively. Our approach is the methodology of separation of concerns: first prove the algorithm at an abstract level; then gradually introduce the peculiarity of the specific protocol.

What is important about our approach is that the fundamental properties, we have proved at the beginning, namely the reachability and the uniqueness of a solution, are kept through the refinement process (provided, of course, the required proofs are done). Our different developments reflect not only the many facets of the problem, but also that there was a learning process involved in understanding the problem and its solution. It seems to us that this sort of approach is highly ignored in the literature of protocol developments [10,11] where, most of the time, things are presented in a flat manner directly at the level of the final protocol itself. In addition, the proposed methodology is generic and can be easily applied to other routing protocols for an ad hoc networks. It can also be applied to large-scale system and to extended ad hoc networks like reactive routing protocols.

Acknowledgments. Work of Neeraj Kumar Singh is supported by grant awarded by the Ministry of University and Research.

References

1. Johnson, D.B., Maltz, D.A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: Mobile Computing. The International Series in Engineering and Computer Science, vol. 353, pp. 153–181. Springer, US (1996) ISSN 0893-3405
2. Project RODIN: Rigorous open development environment for complex systems (2004), http://rodin-b-sharp.sourceforge.net/ (2004-2007)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
4. Hoang, T.S., Kuruma, H., Basin, D.A., Abrial, J.R.: Developing Topology Discovery in Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 1–19. Springer, Heidelberg (2009)
5. Cansell, D., Méry, D.: The Event-B Modelling Method: Concepts and Case Studies, pp. 33–140. Springer, Heidelberg (2007); See [12]
6. Leavens, G.T., Abrial, J.R., Batory, D.S., Butler, M.J., Coglio, A., Fisler, K., Hehner, E.C.R., Jones, C.B., Miller, D., Jones, S.L.P., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: Proceedings of the 5th Inter. Conf. on Generative Programming and Component Engineering (GPCE), pp. 221–236 (2006)
7. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
8. Wibling, O., Parrow, J., Pears, A.: Automatized Verification of Ad Hoc Routing Protocols, pp. 343–358. Springer, Heidelberg (2004)
9. Abrial, J.R., Cansell, D., Méry, D.: A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. Formal Asp. Comput. 14(3), 215–227 (2003)
10. Yang, H., Zhang, X., Wang, Y.: A Correctness Proof of the DSR Protocol. In: Cao, J., Stojmenovic, I., Jia, X., Das, S.K. (eds.) MSN 2006. LNCS, vol. 4325, pp. 72–83. Springer, Heidelberg (2006)
11. Cavalli, A., Grepet, C., Maag, S., Tortajada, V.: A Validation Model for the DSR Protocol. In: Proceedings of the 24th ICDCSW 2004, vol. 7, pp. 768–773. IEEE Computer Society, Los Alamitos (2004)
12. Bjørner, D., Henson, M.C. (eds.): Logics of Specification Languages. EATCS Textbook in Computer Science. Springer, Heidelberg (2007)

Self-Stabilizing De Bruijn Networks

Andréa Richa[1,⋆], Christian Scheideler[2,⋆⋆], and Phillip Stevens[1]

[1] Computer Science and Engineering, SCIDSE, Arizona State University
Tempe, AZ 85287, USA
{aricha,pcsteven}@asu.edu
[2] Department of Computer Science, University of Paderborn,
D-33102 Paderborn, Germany
scheideler@upb.de

Abstract. This paper presents a dynamic overlay network based on the De Bruijn graph which we call *Linearized De Bruijn (LDB) network*. The LDB network has the advantage that it has a guaranteed constant node degree and that the routing between any two nodes takes at most $O(\log n)$ hops with high probability. Also, we show that there is a simple local-control algorithm that can recover the LDB network from any network topology that is weakly connected.

1 Introduction

Peer-to-peer networks (P2P) are characterized by their lack of centralized control and scalability. While such qualities make them highly versatile, they also make it difficult to design efficient routing algorithms that do not break down under high network flux. One approach to designing a P2P network is to start with a classical family of graphs which has all the qualities desired for the P2P network. Then, one would try to design a dynamic variant of this family of graphs that is able to accommodate any number of peers and that can handle a high rate of joining and leaving peers. Such an approach is used in the design of the P2P networks presented, e.g., in [18], [21], [19], [16], and [15].

A general approach to transform classical families of graphs into dynamic P2P networks was formalized by Naor and Wieder in [16] and is called the *continuous-discrete approach*. The basic idea of this approach is to first transform a classical family of graphs into an infinite-size network in a continuous space that preserves essential properties of these graphs. The communication algorithms needed can be easily understood and designed under this space. Following this, one must simply find a way to transform the network from the continuous to the dynamic discrete domain that preserves the basic properties of the network as well as these algorithms. However, while this approach yields P2P networks that are easy to maintain and to use as long as the topology stays in the desired form, it is not clear how these networks can recover from a misconfigured topology.

⋆ Supported in part by NSF awards CCF-0830791 and CCF-0830704.
⋆⋆ Supported in part by DFG awards SCHE 1592/1-1 and SFB 901 (On-the-Fly Computing).

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 416–430, 2011.

In this work, we present the *Linearized De Bruijn (LDB)* network, which is based on a discretization of a continuous variant of the classical De Bruijn network, and which preserves the static network's $O(\log n)$ *routing time and constant node degrees*. Moreover, the LDB network is *self-stabilizing* in the sense that it can recover from any case in which the topology is still weakly connected. Recall that a directed graph G is called *weakly connected* if for any pair of nodes v, w there is a path in G from v to w when considering all edges to be undirected. Other dynamic variants of the De Bruijn network have been proposed in the literature before (e.g., [16]), but none of them is self-stabilizing.

This paper is organized as follows. In Section 2, we present the related work in the literature and Section 3 defines the structure of the LDB network. We then present our routing algorithm in Section 4 and prove its logarithmic bound. Section 5 presents the self-stabilization results for the LDB while Section 6 describes and analyzes join and leave operations. Section 7 concludes the paper and also presents some lines for future work.

2 Related Work

Our work expands on that of Naor and Wieder in [16]. Both our construction and their distance-halving network are based on a continuous extension of the De Bruijn graph to the unit interval. However, our self-stabilizing construction provides additional fault-tolerance while maintaining constant node degree. Naor and Wieder's first construction has constant average degree w.h.p. but does not offer a way to recover from faults. They also suggest a construction which offers stronger results with regards to fault tolerance, but requires an increase in the average degree to $\Theta(\log n)$. The De Bruijn network has also been used as the basis for several other peer-to-peer networks, including [1], [7], [11], and [14]. These constructions also achieve logarithmic routing and constant average degree w.h.p., but none of them is self-stabilizing.

Various self-stabilizing overlay networks have been designed in recent years. Cramer and Fuhrmann [5] present a self-stabilizing ring network and Caron et al. [3] describe a Snap-Stabilizing Prefix Tree for Peer-to-Peer systems. Onus et al. [17] present a linearization technique to transform an arbitrary connected network into a sorted list. Our paper is based on this technique and shows how to extend it to dynamic De Bruijn networks. Clouser et al. [4] propose another variant of the linearization technique to construct a deterministic self-stabilizing skip list. Jacob et al. [10] generalize the linearization technique to two dimensions and present a self-stabilizing construction for Delaunay graphs. In another paper, Jacob et al. [9] present a self-stabilizing variant of the skip graph and show that it can recover its network topology from any weakly connected state in $O(\log^2 n)$ communication rounds with high probability. Gall et al. [8] discuss models that capture the parallel time complexity of locally self-stabilizing networks that avoids bottlenecks and contention. Recently, Kniesburges et al. [13] showed how to obtain a self-stabilizing Chord network. In [2] the authors present a general framework for the self-stabilizing construction of any overlay network.

However, the algorithm requires the knowledge of the 2-hop neighborhood for each node and involves the construction of a clique. In that way, failures at the structure of the overlay network can easily be detected and repaired. All of the constructions above that result in networks of low diameter and high expansion (such as the skip graphs) result in a logarithmic degree while our network guarantees a constant degree.

3 The Linearized De Bruijn Network

The d-dimensional De Bruijn graph is an undirected graph $G = (V, E)$ with node set $V = \{0,1\}^d$ and an edge set E in which every node with label $(x_1, \ldots, x_d) \in \{0,1\}^d$ is connected to the nodes $(0, x_1, \ldots, x_{d-1})$ and $(1, x_1, \ldots, x_{d-1})$. When letting $d \to \infty$ and interpreting every label $(x_1, \ldots, x_d)$ as $x = \sum_{i=1}^{d} x_i/2^i$, the node set converges to $U = [0,1)$ and the edge set to the family $F = \{f_0, f_1\}$ of functions

$$f_0(x) = x/2 \quad \text{and} \quad f_1(x) = (1+x)/2\,.$$

Thus, (U, F) represents a continuous form of the De Bruijn graph. The question is how to transform this continuous form into a dynamic discrete form for any number of peers. We propose the following form.

Definition 1. *The* Linearized De Bruijn network *(LDB) $G = (V, E)$ is a directed graph where the node set V can be partitioned into the set of* real *nodes V_R and a set of* virtual *nodes V_V. Each real node $v \in V_R$ has a real-valued label*[1] *in the interval $(0,1)$; in addition, each $v \in V_R$ hosts two virtual nodes in V_V: a* left *virtual node, $l(v)$, with label $\frac{v}{2}$ and a* right *virtual node, $r(v)$, with label $\frac{v+1}{2}$. The collection of all real and virtual nodes $v \in V$ is arranged in sorted order of their labels, and $(v, w) \in E$ if and only if v and w are consecutive in the linear ordering* (linear edges) *or w is a virtual node of v* (virtual edges)*.*

The definition above reflects the ideal, stable state of the LDB, and our goal is to provide a self-stabilizing mechanism to get to that state from any weakly connected graph. Note that the virtual edges between a real node v and its virtual nodes $l(v)$ and $r(v)$ are static throughout the self-stabilization process as all these nodes are hosted by v, so v can maintain them directly. These virtual edges actually constitute the edges of the continuous De Bruijn construction. In addition to that, v may have a collection of non-virtual edges that start at v, $l(v)$ or $r(v)$ and that eventually are to be transformed into the linear edges in the definition. As we will see, the combination of virtual and linear edges will allow De Bruijn-like routing in the LDB.

In the following, we will say that a node v is to the *right* (resp., *left*) of a node w whenever the label of v is greater (resp., smaller) than the label of w. Given a (real or virtual) node v, we define $N(v)$ as the *neighborhood* of v, which consists of all (real and virtual) nodes that can be reached via non-virtual edges from v. In other words, $N(v)$ represents the current knowledge of v of

[1] We may indistinctly use v to denote a node or its label, when clear from the context.

the other nodes in the network. We define $pred(v) = \max\{w \in N(v) \mid w < v\}$ and $succ(v) = \min\{w \in N(v) \mid w > v\}$. In the ideal state, $pred(v)$ and $succ(v)$ represent the linear edges of v, while in the non-ideal case, $pred(v)$ and $succ(v)$ may just be candidates for v's linear edges.

Analogous to the consistent hashing approach [12], we assume that the real nodes are assigned to points in $(0, 1)$ in a pseudorandom manner, so we can assume that the real nodes are distributed uniformly at random over the interval $(0, 1)$. Note that in the LDB network every real node has a degree of at most 8 (two linear edges for each of v, $l(v)$ and $r(v)$, all hosted by node v, plus the two virtual edges $(v, l(v))$ and $(v, r(v))$). As the LDB network organizes the nodes in a sorted list (in the ideal state), it may be used similar to [12] to construct a distributed hash table.

4 Routing Algorithm

In this section, we outline the algorithm we will use when routing from a node v to a destination node w in the LDB network. In order to implement our algorithm, we must first get a good estimate for $c \log n/n$ (in this paper, all logarithms are to the base 2) for a constant c. For this we use the following lemma adapted from [20]. Recall that the n real nodes are distributed uniformly and independently at random over $(0, 1)$.

Lemma 1. *Let $I(j) \subseteq (0, 1)$ be any interval of size $(1/2)^j$ starting at a real node and let $N(j)$ be the number of real nodes in $I(j)$. For any constant $c > 1$ there is a constant $\epsilon \in (0, 1)$ (that can be arbitrarily small depending on c) so that w.h.p.[2] it holds: if $|I(j)| < (1 - \epsilon)(c \log n)/n$ then $j > N(j)/c - \log N(j)$ and if $|I(j)| > (1 + \epsilon)(c \log n)/n$ then $j < N(j)/c - \log N(j)$.*

Proof. Suppose that $|I(j)| < (1 - \epsilon)c \log n/n$ for some constant $\epsilon > 0$. For $\delta > 0$ with $(1-\delta)^2 = (1-\epsilon)$ it follows from the Chernoff bounds that $N(j) = \alpha c \log n$ for some $\alpha \leq 1 - \delta$ w.h.p. given that c is large enough compared to δ (resp. ϵ). Also, $N(j) \geq 1$ as $I(j)$ starts at a real node, so $\alpha \geq 1/(c \log n)$. As $(1 - \delta)^2 c \log n/n < \alpha c \log n/n^\alpha$ for any $1/(c \log n) \leq \alpha \leq 1 - \delta$ if n is sufficiently large, it follows that in this case $|I(j)| < N(j)/2^{N(j)/c}$ and therefore $j > N(j)/c - \log N(j)$. Thus, it holds that as long as $|I(j)| < (1 - \epsilon)(c \log n/n)$, $j > N(j)/c - \log N(j)$ w.h.p. The other side is shown in a similar way. □

The lemma allows us to accurately estimate $c \log n/n$ which will be helpful for defining appropriate intervals to identify shortcuts in the routing: starting with a real node v, go to the right until a point x is reached so that for the interval $I(j) = [v, x)$ it holds that $j < N(j)/c - \log N(j)$ for the first time. According to the lemma, it holds for this interval that $|I(j)|$ is within $(1 - \epsilon)(c \log n)/n$ and $(1 + \epsilon)(c \log n)/n$ w.h.p. Similar bounds also hold if $I(j)$ is required to end at a real node as the distance between two consecutive real nodes in $(0, 1)$ can

[2] With high probability, that is, with probability at least $1 - 1/n^c$, for some constant $c > 0$.

be shown to be at most $(\epsilon c \log n)/n$ w.h.p. (given that c is sufficiently large compared to ϵ) so that the deviation from $(c \log n)/n$ increases to a factor of at most $(1 \pm 2\epsilon)$. The lemma also allows us to accurately estimate $\log n$: let $I(j)$ be defined as above. Then j is within $\log n - \log\log n - \log c - \log(1 \pm \epsilon)$ and hence $j + \log j + \log c$ is equal to $\log n$ up to some small additive constant independent of c w.h.p.

We can interpret the nodes $x_0, x_1, \ldots, x_k$ in the definition below as the first k nodes one would have followed when routing on the continuous De Bruijn network from node v_0 to a destination given by a node whose highest order k bits are $b_{k-1}, \ldots, b_0$.

Definition 2. *Let b_0, ..., b_{k-1} be a sequence of bits and $v_0 \in (0,1)$. We define a* sequence of ideal De Bruijn hops *x_0, ..., x_k recursively by $x_0 = v_0$ and $x_{i+1} = x_i/2$ if $b_i = 0$ and $x_{i+1} = (1 + x_i)/2$ if $b_i = 1$.*

The routing algorithm is presented in Algorithm 1 and proceeds in three basic stages. Throughout the algorithm, v represents the node at which the message to be routed is currently located. First (Lines 1 – 12), the algorithm determines a close estimation of $\log n$ (according to Lemma 1) so that it can determine how many bits of the destination it needs to fix while emulating the classic De Bruijn routing. It also defines the intervals T_i: $T_i = [i/2^j, (i+1)/2^j)$ for some integer j with $1/2^j \in [(1-\epsilon)c \log n/n, (1+\epsilon)2c \log n/n]$. As we will show later, these intervals are chosen so that our algorithm visits each of these intervals at most once during the routing. The second stage (Lines 13 – 34) is where the network actually uses the virtual edges to emulate routing in the continuous De Bruijn network. Starting from the least significant bit chosen to be fixed and proceeding towards the most significant bit in the destination address, the network will follow the left virtual edge or the right virtual edge depending on whether the bit is a 0 or 1, respectively. It must then proceed linearly from the current virtual node to find a new real node v from which to perform a De Bruijn hop (Lines 21 – 28). If the routing process detects that a later ideal De Bruijn hop, x_k, (as in Definition 2) is in the same interval T_i as the message to be routed (Line 17; we will show that we can always find such a valid x_k, which matches the respective bits of the destination in at least one more position, in the interval T_i, w.h.p.), it may skip ahead and proceed *from the current node* v as if it has already fixed all the less significant matching bits between the destination and x_k. Finally (Lines 35 – 37), the message moves linearly from x_κ towards the destination address.

Figure 1 illustrates some of the variables and actions taken by our algorithm, where v_k is equal to node v at the start of iteration k (a virtual node unless $k = 0$), y_k (resp., $y_{k'}$) is the value of the real node v found at the end of the while loop in Lines 26 – 28 in iteration k (resp., iteration k' immediately preceding iteration k), x_k is as defined in Line 17 in iteration k, T_{i_k} is equal to the interval T_i in iteration k.

Before we show a hop bound of $O(\log n)$ for the routing, we state and prove some basic facts.

Algorithm 1. Routing in the LDB

$v = v_0$
$N = 1$
repeat
  $v = succ(v)$
  if v is a real node **then**
    $N = N + 1$
  end if
until $\log(1/|v - v_0|) \leq N/c - \log N$
$j = \lceil \log(1/|v - v_0|) \rceil$
$\kappa = j + \log j + \log c$
In the following let $T_i = [i(1/2)^j, (i+1)(1/2)^j]$ for all $0 \leq i < 2^j$ and let b_i be the $(\kappa - i)$th bit of the destination address
Fix x_0, ..., x_κ recursively as in Definition 2 based on b_0, ..., $b_{\kappa-1}$ and initial point v_0
$v = v_0$
$k = 0$
while $k \neq \kappa$ **do**
  Let i be such that $v \in T_i$
  $k = max\{k : x_k \in T_i\}$
  if $k = \kappa$ **then**
    Break
  end if
  if v is left of the midpoint of T_i **then**
    Let $next(x) = succ(x)$ for all nodes x
  else
    Let $next(x) = pred(x)$ for all nodes x
  end if
  while v is a virtual node **do**
    $v = next(v)$
  end while
  if $b_k = 0$ **then**
    $v = l(v)$
  else if $b_k = 1$ **then**
    $v = r(v)$
  end if
end while
while $v \neq$ destination node **do**
  $v = pred(v)$ or $v = succ(v)$, whichever is closer to the destination.
end while

Definition 3. *Let I be some interval in $(0,1)$. We define $\Psi(I) = \{x \in (0,1) : x/2 \in I$ or $(x+1)/2 \in I\}$. If $\mathcal{I}$ is some collection of intervals in $(0,1)$ then we define $\Psi(\mathcal{I}) = \bigcup_{I \in \mathcal{I}} \Psi(I)$.*

Lemma 2. *Let I be some interval in $(0,1)$. The total length of the collection of intervals in $I \cup \Psi(I)$ is at most $3|I|$.*

Proof. Suppose $I = (i_1, i_2)$ is an interval with length l. If $I \subseteq (0, 1/2)$ or $I \subseteq (1/2, 1)$, then $\Psi(I)$ is just some interval in $(0,1)$ with length $2l$. If $1/2 \in I$, then $I = I_L \cup I_R$ where $I_L = (i_1, 1/2]$ and $I_R = [1/2, i_2)$, and $\Psi(I) = \Psi(I_L) \cup \Psi(I_R)$ which has length less than or equal to $2l$. Thus in general, if I has length l, the length of $\Psi(I) \leq 2l$ and in particular, the length of $I \cup \Psi(I)$ is at most $3l$. □

Corollary 1. *Let $\mathcal{I}$ be some collection of intervals in $(0,1)$. Then the length of the collection of intervals in $\mathcal{I} \cup \Psi(\mathcal{I})$ is at most three times the length of the collection of intervals in $\mathcal{I}$.*

The following lemma will prove useful for giving an upper bound on the length of a routing path.

Lemma 3. *Let $\mathcal{I}$ be a collection of intervals with total length at most $\frac{p \log n}{n}$ for some constant p. Then w.h.p. the number of nodes in $\mathcal{I}$ is $O(\log n)$.*

Proof. A node is in an interval in $\mathcal{I}$ if there is a real node in an interval in $D = \mathcal{I} \cup \Psi(\mathcal{I})$. So we must show that the number of real nodes in D, which has total length at most $\frac{3p \log n}{n}$, is bounded by $(1+a) \log n$, for some constant a, with high probability. Let v_1, ..., v_n be the real nodes in the network. For each v_i, define

$$X_i = \begin{cases} 1 : v_i \in D \\ 0 : v_i \notin D \end{cases}$$

Note that $E[\sum_{i=1}^{n} X_i] \leq 3p \log n$. So for $3p(1+a') = (1+a)$ we have

$$\begin{aligned} P[D \text{ has more than } (1+a)\log n \text{ real nodes}] &= P[\sum_{i=1}^{n} X_i \geq (1+a')3p \log n] \\ &\leq e^{\frac{-3a'^2 p \log n}{3}} \\ &= n^{-p'a'^2}, \end{aligned}$$

where $p' = p/\ln 2$. The inequality follows from standard Chernoff bounds. □

Now we are ready to prove that the logarithmic length of a routing path holds with high probability.

Theorem 1. *The number of edges on the routing path from a source node v to a destination node w in the LDB network G is $O(\log n)$, w.h.p.*

Proof. First we bound the number of hops performed during Lines 1 – 12. By Lemma 1 the length of the interval traversed to calculate j is $O(\log n/n)$, so

from Lemma 3 it follows that the number of hops traversed in Lines 1 – 12 is $O(\log n)$.

Next we bound the number of hops performed in the middle stage of the algorithm, Lines 13 – 34. In each iteration of loop beginning on Line 15 we have a value of k determined at Line 17, an associated x_k, an interval T_{i_k} from Line 16, a node from which we begin, which we call v_k, and a real node whose virtual node we hop to at the end of the iteration (Lines 26 – 28) which we call y_k, and an interval $S_k = [v_k, y_k]$ (or $[y_k, v_k]$). Note that, by definition, $v_k \in T_{i_k}$ and x_k always exist (since we can always take $k = 0$) — we will actually show below that the sequence of indices k computed in the while loop is strictly increasing w.h.p., implying that the sequence of x_k's found by our algorithm strictly increases the number of most significant bits matched to $b_0, \ldots, b_{k-1}$. We will also show that y_k also belongs to T_{i_k} w.h.p., implying that $S_k \subseteq T_{i_k}$ w.h.p. Figure 1 illustrates $y_{k'}, v_k, y_k, x_k$ and T_{i_k}, where k' was the value of k chosen on the iteration prior to k.

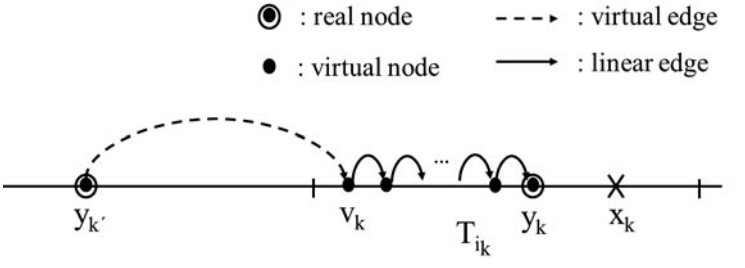


Fig. 1. An example of one iteration of the main routing loop

Claim. At every iteration k it holds w.h.p. that $k > k'$, where k' was the value of k in the previous iteration of the while loop, and $y_k \in T_{i_k}$.

Proof. Note that on the first iteration of the loop, $x_k = v_k = y_k$, so the claim holds.

Now suppose $k > 0$ and that the claim holds for the iteration prior to the current iteration, when k was equal to k'. Either $v_k = l(y_{k'})$ or $v_k = r(y_{k'})$, which implies $x_{k'+1} = l(x_{k'})$ or $x_{k'+1} = r(x_{k'})$ respectively. In either case, since $y_{k'}$ and $x_{k'}$ are both in $T_{i_{k'}}$ and each T_i has length $(1/2)^j$ and is offset from 0, $x_{k'+1} \in T_{i_k}$, implying that $k \geq k' + 1$. Furthermore, if $y_k \notin T_{i_k}$ then $|S_k| \geq c \log n/2n$ since we move in the direction from v_k to the midpoint of T_{i_k} in Lines 26 – 28. Since by definition there are no real nodes in the interior of S_k,

$$P[|S_k| \geq c \log n/2n] \leq \left(1 - \frac{c \log n}{2n}\right)^n < e^{-(c \log n)/2} = n^{-c'/2},$$

where $c' = \frac{c}{\ln 2}$. Thus w.h.p. $y_k \in T_{i_k}$. □

Since we always take k to be maximum index such that $x_k \in T_{i_k}$ (Line 17), this also implies that the message will never return to an interval after leaving it,

else we would have $x_l \in T_{i_k}$ with $l > k$, a contradiction. Hence, since $y_k \in T_{i_k}$ implies that $S_k \subseteq T_{i_k}$, we have that all S_k's are disjoint w.h.p.

Let $S = \bigcup S_k$. We will bound the total length of S w.h.p. Assume $|S| > \frac{2c \log n}{n}$. Since each S_k contains only one real node by definition, there is a set of real nodes V_0 of size at least $n - \kappa$ such that no node in V_0 falls in S. We have

$$\begin{aligned} P[\text{no } v \in V_0 \text{ falls in } S] &\leq (1 - \frac{2c \log n}{n})^{n-\kappa} \\ &\leq exp((\frac{-2c \log n}{n}) \cdot (n - 2\kappa)) \\ &= exp(-2c \log n) \cdot exp(\frac{4c\kappa \log n}{n}) \\ &= O(\frac{1}{n^{2c'}}) \end{aligned}$$

for $c' = \frac{c}{\ln 2}$, since $\kappa = \log n + \Theta(1)$. Thus w.h.p. $|S| \leq \frac{2c \log n}{n}$.

Then according to Lemma 3, w.h.p. the number of virtual nodes in S is $O(\log n)$. Since the S_i's were shown to be disjoint w.h.p., this implies that the number of hops taken before Step 32 of the algorithm is $O(\log n)$ w.h.p.

Finally, we bound the number of hops in Lines 35 – 37. After taking the final virtual hop we arrive at v_κ, which we know is in the same interval, T_{i_κ} as x_κ. Thus $|v_\kappa - x_\kappa| \leq \frac{c \log n}{n}$. Furthermore, since x_κ shares its more significant κ bits with the destination, w, we have

$$|x_\kappa - w| \leq (1/2)^\kappa = (1/2)^{\log n + c'} = \frac{2^{-c'}}{n}.$$

Thus after all bits have been fixed, with high probability, the packet will be at a distance from its destination which is $O(\frac{c \log n}{n})$. Hence by Lemma 3 the total number hops between the current location and the destination is $O(\log n)$ w.h.p. □

5 Self-Stabilization

We use the standard synchronous message passing model already used by Onus et al. [17]: the time steps are synchronized and all messages sent out at time step t arrive at their destinations before the beginning of time step $t + 1$. That is, no messages are ever in transit at the beginning of a new time step so that we do not have to worry about old messages that are still in the system. In addition to that we assume that there are no fake or outdated node identifiers in the system at any time. If so, we would have to worry about failure detectors which we do not want to do here to keep the treatment simple. We also assume that during the self-stabilization process the node set does not change. While joining nodes would be of no danger and would only delay the time needed till the self-stabilization process finishes, leaving nodes would cause outdated node identifiers, which we are not considering here.

Given the assumptions above, we show that the LDB network can self-stabilize from any initial state in which the nodes form a weakly connected graph. In the absense of any outside means that would allow disconnected nodes to reconnect, the assumption that the graph be initially weakly connected is necessary for the self-stabilization mechanism to ensure that at the end all nodes in the system form the desired topology. In the following, whenever we use the word "connected", it means "weakly connected".

Our self-stabilization mechanism builds on and extends the linearization technique of Onus et al. [17] as well as Kniesburges et al. [13]: in each time step, each node v with left neighbors $u_\ell < u_{\ell-1} < ... < u_1$ and right neighbors $w_1 < w_2 < ... < w_r$ replaces every edge (v, u_i) with $i > 1$ with (u_{i-1}, u_i) and every edge (v, w_i) with $i > 1$ with (w_{i-1}, w_i) by contacting the corresponding neighbors (i.e., it *linearizes* its neighborhood which explains the name of the technique as well as the name LDB we gave to our network). Also, it asks u_1 to establish (u_1, v) and w_1 to establish (w_1, v). Due to our message passing model, all notifications sent out by v in order to establish these edges can be received and processed by its neighbors so that their neighborhoods include the new edges at the beginning of the next time step. As shown in [13] (Section 3.1.2), for any weakly connected graph G of size n, at most $O(n)$ time steps are needed by the linearization rule above to transform G into a bi-directed sorted list. However, we have to deal here with the problem that we cannot let the virtual edges participate in the linearization of v, $l(v)$ and $r(v)$ as otherwise we would never reach a stable network (because v would continually introduce $l(v)$ and $r(v)$ to its closest neighbors). Thus, the linearization should only be applied to the non-virtual edges of v, $l(v)$ and $r(v)$ for all nodes v.

Suppose now that we start with some arbitrary directed network $G = (V, E)$ that is weakly connected, where V includes the real as well as the virtual nodes. Let $E' \subseteq E$ be the set of all non-virtual edges and let $G' = (V, E')$. Since the linearization does not include the virtual edges, the possibility remains that G' is not connected even though G is connected. To stabilize from this state to the desired LDB topology, we introduce a light-weight probing algorithm for each node v to determine if there is a path along the non-virtual edges leading from v to v's virtual nodes. We will show that by performing both the probing and the linearization algorithms, a network in any weakly connected state (over both virtual and non-virtual edges) will converge to the LDB in $O(n)$ steps.

5.1 Linear Probing

Let $x \in V$ be a real node. At each time step, x will probe for its left and right virtual node. The probing for the left virtual node is given in Algorithm 2, and the probing for the right virtual node works in an analogous way. The following property immediately follows from Algorithm 2.

Lemma 4. *For every real node v and every graph $G' = (V, E')$ formed by the non-virtual edges, the probing of v for $l(v)$ or $r(v)$ terminates in at most $3n$ steps.*

Algorithm 2. Probing in the LDB

```
1: y = pred(x)
2: while y is a virtual node do
3:    if y = l(x) then
4:       Exit
5:    else if pred(y) does not exist then
6:       Establish a non-virtual edge between x and l(x)
7:       Exit
8:    else
9:       y = pred(y)
10:   end if
11: end while
12: y = l(y)
13: while y < l(x) do
14:   if succ(y) does not exist then
15:      Establish a non-virtual edge between x and l(x)
16:      Exit
17:   end if
18: end while
19: if y ≠ l(x) then
20:   Establish a non-virtual edge between x and l(x)
21: end if
```

In the stable state, i.e., the ideal LDB network has been established, the probing is also very light-weight as stated in the next lemma, which follows from the fact that the real and virtual nodes are distributed uniformly at random in $(0, 1)$.

Lemma 5. *In the ideal LDB network with labels chosen uniformly at random for the real nodes it holds that for every real node v, the expected length of the path travelled by its probe to $l(v)$ and $r(v)$ is a constant.*

Hence, in the stable state, the linearization rule together with the probing rule only involves an expected constant number of steps to check the correctness of the LDB network. Thus, any faults can be detected quickly.

5.2 Convergence of Linearization with Linear Probing

Finally, we prove that linearization with linear probing quickly converges to the desired LDB topology.

Theorem 2. *Using linearization together with linear probing, any weakly connected network (over virtual and non-virtual edges) will converge to the LDB network within $O(n)$ time steps.*

Proof. Let $G = (V, E)$ be the graph containing all virtual and non-virtual edges and $G' = (V, E')$ the the graph containing only the non-virtual edges. We need a sequence of lemmas to prove the theorem. The first lemma shows that weak connectivity is preserved for any pair of nodes.

Lemma 6. *Consider any connected component C in G'. If C is connected at the beginning of step t, then C is also connected at the beginning of step $t+1$.*

Proof. As we do not have any fake or outdated node identifiers, the linearization rule will only perform transformations that preserve connectivity for the non-virtual edges (namely, the neighborhood of each node is transformed into a sorted list). As the probing mechanism may only create additional non-virtual edges, the connectivity property of C will be preserved. □

Lemma 7. *If G is connected but G' is not connected, then there must be a real node v that is not connected to $l(v)$ or $r(v)$ in G'.*

Proof. Suppose that G is connected and every real node $v \in V$ is connected to $l(v)$ and $r(v)$ in G' but G' is not connected. Let $C_1, \ldots, C_k$ be the connected components in G', $k \geq 2$. As these are connected in G, there must exist a real node v in some C_i with $l(v)$ or $r(v)$ being in C_j for some $j \neq i$. However, that contradicts our assumption that all real nodes are connected to their left and right virtual nodes in G' which completes the proof. □

Now we are ready to prove an upper bound on the number of steps it takes until G' is weakly connected.

Lemma 8. *If G is connected, then it takes at most $O(n)$ steps until G' is connected.*

Proof. Suppose that G' is initially not connected. Then it follows from Lemma 7 that there must be a real node v that is not connected to $l(v)$ or $r(v)$ in G'. W.l.o.g. we assume that v is not connected to $l(v)$. Then we follow the left probing of v till it reaches a real node or ends before reaching one. In the latter case v establishes a non-virtual edge to $l(v)$, so v is in the same connected component of G' as $l(v)$. Otherwise, let v' be the real node reached by the probing of v. Suppose that v's probe is not able to reach $l(v)$ from $l(v')$. Then v establishes a non-virtual edge to $l(v)$ and also in this case v and $l(v)$ are in the same connected component in G'. Hence, it remains to consider the case that v's probe succeeds in reaching $l(v)$ from $l(v')$. Then it follows from Lemma 4 that after $O(n)$ steps $l(v)$ and $l(v')$ are in the same connected component in G'. As Lemma 6 guarantees that v and v' remain in the same connected component of G', it follows that if v' and $l(v')$ are in the same connected component of G' after $O(n)$ steps, so are v and $l(v)$. Therefore, instead of focussing on the probe initially sent out by v, we focus on the probe initially sent out by v'. Continuing with the same arguments for v' as for v, it either holds that v' is connected to $l(v')$ after $O(n)$ steps or there is a node v'' with the property that v'' is in the same connected component as v' and $l(v'')$ is in the same connected component as $l(v')$ after $O(n)$ steps. In the latter case we switch to v'' and consider its initial probe for $l(v'')$. Continuing with these arguments, we must end up with a real node w that is in the same connected component as v after $O(n)$ steps, and w is connected to $l(w)$ in G' after $O(n)$ steps, which is in the same connected component as $l(v)$ after $O(n)$ steps. Hence, after $O(n)$ steps v is connected to

$l(w)$ in G'. Since this argument applies to all nodes v that are initially not connected with $l(v)$ in G', the lemma follows. □

Finally, we need the following lemma, which has already been shown in [13].

Lemma 9. *If G' is connected, then the linearization rule ensures that after $O(n)$ steps G' forms a bi-directed sorted list.*

As it is known that the bi-directed sorted list is the only stable structure of the linearization rule [17] and in that case the linear probing will not add any further edges, the theorem follows. □

6 Join and Leave

We rely on the self-stabilization rules above to ensure that we can have very simple join and leave operations and still maintain the network structure. When a node v joins a network via a node w, it simply establishes a non-virtual edge between w and each of v, $l(v)$, and $r(v)$. Then our self-stabilization mechanism will ensure that the nodes will be placed in their proper location in $O(n)$ rounds. Other join operations are possible, such as using the routing algorithm to place the nodes more quickly within the network, but they are not discussed here. Since the self-stabilization rules can repair the network from any weakly connected state, our simple join mechanism can handle arbitrary concurrent join operations, which can be quite tricky to handle with a dedicated join operation. Also, notice that the routing mechanism will still work correctly while nodes are joining for two reasons: (1) the routing just relies on the virtual edges and the edges specified by *pred* and *succ* to proceed,and (2) joining nodes will only affect the *pred* and *succ* values of the nodes already in the system when they reached their right place.

When a node v leaves the network, v must simply introduce $pred(v)$ and $succ(v)$ to one another and the same for $l(v)$ and $r(v)$. Following this, the network will immediately be in a correct state with no additional work.

Finally, we show that if a node leaves without properly introducing its neighbors, the network will remain weakly connected with very high probability. So we know that the self-stabilization algorithms will still be able to return the network to a proper state.

Theorem 3. *Let G be an LDB network and let v be a real node in G. If there exists a real node $w \in G$ such that $l(v) < w < r(v)$, then the graph $G - \{v, l(v), r(v)\}$ is weakly connected.*

Proof. Partition $(0, 1)$ into four regions $R_1 = (0, l(v))$, $R_2 = (l(v), v)$, $R_3 = (v, r(v))$, $R_4 = (r(v), 1)$. Assume $w < v$. The case where $w > v$ is symmetric. First note that from the linearization edges all nodes in a single region R_i are connected. Also note that any region with at least one virtual node but no real nodes is connected to another region, since each virtual node must have an incoming edge from a real node. Since $w < v$, $\frac{w}{2} < \frac{v}{2}$. Thus $l(w) \in R_1$, and so

R_2 is connected to R_1. If there is a real node u in $R_3 \cup R_4$, then consider two cases:

Case 1: $v \leq \frac{1}{2}$. Then $v < \frac{w+1}{2} = r(w) < r(v)$, so R_3 is connected to R_1 and R_2. If $u \in R_3$ then $r(u) > r(v)$ so R_3 is connected to R_4 and so the whole graph is connected. If $u \in R_4$ then $l(u) < r(v)$ so R_4 is connected to $R_1 \cup R_2 \cup R_3$.

Case 2: $v \geq \frac{1}{2}$. Then $\frac{u}{2} < v$. If $u \in R_3$ then this implies R_3 is connected to $R_1 \cup R_2$. Also, since $u > v$, $r(u) > r(v)$ so R_3 is connected to R_4 and thus the entire graph is connected. If $u \in R_4$ then R_4 is connected to $R_1 \cup R_2$ and if there is a real node $y \in R_3$ then $r(y) \in R_4$ so the entire graph is weakly connected. □

Corollary 2. *If a node v leaves in the LDB with no further action, the network remains weakly connected with probability $1 - \frac{1}{2^{n-1}}$.*

Proof. The interval between $l(v)$ and $r(v)$ has length $\frac{1}{2}$. By Proposition 3 the network will be weakly connected if there are no real nodes in $(l(v), r(v))$, which has probability $\frac{1}{2^{n-1}}$. □

7 Conclusion

In this paper we presented the first self-stabilizing dynamic overlay network construction based on a De Bruijn graph, which also retains the main attractive properties of the classical De Bruijn construction, namely, logarithmic time routing and constant node degree. As future work, we believe that our approach, which uses an underlying sorted list of the nodes in order to aid in the network routing and self-stabilization, can also be generalized for other popular topologies, in particular ones that follow the continuous-discrete approach by Naor and Wieder.

References

1. Abraham, I., Awerbuch, B., Azar, Y., Bartal, Y., Malkhi, D., Pavlov, E.: A generic scheme for building overlay networks in adversarial scenarios. In: Proc. of the 17th Intl. Parallel and Distributed Processing Symposium (IPDPS), p. 40 (2003)
2. Berns, A., Ghosh, S., Pemmaraju, S.V.: Brief announcement: a framework for building self-stabilizing overlay networks. In: Proc. of the 29th ACM Symposium on Principles of Distributed Computing (PODC), pp. 398–399 (2010)
3. Caron, E., Desprez, F., Petit, F., Tedeschi, C.: Snap-stabilizing prefix tree for peer-to-peer systems. Parallel Processing Letters 20(1), 15–30 (2010)
4. Clouser, T., Nesterenko, M., Scheideler, C.: Tiara: A self-stabilizing deterministic skip list. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 124–140. Springer, Heidelberg (2008)
5. Cramer, C., Fuhrmann, T.: Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe (2005)
6. De Bruijn, N.: A combinatorial problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)

7. Fraigniaud, P., Gauron, P.: D2B: A De Bruijn based content-addressable network. Theoretical Computer Science 355(1), 65–79 (2006)
8. Gall, D., Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: Time complexity of distributed topological self-stabilization: The case of graph linearization. In: Proc. of the 9th Latin American Theoretical Informatics Symposium, pp. 294–305 (2010)
9. Jacob, R., Richa, A., Scheideler, C., Schmid, S., Taeubig, H.: A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: Proc. of the 28th ACM Symposium on Principles of Distributed Computing (PODC), pp. 131–140 (2009)
10. Jacob, R., Ritscher, S., Scheideler, C., Schmid, S.: A Self-stabilizing and Local Delaunay Graph Construction. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 771–780. Springer, Heidelberg (2009)
11. Kaashoek, M., Karger, D.: Koorde: A Simple Degree-Optimal Distributed Hash Table. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, Springer, Heidelberg (2003)
12. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: Proc. of the 29th ACM Symposium on Theory of Computing, STOC (1997)
13. Kniesburges, S., Koutsopoulos, A., Scheideler, C.: Re-Chord: A self-stabilizing Chord overlay network. To appear in Proc. of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA (2011)
14. Loguinov, D., Kumar, A., Rai, V., Ganesh, S.: Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience. In: Proc. of the 2003 ACM SIGCOMM Conference, pp. 395–406 (2003)
15. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A scalable and dynamic emulation of the butterfly. In: Proc. of the 21st ACM Symposium on Principles of Distributed Computing, PODC (2002)
16. Naor, M., Wieder, U.: Novel architectures for P2P applications: the continuous-discrete approch. In: Proc. of the 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 50–59 (2003)
17. Onus, M., Richa, A., Scheideler, C.: Linearization: Locally Self-Stabilizing Sorting in Graphs. In: Proc. of the 9th Workshop on Algorithm Engineering and Experiments, ALENEX (2007)
18. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proc. of the ACM SIGCOMM Data Communication Festival (2001)
19. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Liu, H. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
20. Scheideler, C., Schmid, S.: A distributed and oblivious heap. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 571–582. Springer, Heidelberg (2009)
21. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer look-up protocol for internet applications. IEEE/ACM Transactions on Networking 11(1), 17–32 (2003)

Brief Announcement: A Conjecture on Traceability, and a New Class of Traceable Networks

H.B. Acharya[1], Anil K. Katti[1], and Mohamed G. Gouda[1,2]

[1] The University of Texas at Austin, USA
acharya @cs.utexas.edu,
[2] The National Science Foundation, USA

Abstract. The problem of reconstructing the topology of a network, given a set of hop-by-hop traces of packet paths through it, is called 'network tracing'. Unfortunately, there are only a few known classes of networks (trees and odd rings) that are traceable, i.e. that have the property that a unique network topology can be reconstructed from a trace set. Here, we suggest a property that may be the reason such networks are traceable, and use it to identify a new class of traceable networks.

1 Introduction

Many applications improve performance by using network locality; hence, knowledge of the network topology is important, but not readily available. There have been many attempts to map networks, such as Rocketfuel [2]; these mappers use Traceroute to obtain hop-by-hop paths between known start and end nodes, called *terminals*, then reconstruct a candidate topology that contains the known paths.

Unfortunately, this approach leads to wildly inaccurate maps. In a trace, every node other than the (known) terminals may refuse to reveal its IP address, so nodes are anonymous; some nodes have many IP addresses (eg. for multi-homing) hence there is aliasing; and many nodes and edges do not appear in the trace set at all.

In our previous work [1], we demonstrated that even if we can ensure complete coverage, knowledge of aliases, and consistent routing, a single anonymous or irregular (sometimes-anonymous) node can make it impossible to uniquely reconstruct a general network from a trace set. Networks are traceable (i.e. can be uniquely reconstructed from their trace sets) despite the presence of irregular nodes if the topology of the network is known to be a tree or an odd ring.

In our present work, we make a conjecture as to the property of trees and odd rings that makes them traceable (possible to uniquely identify from a trace set), and report on our progress toward proving the correctness of our conjecture. Using this conjecture, we identify a new class of traceable network.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 431–432, 2011.

2 The Unique Path Length Conjecture

We begin by noting that a trace set only reveals the distance between nodes along one path (the path of the trace). But this is disproportionately important for a tree network, where there is exactly one path between any two nodes; the algorithm to trace a tree network identifies the nodes using their distances from two terminals, and if there were many paths between the same two nodes, there would be multiple nodes with the same distances to both terminals.

What of odd rings? In any ring, there are two paths between any two nodes a and c; if we take rings with a third node b, there is a path from a to c passing through b, and a path that does not pass through b. In answer, we note that in addition to which terminals it runs between, a trace has another defining characteristic: its length. In an odd ring, there is exactly one path of a given length between any two nodes. (Let $|abc|$ denote the length of the arc connecting a to c through b, and $|ab'c|$ the length of the other arc from a to c. In a ring where $|abc| = |ab'c|$, the total length of a circuit around the ring, i.e. $|abc| + |ab'c|$, is even - and even rings are known to not be traceable.)

In fact, the algorithm to trace odd rings does make use of this characteristic. In the algorithm, we begin by fixing a terminal node a and marking the nodes of one trace $(a...b)$ randomly around the ring (clockwise or counterclockwise). Next, we place all remaining traces with terminal node a on the ring. For such a trace $(a...c)$, we choose the opposite direction as $(a...b)$ if $|ac|+|ab|+|bc| = ring.length$, and the same direction as $(a...b)$ otherwise. At the end, all the terminal nodes are placed, and we can fill in any missing labels using the fact that there is only one path of a given length between terminals (so if, say, x is 2 hops from terminal y along the arc $(y...z)$ of length 5, this uniquely specifies its position on the ring).

Based on the above discussion, we suggest the following conjecture.

Conjecture 1. *A network is traceable if, given any two nodes in N and a length l, there is no more than one path of length l connecting these nodes in the network.*

We are currently investigating the truth of this conjecture. We have found the encouraging partial result that another class of network satisfying the given condition, unicyclic networks in which the length of the cycle is odd, are also traceable (provided that all nodes shared by a tree and the cycle are terminal, and the terminal nodes are regular).

References

1. Acharya, H.B., Gouda, M.G.: On the hardness of topology inference. In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 251–262. Springer, Heidelberg (2011)
2. Spring, N., Mahajan, R., Wetherall, D., Anderson, T.: Measuring isp topologies with rocketfuel. IEEE/ACM Trans. Netw. 12(1), 2–16 (2004)

Brief Announcement: A Stabilizing Algorithm for Finding Two Edge-Disjoint Paths in Arbitrary Graphs

Fawaz M. Al-Azemi and Mehmet Hakan Karaata

Kuwait University, Department of Computer Engineering,
P.O. Box 5969, Safat 13060, Kuwait
fawaz.mohammed@gmail.com
karaata@eng.kuniv.edu.kw

Keywords: Disjoint paths, fault-tolerance, stabilization.

Introduction. Given two distinct nodes s and t of a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of arcs, the problem of identifying *two edge-disjoint paths* from s to t is to identify two distinct paths Q_1 and Q_2 from s to t such that Q_1 and Q_2 share no common arc.

As presented in [2], identifying edge-disjoint paths has a wide range of applications in various areas including VLSI layout, reliable network routing, secure message transmission, and network survivability. For instance, edge-disjoint paths can be used for secure transmission as follows. The simple expedient of breaking up data into several shares and sending them along the disjoint paths makes it difficult for an adversary with bounded eavesdropping capability to intercept a transmission or tamper with it. Alternatively, the same crucial message can be sent over multiple edge-disjoint paths in a network that is prone to message losses to avoid omission failures, or information on the re-routing of traffic along non-faulty disjoint paths can be provided in the presence of faults in some disjoint paths. Other applications of disjoint paths include network coding to provide $1 + N$ protection against single link failures in optical hypercube networks, where N is the dimension of the network. Moreover, edge-disjoint paths between two processes, present additional benefits, such as allowing a process to establish communication with a process by distributing the communication load in the network on two edge-disjoint paths without congesting communication channels.

The concept of *stabilization* was introduced by Dijkstra in [1]. A system is referred to as a stabilizing system if it eventually enters a legitimate configuration regardless of the current configuration in a bounded number of steps and the system state remains legitimate thereafter. In addition, stabilizing system are able to withstand *transient faults*. We view a fault that perturbs the state of the system but not its program as the transient fault. Furthermore, many stabilizing systems are adaptive to topology changes in the form of addition/removal of processes and links.

Considerable effort has been devoted to the design of sequential and distributed algorithms for finding edge-disjoint paths. A simple sequential algorithm

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 433–434, 2011.

for identifying k disjoint paths based on network flow and shortest augmented path is presented in [5]. A more efficient implementation of [5] is given in [6]. [3] presented a distibuted algorithm to solve a closely related problem of identifying two disjoint paths based on the concept of kernel. Another distributed algorithm for disjoint paths between a pair of nodes has been proposed in [4] that reduced [5] approach into the problem of finding minimal shortest path instead of augmented path. Moreover, the first self-stabilizing distributed algorithm for finding disjoint paths in mesh networks is presented in [2].

In this paper, we present a stabilizing algorithm for finding two edge-disjoint paths problem for directed graphs based on Suurballe's algorithm, as presented in [5]. Two edge-disjoint paths are optimal with respect to the sum of their path lengths. The algorithm identifies two edge-disjoint paths Q_1 and Q_2 in three concurrent phases, namely, *first shortest-path phase, second shortest-path phase and edge-disjoint phase*. The progress in a phase is made only after the previous phase has terminated. In the first shortest-path phase, a shortest path P_1 from s to t in G is identified. In the second shortest-path phase, a shortest path P_2 from s to t in G_2 is identified, where G_2 is the same as G with P_1 reversed. Edge-disjoint phase combine paths P_1 and P_2, after the removal of all *oppositely directed* pairs of arcs, to form G_3 containing only two edge-disjoint paths Q_1 and Q_2 that are also paths in G. A pair of arcs, say $(a, b) \in P_1$ and $(c, d) \in P_2$, are *oppositely directed* if and only if $b = c$ and $a = d$.

The algorithm finds Q_1 and Q_2 in $O(D)$ *rounds*, where D is the diameter of the graph G. A *round* is an execution sequence in which the slowest process executes one step during a defined period of time. The state space complexity of the algorithm is $O(log\ n)$ bits per process. Since the algorithm is stabilizing, it withstand transient faults. In addition, the proposed algorithm is capable of dealing with topology change in the form of addition/removal of arcs and nodes as well as changes in the directions of arcs. Although the proposed algorithm works only on directed graphs, its undirected version can be obtained by adding two arcs for each undirected edge in opposite directions. The implementation of such a variation of the algorithm is easy under a realistic model.

References

[1] Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
[2] Hadid, R., Karaata, M.H.: An adaptive stabilizing algorithm for finding all disjoint paths in anonymous mesh networks. Computer Communications 32(5), 858–866 (2009)
[3] Ishida, K., Kakuda, Y., Kikuno, T., Amano, K.: A distributed routing protocol for finding two node-disjoint paths in computer networks
[4] Ogier, R.G., Rutenburg, V., Shacham, N.: Distributed algorithms for computing shortest pairs of disjoint paths. IEEE Transactions on Information Theory 39(2), 443–455 (1993)
[5] Suurballe, J.W.: Disjoint paths in a network. Networks 4, 125–145 (1974)
[6] Suurballe, J.W., Tarjan, R.E.: A quick method for finding shortest pairs of disjoint paths. Networks 14, 325–379 (1984)

Brief Announcement: Towards Interoperability Standards and Services for Autonomic Systems

Richard Anthony, Mariusz Pelc, and Haffiz Suahib

University of Greenwich,
Park Row, SE10 9LS, London, United Kingdom
{r.j.anthony,m.pelc,h.suahib}@gre.ac.uk

Abstract. This paper advocates a generic, standardised approach to the problem of interoperability and proposes introduction of a centralised Interoperability Service (IS) with which Autonomic Managers (AM) register their management interests and capabilities, using a standardised management description language. A fuzzy mapping technique is used to identify potential conflicts of management interest in a conflict-risk model. The main contribution of this work is that the interoperability support is integrated into autonomic components making them *interoperability ready* in advance of their deployment.

Keywords: interoperability, autonomic systems, systems stability.

1 Introduction

The popularity of Autonomic Computing is driving expansion into diverse application domains and increasing the variety and number of functionalities that can be automatically managed within a given system. Thus, for many current and near-future AMs, it is not safe to assume isolated management operation; it will be increasingly common for multiple AM to coexist in any moderately sized computer system. Unplanned coexistence, or unexpected interactions could arise for many reasons, including the multivendor nature of many large systems. These interactions can take many forms, but fall into two classes: direct conflicts occur where two AMs attempt to manage the same explicit resource and indirect conflicts arising when AMs control different resources, but the management effects of one have an undesirable impact on the management function of the other. AMs are supposed to work with no or limited human intervention, and it is generally not possible to predict all run-time scenarios, including the presence of other AMs, in advance at design time. Therefore we propose a universal Interoperability Service (IS) based solution which automatically detects and resolves conflicts between independently-developed IS-compliant AMs leading ultimately to a safe co-existence or even co-operation of various AMs.

2 Interoperability Service Interfaces

We propose an Interoperability Service (IS) which is responsible for detecting possible conflicts of management interest, and granting or withholding management

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 435–436, 2011.

rights to specific AMs as appropriate. A number of communication interfaces are specified, and form three groups:

1. IS-AM interaction is supported by two interfaces.
   IAdvertise {*Advertise, Unregister, Heartbeat*} is used by AMs to signal joining, leaving and heartbeat messages to the IS. **IInteroperate** {*Run, Stop, Suspend, Resume, Throttle*} is used to receive directives from the IS. The AM developer uses the IS API to map these directives onto the AM-internal behaviour.
2. IS-IS interaction is facilitated by a single interface.
   ICommunicate {*Forward, Locate, Elect, SetISLevel, GetISLevel*} supports hierarchical operation, necessary in large or complex systems when AMs operate at different levels within a system and may be involved in local or system-wide conflicts.
3. The IS provides an external management interface.
   IConfigure {*SetMode, GetMode, SetSensitivity, GetSensitivity, StatusReport*} is a configuration and reporting interface which allows external system management utilities to perform system-specific configuration and generate status reports and statistics.

3 A Management Description Language

The foremost role of the IS is to facilitate interoperability amongst (unknown in advance) AMs which have been developed independently of each other, and thus do not directly support interoperability amongst themselves. The standard management description should include **Category** (mandatory) to identify the AM's domain of interest, **Zone** (mandatory) to differentiate between specific management functions, **Impact Factor** (mandatory) to express the strength of the management influence (where $0 < IF \leq 1$), **Scope** (mandatory) defining local or global impact, **Specificity** (optional) to express the extent of manager operation, **Trigger** (optional) expressing of temporal aspects such as periodicity or operating timescale, **Parameter** (optional) identifying specific context parameters, **Envelope** (optional) expressing the number of dimensions of control freedom.

4 Conflict Detection

Conflict detection is based on comparing a newly registering AM's management description with those of the already registered AMs to determine a similarity measure. A dynamically configurable weighted conflict threshold ($0 < Thresh_C \leq 1$) is used to tune the conflict detection sensitivity (via *SetSensitivity*, on *IConfigure*). A potential conflict is detected if the similarity measure of a pair of vectors exceeds $Thresh_C$. The sensitivity level can be configured at run time as necessary. This enables some systems to operate with very low tolerance to potential manager conflicts, whereas in other domains a higher tolerance can lead to benefits of having a greater number of AMs working simultaneously.

Brief Announcement: Distributed Self-organizing Event Space Partitioning for Content-Based Publish/Subscribe Systems*

Roberto Beraldi, Adriano Cerocchi, Fabio Papale, and Leonardo Querzoni

University of Rome "La Sapienza", Rome, Italy
{beraldi,cerocchi,querzoni}@dis.uniroma1.it

Context and Motivations. Publish/subscribe systems have commonly been divided in two large families on the basis of their event-selection model [2]: topic-based and content-based systems. The former trade reduced subscription expressiveness with simpler implementations and higher performance. Conversely, the latter allow to accurately map published data in a complex event schema on top of which expressive subscriptions can be defined, but incur the cost of more complex implementations that delivers reduced performance on large distributed settings. System developers are thus faced with a choice about which kind of system is best suited to the target application. A common solution to this dilemma lies in the *event space partitioning* [4] technique: the event schema is partitioned in a number of subspaces that are then statically mapped to topics. The partitioning must be globally known and subscribers are expected to subscribe those topics where subspaces that have a non-empty intersection with their content-based subscriptions have been mapped. Undesired events (false positives) can be filtered out at the receiver side. The event space partitioning granularity strongly affects the performance of such systems: if it is excessively coarse-grained too much resources are wasted to deliver false positives, while if it is too fine-grained the number of topics that will be generated, and that must be managed by the topic based system, could easily become huge. Current solutions [3] provide sub-optimal approximations that are calculated offline and then statically applied to the system.

Contribution. We propose a self-organizing algorithm that builds and dynamically adapts at run-time an event space partitioning that eventually provides subscribers with a desired level of performance (i.e. percentage of false positives below a specified threshold) while striving to limit the number of topics that must be managed. The novelty of our solution lies in its ability to work on the basis of run-time performance indices measured by subscribers. Each subscriber monitors the ratio between the false positives (an event notified to the subscriber that does not match any of its content-based subscriptions) and the total number of events received for each topic it is subscribed to. If this ratio raises above a predefined global threshold T_{FP} the subscriber starts a procedure for partitioning the subspace mapped to that topic. At the end of this procedure it updates its topic subscriptions and starts again monitoring performance statistics. Figure 1

* This work was partially supported by the BLEND and SOFIA European projects and by the DOTS-LCCI Italian project.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 437–438, 2011.

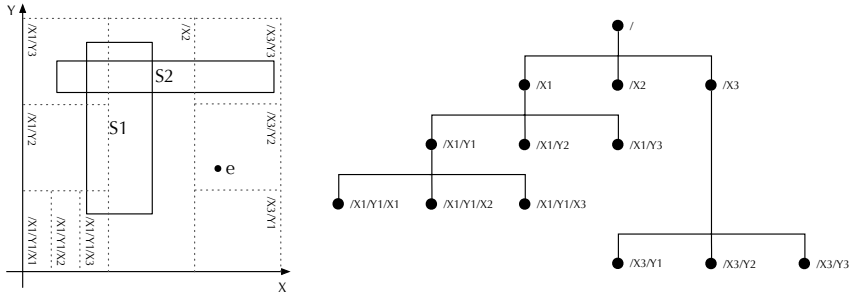


Fig. 1. A partitioned event space with the corresponding partitioning tree

shows an example based on a two dimensions event space with attributes X and Y; the light dotted lines represent a possible partitioning on the event space where two subscriptions $S1$ and $S2$ has been defined; the right side of the figure offers a representation of the corresponding *partitioning tree*, the data structure hosted in a distributed fashion at the subscriber side to keep track of the current partitioning. Each node in the tree corresponds to a partition (or subspace). The root node corresponds to an initial single partition that matches the whole event space. Each partition can be subdivided in multiple sub-partitions that are represented in the tree as children nodes. Two topics are defined for each node in the tree: a *data topic* and a *control topic*. The former is used to diffuse events while the latter is used to diffuse information about the partitioning. The proposed algorithm can be embedded within an architectural component that provides a content-based publish/subscribe interface and leverages services offered by a plain topic-based system. Simulation-based experiments show that (i) the proposed algorithm converges to a stable event space partitioning in a limited amount of time, (ii) it adapts the partitioning with bounded oscillations even in presence of abrupt changes in the workload, (iii) the obtained partitioning provides the desired level of performance and (iv) the associated cost (average number of subscribed topics per subscriber) is lower than the cost required by a static partitioning as long as the desired performance threshold is kept low (i.e. performance similar to a pure content-based system with no false positives). A detailed description of the algorithm and an extensive experimental evaluation are available in [1].

References

1. Beraldi, R., Cerocchi, A., Papale, F., Querzoni, L.: Distributed self-organizing event space partitioning for content-based publish/subscribe systems. Tech. rep., Università di Roma "La Sapienza" (2011), http://www.dis.uniroma1.it/~midlab/publications.php
2. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
3. Opyrchal, L., Astley, M., Auerbach, J., Banavar, G., Strom, R., Sturman, D.: Exploiting IP multicast in content-based publish-subscribe systems. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 185–207. Springer, Heidelberg (2000)
4. Wang, Y., Qiu, L., Achlioptas, D., Das, G., Larson, P., Wang, H.J.: Subscription partitioning and routing in content-based publish/subscribe systems. Tech. rep., Microsoft Research (2002)

Brief Announcement: A Note on Replication of Documents*

Jacek Cichoń, Rafał Kapelko, and Karol Marchwicki

Institute of Mathematics and Computer Science,
Wrocław University of Technology, Poland
{jacek.cichon,rafal.kapelko,karol.marchwicki}@pwr.wroc.pl

Abstract. We prove one combinatorial result which we found useful in investigations of replication of documents in various storage systems like P2P systems or clouds.

1 Main Result

Let U be a fixed set of cardinality $N = n \cdot a$ and let $\{U_i\}_{i=1,\ldots,n}$ be a fixed partition of U into sets of cardinality a, i.e. $\bigcup_{i=1}^{n} U_i = U$, $U_i \cap U_j = \emptyset$ for $i \neq j$ and $|U_i| = a$ for each i. Suppose that we are successively and randomly removing distinct elements from the set U, let $\omega_1, \omega_2, \ldots$ be a realization of this process and let

$$K_{N,a} = \min\{k : (\exists i)(U_i \subseteq \{\omega_1, \ldots, \omega_k\})\} .$$

Theorem 1. *Let $a \geq 1$. Then*

$$\mathrm{E}\left(K_{N,a}\right) = (N+1)\frac{\Gamma\left(1+\frac{1}{a}\right)\Gamma\left(\frac{N}{a}+1\right)}{\Gamma\left(\frac{N}{a}+1+\frac{1}{a}\right)} \tag{1}$$

and

$$\lim_{N\to\infty} \frac{\mathrm{std}\left(K_{N,a}\right)}{\mathrm{E}\left(K_{N,a}\right)} = \sqrt{\frac{\Gamma\left(1+\frac{2}{a}\right)}{\Gamma^2\left(1+\frac{1}{a}\right)} - 1} . \tag{2}$$

In this Theorem Γ denotes the Euler Gamma function, i.e. the standard generalization of the factorial function. By $\mathrm{E}\left(X\right)$ we denoted the expected value of X and by $\mathrm{std}\left(X\right)$ we denoted the standard deviation of X.

Remark 1. We proved Theorem 1 using the inclusion-exclusion principle, some known binomial tautologies and the Rice method described in [1].

Corollary 1. $\mathrm{E}\left(K_{N,a}\right) = a^{\frac{1}{a}}\Gamma\left(1+\frac{1}{a}\right)N^{1-\frac{1}{a}} + \mathrm{O}\left(\frac{1}{N^{\frac{1}{a}}}\right)$

* Supported by grant nr 2010/342643 of the Institute of Mathematics and Computers Science of the Wrocław University of Technology.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 439–440, 2011.

Obviously $\mathrm{E}(K_{N,1}) = 1$. For $a = 2$ we can obtain from (1) the following closed formula $\mathrm{E}(K_{N,2}) = 2^N / \binom{N}{\frac{N}{2}}$ and the approximation formula $\mathrm{E}(K_{N,2}) = \sqrt{\frac{\pi}{2}}\sqrt{N} - \frac{1}{3} + \mathrm{O}\left(\frac{1}{\sqrt{N}}\right)$. Notice that, this formula is very close to the analogical formula for the the classical birthday paradox (see [1]). For $a = 3$ we get $\mathrm{E}(K_{N,3}) = \sqrt[3]{3}\Gamma(\frac{4}{3})N^{\frac{2}{3}} + \mathrm{O}\left(\frac{1}{\sqrt[3]{N}}\right) \approx 1.2879N^{\frac{2}{3}} + \mathrm{O}\left(\frac{1}{\sqrt[3]{N}}\right)$ and so on. Our formulas are similar to Klamkin and Newman's formulas from [2] for a generalization of the classical birthday paradox.

Let $\mathrm{cv}(a) = \lim_{N\to\infty} \mathrm{std}(K_{N,a}) / \mathrm{E}(K_{N,a})$. It is easy to check that the function $\mathrm{cv}(a)$ is decreasing. From (2) we can deduce the following result.

Corollary 2. $\mathrm{cv}(a) = \frac{\pi}{\sqrt{6}}\frac{1}{a} + \mathrm{O}\left(\frac{1}{a^2}\right)$

Therefore the random variables $K_{N,a}$ are becoming more and more concentrated with the increase of the parameter a.

2 Discussion

One of the techniques to increase the durability of documents stored in P2P systems is to store several copies of each document. The replication mechanism called *Global Policy* uses to store multiple copies of each document fixed family of independent hash function (see e.g. [3]). In the *Buddy Policy* each document is stored in some set of nodes of fixed cardinality (see [4] for details).

When we interpret the set U from the previous section as a collection of nodes in a P2P system then the random variable $K_{N,a}$ models the resistance of the system based on the Buddy Policy with blocks of size a on nodes failure. We see, for example, that when $a = 2$ then a simultaneous failure of $\sqrt{N}$ nodes is dangerous for the system with high probability. But when $a = 3$ then a failure of $\sqrt{N}$ nodes should not lead to loss of documents - problem occurs after a simultaneous failure of approximatively $N^{\frac{2}{3}}$ nodes.

Similar results for the Global Policy applied to the Chord P2P system where obtained and discussed in [5].

References

1. Knuth, D.E.: The art of computer programming, 3rd edn. Sorting and Searching. Addison-Wesley, Reading (1997)
2. Klamkin, M., Newman, D.: Extensions of the birthday surprise. J. Combin. Theory 3, 279–282 (1967)
3. Martins, V.: Data Replication in P2P Systems. PhD thesis, Universite De Nantes Faculte Des Sciences Et Des Techniques (2007)
4. Caron, S., Giroire, F., Mazauric, D., Monteiro, J., Pérennes, S.: Data life time for different placement policies in p2p storage systems. In: Hameurlain, A., Morvan, F., Tjoa, A.M. (eds.) Globe 2010. LNCS, vol. 6265, pp. 75–88. Springer, Heidelberg (2010)
5. Cichoń, J., Jasiński, A., Kapelko, R., Zawada, M.: How to Improve the Reliability of Chord? In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM-WS 2008. LNCS, vol. 5333, pp. 904–913. Springer, Heidelberg (2008)

Brief Announcement: A Stable and Robust Membership Protocol

Ajoy K. Datta[1], A.-M. Kermarrec[2], Lawrence L. Larmore[1], and E. Le Merrer[3]

[1] School of Computer Science, University of Nevada Las Vegas
[2] INRIA, Rennes, Bretagne Atlantique
[3] Technicolor

In this paper, we summarize the core ideas of our stable and robust membership protocol, which is fully decentralized. After convergence, each node of the overlay graph has expected in- and out-degrees scaling logarithmically with the size of the network (around $2\ln(n)$), and that the diameter of the overlay graph remains at $\frac{\ln(n)}{\ln(2\ln(n))} + O(1)$. Our protocol restores the desirable properties of the overlay network from an arbitrary state, which might result from a massive but temporary disruption.

Our membership protocol has the following properties, summarized by state of the art work [3]: (M1) small neighborhood sets, regarding system size (here logarithmic); (M2) load balance, as we provide in-degrees for nodes that match their out-degree (here logarithmic); (M3) uniform random neighborhood per node (we provide a random graph); (M4) spatial independence, meaning that neighbors of a node are independently chosen. Our work does not aim at providing aggressively (M5) temporal independence, as we update neighborhoods only when needed, instead of continuously doing it. Due to churns and optimizations, the resulting graph eventually converges towards this property.

One recurrent problem with state of the art work on membership management is that the out-degree of nodes must be given as a parameter of the system. If out-degrees are set at too high for the current network size, useless overhead is generated. On the other hand, too low of a value can cause the network diameter to increase dramatically. Maintaining out-degree that evolves with the size of the system usually requires continuous monitoring of the system, computing the size of the group, and adjusting the value of the out-degree parameter on all nodes as needed. Our protocol is designed to operate without such an input.

1 Overview of the Membership Protocol

The heart of our protocol is the Balancing protocol, which causes convergence of both diameter and degree by *churning* the edges. Edges are either *active* or *passive*; passive edges are those which have been marked for subsequent deletion.

Edges are continually removed or marked as *passive* and new edges are continually added. The Balancing protocol marks an edge (x, y) to be passive only if it determines that there is a *detour* of (x, y), namely a path from x to y which uses only active edges and does not use the edge (x, y). If it does not find such a detour, it increases the out-degree of x.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 441–442, 2011.

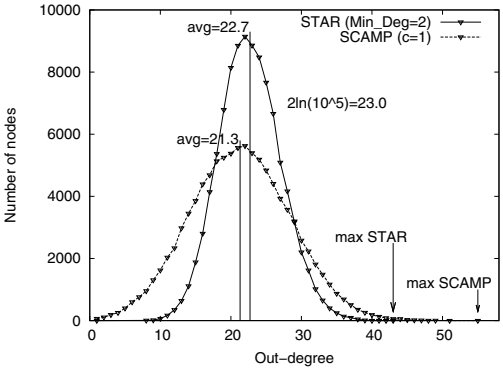


Fig. 1. Distribution of out-degrees compared to SCAMP, resulting from 10^5 joins

The Balancing protocol is designed to search extensively for a detour of (x, y) if the essential out-degree of x is large, and to severely limit that search if x has small out-degree. Thus, a dynamic equilibrium is achieved, where the essential out-degree of every node tends to oscillate around the average. Simultaneously, the local balancing protocol keeps the essential in-degree to within a constant (not a constant factor) of the essential out-degree.

The Balancing protocol is also designed so that, when equilibrium is achieved, the average out-degree of each node will be approximately $2\ln(n)$, and yet there is never a need for any node to compute the value of n. It also achieves the desired diameter merely by the fact that all edges, including "bad" edges, will eventually be removed, while the new edges that are added tend to be randomly placed, and hence "good." The diameter of the subgraph consisting of all nodes and just the active edges, thus converges to approximately the diameter of a random directed graph of the same cardinality and degree, which is $\frac{\ln(n)}{\ln(2\ln(n))} + O(1)$ [1].

Whenever a new node joins, our protocol causes the number of edges in the graph increase by $d + 2$, where d is approximately the average degree of each node. The parameter d is not known in advance, but corresponds to the degree of a node randomly chosen in the graph at join time. Figure 1 presents degree distribution of both our protocol and SCAMP [2]. SCAMP is another membership protocol that does not need system size as an input, but that lacks self-stabilization properties. Whenever a node leaves, our protocol does not decrease the number of edges immediately (it creates new edges in reaction). The Balancing mechanism then compensates, causing the graph to converge to an average degree of approximately $2\ln(n)$.

References

1. Fenner, T.I., Frieze, A.M.: On the connectivity of random m-orientable graphs and digraphs. Combinatorica 2, 347–359 (1982)
2. Ganesh, A.J., Kermarrec, A.-M., Massoulié, L.: Peer-to-peer membership management for gossip-based protocols. IEEE Trans. Computers 52(2), 139–149 (2003)
3. Gurevich, M., Keidar, I.: Correctness of gossip-based membership under message loss. SIAM J. Comput., 3830–3859 (2010)

Brief Announcement: Sorting on Skip Chains*

Ajoy K. Datta[1], Stéphane Devismes[2], and Lawrence L. Larmore[1]

[1] School of Computer Science, University of Nevada Las Vegas, USA
[2] VERIMAG UMR 5104, Université Joseph Fourier

1 Introduction

Sorting values on a chain of processes is a well-known problem, and a number of algorithms has been published [1,2]. We consider here a generalization of this problem, where the processes that have values, called *major processes*, are separated from each other by any number of intermediate processes, called *relay processes*, which do not have their own values, although they can read and write the major values while doing their job of relaying those values.

More precisely, we consider a chain network of n processes. Some of those processes, including the two end processes, are *major* processes, and the rest are *relay* processes. We call this structure a *skip chain*. The problem is then to sort the values held by the major processes. We call this problem the *skip chain sorting problem*.

We propose a silent self-stabilizing distributed algorithm for the skip chain sorting problem. Our algorithm is written in the locally shared memory model and works under an unfair daemon. Its stabilization time is $O(md)$ rounds, where m is the number of major processes and d is the maximum number of processes in the chain from one major process to the next. Note that $md = O(n)$ if the spacing between major processes is roughly equal.

2 Formal Statement of the Problem

We are given a chain of processes. Some of those processes, including the two end processes (which we call *L* and *R*) are *major* processes, and the rest are *relay* processes. We call this structure a *skip chain*. We assume that only major processes have values, and the problem is to sort those values. The specification of the skip chain sorting problem is given below.

1. In an arbitrary configuration of a skip chain, there is a *canonical value* $V(x)$ associated with each major process x. This value may or may not be stored at x.
2. At each step, the multiset of canonical values does not change, although the canonical values of two different major processes can be exchanged.
3. Every computation eventually results in a *legitimate configuration*, where the following conditions hold:
   (a) The canonical values of the major processes are in increasing order from left to right.
   (b) The canonical value of each major process x is stored at x.
   (c) No action is enabled.

* The full version of this paper is available at tinyurl.com/3dydywq. This work has been partially supported by the ANR project *ARESA*2.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 443–444, 2011.

We assume that n is the number of processes in the chain, m is the number of major processes, and d is the *relay chain length*, which is the maximum number of processes in the chain from one major process to the next. For example, if all processes are major processes, then $d = 2$, and $d = n$ if only the two end processes are major.

3 Overview of the Solution

We give an algorithm, *skip chain sort* (SCS) essentially a distributed version of the well-known algorithm *bubblesort*, which satisfies the requirements listed above.

SCS is self-stabilizing, which implies that it converges to a legitimate configuration regardless of the initial configuration. Given any skip chain $\mathcal{S}$, let $\mathcal{C}$ be the set of all configurations of SCS on $\mathcal{S}$. A certain subset $\mathcal{N} \subseteq \mathcal{C}$ consists of what we call *normal* configurations. These configurations are those where the states of all processes are correct, except that the canonical values may not be sorted. $\mathcal{N}$ is closed under the actions of SCS and is an attractor of $\mathcal{C}$.

The first phase of SCS, which we call *error correction*, results in a normal configuration. The second phase of SCS sorts the canonical values of the major processes, and eventually halts in a *legitimate* configuration, where each major process stores its own canonical value, and no process is enabled to execute.

Every major process x, except L, contains two embedded relay processes, which we call x.*l_relay* and x.*r_relay* (at the end, each major node stores its canonical value in its right relay); L contains only one embedded relay process, L.*r_relay*. We call the other relay processes *free relay processes*. If x is any process, then we define *Right_Major*(x) and *Left_Major*(x) to be the nearest major processes to the right and left of x (if any) respectively.

If x is a major process, we define the *right relay chain* of x to be the chain of relay processes starting with x.*r_relay* and ending with *Right_Major*(x).*l_relay*; the *left relay chain* of x is simply defined to be the right relay chain of *Left_Major*(x).

Two values can only be swapped by a major process if it holds both. If x is a major process and $y =$ *Right_Major*(x), then $V(x)$ and $V(y)$ can be compared, and possibly swapped, by y. The mechanism is to move $V(x)$ along the right relay chain of x to y.*l_relay*, while $V(y)$ is at y.*r_relay*. The values are then compared and possibly swapped. Afterward, the new value of $V(x)$ can move back to x, while the new value of $V(y)$ can move to *Right_Major*(y). After at most $\binom{n}{2}$ such comparisons, the canonical values will be sorted.

SCS uses *color waves* to control the movement of the values along the relay chains. A value moves to the left at the crest of a wave of color 0, and to the right at the crest of a wave of color 1. Two additional colors, 2 and 3, complete the color wave cycle to avoid ambiguity between waves. Additionally, there is an "error color," **E**.

When the canonical values are sorted, a *silence wave*, generated by the rightmost process, moves to the left, eventually causing all execution to cease.

References

1. Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F., Santoro, N.: Sorting and election in anonymous asynchronous rings. Journal of Parallel and Distributed Computing 64, 254–265 (2004)
2. Sasaki, A.: A time-optimal distributed sorting algorithm on a line network. Information Processing Letters 83, 21–26 (2002)

Brief Announcement: A Concurrent Partial Snapshot Algorithm for Large-Scale and Dynamic Distributed Systems

Yonghwan Kim[1], Tadashi Araragi[2], Junya Nakamura[1], and Toshimitsu Masuzawa[1]

[1] Graduate School of Information Science and Technology, Osaka University, Japan
{y-kim,junya-n,masuzawa}@ist.osaka-u.ac.jp
[2] NTT Communication Science Laboratories, Kyoto, Japan
araragi@cslab.kecl.ntt.co.jp

Contribution. We propose a concurrent partial snapshot algorithm (*CSS algorithm*) to extend a previously proposed sub-snapshot algorithm (SSS algorithm) [2] by introducing a method of merging multiple snapshots that are concurrently initiated by different nodes. In earlier work [5,6], efficient merging algorithms have already been introduced for CL algorithm. On the other hand, the main issue of our merging algorithm is to cope with dynamic situations based on SSS algorithm. Since the SSS algorithm is an extension of Chandy-Lamport snapshot algorithm (CL algorithm) [1], it allows large-scale and dynamic situations in snapshots. A dynamic situation means that nodes can join and leave freely during the execution of a snapshot algorithm. A snapshot algorithm for the dynamic situation has also been proposed [4]; however in this algorithm, nodes must stop sending application messages during its execution of the snapshot algorithm. Moreover, for concurrent snapshots, it has to cancel a portion of snapshot algorithms. Our algorithm has successfully removed these restrictions.

Basic Idea of *CSS Algorithm*. In SSS algorithm [2], each node maintains a set of node IDs, called a dependency set (DS). The DS includes the IDs of nodes that the node has communicated with, and it is reset at each checkpoint of the node. The DSs of the nodes involved in a snapshot are collected by the initiator and are used to dynamically decide the group by which to use in taking a partial consistent snapshot. In *CSS algorithm*, among multiple initiators, one initiator is elected as the main initiator, and it collects DSs from the other initiators and dynamically decides the group of the nodes whose checkpoints constitute a consistent merged partial snapshot.

The merging procedure is as follows. When a node executing *CSS algorithm* receives a snapshot request (marker) from another initiator directly or indirectly, which we call a collision, the node informs its current main initiator of the collision. If the initiator has not decided the group, the two snapshots are regarded as concurrent ones and merged into a single snapshot. One of the initiators is elected as the main initiator of the resultant snapshot. If collisions occur simultaneously, synchronization is necessary

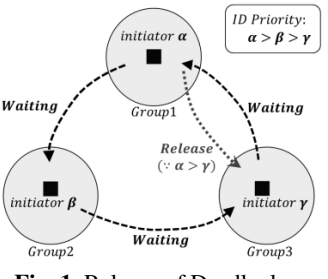


Fig. 1. Release of Deadlock

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 445–446, 2011.

for consistent merging. We also introduce a release mechanism for synchronization based on a predefined priority among the nodes to avoid deadlock (Fig. 1).

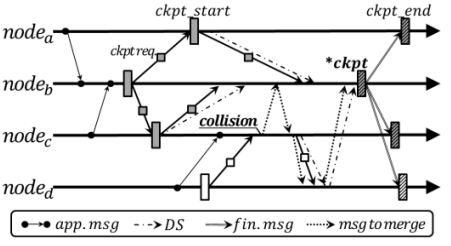


Fig. 2. Execution of CSS algorithm

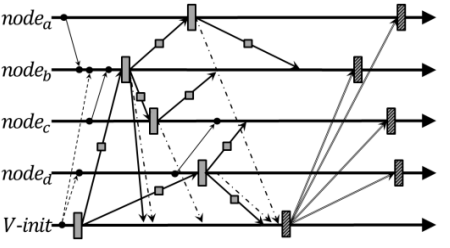


Fig. 3. CSS transformed into SSS

Execution of *CSS Algorithm*. Figure 2 illustrates the execution of *CSS algorithm*. $node_b$ and $node_d$ initiate snapshots. A collision occurs on $node_c$, and it informs $node_b$ of the collision. $node_b$, the current main initiator of $node_c$, has not decided the group, and these two executions of the snapshot algorithm are merged. By a predefined priority among nodes' IDs, $node_b$ becomes the main initiator of the merged snapshot. At **ckpt*, $node_b$ receives the DSs that $node_d$ has collected and thus can decide the group of the snapshot, because all received DSs are closed: $\{id \mid received\ DS(id)\} = \cup DS(id)$, where *DS(id)* is DS maintained by the node of id. Then, $node_b$ sends a message for terminating the algorithm to the members. Note that $node_d$ was not involved in communication related to $node_c$ when $node_c$ started the snapshot algorithm, and the group of snapshot is determined dynamically.

Correctness. The consistency of snapshots by *CSS algorithm* is proved by transforming an execution of *CSS algorithm* (Fig. 2) into the execution of SSS algorithm (Fig. 3), whose consistency has already been proved [3]. For an execution of CSS algorithm, we introduce a virtual initiator and dummy application messages so that the merged snapshots can be treated as a snapshot initiated by a single initiator.

References

1. Chandy, K., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Computer Systems 3(1), 63–75 (1985)
2. Moriya, S., Araragi, T.: Dynamic Snapshot Algorithm and Partial Rollback Algorithm for Internet Agents. In: DISC 2001 Brief Announcements, pp. 23–28 (2001)
3. Moriya, S., Araragi, T.: Dynamic Snapshot Algorithm and Partial Rollback Algorithm for Internet Agents. IEICE Transactions on Information and Systems J86-D-I, 301–317 (2003) (in Japanese)
4. Koo, R., Toueg, S.: Checkpointing and Roll-back Recovery for Distributed Systems. IEEE Transactions on Software Engineering 13(1), 23–31 (1987)
5. Spezialetti, M., Kearns, P.: Efficient Distributed Snapshots. In: Proceedings of the 6th International Conference on Distributed Computing Systems, pp. 382–388 (1986)
6. Prakash, R., Singhal, M.: Maximal Global Snapshot with Concurrent Initiators. In: Proceedings of the 6th IEEE Symposium of Parallel and Distributed Processing, pp. 334–351 (1994)

Brief Announcement: Fault-Tolerant Object Location in Large Compute Clusters

Björn Saballus, Stephan-Alexander Posselt, and Thomas Fuhrmann

Technische Universität München
Boltzmannstrasse 3, 85748 Garching/Munich, Germany
{saballus,posselt,fuhrmann}@in.tum.de

Abstract. A so-called *single system image* (SSI) allows threads in a distributed shared memory (DSM) system to access data and other resources in a location transparent manner. In this brief announcement, we present our ongoing research towards fault-tolerant algorithms that locate objects in such systems. In particular, we build our algorithms on a multi-version, object-based software transactional memory (STM) system, in which objects form a sequence of immutable object versions. Our algorithms are fully decentralized and allow resources to be added and removed at run-time without disturbing the application.

Today, computers grow in parallelism not in single core processing speed. Still, many programmers are not yet used to genuinely parallel programming. Many prefer concepts like distributed shared memory that keep up the traditional system model. However, location transparency and access latency optimization are conflicting goals.

In our work, we found STM to be a promising paradigm to hide latency and increase parallelism in large scale distributed systems. STM can be viewed as if data is processed in thread-local memory and only published when the transaction commits. A conflict resolution mechanism ensures the commit to be atomic, however, at the expense of potentially rolling back the transaction if its atomicity has been violated by concurrent transactions.

The details of the thread-local processing and the commit protocol depend on the STM system. We base our work on the DecentSTM algorithm [1], a multi-version, object-based STM. There, a transaction creates copies of all objects it modifies. These copies become the respective objects' new versions when the transaction commits. A fully decentralized consensus protocol resolves conflicting commits while the committing thread may proceed speculatively.

At first sight, creating copies might seem to create undue overhead. But current hardware creates copies in the caches anyway. Thus, if we assume that future processors support transactional memory and explicit cache control, our proposed copying does not create additional overhead.

As a result of DecentSTM's mode of operation, objects are in fact sequences of immutable object versions. Only the meta data, which is attached to the object versions, is modified as part of the consensus protocol and our proposed object location algorithm. As we have shown previously [2], outdated object versions

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 447–448, 2011.

can serve as checkpoints and thereby provide fault-tolerance, for example, in face of failing memory chips. For this idea to be viable, we need a fault-tolerant way to locate objects and retrieve a consistent – if not current – snapshot of their contents.

In our system, objects are virtual entities, which manifest in memory via their versions. Thus, upon access, our system needs to resolve an object reference to an object version. In general, there are two straight forward ways to resolve a reference: Most DSM systems are single-version systems; they use the storage location as reference, i.e. they use a trivial mapping, which is easy to implement, but fragile in face of hardware failures. Other DSM systems use a resolver that maintains a mapping between references and storage location. Such a mapping is more flexible; for example, it can easily handle object migrations; but it is not fault-tolerant unless the system explicitly creates and consistently maintains redundancy.

Our system exploits the fact that our STM consensus protocol already provides consistency. We use the storage location of an object version as a reference to the object. Thereby, we spare an explicit resolver service, and, more importantly, we spare the effort to keep the redundancy consistent. This mechanism works as follows:

When a transaction reads an object, it creates a thread-local copy. If it has modified the object, it must publish the modification as new version of that object upon commit. Thereby, the system creates *vertical* redundancy. Even if the transaction has not modified the object, it may nevertheless publish the data and thereby create *horizontal* redundancy.

The DecentSTM consensus protocol requires the node that wants to publish a new version to communicate with the node that stores the previous head version of the object. As part of this protocol, both object versions reference each other. Furthermore, as we already mentioned, we use the object version's storage location as an object reference. Thus, when a transaction reads an object, it can directly contact one of its versions. If that version has been overwritten since the reference was created, the request is forwarded to the node that stores the object's current head version. As a result, the requesting transaction obtains a copy of the referenced object and the storage location of a potential further version of the object. Conversely, if the requestor is willing to contribute to horizontal redundancy, the responding node keeps the requestor's address so that it can hand it out to transactions that request the object afterwards. These addresses accumulate so that – after a while – object references are sets of redundant pointers to different versions (or copies) of the same object.

References

1. Bieniusa, A., Fuhrmann, T.: Consistency in hindsight – a fully decentralized STM algorithm. In: Proc. 24th IEEE Intl. Parallel and Distributed Processing Symposium, IPDPS 2010 (2010)
2. Posselt, S.-A.: Design of a reliable, fully decentralized software transactional memory protocol. Diploma thesis, Technische Universität München (2010)

Brief Announcement: Faster Gossiping in Bidirectional Radio Networks with Large Labels

Shailesh Vaya

Department of Computer Science and Engineering
Indian Institute of Technology Patna
India - 800013

Abstract. We study the problem of deterministic gossiping in unknown ad-hoc bidirectional radio networks, when nodes can have polynomially large labels. We present a deterministic protocol which takes $O(n \lg^2 n \lg \lg n)$ rounds, improving upon the previous best result for the problem by Gasienec, Potapov, Pagourtizis [*Deterministic Gossiping in Radio Networks with Large labels*, Algorithmica 47(1) (2007), pp 97-117], by a $O(\lg n)$ factor.

Keywords: Gossiping, Unknown radio networks, Large labels.

1 Introduction

We study the problem of deterministic gossiping in unknown bidirectional radio networks, when nodes can have large labels, for synchronous model. In every round, a node can act only as a transmitter or a receiver. Whether a node is actually able to receive a message or the message sent by it is received by another node depends on the following feature of radio networks: *If two in-neighbors of a node transmit any message in the same round, then a collision occurs and the receiving node receives nothing.* It is assumed that nodes only know the number of nodes in network. This problem has been studied in [2], who give a complex protocol which takes $O(n \lg^3 n \lg \lg n)$ rounds (motivation the problem for large labels is also given in [2]). It proceeds by first learning topology of the network, which invokes leader election protocol on network $O(\lg n)$ times. We observe that one can conduct a DFS on the network, without learning neighborhood, even if nodes have large labels. Once a leader has been elected, one can use the technology developed in [3] to discover an undiscovered neighbor in $O(\lg n)$ rounds and explore network in DFS like manner. The leader finally retransmits collected messages to network by conducting another DFS. The problem of large labels does not pose a challenge as Binary-search on a space of $O(n)$ or $O(n^c)$ size takes $O(\lg n)$ rounds.

2 Deterministic Gossiping in Bi-directional Networks with Large Labels

We review a few standard sub-protocols from literature used by us. (1) **RB(n, n^c):** A deterministic protocol, which can be initiated by single source

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 449–450, 2011.

s or multiple sources, to transmit a message to all nodes of a directed network of size n, when nodes can have large labels i.e., in range $[1, \ldots, n^c]$. It takes $O(n \lg n \lg \lg n)$ rounds, [4]. (2) **Estimate**(h, X, Y)**:** Protocol initiated by a node s, to discover if a neighbor exists with label belonging to Y, but not belonging to X, using assistance of neighbor h (designed using technology in [3]). (3) **Binary-Select**$(h, X, [1, \ldots, n^c])$**:** Initiated by s to select an undiscovered neighbor of s, with label in range $[1, \ldots, n^c] - X$, in $O(\lg n)$ rounds using Estimate(.). (4) **Leader-Elect**$(N, [1, \ldots, n^c]$**:** A standard protocol for nodes of N, to elect the node with maximum label as their leader, using RB$(n+1, n^c)$, $O(\lg n)$-times.

2.1 Gossiping in Bidirectional Networks with Large Labels

Protocol *Bidirectional-Gossip* proceeds in three stages:

1. All nodes execute $Leader - Elect(N, [1, \ldots, n^c])$ to elect a leader l.
2. l initiates the rest of the protocol. It designates a helper node h from its neighborhood arbitrarily. Leader l explores the entire network, collecting messages from each node using DFS as follows. Subset X is used to maintain marked nodes. Initially, $X = \{l\}$ is passed along with token to h. In i^{th} step: Let r be a node which received the token in the $i - 1^{th}$ Step, from node r_h. Then, r updates $X = X \bigcup \{r\}$ and set of messages collected so far. Using r_h, r attempts to discover a new undiscovered neighbor by executing Binary-Select$(h, X, [1, \ldots, n^c)$, for which two possibilities can happen:
   - New node t is discovered: Token is passed to t, along with $X = X \bigcup \{t\}$, set of messages and r identified as a helper to t.
   - If no new node is discovered: Token is returned back, along with X, set of messages to r_h.
3. After token finally returns to s and it has no undiscovered neighbor, it initiates execution of DFS once more to disperse the collected messages.

We have:

Theorem 1. *Deterministic protocol Large-Gossip takes $O(n \lg^2 n \lg \lg n)$ rounds to complete gossiping on all bidirectional networks of n nodes, when node labels can have values in range $[1, \ldots, n^c]$.*

Formal description of the protocol and proof of correctness are available in [5].

References

1. Gąsieniec, L.: On efficient gossiping in radio networks. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 2–14. Springer, Heidelberg (2010)
2. Gasieniec, L., Pagourtzis, A., Potapov, I.: Deterministic communication in radio networks with large labels. Algorithmica 47(1), 97–117 (2007)
3. Kowalski, D., Pelc, A.: Time of deterministic broadcasting in radio networks with local knowledge. SICOMP 33(4), 870–891 (2004)
4. De Marco, G.: Distributed Broadcast in Unknown Radio Networks. SICOMP 39(6), 2162–2175 (2010)
5. Vaya, S.: Faster Gossiping in Bidirectional Networks with Large Labels (2011), http://arxiv.org/abs/1105.0479

Author Index