Toshimitsu Masuzawa
Sébastien Tixeuil (Eds.)

LNCS 4838

Stabilization, Safety, and Security of Distributed Systems

9th International Symposium, SSS 2007
Paris, France, November 2007
Proceedings

SSS
2007

Springer

Toshimitsu Masuzawa   Sébastien Tixeuil (Eds.)

Stabilization, Safety, and Security of Distributed Systems

9th International Symposium, SSS 2007
Paris, France, November 14-16, 2007
Proceedings

Springer

Volume Editors

Toshimitsu Masuzawa
Osaka University
Graduate School of Information Science and Technology
1-3 Machikaneyama, Toyonaka, Osaka 5608531, Japan
E-mail: masuzawa@ist.osaka-u.ac.jp

Sébastien Tixeuil
Université Pierre et Marie Curie - Paris 6
LIP6 - CNRS 7606
104 avenue du Président Kennedy, 75016 Paris, France
E-mail: Sebastien.Tixeuil@lip6.fr

Library of Congress Control Number: 2007938253

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

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

ISSN	0302-9743
ISBN-10	3-540-76626-X Springer Berlin Heidelberg New York
ISBN-13	978-3-540-76626-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12184118 06/3180 5 4 3 2 1 0

Preface

This volume contains the 27 regular papers and the abstracts of three invited keynotes that were presented at the Ninth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS) held November 14–16, 2007 in Paris, France.

SSS, the International Symposium on Stabilization, Safety, and Security of Distributed Systems, is a prestigious international forum for researchers and practitioners in design and development of fault-tolerant distributed systems with self-* properties, such as self-stabilizing, self-configuring, self-organizing, self-managing, self-repairing, self-healing, self-optimizing, self-adaptive, and self-protecting properties. It started as the Workshop on Self-Stabilizing Systems (WSS), which was first held at Austin in 1989. From the second WSS in Las Vegas in 1995, the forum was held biennially, at Santa Barbara (1997), Austin (1999), Lisbon (2001), San Francisco (2003), and Barcelona (2005). With the growth of the research field of self-stabilization, the title of the forum changed to the Symposium on Self-Stabilizing Systems (SSS) in 2003. Since 2005, SSS was run annually to encourage the rapid and sustained growth of the field, and the 2006 edition was held in Dallas. In 2006, following the demand for self-stabilization in various areas of distributed computing including peer-to-peer networks, wireless sensor networks, mobile ad hoc networks, and robotic networks, the scope of the symposium was extended to cover all safety and security-related aspects of self-* systems. The title of the symposium changed to the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS).

This year, we received 64 submissions from 22 countries. Each submission was carefully reviewed by four to six Program Committee members with the help of external reviewers, and the Program Committee selected the 27 papers. It is worthwhile noticing that the overall quality of submissions was excellent and there are many papers that we had to reject because of organization constraints yet deserved to be published. The three invited keynotes dealt with hot topics absorbing the interest of attendees: "The Power of Cryptographic Attacks: Is Your Network Really Secure Against Side Channels Attacks and Malicious Faults?" by Jean-Jacques Quisquater, "Role-Based Self-Configuration of Sensor Networks" by Kay Römer, and "Robots and Molecules" by Masafumi Yamashita. Following the recent tradition of SSS, Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao received the Best Paper Award for their paper, "Probabilistic Fault-Containment."

On behalf of the Program Committee, we would like to thank all authors of submitted papers for their support. We also thank the members of the Steering Committee for their invaluable advice. We wish to express our appreciation to the Program Committee members and additional external reviewers for their tremendous effort and excellent reviews. We gratefully acknowledge the Organizing Committee members for their generous contribution to the success of the symposium. Paper submission, selection, and generation in the proceedings was greatly eased by the use of the EasyChair conference system (`http://www.easychair.org`). We wish to thank the EasyChair creators and maintainers for their commitment to the scientific community.

"We gratefully acknowledge the financial support from [illegible], the [illegible], the [illegible], and the [illegible]".

November 2007

Toshimitsu Masuzawa
Sébastien Tixeuil

Conference Organization

Steering Committee

Anish Arora, Ohio State Univ., USA
Ajoy K. Datta, Univ. of Nevada, Las Vegas, USA
Shlomi Dolev, Ben-Gurion Univ. of the Negev, Israel
Sukumar Ghosh (Chair), Univ. of Iowa, USA
Mohamed G. Gouda, Univ. of Texas at Austin, USA
Ted Herman, Univ. of Iowa, USA
Shing-Tsaan Huang, National Central Univ., Taiwan
Vincent Villain, Univ. de Picardie, France

Program Chairs

Toshimitsu Masuzawa, Osaka Univ., Japan
Sébastien Tixeuil, Univ. Pierre et Marie Curie - Paris 6, France

Program Committee

Tadashi Araragi, NTT Co., Japan
Anish Arora, Ohio State Univ., USA
James Aspnes, Yale Univ., USA
Doina Bein, Univ. of Texas at Dallas, USA
Jorge A. Cobb, Univ. of Texas at Dallas, USA
Frederic Cuppens, ENST Bretagne, France
Ajoy K. Datta, Univ. of Nevada, Las Vegas, USA
Hervé Debar, Orange Labs, France
Sylvie Delaët, Univ. Paris-Sud, France
Danny Dolev, Hebrew Univ. of Jerusalem, Israel
Shlomi Dolev, Ben-Gurion Univ., Israel
Eric Filiol, INRIA, France
Paola Flocchini, Univ. of Ottawa, Canada
Felix Freiling, Univ. of Mannheim, Germany
Toru Fujiwara, Osaka Univ., Japan
Sukumar Ghosh, Univ. of Iowa, USA
Dieter Gollmann, Hamburg Univ. of Technology, Germany
Maria Gradinariu, Univ. Pierre et Marie Curie - Paris 6, France
Isabelle Guérin-Lassous, ENS Lyon, France
Rachid Guerraoui, EPFL, Switzerland
Phuong Ha, Univ. Tromsø, Norway
Ted Herman, Univ. of Iowa, USA

Jaap-Henk Hoepman, TNO / Radboud Univ. Nijmegen, The Netherlands
Chin-Tser Huang, Univ. of South Carolina at Columbia, USA
Shing-Tsaan Huang, National Central Univ., Taiwan
Michiko Inoue, Nara Institute of Science and Technology, Japan
Hirotsugu Kakugawa, Osaka Univ., Japan
Mehmet H. Karaata, Kuwait Univ., Kuwait
Yoshiaki Katayama, Nagoya Institute of Technology, Japan
Boris Koldehofe, Univ. of Stuttgart, Germany
Sandeep S. Kulkarni, Michigan State Univ., USA
Shay Kutten, Technion, Israel
Toshimitsu Masuzawa (Program Chair), Osaka Univ., Japan
Ludovic Mé, Supelec Rennes, France
Miodrag Mihaljevic, Serbian Academy of Sciences and Arts, Belgrade / RCIS-AIST, Japan
Mikhail Nesterenko, Kent State Univ., USA
Marina Papatriantafilou, Chalmers Univ., Sweden
Andrzej Pelc, Univ. du Quebec en Outaouais, Canada
Franck Petit, LaRIA, Univ. de Picardie, France
Scott Pike, Texas A&M Univ., USA
Sergio Rajsbaum, Univ. Nacional Autonoma de Mexico, Mexico
Matthieu Roy, LAAS-CNRS, France
Kouichi Sakurai, Kyushu Univ., Japan
Pierre Sens, Univ. Pierre et Marie Curie - Paris 6, France
Neeraj Suri, TU Darmstadt, Germany
Sébastien Tixeuil (Program Chair), Univ. Pierre et Marie Curie - Paris 6, France
Eric Totel, Supelec Rennes, France
Tatsuhiro Tsuchiya, Osaka Univ., Japan
Masafumi Yamashita, Kyushu Univ., Japan

Organizing Committee

Luciana Arantes, Univ. Pierre et Marie Curie - Paris 6, France
Sylvie Delaët (Organizing Chair), Univ. Paris-Sud, France
Stéphane Devismes, Univ. Paris-Sud, France
Maria Gradinariu, Univ. Pierre et Marie Curie - Paris 6, France
Pierre Sens, Univ. Pierre et Marie Curie - Paris 6, France
Sébastien Tixeuil, Univ. Pierre et Marie Curie - Paris 6, France
Véronique Varenne, Univ. Pierre et Marie Curie - Paris 6, France

Additional Reviewers

Fabien Autrel
J. Alfonso Briones
Joffroy Beauquier
Zinaida Benenson
Bruhadeshwar Bezawada
Christophe Bidan

Erik-Olver Blaß
Christian Boulinier
Rainer Böhme
Hui Cao
Eddy Caron
Julien Clement
Nora Cuppens
Stéphane Devismes
Yoann Dieudonne
Dinil M. Divakaran
Hen Fitoussi
Zhang Fu
Kazuhide Fukushima
Joaquin Garcia-Alfaro
Leszek Gasieniec
Giorgos Georgiadis
Anders Gidenstam
Seth Gilbert
Andreas Grau
Sammy Haddad
Rachid Hadid
Yoshiaki Hori
Michel Hurfin
Yasunori Ishihara
Taisuke Izumi
David P. Jacobs
Mark Jelasity
Hyung Chan Kim
Gerald Georg Koch
Dariusz Kowalski
Rastislav Kralovic
Ioannis Krontiris
Vinod Kulkathumani
Ouiddad Labbani-Igbida
Limor Lahiani
William Leal
Francois Lesueur
Benjamin Morin
Vinayak Naik
Boaz Patt-Shamir
David Peleg
Lucia Draque Penso
Giuseppe Prencipe
Rami Puzis
Marisuz Rokicki
Elad M. Schiller
Mukundan Sridharan
Messika Stephane
Kenichi Takahashi
Cedric Tedeschi
Julien Thomas
Frederic Tronel
Nir Tzachar
Remi Vannier
Koichi Wada
Yi Xian
Reuve Yagel
Hongwei Zhang

Table of Contents

The Power of Cryptographic Attacks: Is Your Network Really Secure Against Side Channels Attacks and Malicious Faults?

Jean-Jacques Quisquater

UCL Crypto Group, Microelectronics Laboratory
Université Catholique de Louvain, Belgium
quisquater@dice.ucl.ac.be

Abstract. When speaking about attacks against networks and computers, people mainly think today about viruses, worms, Trojans, keyloggers, denial of services, etc.

In the last ten years a lot of new attacks were found against servers and smart cards. First are side-channels attacks: those are by using "esoteric" channels to obtain protected, secure and private informations. Esoteric here means very often channels related to the communication channel (time of interaction), processors (power, electromagnetic radiations, caches, branching, etc.). Second are the malicious faults related to secret key cryptography. The interaction of cryptographic algorithms with malicious faults must be carefully known and understood: one error sometimes means a totally broken system.

We will survey the field with a focus on distributed systems and networks.

Role-Based Self-configuration of Sensor Networks

Kay Römer

Institute for Pervasive Computing
ETH Zurich, Switzerland
roemer@inf.ethz.ch

Abstract. Wireless sensor networks consist of so-called sensor nodes – small untethered computing devices equipped with sensors, a wireless radio, a processor, and autonomous power supply. Large and dense networks of these devices can be deployed unobtrusively in the physical environment in order to monitor a wide variety of real-world phenomena with unprecedented quality and scale while only marginally disturbing the observed physical processes.

Many sensor network applications require some form of self-configuration, where sensor nodes take on specific functions in the network. Configuration of a sensor network is particularly challenging, as the anticipated large number of sensor nodes participating in a network typically precludes manual configuration of individual nodes. Additionally, pre-deployment configuration is often infeasible because some configuration parameters such as node location and network neighborhood are typically unknown prior to deployment. Also, node parameters may change over time, necessitating dynamic re-configuration.

In this talk we present a framework for the development of self-configuring sensor networks known as *generic role assignment*. The key idea is to consider self-configuration as the problem of assigning a *role* to each sensor node such that certain global constraints are satisfied. Both the set of available roles and the constraints can be specified by the developer using a declarative specification language. These specifications are compiled and executed in the sensor network, where a distributed role assignment algorithm finds an assignment of roles to sensor nodes that is compliant with the specification. Assigned roles are updated to reflect changes in the sensor network resulting, for example, from addition or removal of nodes. The role assignment algorithms are efficient regarding the communication overhead and robust with respect to message loss.

Using this framework, a variety of different self-configuration problems can be implemented. Example problems include *coverage* (assign roles `ACTIVE` and `SLEEP`, such that few active nodes cover the area of interest with their sensors while the remaining nodes can be turned into a power-saving sleep mode) or *clustering* (assign roles `SLAVE`, `GATEWAY`, and `HEAD` such that each slave has a cluster head neighbor where cluster heads are interconnected by gateway nodes to form a connected backbone). These and other problems can be specified with few lines of code using the declarative specification language, which effectively shields the developer from low-level implementation details.

Robots and Molecules

Masafumi Yamashita

Kyushu University, Fukuoka 819-0395 Japan
mak@csce.kyushu-u.ac.jp

Abstract. We survey some of the recent theoretical works about autonomous mobile robot systems and then discuss the possibility of extending robot models to analyze molecular computing systems. There are two types of robot systems appearing in the Distributed Computing literature. One is a self-reconfigurable system, which consists of a number of identical robotic modules that can connect to, disconnect from, and relocate relatively to adjacent modules. A behaviour of a self-reconfigurable system looks like the life game and we are interested in desigining an algorithm (local map) that makes the whole system behave in a consistent way. The other model is an autonomous mobile robot system, which consists of mobile robots with eye sensors as communication devices. The formation problem of a given geometrical pattern has been discussed extensively. Although those two systems have some common features and common goals, they were proposed independently and have been investigated separetely.

After surveying some of works on those robot systems, we introduce some works in molecular computing. We explain some of the examples in which those robot systems appear naturally. We then argue some problems that arise from the molecular computing applications. A key coincidence between the robot models and the molecular computing is that they are systems composed of anonymous modules.

Relating Stabilizing Timing Assumptions to Stabilizing Failure Detectors Regarding Solvability and Efficiency

Martin Biely[1,⋆], Martin Hutle[2], Lucia Draque Penso[3], and Josef Widder[1,4,⋆⋆]

[1] Technische Universität Wien, Austria
biely@ecs.tuwien.ac.at
[2] École Polytechnique Fédérale de Lausanne, Switzerland
martin.hutle@epfl.ch
[3] University of Mannheim
lucia@rumms.uni-mannheim.de
[4] École Polytechnique, France
widder@lix.polytechnique.fr

Abstract. We investigate computational models with stabilizing properties. Such models include e.g. the partially synchronous model [Dwork et al. 1988], where after some unknown global stabilization time the system complies to bounds on computing speeds and message delays, or the asynchronous model augmented with unreliable failure detectors [Chandra et al. 1996], where after some unknown global stabilization time failure detectors stop making mistakes.

Using algorithm transformations (a notion we introduce in this paper) we show that many (families of such) models are equivalent regarding solvability. We also analyze the efficiency of such transformations regarding not only the number of steps in a model M_1 necessary to emulate a step in a model M_2, but also the stabilization shift, which bounds the number of steps in M_2 required to provide properties of M_2 after the stabilization of M_1.

1 Introduction

We consider distributed message passing systems that are subject to crash failures. Due to the well-known impossibility result for deterministic consensus in asynchronous systems [1], a lot of research was done about adding assumptions to the asynchronous model in order to allow solving the problem. These include assumptions on the timing behavior of processes and communication links [2,3] as well as assumptions on the capability of processes to retrieve information on failures of others [4].

[illegible]. The previous sentence is stated in many research papers and sometimes even the required

⋆ Supported by the Austrian BM:vit FIT-IT project *TRAFT* (proj. no. 812205).
⋆⋆ Partially supported by the Austrian FWF project *Theta* (proj. no. P17757).

amount of synchrony to solve a problem is expressed via the weakest failure detector necessary (e.g. [5]). Interestingly, Charron-Bost et al. [6] have shown that in general, failure detectors do [illegible] encapsulate timing assumptions properly. For perpetual kind failure detectors as the perfect failure detector $\mathcal{P}$ it was shown that the synchronous system has "a higher degree of synchrony" than expressed by the axiomatic properties of $\mathcal{P}$. Being optimistic, one could only hope that (weaker) failure detectors that are just eventually reliable are equivalent to the timing models sufficient for implementing them. This is the first issue we address.

Another line of research considers "asymmetric" models in which timing assumptions need not hold at all links and all correct processes — as in [2,3,7] — but only for a subset of components in a system. This stems from the following question in [8]: Is there a model that allows implementing the eventually strong failure detector $\diamond\mathcal{S}$ (which can be reduced to the eventual leader oracle Ω), but does not allow to implement $\diamond\mathcal{P}$ (i.e., the eventually perfect failure detector, whose output stabilizes to complete information on remote process crashes)? Indeed, it was shown in [9] that such models exist. Since [9], much interest [10,11,12,13] arose in weakening the synchrony assumption of models (or adding as little as possible to the asynchronous model) in order to be able to implement Ω. Regarding solvability, if such models are stronger than the asynchronous one, then these models would allow to solve all problems that can be solved with Ω but would not allow to solve problems where Ω is too weak. The second issue addressed in this paper is thus whether the different spatial distributions of timing assumptions proposed make a difference in the set of problems which they allow to solve.

To tackle these challenges, we consider two main types of models: [illegible] and [illegible]. On one hand, [illegible], such as round-based models and failure detector based models, do not consider the timing behavior of distributed systems. For instance, round-based models restrict the sets of messages which have to be received in the round they were sent, while failure detector based models introduce axiomatic properties to guarantee access to information about failures. On the other hand, [illegible], such as the partially synchronous and eventually synchronous models, have explicit assumption on processing speeds and message delays. However, note that a property which is shared by all models we consider is that they are stabilizing, i.e., they restrict the communication in a distributed computation only from some unknown stabilization time on.

Finally, we introduce the notion of [illegible], which we use to compare different models from both a solvability and an efficiency viewpoint.

1.1 Contribution

[illegible] In this paper we show that the result of [6] is in fact limited to perpetual type failure detectors. To this end we introduce a new parametrized failure detector family, of which both $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$ are special cases. Additionally, we define new parametrized (with respect to the number and distribution of eventually timely links and processes) partially and eventually synchronous model families. In terms of solvability, we show equivalence when instantiating both families with the same parameter k. As a corollary we show that the asynchronous system with $\diamond\mathcal{P}$ allows to solve the same problems as the classic partially synchronous model in [3], as well as that the asynchronous model augmented with Ω is equivalent to several source models, i.e., models where just the links from at least one process (the source) are timely.

[illegible] We introduce the notion of [illegible] for partially synchronous systems and for asynchronous systems augmented with failure detectors. While [illegible] (or their close relatives simulations [14]) are well understood in the context of synchronous as well as asynchronous systems, they have to the best of our knowledge never been studied before for partially synchronous systems.

In the case of synchronous systems, the simplifying assumption is made that no additional local computation and no number of messages that has to be sent for a simulation may lead to a violation of the lock-step round structure. It follows that layering of algorithms as proposed in [14] can be done very easily.

In contrast, for asynchronous systems no time bounds can be violated anyhow. Consequently, the coupling of the algorithm with the underlying simulation can be done so loosely that between any two steps of the algorithm an arbitrary number of simulation steps can be taken. Thus, asynchronous simulations can be very nicely modeled e.g. via I/O automata [15].

For partially synchronous systems, transformations are not straight forward. Based on primitives of lower models, primitives of higher models must be implemented in a more strongly coupled way than in asynchronous systems, while it has to be [illegible][1] that the required timing properties are achieved. For that purpose, we define algorithm transformations, which we discuss in Section 2.6.

[illegible] We also discuss the cost of these algorithm transformations, by examining two diverse measures. The first considers the required number of steps in one model in order to implement one step in the other one. To this end we introduce the notion of B [illegible] transformations which means that any [illegible] of the higher model can be implemented by at most B [illegible] of the lower model.

The second parameter considers how many steps are required to stabilize the implemented steps after the system has stabilized. For this we use the notion of D [illegible], which means that after the system stabilizes, the implemented steps stabilize at most after D steps. Further we introduce the notion of efficiency preserving: A transformation is [illegible], if it is B-bounded, has D-bounded shift, and B and D are known in advance. We show between which pairs of models efficiency-preserving transformations exist.

2 System Models

In this paper we consider multiple models of distributed computations, which vary in their abstraction. For example, failure detector based models are at a

[1] In sharp contrast to the synchronous case, where timing *is assumed to hold*.

higher level of abstraction than partially synchronous systems as they abstract away the timing behavior of systems [4], while round-based models can be seen as being situated at an even higher level as they abstract away how the round structure is enforced — which can be done based on either timing assumptions [16] or on failure detector properties [4]. Whether these abstractions have the "cost" of losing relevant properties of the "lower level" model is the central question of this paper.

We first turn to the common definitions, and then describe the specifics of each model separately.

2.1 Common Definitions

A system is composed by a set Π of n distributed processes $p_1, \dots p_n$ interconnected by a point-to-point message system. Each process has its own local memory, and executes its own automaton. In every execution all processes stick to their specified automaton, except for f which prematurely halt. We call such processes crashed. Process that do not crash are correct as they take an infinite number of steps in infinite executions. All system models we consider assume an upper bound $t \geq f$ on the number of crashes in every execution.

A system model defines the behavior of the environment of the automatons with respect to a set of operations that bind together the automatons by allowing them to interact with each other by manipulating or querying their environment (e.g., the message system). In this paper, these operations are send and receive operations used to exchange messages from an alphabet $\mathcal{M}$. To simplify presentation, messages are assumed to be unique.

We define a partial run as an (infinite) sequence of global states C_i. A global state C_i is composed of the local states of the n automata corresponding to n processes and the message system (i.e., the messages in transit). We say that a process [illegible], when its local state and possibly also the state of the message system changes. A run is a partial run starting in an initial configuration.

A run is said to be admissible in a system model if the run sticks to the relations between the operations that are defined in the system model. As an example, all models in this paper define the message system to be composed of reliable channels. Without going into the particular definitions of the send and receive operations, we can describe the abstract notion of reliable channels by the following three properties[2]:

Reliability. Every message sent to a correct process is eventually delivered.
Integrity. A message is delivered only if it was actually sent.
No Duplication. No message is received more than once.

Below (in Subsections 2.2, 2.3, 2.4, and 2.5) we will complement these definitions with additional assumptions. Since we are interested in the spatial distribution of synchrony in this paper, these assumptions will only hold for some

[2] Note that we did not choose the buffer representation as in [3] but used equivalent separate properties instead to characterize reliable channels.

parts of the system. In fact, we will define families of synchrony models, which only differ in the size of these subsets. Note that (in contrast to [17]) the subset for which the synchrony holds is not known. Just the smallest size of this subset is known.

2.2 The Failure Detector Model

In an asynchronous system model with a failure detector, a process p that executes a [illegible] may execute during every computational step the following operations [illegible]:

***a-receive$_p$*()**: Delivers a message m, $\langle m, q\rangle \in \mathcal{M} \times \mathit{\Pi}$, sent from q to p.
***a-query$_p$*()**: Queries the failure detector of p.
***a-send$_p$*(m, q)**: Sends a message m to process q.

Not all of these operations (but at least one) have to be performed in each step. Algorithms for the asynchronous model do not have access to global time.

A failure detector is of class $\mathcal{G}_k$, if it outputs a set of processes, k is a (possibly constant) function of the failure pattern, and the failure detector fulfills:

***k*-Eventual Trust.** In every execution, there exists a set $\mathit{\Pi}'$ consisting of at least k of correct processes, such that there exists a time τ from which on the failure detector output of all correct processes is a set of correct processes and a superset of $\mathit{\Pi}'$.

The minimal instant τ is called [illegible]. Although this failure detector might seem artificial at first sight, it turns out to unify most of the classical stabilizing failure detectors in literature:

– The eventual strong failure detector $\diamond\mathcal{S}$ [4] guarantees that eventually at least one correct process is not suspected by any correct process. Therefore its output is the converse of the output of $\mathcal{G}_1$. That is, the processes that are not suspected by $\diamond\mathcal{S}$ are those that are trusted by $\mathcal{G}_1$ at each process, and the (at least) one process which is not suspected by any process' $\diamond\mathcal{S}$ is the (at least) one process that is in the intersection of the failure detector outputs of all correct processes.
– The eventual perfect failure detector $\diamond\mathcal{P}$ is the converse of $\mathcal{G}_{n-f}$.
– Finally, the eventual leader election oracle $\mathit{\Omega}$ [7] chooses eventually exactly one leader at all correct processes. This corresponds to a (stronger) variant of $\mathcal{G}_1$, where the output always has only one element.

In the following, we will denote the family of asynchronous systems augmented with $\mathcal{G}_k$ for some k by Async+$\mathcal{G}$.

2.3 The Partially Synchronous Model

This section's model is a variant of the generalized partially synchronous model given in [4]. Based on the steps in a run, we define a discrete global timebase with instants $\tau \in \mathbb{N}$, which is inaccessible to processes. At every instant of this time, every process may execute at most one step and at least one process executes a step. A process p that runs a [illegible] executes at most one step at every discrete time τ and uses [illegible] of the following operations in every step:

***par-send$_p$(m, q)*:** Sends a message m to q.
***par-receive$_p$()*:** Delivers a set S, s.t. $\emptyset \subseteq S \subseteq M \times \Pi$ of message-sender pairs.

Definition 1. [illegible] Δ holds between p and q [illegible] τ, [illegible] m [illegible] p [illegible] q [illegible] τ [illegible] p, [illegible] $_q()$ [illegible] $\tau' \geq \tau + \Delta$ m [illegible] p [illegible] τ'

Definition 2. [illegible] Φ [illegible] p [illegible] τ [illegible] $\Phi + 1$ [illegible] τ, [illegible] p [illegible]

In contrast to [3], Δ only holds for all [illegible] of k processes, which are called sources. We thus assume that in every execution there is a set of processes Π' of cardinality at least k such that:

***k*-Partial Sources.** Eventually some unknown Δ holds for all outgoing links of processes in Π'.
***k*-Partially Synchronous Processes.** Eventually some unknown Φ holds for each process in Π'.

The minimal time from which on the two properties hold for Π' is called stabilization time. (In contrast to [3], this stabilization time is not global, as it only holds for a subset of the system.)

We denote the system model where all executions fulfill [illegible], [illegible], k[illegible], and k[illegible], as ParSync$_k$. If we do not fix k, we denote this family of models as ParSync.

[illegible] Often a variant of partial synchrony is considered where message loss before the global stabilization time may occur. Here we consider only the case with [illegible], for the following reason: In [18] it is shown that fair lossy links can be transformed into reliable ones, if $n > 2t$, and that it is impossible to transform eventually reliable links into reliable links if $n \leq 2t$. So in the former case, which is also the relevant one for consensus, our results regarding solvability hold as well, whereas for the latter case, the opposite of our result is trivially true: It is not possible to build an asynchronous system with reliable links plus a failure detector in a partially synchronous system with message loss before the stabilization time.

Note that Dwork et al. [3] define another variant for partially synchronous communication, where Δ holds always, but is unknown. Since we have reliable channels, this is equivalent to our definition.

2.4 The Eventually Synchronous Model

The eventually synchronous model is a variant of partial synchrony, where the bounds on the communication delay and relative speeds are known, but hold only eventually. This is one of the two models in [3].

This model is very similar to the model of partial synchrony, for sake of brevity we do not go into much detail and only state the operations and properties of the model:

***ev-send*$_p(m, q)$:** Sends a message m to q.
***ev-receive*$_p()$:** Delivers a set S, s.t. $\emptyset \subseteq S \subseteq M \times \Pi$ of message-sender pairs.

As in the partially synchronous case, we consider a set of processes Π' of cardinality at least k, and we assume the following two properties:

k-Eventual Sources. A known Δ eventually holds for all outgoing links of the processes in Π'.
k-Eventually Synchronous Processes. A known Φ eventually holds for each process in Π'.

We denote the system model where all executions fulfill [illegible], [illegible], [illegible], k [illegible] and k [illegible], as $\diamond$Sync$_k$. If we do not fix k, we denote this family of models as $\diamond$Sync.

Observe that ParSync$_k$ is — by definition — [illegible] than $\diamond$Sync$_k$, that is, for any k, every execution in $\diamond$Sync$_k$ is also an execution in ParSync$_k$.

In order to distinguish the Δ for ParSync and for $\diamond$Sync, we will use $\Delta_?$ for the former, and $\Delta_\diamond$ for the latter. When no ambiguity arises we will however only use Δ.

2.5 The Round Model

In our round-based system, processes proceed in rounds $r = 0, 1, 2, \ldots$. For a [illegible], in every round r, a process p executes exactly one step comprising a send operation followed by exactly one step comprising a receive operation, where the operations are defined as:

***rd-send*$_p(r, S)$:** Sends a set $S \subseteq M \times \Pi$ of messages. For every process q, S contains at most one message m_q.
***rd-receive*$_p(r)$:** Delivers a set $S \subseteq M \times \Pi \times \mathbb{N}$ of messages to p, where a tuple $\langle m, q, r' \rangle$ denotes a message m sent by q to p in round $r' \leq r$.

Further, we define the property:

k-Eventual Round Sources. There is a set Π' of k correct processes, and a round r, such that every message that is sent by some $p \in \Pi'$ in some round $r' \geq r$ is received in round r' by all correct processes.

We denote the system model where all executions fulfill [illegible], [illegible], [illegible] and k [illegible], as Round$_k$. The family of all these models is denoted Round.

In our definition of [illegible] above we do not allow the reception of messages from future rounds. This implies that for each round r the [illegible] (r) operations form a consistent cut. By [19, Theorem 2.4] this is equivalent to these operations taking place in lockstep. Note also, that our model is communication open, and thus contrasts the communication closed round models used for example in [20,21].

2.6 Algorithm Transformations

In order to relate our models, we use [illegible]. An algorithm transformation $\mathcal{T}_{A \rightharpoonup B}$ generates from [illegible] algorithm $\mathcal{A}$ that is correct (with respect to some problem specification) in some system $\mathcal{S}_A$ an algorithm $\mathcal{B}$ that is correct (with respect to the same specification) in some other model $\mathcal{B}$ by implementing the operations in $\mathcal{S}_A$ by operations in $\mathcal{S}_B$. For the correctness of such transformations, it has to be shown that $\mathcal{B}$ is well-formed as well, and that the implemented operations are those defined in $\mathcal{S}_A$. Moreover it has to be shown that the assumptions on the operations of $\mathcal{S}_A$ that $\mathcal{A}$ is based on, hold for their implementations (given by the transformation) in $\mathcal{S}_B$. This is captured by the notion of a trace: The [illegible] $\mathcal{S}_A$ [illegible] is the sequence of implementations of $\mathcal{S}_A$ operations (in $\mathcal{S}_B$) the algorithm calls when being executed via the transformation $\mathcal{T}_{A \rightharpoonup B}$ in $\mathcal{S}_B$.

Obviously, problem statements only make sense here if they can be stated independently of the model. Consequently, defining e.g. termination as “the consensus algorithm terminates after x rounds” is not model independent as there is no formal notion of “a round” e.g. in partially synchronous system models. Therefore the notion of a round in such models depends on the algorithms which implement them. Hence, such properties should be regarded as belonging to the algorithm and not to the problem and will be dealt with in the discussions on efficiency of transformations below. In the literature on models with stabilizing properties, algorithms which decide (or terminate) within an a priori bounded number of steps after stabilization time are termed [illegible]. Therefore we are interested in the possibility of transforming efficient algorithms into efficient algorithms.

An algorithm transformation is B [illegible], iff any [illegible] of the higher model can be implemented by at most B [illegible] of the lower model.

Another measure for the efficiency of a transformation is how it behaves with respect to the stabilization in the two models involved. To motivate this measure, consider some implementation of e.g. the eventual perfect failure detector based on some partially synchronous system which has some global stabilization time τ. Since the timing before τ is arbitrary, the processes that some process p suspects may be arbitrary at τ as well. It may take some time until the set of suspected processes at p is consistent. Until then, the asynchronous algorithm using the failure detector may query the failure detector a couple of times — say x times — before the failure detector becomes consistent. Informally, x may be used as measure for the transformation of stabilizing properties.

More formally, since we are considering models with stabilizing properties there is usually a step s from which on $\mathcal{S}_B$ guarantees some properties. This step s is part of some step S in $\mathcal{S}_A$. For a valid transformation, there must also be a step S' from which on the stabilizing properties of the implemented model are guaranteed to hold. When $S \neq S'$ the transformation is [illegible]. Moreover, we say that a transformation has D [illegible], when the transformation guarantees that S' does not occur more than D steps after S.

Definition 3. [illegible] efficiency preserving [illegible] B [illegible] D [illegible] B [illegible] D [illegible]

Note that this definition implies that parameters unknown in advance, e.g., $\Delta_?$ and $\Phi_?$ in the ParSync models, cannot occur in the expressions given for B, while the size of the system, n, can.

3 Equivalence of Solvability

3.1 Possibility of Async+$\mathcal{G}$⇀ParSync

As there are no assumptions on the relative processor speeds and the time it takes for the network to transmit a message, all we need to show is that there is a set of processes that eventually meets the requirements of k-Eventual Trust.[3]

Lemma 1. [illegible] ParSync [illegible] $\mathcal{A}$ [illegible] Async+$\mathcal{G}_k$ [illegible] k [illegible] Async+$\mathcal{G}$ [illegible]

All messages that are received via [illegible] are stored into [illegible]$_p$ and then appended to [illegible]$_p$, from where [illegible] takes them while filtering out the additional $\perp$ messages, therefore our reliable channel assumptions follow from their counterparts of the ParSync-steps, and we obtain that:

Lemma 2. [illegible] Reliability Integrity [illegible] No Duplication

Note that the ever increasing [illegible]$_p$ affects the detection time of the failure detector and has no impact on whether the transformation is bounded. However, for the Async+$\mathcal{G}$-algorithm stabilization only occurs after [illegible]$_p$ is greater than $\Delta_? + (n+1)\Phi_?$ (cf. Section 4.2). As $\Delta_?$ and $\Phi_?$ are unknown there is no a priori known bound for the stabilization shift.

Corollary 1. [illegible] $\mathcal{A}$ [illegible] Async+$\mathcal{G}_x$ [illegible] ParSync$_x$ [illegible]

3.2 Possibility of ParSync⇀◇Sync

Since the properties of ◇Sync$_k$ imply the properties of ParSync$_k$, for this transformation it suffices to replace [illegible] with [illegible], and [illegible] with [illegible], respectively. Also note that there is no stabilization shift either (since abiding to known bounds implies abiding to unknown ones). We thus have:

Corollary 2. [illegible] ParSync [illegible] ◇Sync [illegible]

[3] Due to space limitations, all proofs are omitted. They are included in the full version of the paper [22].

Algorithm 1. Transforming ASYNC+$\mathcal{G}$ algorithms to PARSYNC algorithms

```
variables
  $\forall q \in \Pi$ :  $send_p[q]$, initially $\perp$
  $trusted_p \subseteq \Pi$, initially $\emptyset$
  $buffer_p$, $undelivered_p$ FIFO queue with elements in $\mathcal{M} \times \Pi$, initially empty
  $counter_p$, $threshold_p \in \mathbb{N}$, initially 0 and 1, resp.
end variables

operation $a\text{-}send_p(m, q)$
  $send_p[q] \leftarrow m$
  $update()$
end operation

operation $a\text{-}receive_p()$
  $update()$
  if $undelivered_p$ does not contain non-$\perp$ messages then
    return $\perp$
  else
    return first $\langle m, q\rangle$ from $undelivered_p$ where $m \neq \perp$
      [removing all pairs up to and including $\langle m, q\rangle$]
  end if
end operation

operation $a\text{-}query_p()$
  $update()$
  return $trusted_p$
end operation

procedure $update()$
  for all $q \in \Pi$ do
    $par\text{-}send(send_p[q], q)$
    $send_p[q] \leftarrow \perp$
  end for
  append all $\langle m, q\rangle \in par\text{-}receive_p()$ to $buffer_p$
  $incr(counter_p)$
  if $counter_p = threshold_p$ then
    $counter_p \leftarrow 0$
    $trusted_p \leftarrow \{q : \langle *, q\rangle \in buffer_p\}$
    append $buffer_p$ to $undelivered_p$
    $buffer_p \leftarrow \emptyset$
    $incr(threshold_p)$
  end if
end procedure
```

3.3 Possibility of $\diamond$SYNC$\rightharpoonup$ROUND

The next transformation we consider is one that transforms any algorithm for $\diamond$SYNC to an algorithm for the ROUND model. While each $\diamond$SYNC$_k$ model can be instantiated with any values for $\Delta_\diamond$ and $\Phi_\diamond$, we implement a particular instance, i.e., $\diamond$SYNC$_k$ with $\Delta_\diamond = 0$ and $\Phi_\diamond = 1$. The basic idea of the transformation is to execute one round of the ROUND model in each $\diamond$SYNC step.

Lemma 3. [illegible] ROUND [illegible] $\mathcal{A}$ [illegible] $\diamond$SYNC [illegible] k-Eventual Sources [illegible] k-Eventually Synchronous Processes [illegible] $\Phi = 1$ [illegible] $\Delta = 0$ [illegible] $\diamond$SYNC [illegible]

Lemma 4. [illegible]

Lemma 5. [illegible] Reliability Integrity [illegible] No Duplication

Algorithm 2. Transforming ◇Sync algorithms to Round algorithms

```
 1: variables
 2:    $r \in \mathbb{N}$, initially 0
 3:    $buffer_p \subseteq \mathbb{N} \times \mathcal{M} \times \Pi$, initially $\epsilon$
 4: end variables

 5: operation $ev\text{-}send_p(m, q)$
 6:    $rd\text{-}send_p(r, \{\langle m, q\rangle\})$
 7:    $buffer_p \leftarrow buffer_p \cup rd\text{-}receive_p(r)$
 8:    $r \leftarrow r + 1$
 9: end operation

10: operation $ev\text{-}receive_p()$
11:    $rd\text{-}send_p(r, \emptyset)$
12:    $buffer_p \leftarrow buffer_p \cup rd\text{-}receive_p(r)$
13:    $r \leftarrow r + 1$
14:    $del \leftarrow \{\langle m, q\rangle \mid \langle m, q, *\rangle \in buffer_p\}$
15:    $buffer_p \leftarrow \emptyset$
16:    return $del$
17: end operation
```

It can be easily seen that we have:

$$[illegible] \mapsto \begin{cases} [illegible] \\ [illegible] \end{cases} \qquad [illegible] \mapsto \begin{cases} [illegible] \\ [illegible] \end{cases}$$

yielding that this transformation is 2-[illegible], since only one of the two operations is possible in each step of our delay-bounded models. Moreover, from Lemmas 3, 4 and 5 it is obvious that:

Corollary 3. [illegible] [illegible] $\mathcal{A}$ [illegible] ◇Sync$_x$ [illegible] Round$_x$ [illegible] 2 [illegible]

3.4 Possibility of Round⟶Async+$\mathcal{G}$

With this section's transformation, we close the circle of transformations thereby establishing that the same problems are solvable in all four model families.

For implementing a round structure on top of our failure detector we simply wait in each round until we have received messages for the current round from all trusted processes.

Lemma 6. [illegible] Async+$\mathcal{G}$ [illegible] $\mathcal{A}$ [illegible] Round [illegible] k-Eventual Round Sources [illegible] Round [illegible]

Lemma 7. [illegible] Reliability Integrity [illegible] No Duplication

Again, this lemma follows directly from the transformation and from the fact that the properties are provided by Round (cf. for example lines 14 and 19 for [illegible] and [illegible], resp.). Transforming Round to Async+$\mathcal{G}$ with Algorithm 3, we have:

Algorithm 3. Transforming Round algorithms to Async+$\mathcal{G}$ algorithms

```
variables
   $undelivered_p \subseteq \mathbb{N} \times \mathcal{M} \times \Pi$, initially empty
end variables

operation $rd\text{-}send_p(S, r)$
   for all $\langle m, q \rangle \in S$ do
      $a\text{-}send_p(\langle r, m \rangle, q)$
   end for
   for all $q \notin \{q' \mid \langle m', q' \rangle \in S\}$ do
      $a\text{-}send_p(\langle r, \perp \rangle, q)$
   end for
end operation

operation $rd\text{-}receive_p(r)$
   repeat
      $undelivered_p \leftarrow undelivered_p \cup \{a\text{-}receive_p()\}$
      $trusted_p \leftarrow a\text{-}query_p()$
      $Q \leftarrow \{q \mid \langle r, *, q \rangle \in undelivered_p\}$
   until $Q \supseteq trusted_p$
   $del \leftarrow \{\langle r', m, q \rangle \in undelivered_p \mid r' \leq r \wedge m \neq \perp\}$
   $undelivered_p \leftarrow undelivered_p \setminus del$
   return $del$
end operation
```

$$[illegible] \mapsto \begin{cases} [illegible] & (*, 1) \\ \vdots & \\ [illegible] & (*, n) \end{cases} \qquad [illegible] \mapsto \begin{cases} [illegible] \\ [illegible] \\ [illegible] \\ [illegible] \\ \vdots \end{cases}$$

yielding that this transformation is [illegible]. (Although the [illegible] are not necessarily executed in the given order.) Therefore we conclude this subsection with observing that:

Corollary 4. [illegible] $\mathcal{A}$ [illegible] Round$_k$ [illegible] Async+$\mathcal{G}_k$ [illegible]

4 Efficiency of Transformations

In the previous section we showed that from a solvability point of view, all four models are equivalent. The chain of transformations of a ParSync algorithm to a ◇Sync algorithm to a Round algorithm can be done with bounded transformation, i.e., the transformation is [illegible]. That means, e.g., if there is already an efficient algorithm for the ParSync model, the transformed algorithm is also efficient in the ◇Sync and the Round model. On the other hand, in this section we show that there is no transformation that maintains this efficiency for (1) the other transformations in the previous section and (2) for the transformations going backwards in the efficient chain. Note, however, that this does not imply the non-existence of efficient algorithms for these models, but just that these cannot be obtained by a (general) transformation from an efficient algorithm of the other model.

4.1 Lower Bounds

The aim of this section is to show that which transformations cannot be efficiency preserving. The (trivial) idea behind the proof that no transformation ParSync⇀Async+$\mathcal{G}$ can be efficiency preservingis that no message with an unbounded delay can be received in a bounded number of steps.

Theorem 1. [illegible] $\mathcal{A}$ [illegible] ParSync$_k$ [illegible] Async+$\mathcal{G}_k$

We can also show that no transformation in the opposite direction can be efficiency preserving. The main idea is that no reliable suspicion of faulty processes can be made within known time in a system with unknown delays.

Theorem 2. [illegible] $\mathcal{A}$ [illegible] Async+$\mathcal{G}_k$ [illegible] ParSync$_k$

While the transformation of ParSync algorithms to ones for the ◇Sync model is rather simple (recall that by definition every ◇Sync execution is also a ParSync execution) there is no efficiency-preserving transformation for the opposite direction. The reason for this is, informally speaking, that fixed bounds ($\Delta_\diamond$ and $\Phi_\diamond$) have to be ensured in a system where there are only unknown bounds ($\Delta_?$ and $\Phi_?$).

Theorem 3. [illegible] $\mathcal{A}$ [illegible] ◇Sync$_k$ [illegible] ParSync$_k$

Our next lower bound follows from Theorem 1 and Corollary 2: If there was an efficiency-preserving transformation ◇Sync⇀Async+$\mathcal{G}$ these two results would be contradictory.

Theorem 4. [illegible] $\mathcal{A}$ [illegible] ◇Sync [illegible] Async+$\mathcal{G}$

To prove that the transformations considered above are necessarily not efficiency preserving it was sufficient to examine only stabilization shift. Conversely, our proof there is no efficiency-preserving transformation Round⇀◇Sync is based on showing that no transformation can be B-bounded and, at the same time, cause only D-bounded shift.

Theorem 5. [illegible] $\mathcal{A}$ [illegible] Round$_k$ [illegible] ◇Sync$_k$

4.2 Upper Bound on Async+$\mathcal{G}$⇀◇Sync

We use a modified version of Algorithm 1, with the following changes: [illegible] p is initialized to $\Delta_\diamond + (n+1)\Phi_\diamond$ and the last line is omitted. This incorporates that we know $\Delta_\diamond$ and $\Phi_\diamond$ in advance, and thus we do not have to estimate it.

It is clear that the proof of Algorithm 1 analogously applies here and thus this transformation is correct. It is also easy to see that it is also $(n+1)$-bounded:

$$\text{[illegible]} \mapsto \begin{cases} \text{[illegible]} \quad (*, 1) \\ \vdots \\ \text{[illegible]} \quad (*, n) \\ \text{[illegible]} \end{cases}$$

To determine the bound on the stabilization shift, note that since every source s will send a message to p in every Async+$\mathcal{G}$ operation (taking at most $(n+1)\Phi$ ◇Sync steps) and messages are delivered within $\Delta_\diamond$ ◇Sync steps, p will always receive a message from s before updating [illegible]$_p$, thus never suspecting s anymore. Therefore the maximal shift is bounded by $\Delta_\diamond + (n+1)\Phi_\diamond$ ◇Sync steps, and consequently:

Theorem 6. [illegible] $\mathcal{A}$ [illegible] Async+$\mathcal{G}_k$ [illegible] ◇Sync$_k$

5 Discussion

[illegible] Figure 1 presents a graphical overview of our results. It can easily be seen that the directed subgraph consisting of solid arrows is strongly connected.

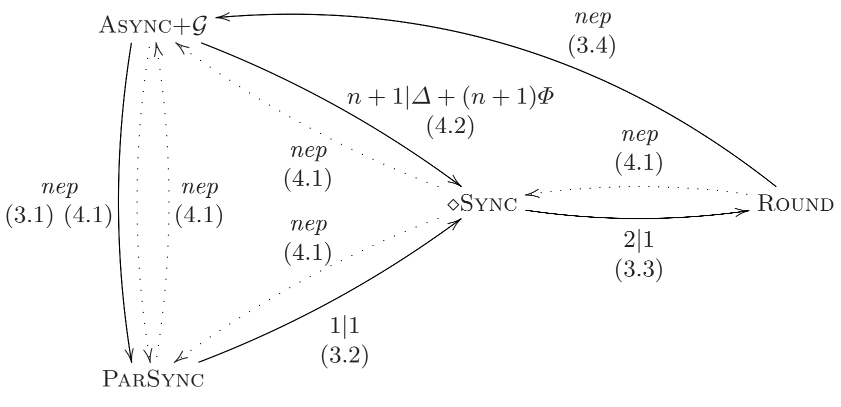


Fig. 1. Relations of models with pointers to sections of this paper; an arrow from model M_1 to model M_2 indicates that a result on algorithm transformations from model M_1 to model M_2 can be found in this paper. Solid lines indicate upper bounds, dotted lines indicate lower bounds. $B|D$ means that a transformation exists which is B-bounded and has D-bounded shift. We use *nep* to mark non-efficiency-preserving transformations.

This subgraph presents the transformation algorithms we have provided in this paper. Thus, all model families presented are equivalent regarding solvability.

[illegible] Setting $k = n - f$ in our models, ROUND$_{n-f}$ is in fact the classic basic round model by Dwork et al. [3] and the asynchronous model with $\mathcal{G}_{n-f}$ is the asynchronous model augmented with the eventually perfect failure detector [4]. We use $k = n - f$ in order to intuitively discuss why these stabilizing models are equivalent from a solvability viewpoint while their perpetual counterparts are not.

The main observation in the relation to [6] is concerned with the term "eventually" in the model definition: In the asynchronous model augmented with $\diamond\mathcal{P}$, two things happen in every execution: (1) eventually, the failure detector becomes accurate, and (2) eventually, the last process crashes and all its messages are received.

The model considered in [6] (asynchrony with $\mathcal{P}$), however shares only (2) while the failure detector taken into account satisfies perpetual strong accuracy. It was shown in [6] that given (2), even perpetual strong accuracy — and thus $\mathcal{P}$ — is too weak to implement a model where every round is communication closed and reliable, as is required by the synchronous model of computation: If a process p crashes, $\mathcal{P}$ does not provide information on the fact whether there are still messages sent by p in transit (cf. [23]).

For showing our equivalence result in the special case of $n - f$, one has to show that communication closed rounds are ensured eventually. We observe that (1) and (2) are sufficient to achieve this. After (1) and (2) hold, all processes are correct, they will never be suspected and thus all their messages are received in the round they were sent. Thus we achieve communication closed rounds eventually which is equivalent to eventual lock-step and thus eventual synchrony.

[illegible] Our results show the equivalence of diverse models with stabilizing properties in which the properties that are guaranteed to hold have the same spatial distribution. Informally, for the models we present, it is equivalent if a source is defined via timing bounds, restrictions on the rounds its messages are received, or whether the "source" has just the property that it is not suspected by any process. Consequently, we conjecture that similar results hold for other timing assumptions as the FAR model [24] or models where the function in which timing delays eventually increase is known given that the spatial distribution of timing properties is the same as in our paper.

[illegible] An interesting consequence of our results concerns models where only t links of the sources are eventually timely [10]. In models stronger than the asynchronous one where at least one such source exists, Ω and thus $\diamond\mathcal{S}$ can be implemented. As $\diamond\mathcal{S}$ is equivalent to $\mathcal{G}_1$, our results reveal that one can transform any algorithm which works in PARSYNC$_1$ (one source with n timely links) to an algorithm which works in a partially synchronous systems with a source with $t < n$ timely links. Consequently, although the number of timely links was reduced in the model assumptions, the set of solvable problems remained the same.

[illegible] An algorithm solving some problem in a stabilizing model is efficient, if it decides (i.e., terminates) after a bounded number of steps after stabilization. Consider some efficient algorithm $\mathcal{A}$ working in model M. If there exists an efficiency-preserving transformation from M into some model M', this implicates that there exists an efficient algorithm in M' as well (the resulting algorithm when $\mathcal{A}$ is transformed). The converse, however, is not necessarily true. The dotted lines with a [illegible] label in Figure 1 show that no efficiency-preserving transformations exist. Consequently, by means of transformations (which are general in that they have to transform [illegible] algorithms) nothing can be said about the existence of an efficient solution in the absence of an efficiency-preserving transformation from M to M'. As an example, consider ROUND_{n-f} and $\diamond\text{SYNC}_{n-f}$ for which efficient consensus algorithms were given in [25] and [26], respectively. Despite this fact, our results show that there cannot be a (general) efficiency-preserving transformation from ROUND to ◇SYNC algorithms.

References

1. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32, 374–382 (1985)
2. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM 34, 77–97 (1987)
3. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM 35, 288–323 (1988)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43, 225–267 (1996)
5. Greve, F., Tixeuil, S.: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: DSN 2007. Dependable Systems and Networks, pp. 82–91 (2007)
6. Charron-Bost, B., Guerraoui, R., Schiper, A.: Synchronous system and perfect failure detector: solveability and efficiency issues. In: Proceedings of the International Conference on Dependable System and Networks, IEEE Computer Society Press, Los Alamitos (2000)
7. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43, 685–722 (1996)
8. Keidar, I., Rajsbaum, S.: Open questions on consensus performance in well-behaved runs. In: Schiper, A., Shvartsman, A.A., Weatherspoon, H., Zhao, B.Y. (eds.) Future Directions in Distributed Computing. LNCS, vol. 2584, pp. 35–39. Springer, Heidelberg (2003)
9. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing Omega with weak reliability and synchrony assumptions. In: Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York (2003)
10. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, pp. 328–337. ACM Press, New York (2004)
11. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 108–122. Springer, Heidelberg (2001)

12. Malkhi, D., Oprea, F., Zhou, L.: Ω meets paxos: Leader election and stability without eventual timely links. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 199–213. Springer, Heidelberg (2005)
13. Hutle, M., Malkhi, D., Schmid, U., Zhou, L.: Brief announcement: Chasing the weakest system model for implementing Ω and consensus. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 576–577. Springer, Heidelberg (2006)
14. Attiya, H., Welch, J.: Distributed Computing, 2nd edn. John Wiley & Sons, Chichester (2004)
15. Lynch, N.: Distributed Algorithms. Morgan Kaufman Publishers, San Francisco (1996)
16. Hutle, M., Schiper, A.: Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In: DSN 2007. Dependable Systems and Networks, pp. 92–101 (2007)
17. Biely, M., Widder, J.: Optimal message-driven implementation of Omega with mute processes. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 110–121. Springer, Heidelberg (2006)
18. Basu, A., Charron-Bost, B., Toueg, S.: Simulating reliable links with unreliable links in the presence of process crashes. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 105–122. Springer, Heidelberg (1996)
19. Mattern, F.: On the relativistic structure of logical time in distributed systems (1992)
20. Santoro, N., Widmayer, P.: Time is not a healer. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 304–313. Springer, Heidelberg (1989)
21. Charron-Bost, B., Schiper, A.: The heard-of model: Unifying all benign failures. Technical Report LSR-REPORT-2006-004, EPFL (2006)
22. Biely, M., Hutle, M., Penso, L.D., Widder, J.: Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. Technical Report 54/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstrasse 3, 2nd floor, 1040 Wien, E.U. (August 2007)
23. Gärtner, F.C., Pleisch, S.: Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 280–294. Springer, Heidelberg (2002)
24. Fetzer, C., Schmid, U., Süßkraut, M.: On the possibility of consensus in asynchronous systems with finite average response times. In: ICDCS 2005. Proceedings of the 25th International Conference on Distributed Computing Systems, pp. 271–280. IEEE Computer Society Press, Los Alamitos (2005)
25. Dutta, P., Guerraoui, R., Keidar, I.: The overhead of consensus failure recovery. Distributed Computing 19, 373–386 (2007)
26. Dutta, P., Guerraoui, R., Lamport, L.: How fast can eventual synchrony lead to consensus? In: Proceedings of the 2005 International Conference on Dependable Systems and Networks, pp. 22–27 (2005)

Distributed Synthesis of Fault-Tolerant Programs in the High Atomicity Model*

Borzoo Bonakdarpour, Sandeep S. Kulkarni, and Fuad Abujarad

Department of Computer Science and Engineering,
Michigan State University,
East Lansing, MI 48824, USA
{borzoo, sandeep, abujarad}@cse.msu.edu
http://www.cse.msu.edu/

Abstract. In this paper, we concentrate on distributed algorithms for automated synthesis of fault-tolerant programs in the high atomicity model, where all processes can read and write all program variables in one atomic step. Although there has recently been an increasing interest in using parallel and distributed techniques in the model checking community, these technique have not been investigated in program synthesis. Developing such techniques is crucial as a means to cope with the state explosion problem in the context of program synthesis and transformation as well. We propose two distributed multithreaded algorithms for adding two levels of fault-tolerance, namely *failsafe* and *masking*, to existing fault-intolerant programs whose state space is distributed over a network or cluster of workstations.

Keywords: Program transformation, Program synthesis, Distributed algorithms, Fault-tolerance, Parallel synthesis.

1 Introduction

Automated program synthesis is the problem of designing an algorithmic method to find a program that satisfies a set of required behaviors. Such automated method is desirable, as it ensures that the synthesized program is *correct-by-construction*. Similar to verification algorithms, synthesis algorithms often suffer from two factors of time and space complexity. In order to overcome the time complexity problem, several approaches have been proposed in the literature to incrementally *add* properties to existing programs [1,2,3,4,5,6,7]. These approaches (called *local redesign*) make it possible to start from an existing program and, hence, *reusing* the previous efforts made for synthesizing them effectively. As opposed to local redesign, the traditional synthesis algorithms (called *comprehensive redesign*) [8,9] start from specification. Hence, for adding a newly identified property, one should synthesize a new program by starting from the conjunction of the new property and the existing properties from scratch.

* This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

In order to overcome the space explosion problem, recently, an increasing interest in parallel and distributed techniques has emerged in the model checking community (e.g., [10,11,12,13,14]). Such techniques parallelize *enumerative* or *symbolic* state space of a given model over a network or cluster of workstations and run a distributed verification algorithm over the parallelized state space. On the other hand, the space explosion problem remains unaddressed in the context of automated program synthesis.

With this motivation, in this paper, we concentrate on the problem of designing distributed algorithms for automated program synthesis. More specifically, we parallelize two synthesis algorithms (from [7]) for adding two levels of fault-tolerance, namely failsafe and masking, to existing fault-intolerant programs. Intuitively, in the presence of faults, a *failsafe* fault-tolerant program satisfies only its safety specification, but a *masking* fault-tolerant program satisfies both its safety and liveness specifications. We assume that programs are in the *high atomicity model*, where all processes can read and write all program variables in one atomic step. We note that the aforementioned synthesis algorithms *solely* add fault-tolerance to a fault-intolerant program in the sense that they add no new behaviors to the input program in the *absence* of faults.

Similar to distributed model checking techniques, developing distributed synthesis algorithms consists of two phases: (1) parallelizing the state space over a network of workstations, and (2) designing a distributed algorithm that runs on each partition of the state space. In this paper, we only focus on the second phase. In particular, we assume that parallelization of state space is already done using one of the known enumerative techniques in the literature. Precisely, we use the state space parallelization technique proposed by Garavel, Mateescu, and Smarandache [10] with some modifications tailored for the purpose of synthesis rather than model checking. Although there exist more efficient ways for parallel construction of state space (e.g., using abstract interpretation), we cannot trivially apply them as a means for synthesizing programs. This is due to the fact that in synthesis (unlike model checking), we usually require full information about the program being synthesized, as we need to manipulate a program by removing or adding computations. Thus, we conservatively choose to develop distributed algorithms that run over the detailed parallelized enumerative state space.

Since the essence of the proposed algorithm in [7] for synthesizing failsafe fault-tolerant programs is calculating fixpoint of formulas, in this paper, we propose a distributed multithreaded algorithm for calculating smallest and largest fixpoints. Furthermore, since a masking fault-tolerant program recovers to its normal behavior after the occurrence of faults, we also propose a distributed algorithm for synthesizing recovery paths.

Contributions of the paper. The main results of this paper are as follows. We propose (i) a distributed multithreaded synthesis algorithm for adding failsafe fault-tolerance, and (ii) a distributed multithreaded synthesis algorithm for adding masking fault-tolerance to existing fault-intolerant programs. These algorithms involve designing distributed techniques for fixpoint calculations and adding recovery computations to a program. To the best of our knowledge, this paper is the first work that addresses challenges and proposes solutions for designing distributed algorithms in the context of program synthesis and transformation. We believe that this study paves the way for further research on designing distributed synthesis algorithms.

Organization of the paper. In Section 2, we present the preliminary concepts. In Section 3, we formally state the problem of addition of fault-tolerance to existing fault-intolerant programs. Then, we present our distributed synthesis algorithms for adding failsafe and masking fault-tolerance in Section 4. Finally, we make the concluding remarks and discuss future work in Section 5.

2 Preliminaries

In this section, we present formal definitions of programs, specifications, faults, and fault-tolerance. We specify *programs* in terms of their state space and their transitions. The definition of *specifications* is adapted from Alpern and Schneider [15]. The definition of *faults* and *fault-tolerance* is adapted from Arora and Gouda [16].

2.1 Programs and Specifications

A *program* p is specified by a tuple $\langle S_p, \delta_p \rangle$, where S_p is the finite *state space* of p and δ_p is a finite set of *transitions* (i.e., a subset of $S_p \times S_p$). A sequence of states, $\sigma = \langle s_0, s_1, ... \rangle$, is a *computation* of p where $s_i \in S_p$ for all $i \in \mathbb{Z}_{\geq 0}$ iff the following two conditions are satisfied: (1) if σ is infinite then $\forall j > 0 : (s_{j-1}, s_j) \in \delta_p$, and (2) if σ is finite and terminates in state s_n then there does not exist state s such that $(s_n, s) \in \delta_p$, and the condition $\forall j \mid 0 < j \leq n : (s_{j-1}, s_j) \in \delta_p$ holds. A *computation prefix* of p is a finite sequence of states $\langle s_0, s_1, ..., s_k \rangle$, where k is a positive integer and $\forall j \mid 0 < j \leq k : (s_{j-1}, s_j) \in \delta_p$. Note that when it is clear from the context, we use p and δ_p interchangeably.

A *state predicate* of p is any subset of S_p. A state predicate S is *closed* in program p iff $\forall (s_0, s_1) \in p \ : \ (s_0 \in S \Rightarrow s_1 \in S)$. The *projection* of program p on a state predicate S (denoted $p \mid S$) consists of transitions $\{(s_0, s_1) \mid (s_0, s_1) \in p \ \wedge \ s_0, s_1 \in S\}$, i.e., transitions of p that start in S and end in S.

A *specification* Σ is a set of infinite sequences of states. Given a program p, a state predicate S, and a specification Σ, we say that p *satisfies* Σ *from* S (denoted $p \models_S \Sigma$) iff (1) S is closed in p, and (2) every computation of p that starts in a state where S is true is in Σ. If $p \models_S \Sigma$ and $S \neq \{\}$, we say that S is an *invariant* of p for Σ.

We say that a finite computation α *maintains* Σ iff there exists a computation suffix β such that $\alpha\beta$ is in Σ. We say that program p *maintains* (does not *violate*) Σ from S iff (1) S is closed in p, and (2) every computation prefix α of p that starts in a state in S maintains Σ. Note that the definition of *maintains* focuses on finite sequences of states, whereas the definition of *satisfies* characterizes infinite sequences of states.

Notation. Whenever the specification is clear from the context, we will omit it; thus, "S is an invariant of p" abbreviates "S is an invariant of p for Σ".

In this paper, we only consider suffix-closed and fusion-closed specifications. *Suffix closure* of a set means that if a state sequence σ is in that set then so are all the suffixes of σ. *Fusion closure* of a set means that if state sequences $\langle \alpha, s, \gamma \rangle$ and $\langle \beta, s, \delta \rangle$ are in that set then so are the state sequences $\langle \alpha, s, \delta \rangle$ and $\langle \beta, s, \gamma \rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state. Intuitively, fusion closure of the specification means that an implementation of the specification must execute its next transition only based on its current state, i.e., the history of a computation does not affect the next move of the program.

Furthermore, following Alpern and Schneider [15], we let the specification be a conjunction of a *safety specification* and a *liveness specification*. For a suffix-closed and fusion-closed specification, the safety specification can be represented as a set Σ_{bt} of *bad transitions* [17] that must not occur in program computations (i.e., the safety specification is a subset of $S_p \times S_p$). Now, let Σ be a specification. Throughout the paper, we let $\Sigma_{\overline{bt}}$ be the specification whose computations, say $\sigma = \langle s_0, s_1, \cdots \rangle$, is in Σ and for all $i \geq 0$, $(s_i, s_{i+1}) \notin \Sigma_{bt}$, i.e., the specification in which safety is never violated. In our algorithms, we do not explicitly specify the liveness specification; the transformed fault-tolerant program satisfies the liveness specification iff the input fault-intolerant program satisfies the liveness specification.

2.2 Faults and Fault-Tolerance

The *faults* that a program p is subject to are systematically represented by a set f of transitions, i.e., a subset of $S_p \times S_p$ where S_p is the state space of p. A sequence of states $\langle s_0, s_1, ... \rangle$ is a *computation of p in the presence of f* iff (1) $\forall j > 0 \; : \; ((s_{j-1}, s_j) \in \delta_p \cup f)$, (2) if the sequence is finite and terminates in s_l then there exists no program transition originating at s_l, and (3) $\exists n \geq 0 : (\forall j > n : (s_{j-1}, s_j) \in \delta_p)$. We note that the last condition (bounded fault model) is only necessary for *masking* fault-tolerance (defined below) where recovery to the invariant is required. This constraint is not necessary for *failsafe* fault-tolerance.

We use $p[]f$ to denote the transitions obtained by taking the union of the transitions in p and the transitions in f. We say that a state predicate T is an f-span (read as *fault-span*) of p from S iff the following two conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $p[]f$. Observe that for all computations of p that start at states where S is true, T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the transitions in f.

We now describe what we mean by *levels of fault-tolerance*. We identify the fault-tolerance level of a program based on its behavior in the presence of faults. We say that p is *failsafe* f-tolerant to Σ_{bt} from S iff (i) $p \models_S \Sigma_{\overline{bt}}$, and (ii) there exists a state predicate T such that T is an f-span of p from S and $p[]f$ maintains $\Sigma_{\overline{bt}}$ from T. We say that p is *masking* f-tolerant to Σ_{bt} from S iff (i) $p \models_S \Sigma_{\overline{bt}}$, and (ii) there exists a state predicate T such that (1) T is an f-span of p from S, (2) $p[]f$ maintains $\Sigma_{\overline{bt}}$ from T, and (3) every computation of $p[]f$ that starts from a state in T has a state in S.

3 Problem Statement

In this section, we reiterate the problem statement from [7]. However, it is important to note that in this paper, we solve the same problem in a distributed fashion. Given are a program p with invariant S, a set of faults f, and safety specification Σ_{bt} such that $p \models_S \Sigma_{\overline{bt}}$. Our goal is to find a program p' with invariant S' such that p' is f-tolerant to Σ_{bt} from S'.

Our synthesis methods obtain p' from p by adding fault-tolerance alone to p, i.e., p does not introduce new behaviors to p when no faults have occurred. Observe that if S' (respectively, $p' \mid S'$) contains states (respectively, transitions) that are not in S (respectively, $p' \mid S$) then, in the absence of faults, p' may include computations that start outside S (respectively, $p' \mid S$). Since we require that $p' \models_{S'} \Sigma_{\overline{bt}}$, it would imply that p' is using a new way to satisfy $\Sigma_{\overline{bt}}$ in the absence of faults. Therefore, we require that $S' \subseteq S$ and $(p' \mid S') \subseteq (p \mid S')$. Thus, the synthesis problem is as follows (we instantiate this problem for failsafe and masking f-tolerance in the obvious way):

Problem Statement 3.1. Given p, S, f, and Σ_{bt} such that $p \models_S \Sigma_{\overline{bt}}$. Identify p' and S' such that:

$(C1)$ $S^{'} \subseteq S$,
$(C2)$ $(p' \mid S') \subseteq (p \mid S')$, and
$(C3)$ p' is f-tolerant to Σ_{bt} from S'. □

4 Distributed Automated Addition of Fault-Tolerance

In this section, we present our distributed algorithms for adding fault-tolerance to existing fault-intolerant programs. Similar to distributed model checking techniques, developing distributed synthesis algorithms consists of two phases: (1) parallelizing the state space over a network of workstations, and (2) designing a distributed algorithm that runs on each portion of the state space. In this paper, we only focus on the second phase. In particular, we assume that parallelization of state space is already done using the construction technique due to Garavel, Mateescu, and Smarandache [10]. However, we make some modifications tailored for the purpose of synthesis rather than model checking.

4.1 Parallel Construction of State Space

In order to represent a program p with state space S_p and invariant S on N machines (numbered from 0 to $N-1$), we use the notion of *partitioned* programs. More specifically, the state space S_p is partitioned to $S_p^0 \cdots S_p^{N-1}$, where $S_p = \cup_{i=0}^{N-1} S_p^i$ and $S_p^i \cap S_p^j = \{\}$ for all $0 \leq i \neq j < N$ (i.e., the state space is partitioned into N classes, one class per machine). Likewise, state predicates are partitioned in the same fashion. For instance, machine i contains S^i and T^i partitions of the invariant S and the fault-span T. From now on, we call S the *global invariant* and each S^i the *local invariant* with respect to machine i. The same concept applies to any other state predicate such as the fault-span T, i.e., T is the global fault-span and T^i is the local fault-span with respect to machine i.

The set p of transitions is partitioned to $p^0 \cdots p^{N-1}$, where $p = \cup_{i=0}^{N-1} p^i$, and $(s_0, s_1) \in p^i$ iff $(s_0 \in S_p^i \vee s_1 \in S_p^i)$ for all $0 \leq i < N$ (i.e., if the source and target of a transition belong to different machines, the transition is stored in both the source and target machines). We call such transitions *cross transitions.* Likewise, f and Σ_{bt} are partitioned in the same fashion. From now on, we call p the *global set of program transitions* and each p^i the *local set of program transitions* with respect to machine i. The same concept applies to any other set of transitions such as the set of faults f and the set of bad transitions Σ_{bt}.

Remark 4.1. We choose to store cross transitions in both source and target machines due to two reasons: (1) as we shall see in Subsections 4.2 and 4.3, such duplication decreases the number of potential broadcast messages considerably, and (2) it allows us to efficiently do both forward and backward reachability analysis at the same time. In fact, this deviation from distributed model checking techniques is due to the nature synthesis as compared to verification.

Assumption 4.2. In our synthesis algorithms, we assume that the input fault-intolerant program is already partitioned over a network using a reasonable static partition function $h : S_p \to [0, N-1]$ using the above parallelization method. In other words, machine i contains a state s iff $h(s) = i$. We also assume that all the synthesis processes over the network have a replica of h.

Revised problem statement. With this setting, we revise the Problem Statement 3.1 as follows. Given are a partition function h, a partitioned program $p^0 \cdots p^{N-1}$ with state space $S_p^0 \cdots S_p^{N-1}$, local invariants $S^0 \cdots S^{N-1}$, a partitioned class of faults $f^0 \cdots f^{N-1}$, and safety specification $\Sigma_{bt}^0 \cdots \Sigma_{bt}^{N-1}$ such that $p \models_S \Sigma_{\overline{bt}}$. Our goal is to design distributed algorithms that synthesize a program p' with invariant S' such that p' is failsafe/masking f-tolerant to Σ_{bt} from S'.

4.2 Distributed Addition of Failsafe Fault-Tolerance

In order to synthesize a failsafe fault-tolerant program, we transform p into p' such that transitions of Σ_{bt} occur in no computation prefixes of p'. Towards this end, we parallelize the proposed centralized algorithm in [7] for adding failsafe fault-tolerance.

Algorithm sketch. The essence of adding failsafe fault-tolerance consists of two parts: (1) a smallest fixpoint calculation for identifying the set of states from where safety may be violated, and (2) a largest fixpoint calculation for computing the invariant of the failsafe program. Our algorithm consists of a set of distributed processes each running on one machine across the network. Each process consists of two threads, namely, Distributed_Add_failsafe (cf. Figure 1)

```
 thread Distributed_Add_failsafe($p^i, f^i, \Sigma_{bt}^i$ : set of transitions,
                 $S_p^i, S^i$ : state predicate, $N$: int, $h$: partition function, $bLeader^i$: Boolean)
{
     $cbSnt^i$, $cbRcvd^i := 0$; $ns^i := \{\}$;                                        (1)
     $ms^i := \{s_0 \mid \exists s_1 \in S_p^i : (s_0, s_1) \in f^i \ \wedge \ (s_0, s_1) \in \Sigma_{bt}^i\}$;     (2)
     $ms^i :=$ FindLocalUnsafeStates($S_p^i, ms^i, f^i$);                               (3)
→    BlkReceive (Trm_dtct); $cbSnt^i$, $cbRcvd^i := 0$;                                 (4)
     $mt^i := \{(s_0, s_1) \mid s_1 \in (ms^i \cup ns^i) \ \vee \ (s_0, s_1) \in \Sigma_{bt}^i\}$;     (5)
     $S^i := S^i - ms^i$; $p^i := p^i - mt^i$;                                          (6)
     $S'^i, p'^i :=$ RemoveLocalDeadlocks($S^i, p^i$);                                  (7)
→    BlkReceive (Trm_dtct);                                                             (8)
     if ($S'^i \neq \{\}$) then return $p' := \cup_{i=0}^{N-1} p'^i$, $S' := \cup_{i=0}^{N-1} S'^i$;   (9)
     elseif ($bLeader^i$) then Send($(i+1) \mod N$, Empt_inv(0));                       (10)
}
procedure FindLocalUnsafeStates($S_p^i, ms^i$: state predicate, $f^i$: set of transitions)
// Returns the set of states from where safety may be violated by faults alone
{
     while ($\exists s_0, s_1 : (s_1 \in ms^i \ \wedge \ (s_0, s_1) \in f^i)$)                (11)
          if $h(s_0) = i$ then $ms^i := ms^i \cup \{s_0\}$;                             (12)
          else Send($h(s_0)$, New_ms($s_0, s_1$)); $cbSnt^i := cbSnt^i + 1$;            (13)
     return $ms^i$;                                                                     (14)
}
procedure RemoveLocalDeadlocks($S^i$ : state predicate, $p^i$ : set of transitions)
// Returns the largest subset of $S^i$ s.t. computations of $p$ within that subset are infinite
{
     while ($\exists s_1 \in S^i : (\forall s_2 \mid (\exists s_0 \mid (s_0, s_2) \in p^i) \ : \ (s_1, s_2) \notin p^i)$)   (15)
          $S^i := S^i - \{s_1\}$;                                                      (16)
          $p^i :=$ EnsureClosure($p^i, S^i, s_1$);                                      (17)
     return $S^i, p^i$                                                                  (18)
}
procedure EnsureClosure($p^i$: set of transitions, $S^i$: state predicate, $s_1$: state)
{
     while ($\exists s_0 \ : \ ((s_0, s_1) \in p^i \ \wedge \ h(s_0) \neq i)$)                (19)
          Send($h(s_0)$, New_ds($s_0, s_1$)); $cbSnt^i := cbSnt^i + 1$;                 (20)
          $p^i := p^i - \{(s_0, s_1)\}$;                                                (21)
     return $p^i - \{(s_0, s_1) \mid s_0 \in S^i\}$                                     (22)
}
```

Fig. 1. Distributed algorithm for adding failsafe fault-tolerance

and MessageHandler (cf. Figure 2). Briefly, the thread Distributed_Add_failsafe is in charge of initiating local fixpoint calculations and managing synchronization points of the algorithm. The thread MessageHandler is responsible for handling messages sent by other synthesis processes across the network and invoking appropriate procedures. The thread Distributed_Add_failsafe consists of three main parts, namely, Lines 1-4 which is a smallest fixpoint computation, Lines 5-8 which is a largest fixpoint computation, and Lines 9-10 where we check the emptiness of the synthesized program (to declare failure or success). It also invokes three procedures, namely, FindLocalUnsafeStates, RemoveLocalDeadlocks, and EnsureClosure.

Assumption 4.3. Throughout the paper, we assume that procedure invocations are *atomic*.

We now describe our algorithm in detail. First, the thread Distributed_Add_failsafe finds the set ms^i of states from where a single fault transition violates the safety

```
thread MessageHandler()
{
   $msg$ := RECEIVE();
   case $msg$ is
      New_ms$_j(s_0, s_1)$:     $ms^i := ms^i \cup \{s_0\}$; $ns^i := ns^i \cup \{s_1\}$; $cbRcvd^i := cbRcvd^i + 1$;   (1)
                                $ms^i$ := FindLocalUnsafeStates($S_p^i, ms^i, f^i$);                                 (2)
      New_ds$_j(s_0, s_1)$:     $p^i := p^i - \{(s_0, s_1)\}$; $cbRcvd^i := cbRcvd^i + 1$;                            (3)
                                $S'^i, p'^i$ := RemoveLocalDeadlocks($S^i, p^i$);                                    (4)
      Empt_inv$_j(k)$:          if $(\neg bLeader \ \wedge \ S^i = \{\})$ then
                                    SEND($(i+1)$ mod $N$, Empt_inv$(k+1)$); return $\{\}$;                          (5)
                                elseif $(\neg bLeader \ \wedge \ S^i \neq \{\})$ then
                                    SEND($(i+1)$ mod $N$, Empt_inv$(k)$);                                           (6)
                                    return $p'^i$, $S'^i$;                                                          (7)
                                if $(bLeader \ \wedge \ (k = N-1))$ then
                                    declare no failsafe program $p'$ exists;                                        (8)
                                    exit;                                                                           (9)
                                else return $p' := \cup_{i=0}^{N-1} p'^i, S' := \cup_{i=0}^{N-1} S'^i$;             (10)
      Report_rcv$_j(k)$:        if $(\neg bLeader^i)$ then
                                    SEND($(i+1)$ mod $N$, Report_rcv$(k + cbRcvd^i)$);                              (11)
                                else $nbTotal := k$;                                                                (12)
                                    SEND($(i+1)$ mod $N$, Report_snd$(cbSnt)^i$);                                   (13)
      Report_snd$_j(k)$:        if $(\neg bLeader^i)$ then
                                    SEND($(i+1)$ mod $N$, Report_snd$(k + cbSnt^i)$);                               (14)
                                elseif $(nbTotal = k)$ then
                                    SEND($[(i+1)$ mod $N..(i+N-1)$ mod $N]$, Trm_dtct);                            (15)
      New_fs$(s_0)$:            $T_1^i := T_1^i - \{s_0\}$; $cbRcvd^i := cbRcvd^i + 1$;                              (16)
                                $T_1^i$ :=ConstructLocalFaultSpan($T_1^i, T_2^i - T_1^i, f^i$);                      (17)
      Search_path$_j(X)$:       $r^i := \{\}$; $cbRcvd^i := cbRcvd^i + 1$;                                          (18)
                                For each $s_0 \in X$ :
                                    if $(\exists s_1 \ : \ (\text{Rank}(s_1) \neq \infty)) \ \wedge (s_0, s_1) \notin mt^i)$ then
                                        $r^i := r^i \cup \{(s_0, s_1, \text{Rank}(s_1) + 1)\}$;                     (19)
                                SEND($j$, New_path$(r^i)$); $cbSnt^i := cbSnt^i + 1$;                                (20)
      New_path$_j(r^i)$:        $q^i := \{\}$; $cbRcvd^i := cbRcvd^i + 1$;                                          (21)
                                For each $(s_0, s_1, a)$ s.t. $((s_0, s_1, a) \in r^i \ \wedge \ s_0 \in (T_2^i - T_1^i)$:
                                    if $(s_0, s_1) \notin p^i$ then
                                        $p^i := p^i \cup (s_0, s_1)$; $q^i := q^i \cup (s_0, s_1)$;                 (22)
                                        Rank$(s_0) := a$;                                                           (23)
                                        $T_1^i, T_2^i := T_1^i \cup \{s_0\}, T_2^i - \{s_0\}$;                      (24)
                                        $p_1^i, T_1^i$ := ConstructLocalRecoveryPaths($S_1^i, T_2^i, p_1^i, mt^i$); (25)
                                        SEND($j$, Confirm_trns$(q^i)$); $cbSnt^i := cbSnt^i + 1$;                   (26)
      Confirm_trns$_j(q^i)$:    $p^i := p^i \cup q^i$; $cbRcvd^i := cbRcvd^i + 1$;                                  (27)
                                SEND($j$, Commit); $cbSnt^i := cbSnt^i + 1$;                                        (28)
      Commit$_j$:               $cbRcvd^i := cbRcvd^i + 1$;                                                         (29)
                                Wait to receive Commit message from all providers;                                  (30)
                                SEND($i+1$ mod $N$, Token); $cbSnt^i := cbSnt^i + 1$;                               (31)
      Token$_j$:                $cbRcvd^i := cbRcvd^i + 1$;                                                         (32)
                                SEND($[(i+1)$ mod $N..(i+N-1)$ mod $N]$,
                                    Search_path$(T_2^i - T_1^i)$); $cbSnt^i := cbSnt^i + 1$;                        (33)
}
```

Fig. 2. The message handler thread

(Line 2). Next, we invoke the procedure FindLocalUnsafeStates where we find the set of states from where faults alone may violate the safety (Line 3). We find this set by calculating the smallest fixpoint of backward reachable states, given the initial set ms^i (Lines 11-12). In this calculation, if we find a fault transition, say (s_0, s_1), where $s_1 \in ms^i$, but s_0 resides in a machine other than i (i.e., $h(s_0) \neq i$), we send a New_ms message to process $h(s_0)$ indicating that s_0 is a state from where faults alone may violate the safety specification (Line 13).

Notation: At the receiver's side, we denote messages by $msg_j(params)$, where msg is the name of message, j is the sender process, and $params$ is a list of parameters sent along with the message. All messages (except `Trm_dtct`) are handled in the thread MessageHandler. At the sender's side, we omit the sender's subscript.

The receiver of a `New_ms` message (cf. Lines 1-2 in Figure 2) adds s_0 to its local ms^i (Line 1) and invokes the procedure FindLocalUnsafeStates (Line 2) so that by taking s_0 into account, new states from where faults alone may violate the safety specification are explored. The set ns^i consists of states that are in ms^j. Notice that every time a process sends (respectively, receives) such messages, it increments the variable $cbSnt^i$ (respectively, $cbRcvd^i$). We shall use these variables for termination detection as a means to synchronize processes at certain points.

The next phase of the algorithm is removing the states of global ms from the global invariant. To this end, we need to have a synchronization mechanism to ensure that calculation of ms^i is completed for all $i \in [0..N-1]$. In particular, we use the termination detection technique proposed by Mattern [18]. More specifically, in Line 4, the thread Distributed_Add_failsafe waits to receive a `Trm_dtct` message indicating that all processes are finished by calculating their local ms^i and all communication channels are empty. The arrows ($\rightarrow$) in Figure 1 mark the synchronization barriers. We will describe the termination detection technique later in this subsection.

After calculating the global set ms, we remove this set from the invariant to ensure that no computation of p' that starts from a state in S' violates the safety specification. We also remove the transitions of the set mt^i from p^i, where mt^i consists of transitions whose target states are in ms^i or directly violate the safety specification (Line 6). Notice that this removal may create *deadlock* states (i.e., states from where there exist no outgoing transitions). Thus, the thread Distributed_Add_failsafe invokes the procedure RemoveLocalDeadlocks (Line 7) to remove deadlock states which is in turn calculating the largest fixpoint of backward reachable states, given the initial set S^i. In other words, it keeps removing deadlock states until it reaches a fixpoint (Lines 15-16). In this calculation, since removal of a deadlock state, say s_1, may create transitions, say (s_0, s_1), such that (s_0, s_1) violates the closure of invariant, we invoke the procedure Ensure-Closure (Line 17) to ensure that no such transitions exist in the final synthesized program. Furthermore, if we encounter a program transition, say (s_0, s_1), where s_1 is a deadlock state and s_0 resides in a machine other than i (i.e., $h(s_0) \neq i$), then we send a `New_ds` message to process $h(s_0)$ indicating that s_0 *might* be a deadlock state (Line 20). Upon receipt of such a message (cf. Line 3 in Figure 2), the receiver removes the transition (s_0, s_1) to maintain consistency of transitions and then invokes the procedure RemoveLocalDeadlocks (cf. Line 4 in Figure 2) to remove possible new deadlock states due to removal of (s_0, s_1). Similar to the calculation of ms, our algorithm ensures completion of calculation of the largest fixpoint S' using the same termination detection technique (Line 8 in Figure 1).

At this point, each process has synthesized a local set of program transitions p'^i with a local invariant S'^i. The union of these portions is the final synthesized program, i.e., $p' = \cup_{i=0}^{N-1} p'^i$ and $S' = \cup_{i=0}^{N-1} S'^i$. However, since invariant predicates cannot be empty, if S' turns out to be equal to the empty set, the algorithm declares failure. To test the emptiness of S', a pre-specified *leader* process identified by the variable *bLeader* initiates an emptiness polling of the global invariant S' as follows. For this polling (and also termination detection), we consider a unidirectional virtual ring which connects every machine i to its successor machine $(i+1) \mod N$. Note that this virtual ring is independent of the fully connected topology of the network. Now, if the local invariant of the leader is equal to the empty set then it sends an `Empt_inv`(0) message to its first neighbor on the virtual ring (process $(i+1) \mod N$) indicating that its own local invariant is equal to the empty set (cf. Line 10 in Figure 1). If the local invariant of the $((i+1) \mod N)^{\text{th}}$ process is equal to the empty set as well, it increments the value of k (the integer received along with the message `Empt_inv`) by one and sends the same message to the next process on the ring (cf. Line 5 in Figure 2). Otherwise, it does not change the value of k and sends an `Empt_inv`(k) message to the next process (Line 6). Upon the completion of one round of sending the `Empt_inv` messages, the leader finally finds out whether the global invariant S' is equal to the empty set or not (Lines 8-10). If the global invariant S' is indeed equal to the empty set then the leader declares failure (Line 8). Otherwise, it calculates and returns p' and S' (Line 10). Notice that Lines 9 and 10 in Figures 1 and 2 respectively *describes* that the output of the distributed algorithm is indeed a program which is the union of all local sets of transitions and local invariants.

Termination detection. In order to detect the termination of the fixpoint calculations, we use a virtual ring-based algorithm inspired by Mattern [18]. According to the general definition, global termination is reached when all local computations are complete (i.e., each machine i has calculated a local fixpoint) and all communication channels are empty (i.e., all sent transitions have been received). The core of the termination detection algorithm is as follows. Every time the leader process finishes its local fixpoint calculations, it checks whether global termination has been reached by generating two successive waves of `Report_rcv` (respectively, `Report_snd`) messages on the virtual ring to collect the number of messages received (respectively, sent) by all machines. A message `Report_rcv`$_j(k)$ (respectively, `Report_snd`$_j(k)$) received by machine i indicates that k messages have been received (respectively, sent) by the machines from the leader up to $j = (i-1) \mod N$. Each machine i counts the messages it has received and sent using two integer variables $cbRcvd^i$ and $cbSnt^i$, and adds their values to the numbers carried by `Report_rcv` and `Report_snd` messages (Lines 11, 13, and 14). Upon receipt of the `Report_snd`$_j(k)$ message ending the second wave, the leader machine checks whether the total number k of messages sent is equal to the total number *nbTotal* of messages received (the result of the `Report_rcv` wave). If this is the case, it informs the other machines that termination has been reached, by sending a broadcast `Trm_dtct` message. Otherwise, the leader concludes that termination has not been reached yet and will generate a new termination detection wave later (Line 15).

Theorem 4.4. The algorithm Distributed_Add_failsafe is sound and complete.

Performance of parallelized addition of failsafe. Distributing the synthesis algorithm is aimed at reducing the space complexity and time complexity. Of these, similar to the goals for distributed model checking, reducing the space complexity is a higher priority. We expect that our approach would assist in this case. In particular, if N machines are used to perform synthesis then each of them is expected to have $(1/N)^{\text{th}}$ number of states and at most $(2/N)^{\text{th}}$ number of transitions (because a transition may be stored in up to two machines). Regarding time complexity, in each phase, a machine performs some local computation that results a set of queries (e.g., `New_ms`, `New_ds`, etc.) for other machines. Now, consider the role of the two threads Distributed_Add_failsafe and MessageHandler. The thread MessageHandler provides a new list of tasks (received from other machines) that should be performed by Distributed_Add_failsafe. Since Distributed_Add_failsafe begins with a list of tasks (based on its local states and transitions) and MessageHandler continues to provide new tasks based on requests received from others, we expect that the list of tasks that Distributed_Add_failsafe needs to perform will typically be nonempty at all times. In other words, communication cost will not be in the critical path for the synthesis. Therefore, we expect that the distributed synthesis algorithm will be able to provide significant benefits regarding time complexity as well.

4.3 Distributed Addition of Masking Fault-Tolerance

In order to synthesize a masking program, we should generate a program p' with invariant S' and fault-span T', such that p' never violates its safety specification and if faults perturb the state of p' to a state in T', it recovers to S' within a finite number of recovery steps. Similar to the distributed algorithm for adding failsafe fault-tolerance, our algorithm for adding masking fault-tolerance consists of two threads Distributed_Add_masking (cf. Figure 3) and MessageHandler (cf. Figure 2).

Our first estimate of a masking program is a failsafe program. Hence, we let our first estimate S_1^i be the local invariant of its failsafe fault-tolerant program (cf. Line 2 in Figure 3). Likewise, we estimate the local fault-span to be T_1^i where T_1^i includes all the states in the local state space minus the states from where safety of p' may be violated (Line 3). Next, we compute the local set of transitions p_1^i, local fault-span T_1^i, and local invariant S_1^i in a loop (Lines 5-15). This loop consists of three main steps for constructing recovery paths, calculating fault-span, and calculating invariant as follows:

1. In order to compute the local set of transitions p_1^i, we construct recovery paths from each state in the fault-span to a state in the invariant. To this end, we identify two types of recovery paths: (1) recovery paths consist of only local program transitions, and (2) recovery paths consist of both local program transitions as well as cross transitions. We note that since these transitions originate outside the invariant, they do not violate the second

```
thread Distributed_Add_masking($p^i, f^i, \Sigma^i_{bt}$ : set of transitions,
                $S^i_p, S^i$ : state predicate, $N$: int, $h$: partition function, $bLeader^i$: Boolean)
{
    Compute $ms^i$ and $mt^i$ as in Distributed_Add_failsafe;                          (1)
    Let $S^i_1$ be the local invariant of the failsafe version of $p$;                   (2)
    $T^i_1 := S^i_p - ms^i$; $\forall s \in (T^i_1 - S^i_1)$ : Rank$(s) := \infty$;     (3)
    $p^i_1 := p^i$;                                                                       (4)
    repeat
        $T^i_2, S^i_2 := T^i_1, S^i_1$; $cbSnt^i$, $cbRcvd^i := 0$                        (5)
        $p^i_1, T^i_1 :=$ ConstructLocalRecoveryPaths$(S^i_1, T^i_1, p^i, mt^i)$;         (6)
→       BlkReceive (Trm_dtct); $cbSnt^i$, $cbRcvd^i := 0$;                                (7)
        if $(bLeader)$ then
            Send$([(i+1) \mod N..(i+N-1) \mod N]$, Search_path$(T^i_2 - T^i_1)$);         (8)
            $cbSnt^i := cbSnt^i + 1$;                                                     (9)
→       BlkReceive (Trm_dtct); $cbSnt^i$, $cbRcvd^i := 0$;                                (10)
        $T^i_1 :=$ ConstructLocalFaultSpan$(T^i_1, T^i_2 - T^i_1, f^i)$;                  (11)
→       BlkReceive (Trm_dtct); $cbSnt^i$, $cbRcvd^i := 0$;                                (12)
        $S^i_1 :=$ RemoveLocalDeadlocks$(S^i_1 \cap T^i_1, p^i_1)$;                       (13)
→       BlkReceive (Trm_dtct);                                                            (14)
        if $(S^i_1 = \{\} \vee T^i_1 = \{\})$ then break;                                 (15)
    until $(T^i_1 = T^i_2 \wedge S^i_1 = S^i_2)$
→   BlkReceive (Trm_dtct);                                                                (16)
    $T'^i, S'^i := T^i_1, S'^i$;                                                          (17)
    if $(bLeader^i \wedge (S'^i = \{\}))$ then
        Send$((i+1) \mod N$, Empt_inv$(0))$;                                              (18)
    if $(bLeader^i \wedge (T'^i = \{\}))$ then
        Send$((i+1) \mod N$, Empt_fs$(0))$;                                               (19)
}
procedure ConstructLocalRecoveryPaths$(S^i, T^i$ : state predicate, $p^i, mt^i$: set of transitions)
    $X^i_1 := S^i$; $j = 0$; $X^i_2 := \{\}$;                                             (20)
    repeat
        $\forall s \in (X^i_1 - X^i_2)$ : Rank$(s) := j$;                                 (21)
        $X^i_2 := X^i_1$; $j := j + 1$;                                                   (22)
        $r^i := \{(s_0, s_1) \mid s_0 \in (T^i_1 - X^i_1) \wedge s_1 \in X^i_1\} - mt^i$; (23)
        $p^i := p^i \mid S^i \cup r^i$;                                                   (24)
        $X^i_1 := X^i_1 \cup \{s_0 \mid \exists s_1 : (s_0, s_1) \in r_i\}$;              (25)
    until $(X^i_1 = X^i_2)$
    return $p^i, X^i$;                                                                    (26)
}
procedure ConstructLocalFaultSpan$(T^i_1, T^i_2$ : state predicate, $f^i$ : set of transitions)
// Returns the largest subset of $T^i_1$ that is closed in $f$
{
    while $(\exists s_0, s_1 : ((s_0 \in T^i_1) \wedge (s_1 \in T^i_2) \wedge (s_0, s_1) \in f^i))$
        $T^i_1 := T^i_1 - \{s_0\}$;                                                       (27)
    For each $s_1 \in T^i_2$ :
        if $(\exists s_0 : ((s_0, s_1) \in f^i \wedge h(s_0) \neq i))$
            Send$(h(s_0)$, New_fs$(s_0))$; $cbSnt^i := cbSnt^i + 1$;                      (28)
    return $T^i_1$;                                                                       (29)
}
```

Fig. 3. Distributed algorithm for adding masking tolerance

constraint of the problem statement (i.e., in the absence of faults, no new computation is introduced to fault-tolerant program).

Recovery paths through local transitions. The thread Distributed_Add_masking invokes the procedure ConstructLocalRecoveryPaths (Line 6), which identifies layers of states in the local fault-span corresponding to the number of steps of recovery paths, in a loop (Lines 21-25). In the beginning of the loop it assigns a rank to each state which is equal to the number of recovery steps from that state to a state in the local invariant. In this setting, the rank of states in the local invariant are zero. In the first iteration of the loop, we identify the set of states from where one-step recovery to the local invariant is possible while maintaining the safety, i.e., $X_1^i = \{s_0 \mid s_0 \in (T_1^i - S_1^i) \ \wedge \ \exists s_1 \in S_1^i \ : \ (s_0, s_1) \notin mt^i\}$. Thus, we add the transitions, say (s_0, s_1) where $s_0 \in X_1^i$ and $s_1 \in S^i$, to the set of local program transitions. In the second iteration of the loop, we identify the set of states from where two-step recovery is possible. Indeed, this is equivalent to identifying the set of states from where one-step recovery is possible from $T_1^i - X_1^i$ to the set $X_1^i \cup S_1^i$. Continuing thus inductively, we identify layers of states from where multi-step recovery is possible. Finally, we reach a point where we identify the set X_1^i of states from where recovery to the local invariant using local transitions is possible and the set $T_1^i - X_1^i$ of states from where such recovery is not possible.

Recovery paths through cross transitions. After constructing local recovery paths, the leader process initiates a wave of communication among all processes to identify the set of states from where local recovery is not possible, but recovery through cross transitions is possible. More specifically, the leader process sends a `Search_path` message to all other processes (Line 8 in Figure 3). Let us call the process which sends a `Search_path` the *requester* process. Upon receipt of this message along with the set X of states from where local recovery is not possible (Line 18 in Figure 2), each process offers a recovery cross transition, say (s_0, s_1), provided $(s_0, s_1) \notin mt^i$ and there exists a recovery path from s_1 (i.e., Rank$(s_1) \neq \infty$), for each state $s_0 \in X$ (Line 19). Let us call such processes the *providers*. Each provider sends a `New_path` message carrying the set r^i of cross recovery transitions along with the rank of state s_0 to the requester (Line 20). Obviously, if the requester accepts the provider's transitions, the rank of s_0 will be Rank$(s_1) + 1$.

Upon receipt of this message (Line 21 in Figure 2), the requester adds the new recovery cross transitions, say (s_0, s_1), to its set of local program transitions (Line 22) and sets the rank of source states s_0 (Line 23). These states should be added to the local fault-span (Line 24). Next, it invokes the procedure ConstructLocalRecoveryPaths to add new possible local recovery transitions by taking the newly added recovery cross transitions into account (Line 25). Then, it sends a `Confirm_trns` message to the providers of the cross transitions so that the set of cross transitions of providers and the requester processes are consistent (Line 26). Obviously, if the requester receives other offers for a cross transition originated at s_0, say (s_0, s_1) with rank a, where the current rank of s_0 is greater than a, then the requester can replace its current cross transition with (s_0, s_1). However, we do not illustrate such implementation details in the algorithms.

Finally, upon the receipt of a `Confirm_trns` message, the providers add the set q^i of cross transitions (selected by the requester) to their set of local program transitions as well (Line 27). At this point, providers send a

Commit message to the requester (Line 28) indicating that the changes are committed. Upon receipt of Commit message from all providers (Lines 29-30), the requester sends a Token message to the next process on the virtual ring (Line 31) so that it starts identifying the cross recovery transitions in the same fashion (Line 33). We continue doing this until no cross transition is added across the network.

   Notice that in both types of recovery paths, we do not introduce cycles to the fault-span, as we do not add transitions from a state with a lower rank to a state with higher rank. Hence, after occurrence of faults, recovery within a finite number of steps is guaranteed. We synchronize the completion of construction of recovery paths in Line 7.

2. Since there may exist states from where recovery to the invariant is not possible, we need to recompute the local fault-span by removing the states from where closure of fault-span is violated through fault transitions. To this end, we invoke the procedure ConstructFaultspan which is a largest fixpoint calculation (Line 11 in Figure 3) to calculate the largest fault-span which is closed in $p[]f$. Since this removal may cause other states in the local fault-span of other processes to violate the closure of the global fault-span, we send a New_fs message to such processes to indicate this fact (Line 28). Note that in order to synchronize the completion of calculation of local fault-spans, here as well, we need a barrier synchronization (Line 12).
3. Due to the removal of some states in step 2, we recompute the local invariant by invoking the procedure RemoveLocalDeadlocks. Notice that since S_1^i must be a subset of T_1^i, this invocation is parameterized by $S_1^i \cap T_1^i$ (Line 13). At this point, if both S_1^i and T_1^i are nonempty, we jump back to step 1 and we keep repeating the loop until a fixpoint is reached, i.e., $(T_1^i = T_2^i \wedge S_1^i = S_2^i)$.

Upon the termination of the repeat-until loop, recovery without violation of the safety specification from T_1' to S_1' is provided. At this point, if there exist processes i and j such that S'^i and T'^j are both nonempty then we have a solution to the synthesis problem. Thus, similar to addition of failsafe, we run an emptiness poll among the processes (Lines 18-21). To this end, we send a Empt_fs(0), which is similar to Empt_inv, except the message handler tests the emptiness of the local fault-span rather than local invariant. We skip including this message in Figure 2.

We note that, in this paper, we have modified the recovery mechanism of the centralized algorithm in [7]. This is due to the fact that in that algorithm the authors add all possible transitions, i.e., the set $p_1|S_1 \cup \{(s_0, s_1) \mid s_0 \in T_1 - S_1 \;\wedge\; s_1 \in T_1\}$, and then remove non-progress cycles. However, since the size of this set in worst case is in the square order of the size of the state space, it implies that in worst case, each machine i must store a set whose size is in the square order of the state space which obviously does not make sense. Hence, instead of adding all possible transitions and removing cycles, we construct recovery paths in a more space-efficient way in a stepwise manner using the notion of layered fault-span (cf. the Procedure ConstructLocalRecoveryPaths).

Theorem 4.5. The algorithm Distributed_Add_masking is sound. □

5 Conclusion and Future Work

In this paper, we focused on the problem of automated addition of fault-tolerance to existing fault-intolerant programs where the state space of the fault-intolerant program is distributed over a network or cluster of workstations. We addressed this problem in the high atomicity model where all processes of the program are able to read and write all program variables in one atomic step. We presented two distributed multithreaded algorithms for adding failsafe and masking fault-tolerance to a given fault-intolerant program. To this end, we parallelized calculation of smallest and largest fixpoints of a given formula and also addition of safe recovery paths.

As future work, we plan to implement the algorithms proposed in this paper in our tool FTSyn. This implementation will enable us to synthesize fault-tolerant programs with large state space. We also plan to study the problem of designing distributed algorithms for adding fault-tolerance to distributed [6] and real-time [2] programs. We are currently investigating the possibility of reducing the number of synchronization points in our algorithms. Such synchronization barriers decrease the level of parallelism and, hence, efficiency of distributed algorithms.

References

1. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In: ICDCS. IEEE International Conference on Distributed Computing Systems, pp. 3–10 (2007)
2. Bonakdarpour, B., Kulkarni, S.S.: Incremental synthesis of fault-tolerant real-time programs. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 122–136. Springer, Heidelberg (2006)
3. Bonakdarpour, B., Kulkarni, S.S.: Automated incremental synthesis of timed automata. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) FMICS 2006. LNCS, vol. 4346, pp. 261–276. Springer, Heidelberg (2007)
4. Ebnenasir, A., Kulkarni, S.S., Bonakdarpour, B.: Revising UNITY programs: Possibilities and limitations. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 275–290. Springer, Heidelberg (2005)
5. Kulkarni, S.S., Ebnenasir, A.: Automated synthesis of multitolerance. In: DSN. International Conference on Dependable Systems and Networks, pp. 209–219 (2004)
6. Kulkarni, S.S., Arora, A., Chippada, A.: Polynomial time synthesis of Byzantine agreement. In: SRDS. Symposium on Reliable Distributed Systems, pp. 130–140 (2001)
7. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 82–93. Springer, Heidelberg (2000)
8. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Science of Computer Programming 2(3), 241–266 (1982)
9. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 6(1), 68–93 (1984)

10. Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: Dwyer, M.B. (ed.) Model Checking Software. LNCS, vol. 2057, pp. 217–234. Springer, Heidelberg (2001)
11. Stern, U., Dill, D.L.: Parallelizing the murφ verifier. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 256–278. Springer, Heidelberg (1997)
12. Heyman, T., Geist, D., Grumberg, O., Schuster, A.: Achieving scalability in parallel reachability analysis of very large circuits. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 20–35. Springer, Heidelberg (2000)
13. Leucker, M., Somla, R., Weber, M.: Parallel model checking for LTL, CTL*, and L_2^μ. In: PDMC. International Workshop on Parallel and Distributed Model Checking (2003)
14. Chung, M.-Y., Ciardo, G.: A dynamic firing speculation to speedup distributed symbolic state-space generation. In: IPDPS. International Parallel and Distributed Processing Symposium (2006)
15. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21, 181–185 (1985)
16. Arora, A., Gouda, M.G.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering 19(11), 1015–1027 (1993)
17. Kulkarni, S.S.: Component-based design of fault-tolerance. PhD thesis, Ohio State University (1999)
18. Mattern, F.: Algorithms for distributed termination detection. Journal of Distributed Computing 2(3), 161–175 (1987)

Decentralized Detector Generation in Cooperative Intrusion Detection Systems

Rainer Bye[1], Katja Luther[1], Seyit Ahmet Çamtepe[1], Tansu Alpcan[2], Şahin Albayrak[1], and Bülent Yener[3]

[1] DAI-Labor, Technische Universität Berlin
[2] Deutsche Telekom Laboratories, Berlin
[3] Department of Computer Science, Rensselaer Polytechnic Institute, NY

Abstract. We consider *Cooperative Intrusion Detection System* (CIDS) which is a distributed AIS-based (Artificial Immune System) IDS where nodes collaborate over a peer-to-peer overlay network. The AIS uses the negative selection algorithm for the selection of detectors (e.g., vectors of features such as CPU utilization, memory usage and network activity). For better detection performance, selection of all possible detectors for a node is desirable but it may not be feasible due to storage and computational overheads. Limiting the number of detectors on the other hand comes with the danger of missing attacks. We present a scheme for the controlled and decentralized division of detector sets where each IDS is assigned to a region of the feature space. We investigate the trade-off between scalability and robustness of detector sets. We address the problem of self-organization in CIDS so that each node generates a distinct set of the detectors to maximize the coverage of the feature space while pairs of nodes exchange their detector sets to provide a controlled level of redundancy. Our contribution is twofold. First, we use *Symmetric Balanced Incomplete Block Design*, *Generalized Quadrangles* and *Ramanujan Expander Graph* based deterministic techniques from combinatorial design theory and graph theory to decide how many and which detectors are exchanged between which pair of IDS nodes. Second, we use a classical epidemic model (SIR model) to show how properties from deterministic techniques can help us to reduce the attack spread rate.

1 Introduction

In this paper, we introduce self-organizing, self-adaptive and self-healing capabilities to the *Cooperative Intrusion Detection System* (CIDS) which is a distributed Artificial Immune System (AIS) based Intrusion Detection System (IDS) where nodes collaborate over a peer-to-peer overlay network to improve the detection capability and to decrease the false alarm rate. Hence, CIDS becomes a promising step towards the realization of *Autonomous Security* (AS) framework. We define Autonomous Security as an environment which provides smart and usable security mechanisms, distributed monitoring-detection-prevention with self-* properties and a security simulation/evaluation tool.

The Artificial Immune System (AIS), like the Biological Immune System, is based on the distinction between self and non-self. Initially, an n-dimensional feature space is covered by detectors (i.e., n-dimensional vectors of features such as CPU utilization, memory usage, . . ., number of tcp connections). During the training phase, these detectors are presented to feature vectors describing the self. Matching detectors are eliminated and the remaining are used for the detection of anomalies. Selecting all possible detectors for an IDS node would certainly provide best attack detection capability. But, it may not be feasible to store large amount of detectors and process them while trying to detect attacks in timely manner. Limiting the number of detectors on the other hand comes with the danger of missing attacks. Therefore, creating a detector set which covers the whole feature space may not be feasible, and random selections can create holes which means some critical regions on the feature space may not be covered. Kim *et al.* [1] use evolutionary algorithms and Gonzalez *et al.* [2] use monte carlo based approaches to face the coverage problem. In this paper, we assume that each IDS is assigned to a mutually exclusive region in the feature space due to approaches in [1] and [2]. Each IDS node can independently generate a detector set for the region of the feature space it is responsible for. In this scalable and decentralized approach, at most one IDS can have the proper detector for an attack and the attack may spread faster than the case where each node stores the global detector set. Thus, IDS nodes should have overlapping detector sets; in other words, detectors should be redundantly distributed to IDS nodes.

Goel *et al.* propose to distribute the detector generation [3] where each IDS is responsible for a non-overlapping region in the feature space, and it generates a mutually exclusive subset of the detector space. When an attack is detected, the corresponding detector set is broadcasted to other vulnerable nodes in the network. In this approach, processing and memory overhead of the detector generation is reduced by the expense of increased communication. Moreover, an attack may spread quickly since only one IDS has the proper detector to detect the attack, and vulnerable nodes are updated only when this specific IDS is attacked. Thus, Goel *et al.* [3] consider to create an overlap of detector sets stored on IDS nodes by using random graph $G(N, p)$[1] approach due to Erdős-Rényi [4] where each IDS exchanges its detector set with $(\log N)$ other IDS nodes.

In this paper, we assume a homogeneous network environment where each node has similar capabilities and configuration. Nodes have limited resources so that they can not store and process all possible detectors. Wireless sensor networks, networks of pico-satellites, smartphones, wireless ad-hoc networks of mobile devices can be the examples of such networks. We assume that attacks are equally probable and they are uniformly distributed over the feature space.

1.1 Our Contribution

We investigate the trade-off between scalability and redundancy of detector sets. Our contribution is twofold. First, we use Symmetric Balanced Incomplete

[1] $G(N, p)$ is a graph with N nodes where each pair of nodes have a link in between with probability p.

Block Design (SBIBD), Generalized Quadrangles (GQ) and Ramanujan Expander Graph (REG) techniques from combinatorial design theory and graph theory to decide how many and which detectors are exchanged between which pair of IDS nodes. Each IDS node uses the proposed deterministic techniques to independently decide which nodes to contact with and get their detector sets.

Communication and detector set exchange is done through a peer-to-peer overlay network. Unlike probabilistic approaches (i.e., $G(N,p)$), our deterministic approaches provide regular logical graphs for detector set exchange. In the SBIBD-based approach, every pair of IDS nodes have exactly one detector set in common. This approach provides the highest level of overlap. In GQ, not every pair of IDS nodes share a detector set but if two IDS nodes do not share a detector set, there are other IDS nodes sharing detector set with both. The GQ-based approach decreases the level of overlap in a controlled way. REG-based deterministic approach is comparable to $G(N,p)$-based probabilistic approach due to Goel *et al.* [3]. REG provides better defense against attack spread since REG are the best known expanders meaning that any subset of nodes are connected to a larger subset of nodes for a fixed node degree; equivalently, any subset of IDS nodes share detectors with the largest possible subset of IDS nodes. This property of REG help them to provide a better immunity against attacks.

In general, our approaches can be classified as decentralized, self-organizing, self-adaptive and self-healing. They are *decentralized* because detector set generation task is distributed to networked IDS nodes. Next, they are *self-organizing* because once assigned to a feature subspace, each IDS node independently generates its detector set, and independently decides nodes with which to exchange its detector set to create a controlled level of overlap. Then, they are *self-adaptive* because the network size can be increased by inserting new IDS nodes in which case IDS nodes shall re-organize. Any faulty IDS can be replaced with a new one and new IDS may independently decide which other nodes to contact to recover the detector sets. Finally, they are *self-healing* because when an attack reaches to an IDS node with a proper detector, the detector is sent to other IDS nodes to stop the attack spread.

As the second contribution, we enhance the classical epidemic model (SIR model [5]). We model the spread of both attacks and defenses (detector updates). We show how properties from deterministic techniques can help us to create a regular and controlled level overlap between detector sets of individual IDS nodes to reduce the attack spread rate.

1.2 Organization

Organization of the paper is as follows: in Section 2 we provide background information on Cooperative IDS, classical epidemic model, Balanced Incomplete Block Designs (BIBD), Generalized Quadrangles (GQ) and Ramanujan Expander Graphs (REG). In Section 3 we introduce our combinatorial design and expander graph based approaches. In Section 4 we analyze our approaches by using an epidemic model. Finally, in Section 5 we conclude.

2 Background

2.1 Cooperative IDS

Cooperative Intrusion Detection System (CIDS) utilizes the AIS (Artificial Immune System) introduced by Forrest *et al.* [6,7,8] and a peer-to-peer (P2P) overlay system [9]. The AIS principle bases on the *self/non-self* discrimination. It uses the negative selection algorithm described by Hofmeyr *et al.* [8] as the algorithm for the selection of detectors for a given set of feature vectors. A *feature vector* is an n-dimensional vector of numerical values. A detector denotes a point in the feature space, like a feature vector, with the additional information such as age.

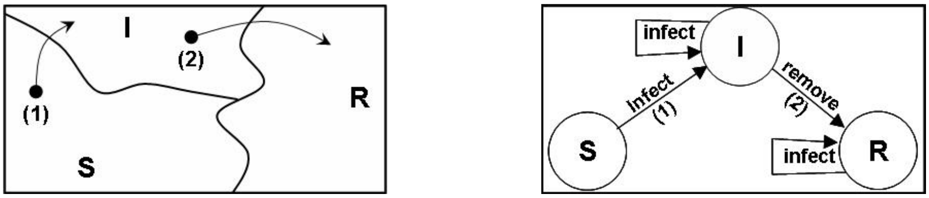
The main idea is to produce detectors randomly and compare them to the normal patterns obtained during the training. Every detector that matches to these normal patterns is eliminated, and hence, the remaining detectors recognize only abnormal patterns. In the case of the Cooperative IDS, the feature vectors might be composed of network traffic statistics measured at an end device in a specific time interval. A sample feature vector might be < *timestamp, number of TCP connections, number of TCP packets, number of UDP packets, number of used ports, number of port scans* >. After the learning period, new measurements are presented to the remaining detectors, and the distances between the feature vectors and the detectors are computed. If the distance between a detector and a feature vector crosses a specific threshold, the matching detector is distributed.

The used P2P infrastructure is based on a hybrid decentralized architecture. The nodes are equal in their abilities but one node acts always as the *super node* for the purpose of the initial look-up of other peers containing AIS-systems. Apart from this, the nodes communicate with all other peers autonomously. The P2P overlay enables the collaborating nodes to exchange status information or detectors in the scenario of this paper. A general classification of P2P systems can be found in [10]. For further details regarding the CIDS, we refer to [9].

2.2 Classical Epidemic Model

Classical epidemic model for the spread of Internet worms [5] considers a group of homogeneously mixed susceptible (S), infected (I) and removed (R) nodes. A node is susceptible if it is not infected and if it has no proper protection for the attack. A node which is attacked by a worm (a.k.a., virus or any malicious code) becomes infected, and immediately starts to spread the infection to other susceptible nodes. An infected node can be recovered and immunized when proper protection mechanisms are enabled in which case the node becomes removed. Removed nodes can not be infected again. Suppose that at time t_i there are $S(t_i)$ susceptible, $I(t_i)$ infected and $R(t_i)$ removed nodes where $S(t_i) + I(t_i) + R(t_i) = N$. Let

$$s(t_i) = \frac{S(t_i)}{N},\ i(t_i) = \frac{I(t_i)}{N},\ r(t_i) = \frac{R(t_i)}{N}$$



(a) (b)

Fig. 1. Classical epidemic model for spread of Internet worms in a group of homogeneously mixed Susceptible (S), Infected (I) and Removed (R) nodes where $S(t_i) + I(t_i) + R(t_i) = N$ at time t_i. (1) Susceptible nodes get infected at the rate given by Equation 3 as they are contacted by the infected nodes. (2) Infected nodes are immunized and they become removed at the rate given by Equation 4. A removed node can not be infected again.

be the ratio of susceptible, infected and removed nodes respectively. Each contact in between susceptible and infected nodes will result in an infection. Therefore, in an interval $\triangle t$, there will be $\beta\ I(t)\ \frac{S(t)}{N}\ \triangle t$ contacts resulting in infection where β is the average number of contacts per infected node. Furthermore, infected nodes are removed from the system at a rate γ due to recovery or immunization. The number of removal at time interval $\triangle t$ is $\gamma\ I(t)\ \triangle t$. That is:

$$\triangle I(t) = \beta\ I(t)\ \frac{S(t)}{N}\ \triangle t - \gamma\ I(t)\ \triangle t, \tag{1}$$

$$\frac{di(t)}{dt} = \beta\ i(t)\ s(t) - \gamma\ i(t), \tag{2}$$

$$\frac{ds(t)}{dt} = -\beta\ i(t)\ s(t), \tag{3}$$

$$\frac{dr(t)}{dt} = \gamma\ i(t). \tag{4}$$

Epidemic can not build up if $[di(t)/dt]_{t_0} \leq 0$. That means, $\beta\ i(t)\ s(t) - \gamma\ i(t) \leq 0$ and $s(t) \leq \gamma/\beta$. The ratio γ/β is called *relative removal-rate* or *threshold density for susceptible nodes* and represented by $\rho = \gamma/\beta$. Figure 1 summarizes the overall model.

2.3 Balanced Incomplete Block Designs (BIBD)

A *BIBD* is an arrangement of v distinct objects into b blocks such that: (i) each object is in exactly r distinct blocks, (ii) each block contains exactly k distinct objects, (iii) every pair of distinct objects is in exactly λ blocks. The design is expressed as (v, b, r, k, λ) (a.k.a., (v, k, λ)) where: $b \cdot k = v \cdot r$ and $\lambda \cdot (v-1) = r \cdot (k-1)$. It is called *Symmetric BIBD* (a.k.a., *Symmetric Design* or SBIBD) when $b = v$ and $r = k$ [11] meaning that not only every pair of objects occurs in λ blocks but also every pair of blocks intersects on λ objects.

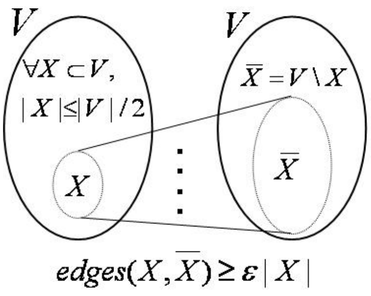
In this paper, we are interested in the *Finite Projective Plane* which is a subset of *Symmetric BIBD*. *Finite Projective Plane* consists of points (a finite set P of points) and lines (a set of subsets of P) of the *projective space* $PG(2, q)$ of dimension 2 and order q. For each prime power q where $q \geq 2$, there exists a *Finite Projective Plane* of order q [12, Theorem 2.10] with following four properties: (i) every line contains exactly $k = q + 1$ points, (ii) every point occurs on exactly $r = q + 1$ lines, (iii) there are exactly $v = q^2 + q + 1$ points, and (iv) there are exactly $b = q^2 + q + 1$ lines. Thus, a *Finite Projective Plane* of order q is a *SBIBD* with parameters $(q^2 + q + 1, q + 1, 1)$ [11].

We consider two methods to construct *SBIBD* of the form $(q^2 + q + 1, q + 1, 1)$. First method is *Difference Set Method* where the construction is done by simple modular addition operations on a *cyclic difference set*. A cyclic (v, k, λ) *difference set* $(mod\ v)$ is a set $B = \{b_1, b_2, \ldots, b_k\}$ of distinct elements in Z_v such that each one of the $(v-1)$ elements, say b, can be expressed in the form of difference $b = b_i - b_j\ (mod\ v)$ in λ different ways where $b_i, b_j \in B$ [11, Definition 2.1.1]. SBIBD blocks can be constructed by $B, B + 1, B + 2, \ldots, B + (v - 1)$ $(mod\ v)$ [11, Theorem 2.1.3] where $B + i = \{(b_1 + i)\ mod\ v, \ldots,\ (b_k + i)\ mod\ v\}$. For example, *difference set* $\{1, 2, 4\}$ can be used to generate $(7, 3, 1)$ SBIBD with blocks $\{1, 2, 4\}$, $\{2, 3, 5\}$, $\{3, 4, 6\}$, $\{4, 5, 7\}$, $\{5, 6, 1\}$, $\{6, 7, 2\}$, $\{7, 1, 3\}$ (0 is replaced with 7). Difference method provides a very efficient construction which can be used on low-resource devices if a cyclic different set for the target design is known. In fact, *cyclic difference sets* for small designs are listed in [11]. But, generating a cyclic difference set for a large design is not trivial [11, Theorem 2.5.2]. Second method is used for large designs where *complete set* of $(q - 1)$ *Mutually Orthogonal Latin Squares (MOLS)* are used to first construct an *affine plane* of order q which is a $(q^2, q, 1)$ design. The *affine plane* of order q is then converted into a *projective plane* of order q which is a $(q^2+q+1, q+1, 1)$ *SBIBD*. Construction can be done in $O(v^{3/2})$ time as described in [13] and references there in.

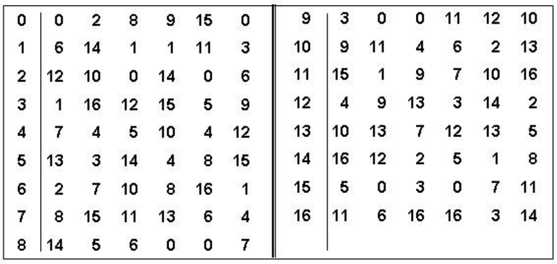
2.4 Finite Generalized Quadrangle (GQ)

A *Finite Generalized Quadrangle* $GQ(s, t)$ is a point-line incidence relation with following properties: (i) each point is incident with $t + 1$ lines $(t \geq 1)$ and two distinct points are incident with at most one line, (ii) each line is incident with $s + 1$ points $(s \geq 1)$ and two distinct lines are incident with (a.k.a., intersect on) at most one point, and (iii) if x is a point and L is a line not incident (I) with x, then there is a unique pair $(y, M) \in Points \times Lines$ for which $x\ I\ M\ I\ y\ I\ L$. In a $GQ(s, t)$, there are $v = (s+1)(st+1)$ points and $b = (t+1)(st+1)$ lines where each line includes $s + 1$ points and each point is incident with $t + 1$ lines. In this work, we are interested in $GQ(q, q)$ from projective space $PG(4, q)$. Probability that two lines intersect in $GQ(q, q)$ is given by the Equation 5.

$$P_{GQ} = \frac{t(s+1)}{(t+1)(st+1)} = \frac{q(q+1)}{(q+1)(q^2+1)} = \frac{q^2+q}{q^3+q^2+q+1} \approx \frac{1}{q}. \tag{5}$$



V
$\forall X \subset V,$
$|X| \leq |V|/2$
X
V
$\overline{X} = V \setminus X$
$\overline{X}$
$edges(X, \overline{X}) \geq \varepsilon |X|$



0	0	2	8	9	15	0
1	6	14	1	1	11	3
2	12	10	0	14	0	6
3	1	16	12	15	5	9
4	7	4	5	10	4	12
5	13	3	14	4	8	15
6	2	7	10	8	16	1
7	8	15	11	13	6	4
8	14	5	6	0	0	7
9	3	0	0	11	12	10
10	9	11	4	6	2	13
11	15	1	9	7	10	16
12	4	9	13	3	14	2
13	10	13	7	12	13	5
14	16	12	2	5	1	8
15	5	0	3	0	7	11
16	11	6	16	16	3	14

Fig. 2. Expander graphs are regular multi-graphs with expansion coefficient ϵ. Adjacency matrix for a sample Ramanujan Expander Graph (REG) $X^{5,17}$.

In $GQ(s,t) = GQ(q,q)$, there are $v = b = q^3 + q^2 + q + 1$ lines and points. Each line contains $s + 1 = q + 1$ points, and each point is incident with $t + 1 = q + 1$ lines. Consider $GQ(s,t) = GQ(2,2)$ for $q = 2$ as an example. There are 15 points $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ and 15 lines $\{1, 8, 9\}$ $\{1, 12, 13\}$ $\{1, 4, 5\}$ $\{3, 12, 15\}$ $\{2, 8, 10\}$ $\{2, 12, 14\}$ $\{2, 4, 6\}$ $\{5, 11, 14\}$ $\{3, 4, 7\}$ $\{6, 11, 13\}$ $\{5, 10, 15\}$ $\{3, 8, 11\}$ $\{7, 9, 14\}$ $\{7, 10, 13\}$ and $\{6, 9, 15\}$ where each line contains $s + 1 = 3$ points and each point is incident with $t + 1 = 3$ lines. Note that lines $\{1, 8, 9\}$ and $\{3, 12, 15\}$ do not intersect but GQ provides three other lines intersecting with both: $\{1, 12, 13\}$, $\{3, 8, 11\}$ and $\{6, 9, 15\}$. $GQ(q,q)$ can be constructed from projective space $PG(4,q)$ with the canonical equation $Q(\boldsymbol{X}) = x_0^2 + x_1x_2 + x_3x_4 = 0$. Each point is a vector of the form $\boldsymbol{X} =< x_0, x_1, x_2, x_3, x_4 >$ in $GF(q)$, and each line contains $q+1$ bilinear points. $GQ(q,q)$ can be constructed in $O(v^2)$ time as described in [13] and references there in.

2.5 Ramanujan Expander Graphs (REG)

An *expander* is a *regular* multi-graph in which any subset of vertices has a large number of neighbors. It is highly connected, it has small diameter, small degree and many alternate disjoint paths between vertices. Formally, a graph $G = (V, E)$ is said to be ϵ-*edge-expander* if for every partition of the vertex set V into X and $\overline{X} = V \backslash X$, where $|X| \leq |V|/2$, the number of *cross edges* is $e(X, \overline{X}) \geq \epsilon|X|$. ϵ is called *expansion coefficient* ([14], [15], [16]). Almost all random bipartite graphs are expanders but it is possible to deterministically construct the best expanders with maximum possible *expansion coefficient* ϵ and with bounded vertex degrees. Figure 2 illustrates the expansion property.

There is a relation between expansion of a graph and eigenvalues of its adjacency matrix. A graph $G(V,E)$ with n vertices can be represented as an $(n \times n)$ *adjacency matrix* $A(G)$. The eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ of $A(G)$ are the *spectrum* of graph G. Spectrum of a k-regular graph has the property that $\lambda_0 = k \geq \lambda_1 \geq \ldots \geq \lambda_{n-1}$. The relation between spectral gap (i.e., $\lambda_0 - \lambda_1$) and expansion coefficient ϵ is given in Equation 6 where larger spectral gap $(\lambda_0 - \lambda_1)$ implies higher expansion. *Ramanujan Expander Graphs* (REG) are asymptotically optimal and best known explicit expanders [17] where $|\lambda_i| \leq 2\sqrt{k-1}$ for $(1 \leq i \leq n-1)$. A REG $X^{s,t}$ is a k-regular graph with $N = t+1$ nodes for $k = s+1$ where s and t are primes congruent to 1 (mod 4). REG can be constructed in $O(st)$ time as described in [18].

$$\frac{k-\lambda_1}{2} \leq \epsilon \leq \sqrt{2k(k-\lambda_1)}, \tag{6}$$

3 Distributed Detector Set Generation

We consider the *Cooperative Intrusion Detection System* (CIDS) where N AIS-based IDS nodes collaborate over a peer-to-peer overlay network. Each IDS is assigned to a subspace in the feature space due to approaches in [1] and [2], and generates a set of detectors. Therefore, IDS nodes have *mutually exclusive sets* of detectors as proposed in [3]. Advantage of this approach is that for a fixed capacity of IDS nodes (i.e., χ detectors per nodes), it is possible to have an aggregated global detector database of $N\chi$ detectors meaning that the feature space coverage is maximized. However, when an attack starts spreading, only one IDS node will have the proper detector and this IDS will not be updating others until the attack reaches itself. Thus, detector sets on IDS nodes should have a level of overlap to be able to stop an attack spread in its earliest stage possible. In the following sections, we describe a probabilistic approach [3] and our three novel deterministic approaches which enable IDS nodes to create controlled level of detector set overlap by using techniques from combinatorial design theory and graph theory.

Algorithm 1 summarizes the decentralized detector generation scheme for a homogeneous network of N AIS-based (Artificial Immune System) IDS enabled nodes. Initially, each IDS node receives a unique ID (i.e., IDS_i) and gets assigned to a mutually exclusive subspace S_i in the feature space. In Algorithm 1 step 1, each IDS_i generates a set of detectors D_i of size d for the subspace S_i based on the approaches described in [9]. Next, through steps 2 to 14, each IDS_i decides on a list of node ID B_i for $|B_i| = k$ by using either one of the Random Graph (RG), Symmetric BIBD (SBIBD), Generalized Quadrangles (GQ) or Ramanujan Expander Graph (REG) based approaches. IDS_i contacts to each node $j \in B_i$ to get the detectors D_j for the subspace S_j through steps 15 to 18. IDS_i aggregates the received detectors $AD_i = \{\bigcup D_j | \forall j \in B_i\}$ where $|AD_i| = k \cdot d$ in step 19.

We assume that there exist a detector for the attack and initially only one node is infected (i.e., $I(t=0) = 1$). Let β be the number of nodes an infected node can attack and γ be the number of nodes that the detector for the attack is sent per unit time. *Probability $P(t)$ that infected nodes will not attack any node which has a proper detector at time t* is an important metric for spread of attack. Because, once an IDS with a proper detector is attacked, it will start distributing the detector at a rate γ as modeled in Section 4. *Feature space coverage* is another metric where we want to maximize coverage while minimizing probability $P(t)$. *Communication overhead* is our third metric to evaluate our approaches. It is the number and size of messages exchanged until every node has its target aggregated detector set.

Algorithm 1. Decentralized Detector Generation

```
Require:
    N {Total number of IDS enabled nodes},
    $IDS_i$ {ID of the IDS node running this algorithm},
    $S_i \subset Feature\ Space$ {Subspace assigned to $IDS_i$},
    d {Number of detectors generated from each subspace $S_i$},
    Algorithm {SBIBD, GQ, REG or RG}.
 1: $IDS_i$ generates the detector set $D_i$ ($|D_i| = d$) from the subspace $S_i$
 2: if Algorithm = Symmetric BIBD then
 3:     Generate $(v, k, \lambda)$-Design where $v = N = q^2 + q + 1$ and $k = q + 1$
 4:     $IDS_i$ selects the block $B_i$ ($|B_i| = k$)
 5: else if Algorithm = Generalized Quadrangles then
 6:     Generate $GQ(q, q)$-Design where $N = q^3 + q^2 + q + 1$ and $k = q + 1$
 7:     $IDS_i$ selects the block $B_i$ ($|B_i| = k$)
 8: else if Algorithm = Ramanujan Expander Graph then
 9:     Generate $X(s, t)$ expander where $N = t + 1$ and $k = s + 1$
10:     $B_i$ for $IDS_i$ is the neighbor list of $i^{th}$ node in $X^{s,t}$
11: else if Algorithm = Random Graph then
12:     $IDS_i$ selects $k = \log N$ IDS nodes
13:     $B_i$ for $IDS_i$ is the list of selected nodes
14: end if
15: for $i = 1$ to $|B_i|$ do
16:     $IDS_i$ contacts $IDS_j$ where $j \in B_i$
17:     $IDS_i$ receives the detector set $D_j$ from $IDS_j$
18: end for
19: Aggregated detector set for $IDS_i$ is $AD_i = \{\bigcup D_j | \forall j \in B_i\}$ where $|AD_i| = k \cdot d$
```

3.1 Random Graph $G(N, p)$ Based Approach

Mapping: Random graph approach is proposed by Goel *et al.* [3]. This approach employs graphs $G(N, p)$ due to Erdős-Rényi [4] where there are N nodes and each pair of nodes have a link in between with probability p. It is shown that each node should have $\log N$ neighbors for the graph $G(N, p)$ to be connected [19]. We define following mapping from random graphs to decentralized detector generation: nodes in graph $G(N, p)$ are mapped to IDS nodes and two nodes share a common detector set if there is an edge in between in the graph.

Construction: IDS_i (for $1 \leq i \leq N$) randomly selects $\log N$ IDS nodes. Selected nodes form the set B_i where it is possible that IDS_i selects itself (i.e., $i \in B_i$). IDS_i contacts each IDS_j for $j \in B_i$ and requests its detector set D_j. IDS_i ends up with the aggregated detector set $AD_i = \{\bigcup D_j | \forall j \in B_i\}$. Overall process is given through steps 11 to 14 of Algorithm 1. Graph $G(N, p)$ should be connected to make sure that a detector set D_j appears in at least two IDS nodes. A new coming node can randomly select $\log N$ nodes to receive their detector sets and $\log N$ nodes to send its detector set. A replacement node should only randomly select $\log N$ nodes to receive their detector sets.

Analysis: Each detector appears in average of $\log N$ nodes because each IDS node is contacted by average of $\log N$ nodes. Probability $P_{RG}(t)$ that infected nodes will not attack any node which has a proper detector at time t:

$$P_{RG}(t) = \left(1 - \frac{\log N}{N}\right)^{\beta I(t)} \tag{7}$$

For a fixed IDS node capacity of χ detectors, each node generates $\frac{\chi}{\log N}$ detectors and there are total of $\frac{N\chi}{\log N}$ distinct detectors. Each IDS contacts $\log N$ nodes and get contacted by $\log N$ nodes at average, *communication overhead* for each nodes is $2 \log N$ messages where each message includes $\frac{\chi}{\log N}$ detectors.

3.2 Ramanujan Expander Graph Based Approach

Problem with the random graph is its lack of regularity. Each node has average of $\log N$ neighbors (a.k.a., a detector appears in average of $\log N$ nodes) meaning also that some nodes may have less (a.k.a., some detectors may appear on less nodes) while some nodes may have more neighbors (a.k.a., some detectors may appear on more nodes). For the same node degree as $G(N, p)$, Ramanujan Expander Graph (REG) based deterministic approach provides best known expansion which means any subset of IDS nodes share detectors with the largest possible subset of IDS nodes. This property of REGs help REG-based approaches to provide better immunity against attacks when compared to RG-based approaches.

Mapping: A Ramanujan Expander Graph (REG) $X^{s,t}$ is a regular multi-graph where there are $N = t + 1$ nodes and where each node has $s + 1$ neighbors. We define following mapping from REGs to decentralized detector generation: nodes of the $X^{s,t}$ are mapped to IDS nodes and two nodes share a common detector set if there is an edge in between in $X^{s,t}$.

Construction: Each IDS_i deterministically generates the list of its neighbors B_i. IDS_i contacts each IDS_j for $j \in B_i$ and requests its detector set D_j. IDS_i ends up with the aggregated detector set $AD_i = \{\bigcup D_j | \forall j \in B_i\}$. Overall process is given through steps 8 to 10 of Algorithm 1. An additional node or a replacement node can generate the design and contact with its neighbors to exchange the detector sets.

Analysis: If we let the REG to have same degree as RG (i.e., $s + 1 = \log N$), it will have same probability $P_{REG}(t) = P_{RG}(t)$, feature space coverage and communication overhead. Advantage of the REG is its regularity and expansion property which provides better immunity against attacks when compared to RG-based approaches as we discuss in Section 4.

3.3 Combinatorial Design Based Approach

Mapping: We propose two novel approaches, Generalized Quadrangles (GQ) and Symmetric Balanced Incomplete Block Designs (SBIBD) from combinatorial

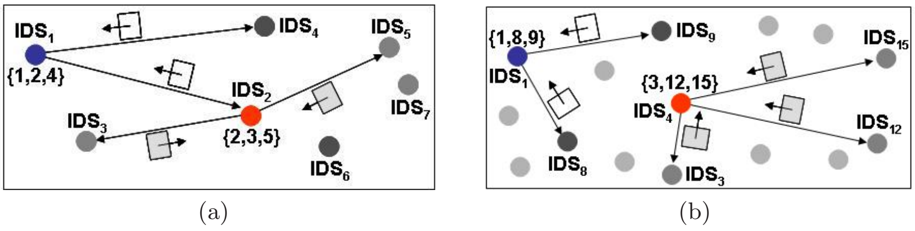


Fig. 3. Decentralized detector generation. (a) Symmetric BIBD (SBIBD) based approach. (b) Generalized Quadrangle (GQ) based approach. Each block is assigned to an IDS node (IDS_1 uses block {1,2,4} in (a)) where each object is a distinct IDS node ID. Each IDS contacts the nodes in its block and collects the detector sets they generated.

design theory. Both SBIBD and GQ design assign v objects into b blocks ($v = b = q^2 + q + 1$ in SBIBD and $v = b = q^3 + q^2 + q + 1$ in $GQ(q, q)$) so that every pair of blocks have exactly one common object in SBIBD, and at most one common object in GQ design (probability of object share is given in Equation 5). We define following mapping from GQ and SBIBD to decentralized detector generation: each object is mapped to a distinct IDS node ID and each block is assigned to an IDS node (IDS_i uses the block B_i). Each IDS contacts the nodes in its block and collects the detector sets they generated. Thus, two IDS have common detectors if they have the same node ID in their blocks which is true for every pair of nodes in SBIBD based approach. Overall process is illustrated in Figure 3.

Construction: After the first step of Algorithm 1, each IDS_i generates the blocks in SBIBD or GQ. Assume that difference set method is used in SBIBD. Given a cyclic difference set B, node IDS_i can construct block B_i by simply performing $O(\sqrt{N})$ modular addition operations. Larger SBIBD can be constructed with MOLS in $O(N^{3/2})$ and GQ designs can be constructed in $O(N^2)$ as described in Section 2. Finally, IDS_i contacts each IDS_j for $j \in B_i$ and requests its detector set D_j. Aggregated detector set for IDS_i becomes $AD_i = \{\bigcup D_j | \forall j \in B_i\}$. Overall process is given through steps 2 to 7 of Algorithm 1. An additional node or a replacement node can generate the design and contact with its neighbors to exchange the detector sets.

Analysis: In SBIBD-based approach, each detector appears in exactly $q + 1$ nodes where $N = q^2 + q + 1$. Thus, probability $P_{SBIBD}(t)$ that infected nodes will not attack any node which has a proper detector at time t:

$$P_{SBIBD}(t) = \left(1 - \frac{q+1}{q^2+q+1}\right)^{\beta I(t)} \approx \left(1 - \frac{1}{\sqrt{N}}\right)^{\beta I(t)} \quad (8)$$

For fixed IDS node capacity, say χ detectors, each node generates $\frac{\chi}{q+1} \approx \frac{\chi}{\sqrt{N}}$ detectors and there are $\sqrt{N}\chi$ distinct detectors. During detector exchange process, each IDS contacts $q+1 \approx \sqrt{N}$ nodes and gets contacted by $\sqrt{N}$ nodes. *Communication overhead* for each node is $2\sqrt{N}$ messages where each message includes $\frac{\chi}{\sqrt{N}}$ detectors. In the Generalized Quadrangle (GQ) based approach, each detector appears in exactly $q+1$ nodes where $N = q^3 + q^2 + q + 1$. Thus, probability $P_{GQ}(t)$ is:

$$P_{GQ}(t) = \left(1 - \frac{q+1}{q^3+q^2+q+1}\right)^{\beta I(t)} \approx \left(1 - \frac{1}{\sqrt[3]{N^2}}\right)^{\beta I(t)} \tag{9}$$

For fixed IDS node capacity of χ detectors, each node generates $\frac{\chi}{q+1} \approx \frac{\chi}{\sqrt[3]{N}}$ detectors and there are $\sqrt[3]{N^2}\chi$ distinct detectors. During detector exchange process, each IDS contacts $q+1 \approx \sqrt[3]{N}$ nodes and gets contacted by $\sqrt[3]{N}$ nodes. *Communication overhead* for each node is $2\sqrt[3]{N}$ messages where each message includes $\frac{\chi}{\sqrt[3]{N}}$ detectors.

4 Analysis and Comparisons

We enhance the classical epidemic model to analyze our decentralized detector distribution approaches in cooperative intrusion detection systems. We consider a homogeneous network of susceptible (S), infected (I), active removed (R^+) and passive removed (R^-) nodes where each node stores a subset of detectors and has the CIDS capability. A node is susceptible for a specific attack if it is not infected and if it does not have the detector for the attack. A susceptible node which is the target of the attack becomes infected, and it immediately starts to spread the attack to other susceptible nodes. A node is removed if it has the detector for the attack. Initially all removed nodes are passive removed nodes. A node becomes an active removed node and spreads the detector for the attack under following conditions: (i) when a passive removed node is attacked by an infected node, (ii) when an infected or susceptible node receives the detector from an active removed node, and (iii) when a passive node is contacted by an active removed node.

Suppose that at time t_i there are $S(t_i)$ susceptible nodes, $I(t_i)$ infected nodes, $R^+(t_i)$ active removed nodes and $R^-(t_i)$ passive removed nodes where $S(t_i) + I(t_i) + R^+(t_i) + R^-(t_i) = N$ and where $R^+(t_0) = 0$. Let

$$s(t_i) = \frac{S(t_i)}{N},\ i(t_i) = \frac{I(t_i)}{N},\ r^+(t_i) = \frac{R^+(t_i)}{N},\ r^-(t_i) = \frac{R^-(t_i)}{N}$$

be the ratio of susceptible, infected, active removed and passive removed nodes respectively. Each contact in between susceptible and infected nodes will result in an infection. Let β be the average number of contacts per infected node and γ be the average number of contacts per active removed node, in an interval $\triangle t$:

1. $\beta\ I(t)\ \frac{S(t)}{N}\ \triangle t$ susceptible nodes will be infected due to the attacks from infected nodes {flow (1) in Figure 4},
2. $\gamma\ R^+(t)\ \frac{I(t)}{N}\ \triangle t$ infected nodes will be removed due to the detector updates from active removed nodes {flow (2) in Figure 4},

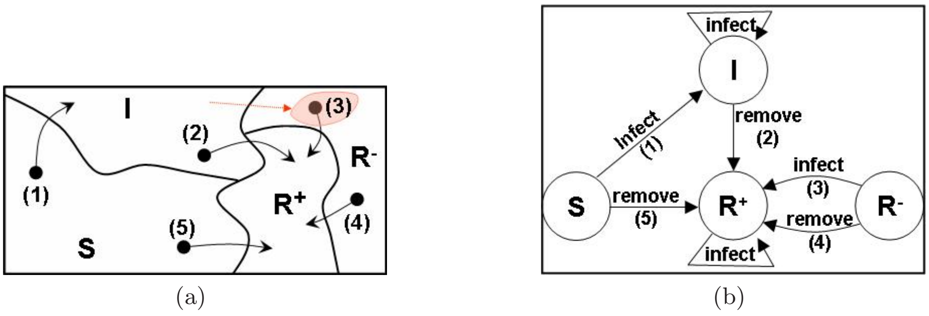


Fig. 4. Epidemic model for Cooperative Intrusion Detection Systems (CIDS) for group of homogeneously mixed susceptible (S), infected (I), active removed (R^+) and passive removed (R^-) nodes where $S(t_i) + I(t_i) + R^+(t_i) + R^-(t_i) = N$ at time t_i. (1) Susceptible nodes become infected when attacked by infected nodes, (2) infected nodes become removed nodes when updated by active removed nodes, (3) passive removed nodes become active removed nodes when attacked by infected nodes, (4) passive removed nodes become active removed nodes when updated by active removed nodes, and (5) susceptible nodes become active removed nodes when updated by active removed nodes.

3. $\beta\ I(t)\ \frac{R^-(t)}{N}\ \triangle t$ passive removed nodes will be active removed due to the attacks from infected nodes {flow (3) in Figure 4},
4. $\gamma\ R^+(t)\ \frac{R^-(t)}{N}\ \triangle t$ passive removed nodes will be active removed due to the detector updates from active removed nodes {flow (4) in Figure 4},
5. $\gamma\ R^+(t)\ \frac{S(t)}{N}\ \triangle t$ susceptible nodes will be active removed due to the detector updates from active removed nodes {flow (5) in Figure 4}.

Thus, infection and removal rates can be formulated as follows:

$$\triangle I(t) = \beta\ I(t)\ \frac{S(t)}{N}\ \triangle t - \gamma\ R^+(t)\ \frac{I(t)}{N}\ \triangle t, \tag{10}$$

$$\frac{di(t)}{dt} = \beta\ i(t)\ s(t) - \gamma\ r^+(t)\ i(t), \tag{11}$$

$$\frac{dr^+(t)}{dt} = \beta\ i(t)\ r^-(t) + \gamma\ r^+(t)\ \big(s(t) + i(t) + r^-(t)\big)\,. \tag{12}$$

Epidemic can not build up if $[di(t)/dt]_{t_i} \leq 0$ when $r^+(t_i) > 0$. The ratio $\frac{\gamma}{\beta}\ r^+(t)$ in Equation 13 is *relative removal-rate* or *threshold density for susceptible nodes.*

$$\beta\ i(t)\ s(t) - \gamma\ r^+(t)\ i(t) \leq 0, \quad s(t) \leq \frac{\gamma}{\beta}\ r^+(t). \tag{13}$$

In all approaches, SBIBD-based approach provides the largest overlap between detector sets and minimum probability $P(t)$ (Equations 7, 8 and 9) at the expense of increased communication overhead and decreased coverage of feature space. In addition to these, deterministic approaches have two advantages. First, unlike $G(N,p)$, deterministic approaches are based on the regular graphs due to SBIBD, GQ and REG. That means, each detector is replicated in equal number of IDS initially. Second, each node can generate the SBIBD, GQ and REG at very low cost. That means, all nodes know exactly which IDS has which detectors; moreover, they know $R^-(t_0)$. We can safely assume that, when an attack reaches to a node in $R^-(t_0)$ all others in $R^-(t_0)$ can be informed because such a broadcast update should have much lower overhead compared to attack and detector spread. Thus, a node in $R^+(t)$ does not need to update the nodes in $R^-(t_0)$ preventing the update collisions. Equation 10 then becomes:

$$\triangle I(t) \;=\; \beta\; I(t)\;\frac{S(t)}{N}\;\triangle t \;-\; \gamma_{det}\; R^+(t)\;\frac{I(t)}{N}\;\triangle t$$

where $\gamma_{det} = \gamma \frac{N}{N - R^-(t_0)}$ is the improved detector update rate due to the deterministic approaches. Problem can also be formulated as a spread of a detector on the logical graph due to underlying SBIBD, GQ and REG techniques. It is shown in [20] and [21] that epidemic spread threshold is related to underlying logical graph properties (i.e., node degree, diameter, spectral radius and largest eigenvalue of the adjacency matrix). More specifically, larger degree and spectral radius mean faster spread. SBIBD, GQ and REG provides regular graphs; in fact, SBIBD provides a complete graph which has the largest spectral radius (n where $N = n^2 + n + 1$).

5 Conclusions

We address the problem of self-organization and decentralized detector generation in Cooperative Intrusion Detection Systems (CIDS). We consider a set of AIS-based IDS nodes each of which is assigned to a distinct subspace of the feature space. Each IDS node generates a subset of the global detector set so that the coverage of the future space is maximized. Then, pairs of IDS nodes exchange detector sets to create controlled level redundancy so that the spread rate of the attack is limited. More specifically, our contribution is twofold. First, we use Symmetric Balanced Incomplete Block Design (SBIBD), Generalized Quadrangles (GQ) and Ramanujan Expander Graph (REG) techniques from combinatorial design theory and graph theory so that each node can independently decide how many and which detectors to exchange with which IDS nodes. Second, we applied classical epidemic model on both spread of attack and spread of detector, and showed that regular structures in deterministic techniques provides better immunity in a self-organized, self-adaptive and self-healing cooperative intrusion detection system when compared to probabilistic approaches.

References

1. Kim, J., Bentley, P.: The artificial immune model for network intrusion detection. In: EUFIT. 7th European Conference on Intelligent Techniques and Soft Computing (1999)

2. Gonzalez, F., Dasgupta, D.: Anomaly detection using using real-valued negative selection. In: Genetic Programming and Evolvable Machines (2003)
3. Goel, S., Bush, S.F.: Kolmogorov complexity estimates for detection of viruses in biologically inspired security systems: a comparison with traditional approaches. Complexity 9(2), 54–73 (2003)
4. Erdős, P., Rényi, A.: On random graphs. Publ. Math. Debrecen 6, 290–297 (1959)
5. Hethcote, H.W.: The mathematics of infectious diseases. SIAM Review 42(4), 599–653 (2000)
6. Forrest, S., Perelson, A.S., Allen, L., Cherukuri, R.: Self-nonself Discrimination in a Computer. In: Proceedings of the IEEE Symposium on Research in Security and Privacy, pp. 202–212. IEEE Computer Society Press, Los Alamitos (1994)
7. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 120–128. IEEE Computer Society Press, Los Alamitos (1996)
8. Hofmeyr, S., Forrest, S.: Architecture for an Artificial Immune System. Evolutionary Computation Journal 8(4), 443–473 (2000)
9. Luther, K., Bye, R., Alpcan, T., Muller, A., Albayrak, S.: A cooperative ais framework for intrusion detection. In: IEEE International Conference on Communications, IEEE Computer Society Press, Los Alamitos (2007)
10. Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM Computing Surveys 36(4), 335–371 (2004)
11. Anderson, I.: Combinatorial designs: construction methods. Ellis Horwood Limited (1990)
12. Stinson, D.R.: Combinatorial designs: construction and analysis. Springer, Heidelberg (2004)
13. Camtepe, S.A., Yener, B.: Combinatorial design of key distribution mechanisms for wireless sensor networks. IEEE/ACM Transactions on Networking 15(2), 346–358 (2007)
14. Linial, N., Wigderson, A.: Expander graphs and their applications. Lecture Notes, Hebrew University, Israel (January 2003)
15. Linial, N.: Expanders, eigenvalues and all that. In: NIPS 2004 Talk (2004)
16. Govindaraju, R.: Design of Scalable Expander Interconnection Networks. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York 12180, USA (1994)
17. Lubotzky, A., Phillips, R., Sarnak, P.: Ramanujan graphs. Combinatorica 8(3), 261–277 (1988)
18. Camtepe, S.A., Yener, B., Yung, M.: Expander graph based key distribution mechanisms in wireless sensor networks. In: IEEE International Conference on Communications, IEEE Computer Society Press, Los Alamitos (2006)
19. Xue, F., Kumar, P.R.: The number of neighbors needed for connectivity of wireless networks. Wireless Networks 10, 169–181 (2004)
20. Draief, M., Ganesh, A., Massoulié, L.: Thresholds for virus spread on networks. In: 1st International Conference on Performance Evaluation Methodolgies and Tools, p. 51 (2006)
21. Wang, Y., Chakrabarti, D., Wang, C., Faloutsos, C.: Epidemic spreading in real networks: An eigenvalue viewpoint. In: 22nd Symposium on Reliable Distributed Computing (2003)

Stabilizing Flocking Via Leader Election in Robot Networks

Davide Canepa and Maria Gradinariu Potop-Butucaru

Université Pierre et Marie Curie (Paris 6), LIP6, CNRS, INRIA, France
canepa.davide@tiscali.it, maria.gradinariu@lip6.fr

Abstract. Flocking is the ability of a group of robots to follow a leader or head whenever it moves in a plane (two dimensional Cartesian space). In this paper we propose and prove correct an architecture for a self-organizing and stabilizing flocking system. Contrary to the existing work on this topic our flocking architecture does not rely on the existence of a specific leader *a priori* known to every robot in the network. In our approach robots are uniform, start in an arbitrary configuration and the head of the group is elected via algorithmic tools.

Our contribution is threefold. First, we propose novel probabilistic solutions for leader election in asynchronous settings under bounded schedulers. Additionally, we prove the impossibility of deterministic leader election when robots have no common coordinates and start in an arbitrary configuration. Secondly, we propose a collision free deterministic algorithm for circle formation designed for asynchronous networks. Thirdly, we propose a deterministic flocking algorithm totally independent of the existence of an *a priori* known leader. The proposed algorithm also works in asynchronous networks.

1 Introduction

Several applications like large-scale constructions, hazardous waste cleanup, space missions or exploration of dangerous or contaminated area motivate the research related to self-organized robot networks (multi-robot systems). The literature proposed so far a significant amount of research towards the operation of a single remote robot, however more work is required towards the operation of networks of autonomous robots. These systems provide interesting solutions to many real problems: manipulation of large objects, system redundancy, reducing time complexity for the targeted tasks, however they bring in discussion some specific difficulties. In particular, these robots should achieve their tasks without human intervention based only on the information provided by the robots in the same group. Moreover, they have to explore unknown or quasi unknown environments while avoiding collisions among themselves. Additionally, they have to be able to reorganize whenever one or more robots in the group stop to behave correctly.

In this paper we propose a self-organized and stabilizing flocking architecture. Flocking is the ability of a group of robots to follow a leader or a flock-head whenever this one changes its position in plane. Our work is developed in

Corda model [1,2] one of the two theoretical models proposed so far for robot networks. The first model proposed in the literature was introduced by Suzuky and Yamashita [3,4,5]. In this model robots are oblivious and perform a cycles of elementary actions as follows : observation (the robot observes the environment), computation (the robot computes its next position based on the information collected in the observation phase) and motion (the robot changes its position by moving to the coordinates returned by the computation phase). In this model robots cannot be interrupted during the execution of a cycle. The Corda model breaks the execution cycle in elementary actions. That is, a robot can be activated/turned off while executing a cycle. Hence, robots are not anymore synchronized.

In both Corda and Suzuki-Yamashita model several problems have been studied under different assumptions on the environment (e.g. schedulers, fault-tolerance), robots visibility, accuracy of compasses: circle formation, pattern formation, gathering [6,7,8,9,10,11,12]. The *flocking problem* although largely discussed for real robots ([13,14] and [15]) was studied from theoretical point of view principally by Prencipe [16,17]. The authors propose non-uniform algorithms where robots play two roles: leader or follower. The leader is unique and all the followers know it. Obviously, when the leader crashes, disappears or duplicates the flock cannot finish its task. Our approach is different, the leader is not known *a priory* but it is elected via algorithmic tools. When the current leader disappears from the system another leader is elected and the network can finish its task. In order to be sound our flocking architecture includes as basic building block a leader election module.

The leader election problem has been studied under a broad class of models. Recent works propose solutions in the population protocol model, [18,19]. The same problem has also been studied in the mobile agents model [20]. These models may seem similar to the robots model however, in these models agents either have a point to point interaction with simultaneous change of their respective state or assume a specific topology of the network guesting the agents (e.g. rings) or make additional assumptions like the existence of whiteboards on the nodes visited by agents. In the robot networks there is no such assumptions since robots move in a Cartesian two dimensional space helped only by the information they can collect at each activation.

In robot networks leader election have been mainly studied in [5]. The authors propose a solution where robots share the same coordinate system. Further in [21] is proposed an algorithm for leader election based on Lyndon words which works if the number of robots is prime and robots are not disposed in a regular n-gon. The previously cited works focus the Suzuki-Yamashita model. In [22] the author prove the leader election impossibility in Corda model when the number of robots is even.

Our contribution. In this paper we propose and prove correct an architecture for a self-organizing and stabilizing flocking system. Contrary to existing work on this topic our flocking architecture does not rely on the existence of a specific leader *a priory* known to every robot in the network. In our approach robots are uniform, start in an arbitrary configuration and the head of the group is elected via algorithmic tools. Our architecture includes three modules: a leader election module, a preprocessing module and a motion module. The leader election module returns to each robot its status : leader or follower. The preprocessing module outputs a moving formation. The motion module provides the rules that will make the robots in the moving formation to change their positions whenever the leader moves. Every modification of robots position preserves the moving formation. For each of these modules we propose deterministic or probabilistic algorithms (in the case when a deterministic solution is impossible). Moreover, we prove their correctness in Corda model. The correctness[1] of the probabilistic algorithms considered in this paper assumes bounded schedulers.

2 Model

The notions and the model description presented in this section are borrowed from [1,11,16]. We consider a system of autonomous mobile robots that work in the Corda model [1]. Each robot is capable of observing its surrounding, computing a destination based on what it observed, and moving towards the computed destination: hence it performs an (endless) cycle of observing, computing, and moving. Each robot has its own local view of the world. This view includes a local Cartesian coordinate system having an origin, a unit of length, and the directions of two coordinate axes (which we will refer to as the x and y axes), together with their orientations, identified as the positive and negative sides of the axes.

The robots are modeled as processes with computational capabilities, which are able to freely move in the plane. They are equipped with sensors that let each robot observe the positions of the others with respect to their local coordinate system. Each robot is viewed as a point, and can see all the other robots in the system.

The robots act totally independently and asynchronously from each other, and do not rely on any centralized directives, nor on any common notion of time. Furthermore, they are oblivious, meaning that they do not remember any previous observation nor computations performed in the previous steps. Note that this feature gives to the algorithms designed in this model the nice property of self-stabilization [24]: in fact, every decision taken by a robot cannot depend on what happened in the system previously, and hence cannot be based on corrupted data stored in its local memory. The robots are anonymous, meaning that they are a priory indistinguishable by their appearances, and they do not have any kind of identifiers that can be used during the computation. Moreover, there are no explicit direct means of communication; hence the only way they

[1] Due to space restrictions, most of the proofs are proposed in the extended version of this work [23].

have to acquire information from their fellows is by observing their positions. The robots are uniform, meaning that they execute the same algorithm, which takes as input the observed positions of the robots, and returns a destination point towards which the executing robot moves.

Schedulers. A scheduler decides at each configuration the set of robots allowed to perform their actions. A scheduler is fair if, in an infinite execution, a robot is activated infinitely often. In this paper we consider the fair version of the following schedulers:

– *k-bounded*: between two consecutive activations of a robot, another robot can be activated at most k times;
– *arbitrary*: at each configuration an arbitrary subset of robots is activated.

In short, robots move asynchronously, are oblivious, anonymous and uniform. Additionally, their activation is managed by a scheduler who decides in each configuration the set of active robots. That is, in this paper we consider the Corda model refined with the above mentioned fair scheduling strategies (i.e. k-bounded and arbitrary).

3 Leader Election and Flocking Problems

Leader election creates an asymmetry whatever the initial configuration. Robots may be in one of the following states: leader or follower and the leader should be unique in the system.

Definition 1 (Leader Election). *A system of robots verifies the leader election specification iff the following two properties hold:*

– *Safety: The system is in a legal configuration where there is an unique robot in the state leader and all the other robots are in the state follower.*
– *Liveness: The legal configuration is reached in a finite number of steps.*

Leader election is the building block for a large class of problems. In this paper we focus on the flocking problem. Intuitively, a flock is a group of robots that moves in the plane in order to execute a task while maintaining a specific formation. The most current definition of the flocking implicitly assumes the existence of an unique leader of the group that will lead the group during the task execution. Robots have as input the same pattern representing the flock to be maintained which is described as a set of coordinates in the plane, relative to a point representing the leader.

Obviously, in order to achieve flocking robots need to re-organize their formation whenever the leader changes its position. Therefore the definition of flocking has to capture the mobility of the flock.

Formally, the flocking problem can be defined as follows:

Definition 2 (Flocking). *Let S be a system of robots and let $\mathcal{P}$ be the flocking pattern. S verifies the flocking specification iff the robots satisfy $\mathcal{P}$ infinitely often.*

4 Architecture of a Flocking System

In the following we define a possible architecture for a flocking system. The architecture is composed of three modules : the leader election module, the preprocessing module and the flocking module.

– The leader election module is the base of the architecture. This module accepts as input a set of robots arbitrarily distributed in the plane and elects a leader. Results related to the impossibility of leader election and detailed description of probabilistic solutions for leader election are proposed in Section 5.
– The preprocessing module prepares the group of robots for the moving formation. All robots but the leader are placed on the smallest enclosing circle. Then, all robots on the smallest enclosing circle form a circular moving formation using as reference point the leader computed by the leader election module. One robot in this set will further act as the head of the flock. The preprocessing module is propose in Section 6.
– The flocking module receives as input a moving formation which initially has a circular form defined by a reference robot and a head and provides the necessary rules to move this formation in the plane whenever the head changes its position. The objective of the flocking module is to ensure the formation moving while keeping its properties. The algorithms for moving the formation are proposed in Section 7.

5 Leader Election Module

In this section we prove the impossibility of deterministic leader election. Generally, the impossibility results can be circumvent by using randomization. In the following we show that probabilistic leader election is impossible for 2 robots systems. However, the probabilistic leader election is possible for systems of size greater than 3.

5.1 Impossibility Results for Leader Election

In this section we prove the deterministic leader election impossible in Suzuki-Yamashita and Corda models.[2]

Theorem 1. *Deterministic leader election is impossible.*

Proof (sketch). Lets consider n robots forming a regular n-gon with the local $x - y$ coordinates of each robot such that the y positive axis is directed towards the next robot in clockwise. Assume also the x positive axis is such that the n-gon has no value of x less than 0. Consider all robots have the same unit of length. Without restraining the generality we consider in the following an equilateral triangle. For a deeper comprehension, lets consider Figure 1.

[2] Note that in [22] is proved the impossibility of leader election for n even, while in [21] is shown that leader election can be deterministically solved for n prime and robots not disposed in a n-gon.

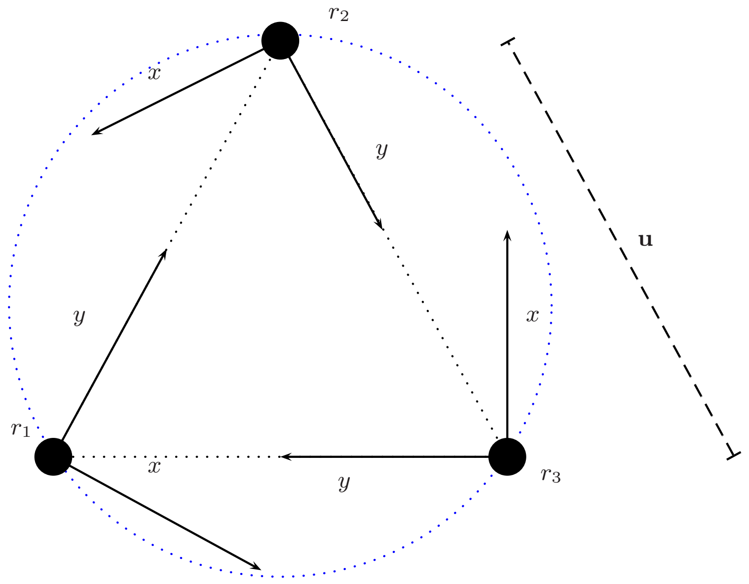


Fig. 1. Symmetric Configuration

Each robot can see a robot in (0,0) (itself) and other two robots in $(\frac{u}{2},\frac{\sqrt{3}}{2}u)$ and in $(u, 0)$. Note that the three robots have the same view.

Assume a configuration such that the leader is the robot in $(u,1)$. In our example for r_1 the leader is r_2, for r_2 the leader is r_3 and for r_3 the leader is r_1. Each robot sees a different leader. Therefore, the safety property is violated.

Assume an initial configuration where there is no leader. In order to reach a legal leader election configuration robots should move. Assume the algorithm executed by each robot makes them move towards a point (x', y') of their system of coordinates and assume the scheduler chooses all robots to move concurrently. The system reaches a configuration where the n-gon structure is maintained. Moreover, in the new configuration robots have the same view. So, each deterministic movement from a symmetric configuration leads to a symmetric configuration. Hence, the system never converges to a legal configuration.

Lemma 1. *There is no probabilistic 2-robots leader election.*

5.2 Probabilistic Leader Election

In this section we propose probabilistic solutions for leader election for systems with three or more robots.

Probabilistic leader election with 3 robots. The algorithm idea is to exploit the asymmetry of a triangle. We choose as leader candidate the robot with the smallest angle or the robot different from the other two robots in the case of an isosceles triangle. The randomization is used only to break the symmetry of equilateral triangles. For this particular case we use randomization in order to create an asymmetric triangle on which we apply the method described above.

```
1) Compute the angle between every two robots.
2) if $my\_angle$ is the smallest then become $Leader$.
3) else if $my\_angle$ is not the smallest but the other two are identical
           then become $Leader$.
4) else if All the angles are identical
           then move perpendicular to segment linking the other
                two robots in opposite direction with probability $\frac{1}{3}$
```

Algorithm 5.1. Leader election algorithm

Lemma 2. *Algorithm 5.1 converges to the leader election specification in finite number of steps in expectation in the Corda model refined with a k-bounded scheduler.*

Probabilistic leader election with more than 3 robots. In the following we propose a leader election algorithm for systems with more than three robots. Intuitively, the leader robot will be the robot whose position is the closest to the center of the smallest enclosing circle (SEC). Additionally, we would like the leader to define a second reference together with the center of SEC. Therefore, the leader should not be placed on the center position. If a robot is initially positioned in the center of the smallest enclosing circle then a preprocessing phase is executed. The robot in the center moves to a free position chosen non-deterministically inside the SEC. The leader election algorithm idea is as follows. Robots randomly change their positions until only one of them is the closest to the SEC.

```
1) Compute the smallest enclosing circle $SEC$.
2) Compute the distance $d\_myself$ to the center of $SEC$.
3) if ($d\_myself < d_k$ $\forall$ k$\neq$ $myself$, where 1$\leq$k$\leq$n )
      then { become leader;
             exit; }
4) if ($d\_myself \leq d_k$ $\forall$ k$\neq$ $myself$, where 1$\leq$k$\leq$n )
      then { move to the center of $SEC$ with probability $p = \frac{1}{n}$ of a
             distance $d\_myself \cdot p$)}
```

Algorithm 5.2. Leader election in systems of size $n > 3$

Definition 3 (Leader election legitimate configuration). *A legitimate configuration for Leader Election is a configuration with an unique robot closest to the center of the smallest enclosing circle.*

Lemma 3. *Algorithm 5.2 converges to a legitimate configuration for the leader election problem in a finite number of steps in expectation in the Corda model refined with a k-bounded scheduler.*

6 Preprocessing Module: Setting a Moving Formation

In this section we gradually set the motion pattern used further in the flocking algorithm. We build on top of the leader election algorithms proposed in Section 5.2.The construction takes two phases. First, all robots but the leader are placed on the smallest enclosing circle. Then, the robots on the circle will be placed in their final positions for motion.

6.1 Phase 1: Placement on the Smallest Enclosing Circle

In this section we propose an algorithm for placing robots on the smallest enclosing circle. This algorithm uses as building block the leader election algorithm proposed in the previous section. Once this algorithm is stabilized all robots but the leader are placed on the smallest enclosing circle. Note that the leader does not change during this phase.

The algorithm works "in waves". First, the robots closest to the bounds of the smallest enclosing circle are placed. Then, recursively all the other robots but the leader are placed. The robots that should occupy a position that is already occupied by another robot will be placed on a free position between the robot that occupied their position and the next one on the smallest enclosing circle. We assume the robots agree on the same direction of the Ox axis given by the center of SEC and the leader position and the same direction of Oy axis. Our algorithm is collision free and works in the Corda model with arbitrary fair scheduler. Note that in [25] the authors propose similar deterministic solutions for Suzuki-Yamashita model. Interestingly, our algorithm has the same time complexity as the solution proposed in [25].

The following definitions introduce key functions used by Algorithm 6.1.

Definition 4 (FreeToMove). *Let $FreeToMove$ be the set of robots without robots between themselves and the SEC (including the border) along the radius passing through them, and that do not belong to the border of SEC.*

Definition 5 (Placed). *Let Placed be the set of robots belonging to the border of the SEC.*

Definition 6. *A legitimate configuration for Algorithm 6.1 is a configuration where all robots but the leader are Placed.*

Note that the algorithm does not change the leader position neither the position of *Placed* robots. Moreover, there is no robot between the leader and the SEC. Otherwise this robot is the closest to the center of the SEC hence the real leader.

The correctness of Algorithm 6.1 comes from the following lemmas.

```
$\forall r_i$ compute the value of the radius passing through $r_i$. Let $rad_{r_i}$ be the value
of the angle between my radius ($rad_{myself} = 0$) and the radius of robot $r_i$, in
clockwise direction
$\forall r_i$ compute the value of $dist_{r_i}$, distance of the robot $r_i$ to the border of the
smallest enclosing circle ($SEC$)

Predicates:
$Leader(myself) \equiv \forall r_i$ with $i \neq myself$, $dist_i < dist_{myself}$

Functions:
$OccupiedPosition(rad_{myself})$ : returns $r_i, i \neq myself, dist_{r_i} = 0$ and $rad_{r_i} = rad_{myself}$ otherwise $\perp$
$NextToMove$ : returns the set of closest robots $r$ to the $SEC$ with $dist_r \neq 0$

1) if ( $\neg Leader(myself) \wedge myself \in FreeToMove$)
then { move to $SEC$ with distance $dist_{myself}$}
2) if ($\neg Leader(myself) \wedge (myself \in NextToMove) \wedge (FreeToMove = \emptyset) \wedge (OccupiedPosition(rad_{myself}) \neq \perp)$)
then { Move to the middle point of the arc between robot OccupiedPosition($rad_{myself}$) and robot $r_j$ belonging to the $SEC$ such that $rad_j$ is minimum.}
```

Algorithm 6.1. Positioning Algorithm executed by robot *my_self*

Lemma 4. *If two robots r_i and r_j belong to the set $FreeToMove$, then their final position will be different.*

Lemma 5. *A robot always moves towards a free position on the SEC.*

Lemma 6. *Algorithm 6.1 is collisions free (two robots never move towards the same free position).*

Lemma 7. *Algorithm 6.1 converges in a finite number of steps, $O(n)$, to a legitimate configuration.*

6.2 Phase 2: Setting the Flocking Configuration

In this section we propose an algorithm that starting from the final configuration of Algorithm 6.1 reaches a flocking pattern or moving formation having the singularity property detailed later.

Initially, we place robots in a circular moving formation then in the final moving formation. The circular moving formation has the following shape: r_0 is inside SEC (the one computed by Algorithm 6.1) and all the other robots are placed on its border. These robots are placed as follows: a robot r_1 is in the position $SEC \cap [O, r_0)$ and the others, uniformly disposed on the semi-circle that does not contain r_1 and that ends in the points given by the intersection of SEC with the perpendicular on $[O, r_0)$ that passes through O ($SEC \cap (\perp [O, r_0)$ *on* O)). In the following this configuration will be referred as *circular moving formation* (see Figure 2 for a seven robots example).

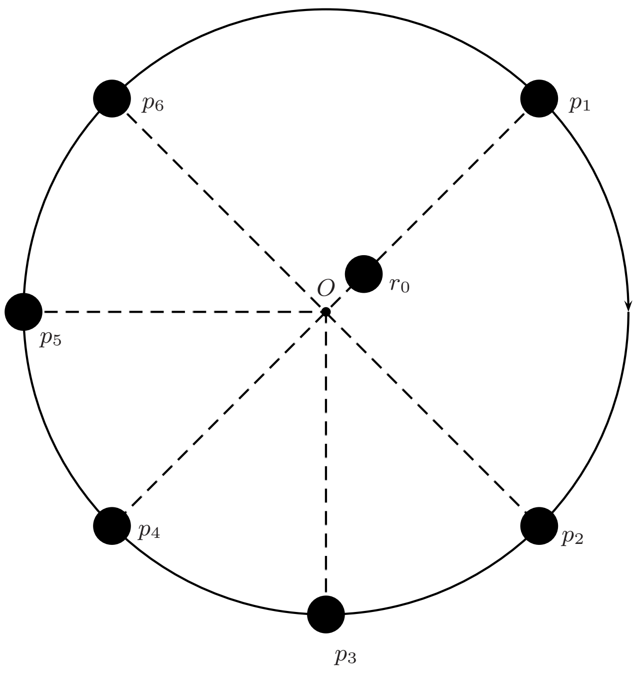


Fig. 2. Circular moving formation

In order to construct the circular moving formation we use the concept of oriented configuration [21]:

Definition 7 (oriented circular configuration). *A configuration is called circular oriented if the following conditions hold:*

1. *All robots are at distinct positions on the same circle SEC, except only one of them, called r_0, located inside SEC ;*
2. *r_0 is not located at the center of SEC;*

Note Algorithm 6.1 verifies point 1 of the above definition and is collisions free contrary to the solution proposed in [21]. Note also the leader election algorithm chooses a leader such that it is the closest to the center of SEC without reaching this center. If the leader is initially in the center, we recall that a preprocessing is performed in order to take care of this particular case. The leader election algorithm is executed only after the end of the preprocessing phase.

We now describe Algorithm 6.2. The algorithm makes use of the following function: FinalPositions(SEC, p_1). This function returns, when invoked by a robot, the set of positions in the circular moving formation with respect to SEC and the point p_1. p_1 is the intersection between the segment $[O, r_0)$ and the circle SEC. The order of positions and robots is given clockwise starting with position p_1. Started in an oriented configuration Algorithm 6.2 eventually converges to a configuration where robots are disposed on SEC following the restrictions imposed by the $FinalPositions$ function.

```
Functions:
    $get\_number(myself)$ returns the number of robots between $myself$ and
                position $p_1$ (including robot myself) clockwise
    $get\_position(myself)$ returns the position $get\_number(myself)$
                in $FinalPositions(SEC, p_1)$
    $FreeToMove(myself)$ returns true if there are no robots between $myself$
                and $get\_position(myself)$

Motion Rule:
        if FreeToMove(myself) then
            move to $get\_position(myself)$
```

Algorithm 6.2. Setting the moving formation executed by robot $myself$

The idea of the algorithm is as follows. Robots started in an oriented configuration reach their final positions. If a robot is blocked by some other robot then it waits until this robot is placed in its final position. In the following we prove no robot is blocked infinitely.

Lemma 8. *In a system with n robots Algorithm 6.2 started in an oriented configuration converges in finite number of steps, $O(n)$, to a configuration where all robots reach their final positions computed via FinalPositions function.*

We formally define the moving formation as follows:

Definition 8 (moving formation). *A set of $n > 4$ robots, $r_0, \dots r_n$, is a moving formation if:*

- *r_1 and r_0 define the Oy axis of the system such that: the y coordinate of r_0 equals 0 and the positive values are in the r_1 direction;*
- *the axis Ox is perpendicular to Oy in r_0 and has positive values at the right of Oy;*
- *all the other robots are such that:*

  1. *$\forall\ r_i \neq r_l$ and $r_i \neq r_0 \Rightarrow y_{r_i} < 0$*
  2. *$\forall r_i, r_j,\ x_{r_i} \neq x_{r_j}$;*
  3. *$\forall\ r_i,\ \exists\ r_j$ such that $x_{r_i} = -x_{r_j}$*
  4. *if $|x_{r_i}| > |x_{r_j}|$ then $|y_{r_i}| < |y_{r_j}|$*
  5. *there exists an unique robot with $x = 0$ and $y < 0$*

The following theorem states the singularity property of the moving formation defined above. More precisely, we show that there is only one formation that satisfies Definition 8 when $n > 4$. Note that for the case $n \leq 4$ the formation defined by Definition 8 is not unique. In the sequel we consider systems with more than 4 robots. For the case $n \leq 4$ simple adhoc algorithms can be designed on top of the algorithms proposed in Section 5.2.

Theorem 2. *The moving formation defined by Definition 8 is singular when* $n > 4$.

Corollary 1. *Algorithm 6.2 started in an oriented configuration eventually places n robots in a circular moving formation if FindPositions returns a set of positions verifying Definition 8.*

7 Flocking Module

In this section we propose a flocking algorithm. The flock of robots verifies the moving formation defined in Definition 8 and follow the head robot (the robot referred as robot r_1) whenever this head changes its position. In the following the robot r_0 of the moving formation will be called *reference robot* and the robot r_1 *leader*. The only constraint imposed to the system is: the leader cannot move quicker than the slowest robot in the set. The algorithm idea is as follows. When the head of the group moves, it is followed within a distance δ (a parameter of the algorithm) by the reference robot. Then the closest robots to the reference move within a distance ϵ (another parameter of the algorithm) to the reference and so on till all the robots in the group move. Note the algorithm has three parameters: the speed of the leader, the distance between the leader and the reference and the distance ϵ between the successive rows of robots. The moving formation can be seen as a virtual tree where levels are linked to each other via virtual springs (Figure 3).

```
Input: $r_0, r_1 \dots r_n$ a moving formation

Functions:
        $TheMostExterior(r_{myself})$ returns true if $|x_{r_{myself}}| \geq |x_{r_i}| \; \forall r_i$
        $YClosestExterior(r_{myself})$ returns the y coordinate of $r_{ext}$, the robot
        such that $(|x_{r_{ext}}| - |x_{r_{myself}}|)$ is minimum and positive

  1. if ($r_{myself} == r_1$): { move ahead at a speed $< v_{max}$ }, $v_{max}$ is a
     parameter of the algorithm

  2. if ($r_{myself} == r_0$): { follow the leader within a distance $\delta$ }

  3. if ($r_{myself} \neq r_0, r_1$ & $TheMostExterior(r_{myself})$ ): { move ahead
     following $y = y_{r_j}$ towards the point ($x_{r_j}$, -$\epsilon$); }

  4. ($r_{myself} \neq r_0, r_1$ &$\neg TheMostExterior(r_{myself})$): { move ahead following
     $y = y_{r_j}$ towards the point ($x_{r_j}$, $YClosestExterior(r_{myself}) - \epsilon$); }
```

Algorithm 7.1. Flocking executed by robot r_{myself}

Lemma 9. *Algorithm 7.1 preserves the moving formation (Definition 8).*

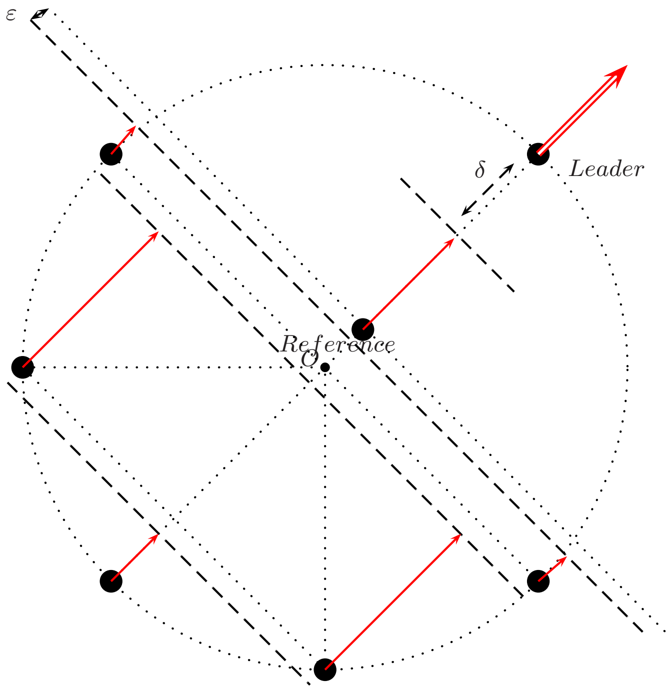


Fig. 3. Animation for Algorithm 7.1

8 Conclusions and Open Problems

In this paper we proposed an architecture for building a self-organizing and stabilizing flocking architecture. Contrary to the existing work on this topic our flocking architecture does not assume the existence of a specific leader a priory known to the robots in the network. In our approach robots are anonymous and uniform.

Our architecture includes three modules: a leader election module, a preprocessing module and a motion module. For each of these modules we propose deterministic or probabilistic algorithms (in the case when deterministic solutions are impossible).

This work can be seen as a preliminary study for the design of a general fault-tolerant flocking architecture where the group of robots verify a generic pattern and follow the head whatever its direction. Additionally, we currently investigate probabilistic algorithms that improve the leader election part of our architecture. In particular, we are looking for leader election solutions in the Corda model refined with an arbitrary scheduler. The idea of these algorithms is to use analogical strategies for election. That is, the robots candidate to a leader position choose probabilistically a free position on their corresponding radius in the smallest enclosing circle.

Acknowledgments

The authors would like to thank Xavier Defago for detailed discussions related to the current work, the suggested open problems and the various hints to solve these problems as discussed in Section 8. We also would like to thank Sebastien Tixeuil and anonymous reviewers for all their comments that greatly improved the presentation of the current work.

References

1. Prencipe, G.: Corda: Distributed coordination of a set of autonomous mobile robots. In: Proc. ERSADS 2001, pp. 185–190 (2001)
2. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Distributed coordination of a set of autonomous mobile robots. In: IVS 2000. IEEE Intelligent Vehicles Symposium, pp. 480–485 (2000)
3. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots—formation and agreement problems. In: SIROCCO 1996. Proceedings of the 3rd International Colloquium on Structural Information and Communication Complexity (1996)
4. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM Journal on Computing 28(4), 1347–1363 (1999)
5. Suzuki, I., Yamashita, M.: A theory of distributed anonymous mobile robots formation and agreement problems. Technical report, Wisconsin Univ. Milwakee, Dep. of Electrical Engineering and Computer Science, 6 (1994)
6. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous mobile robots with limited visibility. Theoretical Computer Science 337, 147–168 (2005)
7. Souissi, S., Défago, X., Yamashita, M.: Eventually consistent compasses for robust gathering of asynchronous mobile robots with limited visibility. Research Report IS-RR-2005-010, JAIST, Ishikawa, Japan (2005)
8. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed memoryless point convergence algorithm for mobile robots with limited visibility. IEEE Trans. on Robotics and Automation 15(5), 818–828 (1999)
9. Cohen, R., Peleg, D.: Convergence of autonomous mobile robots with inaccurate sensors and movements. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 549–560. Springer, Heidelberg (2006)
10. Agmon, N., Peleg, D.: Fault-tolerant gathering algorithms for autonomous mobile robots. In: SODA 2004. Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1070–1078 (2004)
11. Défago, X., Gradinariu, M., Messika, S., Parvédy, P.R.: Fault-tolerant and self-stabilizing mobile robots gathering. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 46–60. Springer, Heidelberg (2006)
12. Kasuya, M., Ito, N., Inuzuka, N., Wada, K.: A pattern formation algorithm for a set of autonomous distributed robots with agreement on orientation along one axis. Systems and Computers in Japan 37(10), 89–100 (2006)
13. Qadi, A., Huang, J., Farritor, S.M., Goddard, S.: Localization and follow-the-leader control of a heterogeneous group of mobile robots. IEEE/ASME Transactions on Mechatronics 11, 205–215 (2006)
14. Renaud, P., Cervera, E., Martiner, P.: Towards a reliable vision-based mobile robot formation control. In: IROS. IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3176–3181 (2004)

15. Lindhe, M.: A flocking and obstacle avoidance algorithm for mobile robots. PhD thesis, KTH Stockholm (2004)
16. Gervasi, V., Prencipe, G.: Flocking by a set of autonomous mobile robots. Technical Report TR-01-24, Universitat di Pisa (2001)
17. Gervasi, V., Prencipe, G.: Coordination without communication: the case of the flocking problem. Discrete Appl. Math. 144(3), 324–344 (2004)
18. Angluin, D., Aspnes, J., Fischer, M., Jiang, H.: Self-stabilizing population protocols. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 79–90. Springer, Heidelberg (2006)
19. Fischer, M., Jiang, H.: Self-stabilizing leader election in networks of finite-state anonymous agents. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 395–409. Springer, Heidelberg (2006)
20. Barri, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Electing a leader among anonymous mobile agents in anonymous networks with sense-of-direction. Technical Report 1310, Technical Report LRI, Laboratoire de recherche en Informatique, Université Paris-Sud, France, (April 2002)
21. Dieudonne, Y., Petit, F.: Circle formation of weak robots and lyndon words. Inf. Process. Lett. 101(4) (2007)
22. Prencipe, G.: Achievable patterns by an even number of autonomous mobile robots. Technical Report TR-00-11, Universitat di Pisa (2000)
23. Canepa, D., Gradinariu, M.: Stabilizing flocking via leader election in robot networks. Technical Report 6268, INRIA, France (2007)
24. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
25. Suissi, S.: Fault resilient cooperation of Autonomous Mobile robots with unreliable compass sensors. PhD thesis, JAIST, Japon (2007)

Stabilization in Dynamic Systems with Varying Equilibrium

Hui Cao and Anish Arora

Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210
{caohu,anish}@cse.ohio-state.edu

Abstract. System design often explores optimality of performance. What is optimal is, however, often not predefined or static in most cases, because it is affected by the context of operation, such as the environment or external system inputs. In this paper, we formulate the maintenance of optimality of performance in dynamical systems in terms of the standard notion of stabilization. For systems with observable external inputs and computable optimality, stabilization may be achieved by adding a stabilizing input estimator to the system. But environments and external inputs are often unobservable. To overcome this difficulty, we present two alternative methods, one based on a game-theoretic MinMax strategy that leads to Nash equilibrium, and the other based on a feedback control mechanism that adds a stabilizing output transformer to the system. We exemplify these two approaches with a pursuit-evasion application and a MAC layer duty cycle adaptation protocol, respectively.

Keywords: Stabilization, MiniMax, Optimality, Equilibrium, Control.

1 Introduction

System design often explores optimality of systems, as measured in terms of some performance metric(s). However, these optimal states are not predefined or static in many systems because they are affected—or even decided—by the context of operation, such as the environment or external system inputs. When operating context changes, the equilibrium state(s) at which optimal performance is achieved also change, which requires the system to reestablish equilibrium continuously. By way of example, fault occurrences, environmental parameter changes, and new user inputs/traffics can each leave the system in a suboptimal state, as the optimal system state is often a single equilibrium point or a narrow region of points. These considerations motivate the importance of designing the property of stabilization in systems with varying equilibrium.

The standard notion of the stabilization of a system implies that regardless of the current state of the system, each computation of the system will eventually converge to a legal state in a finite number of steps and henceforth the system will continue to operate as specified. In the context of systems that seek to maintain optimal performance, their legal states must satisfy the desired optimality. Stabilization of these systems thus implies that when the current state of the system becomes suboptimal, for instance as a result of change in operating context, they will eventually resume optimal performance.

Designing stabilization to ensure optimality is however not an easy task. We attribute this to two facts: (a) Dynamically varying equilibrium: Equilibrium is often not determined by system itself, it is also affected by the operating context. Thus, when operating context changes, equilibrium can also vary. (b) Difficulty of detecting optimality: To determine optimality, all possible outcomes may need to be compared. When several equilibria coexist in a system, local detection of optimality can often leave the system in a local maximum (or minimum). In addition, optimality detection is sensitive to noise. An output spike introduced by noise may be falsely regarded as a maximum.

Our goal in this paper is to investigate new system design methodologies that exploit stabilization techniques to maintain optimality despite changes in the context of system operation.

1.1 Summary of Our Results

In this paper, we study classes of stabilization problems for systems that achieve optimality. Based on a general system model, we provide three techniques that achieve stabilization.

1. For systems with observable external inputs, we add a stabilizing estimator to the system, using which the system infers the external inputs. Since optimality is determined by external inputs and system, when the system is controllable, we can achieve optimal performance by incorporating the estimated values of the external inputs.
2. For systems with unobservable external inputs, we suggest the design of MinMax controller strategies. We prove whenever a MinMax strategy leads to Nash equilibrium, stabilization (to the Nash equilibrium) is achieved. We illustrate this technique via an optimal catch time pursuit-evasion application where the design of a MinMax strategy for a pursuer guarantees stabilization irrespective of the evader's choice (or change) of strategy.
3. When outputs are observable, we suggest the design of a feedback control module. To eschew the difficulty of optimality detection, we add a stabilizing transformer to the system, using which an optimization problem is transformed into a fixed point control problem. We illustrate this technique via an optimal energy-efficiency duty cycle MAC design that uses a feedback control algorithm to guarantee stabilization.

1.2 Related Work

Reactive systems [1] (also called open systems) are systems whose role is to maintain an ongoing interaction with their environment, as opposed to calculating a final value upon termination. The literature on stabilization has considered reactive systems in a number of ways, of which we recall a few. In [2], an adaptive program is defined as a program that changes its behavior based on current state of environment. Operators that compose adaptive programs are developed in that work for both sequential and distributed program classes. A formal definition of stabilization in the presence of changes in operating context and general classes of faults is given in [3], in terms of closure and convergence. In this paper, we also use closure and convergence properties to define stabilizing optimality. [4] focuses on the adaptive stabilization of reactive distributed protocols; it shows that general reactive systems can be implemented in an adaptive way, i.e., the recovery time of stabilizing protocols can be proportional to the number of faults.

In [5], both termination and stabilization are investigated in message-passing systems relative to external input. In [6], stabilization of majority consensus is presented. [7] focuses on mutual exclusion. Both [5] [6] and [7] provide solutions for specific reactive systems. Control theories such as Lyapunov Theory are applied in [8],[9] to explore its application in stabilization. Some new classes of stabilization such as Self-organizing [10][11] and Selfish stabilization [12] are related with game theory approaches.

Compared with current work, we may identify two distinguishing features of our study: (a) Optimality: the set of legal states, which the system should converge upon starting from an arbitrary state, are characterized by one or more optimal properties of interest. (b) Reactivity: as the external inputs changes continuously, the corresponding equilibrium states can vary continuously as well.

1.3 System Model

As shown in Figure 1(a), our system model consists of four major components: external input, system controller, internal subsystem, and faults.

- **Controller:** The system controller is the component that manages the computation of the system. If protocols or algorithms are added to the system to achieve stabilization, their execution is controlled by this component.
- **Internal subsystem:** The internal subsystem accepts commands or data from controller to change its behavior. However, its outputs may also be affected by when external inputs change and/or faults occur. The distinction between internal subsystem and controller is that internal subsystem is governed by its inherent mechanism. In control theory, internal subsystem is called "plant".
- **External inputs:** Based on their influence on the equilibrium states, we choose to classify the operating context into two parts: external inputs and faults. External inputs are defined as that part of the operating context that can directly impact system equilibrium. If external inputs are known, they together with controller, uniquely decide the equilibrium of system.
- **Faults:** In contrast to external inputs, faults affect system equilibrium arbitrarily or transiently. In this paper, we assume that when faults occur the net effect is to perturb the system into a potentially arbitrary state. The goal of stabilization then is to subsequently ensure that continued computation of the system will converge to a legal (i.e., optimal) state eventually (and ideally in a timely manner).

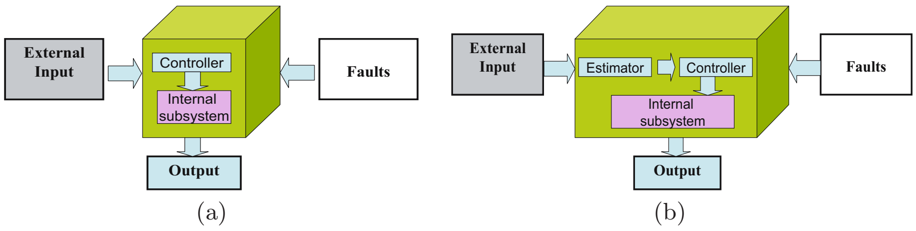


Fig. 1. (a) The system model (b) System stabilization with observable inputs

In summary, our model defines several key components that affect system equilibrium. Many systems such as control system can be generalized by this model.

Notations. We will use the following notations in the rest of the paper.

i: External inputs	I: The set of all possible external inputs
c: Controller value	C: The set of all possible controller values
y: Outputs	Y: The set of all possible outputs

A steady state means a stable condition that does not change over time or in which change in one direction is continually balanced by change in another. We assume that in steady states, external inputs and the controller value determine the outputs through a system function, f, that is, $y = f(i, c)$.

Definition 1. *We say that the system is in an optimal state for a given choice of external inputs $i \in I$ and controller value c iff the system output $f(i, c) = \max\{f(i, c')|(c' \in C)\}$.*

Note that we use max to indicate optimization. However, min and other types optimization can be achieved through a general system function, f.

Definition 2. *We say that the system has computable optimality iff there exists a known function O such that for any choice of external inputs $i \in I$, the system output $f(i, O(i)) = \max\{f(i, c')|c' \in C\}$.*

Definition 3. *We say that external inputs are observable iff they can be measured directly or inferred from outputs.*

Definition 4. *We say that a system has stabilizing optimality iff it satisfies two conditions. (a) Closure: if the system state is optimal, it will remain optimal, unless faults occur or external input changes. (b) Convergence: upon starting from an arbitrary system state, every computation of the system will eventually converge to an optimal state.*

In the following sections, we will illustrate three techniques that focus on external inputs, controller, and output respectively to achieve stabilizing optimality.

2 Stabilizing Optimality Via a Stabilizing Estimator

If external inputs are observable, an estimator can be added into the system, see Figure 1(b). This estimator would execute as an independent program and output a value that estimates the external inputs, by filtering noises in measurements of the input. Examples of estimator include Kalman filters, Wiener filters, and maximum likelihood estimators (MLE). If system has computable optimality as Definition 2, the controller can determine the equilibrium through the known function O, using the estimated value as opposed to the ideal external inputs. The estimator must however be self-stabilizing, to deal with situations when the estimators state itself is corrupted by fault occurrence.

Lemma 1. *If the added estimator is stabilizing and system has computable optimality, the system has stabilizing optimality.*

Using this approach, the stabilizing optimality of a system is achieved through continuous self-stabilizing estimation. (a) Closure: when a system is in equilibrium, if it has computable optimality and its estimation is correct, the system function O will compute the same equilibrium. (b) Convergence: If the system is in an arbitrary state, continued computation of the estimator would eventually lead to the external inputs being estimated correctly, and the system function O will the re-establish the equilibrium.

However, the technique of adding stabilizing estimators is prone to several vulnerabilities. Firstly, the estimator must closely follow the dynamics of external inputs. Inaccurate or severely delayed estimation may produce suboptimal output. In addition, ensuring that the estimator is stabilizing can be nontrivial especially if it depends on system history which may also get corrupted. The implementation of the estimator should also be stabilizing, i.e., implementations of the estimator may introduce their own internal states. When faults happen, those internal states may be corrupted, and the estimator must recover from these as well. Furthermore, calculating the function $c^* = O(i)$ may not be trivial in all applications. Next, we will provide two techniques that avoid those shortcomings.

3 Stabilizing Optimality Via MinMax Strategies

In game theory, the payoff for a player depends on the choices made by other players. We may model the interaction among external inputs, controller, and faults as players in a game, as shown in Figure 2(a). When external inputs and faults are unpredictable, the game becomes a non-cooperative one. In non-cooperative games, simultaneous actions from other players (external inputs and faults) are unobservable. Therefore, maintaining optimal performance based on an estimator is infeasible.

To deal with unobservable simultaneous actions from external inputs and faults, our technique is to design a controller based on MinMax strategy. MinMax is a method in decision theory for minimizing the maximum possible loss.

It can alternatively be thought of as maximizing the minimum gain (MaxMin). In our system model, the MinMax strategy for the controller would minimize the maximum loss introduced by external inputs and faults. This is a conservative approach, but stabilization can be achieved when MinMax strategies lead to Nash equilibrium, as explained below. Nash equilibrium is a solution concept for a game where a player has no gain by changing its own strategy unilaterally. Specifically, if no player can benefit by changing its strategy while the other players keep theirs unchanged, then the current set of strategies and the corresponding payoffs constitute a Nash equilibrium.

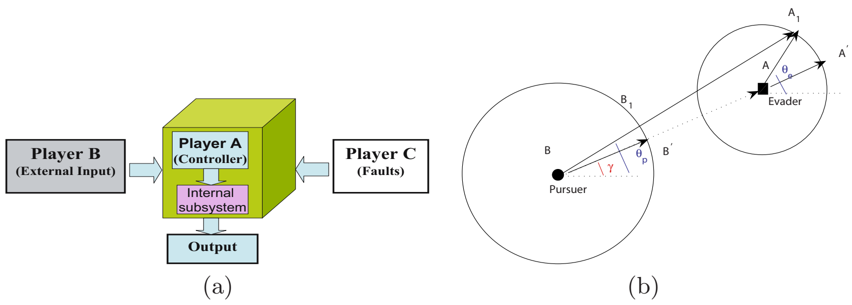


Fig. 2. (a) System model in terms of game theory (b) Pursuit-evasion game for target capture

Theorem 1. *When a Controller adopts a MinMax strategy, the system has stabilizing optimality iff the MinMax strategy leads to Nash equilibrium.*

Proof. We combine the effects of external inputs and faults as one external player input e, where $e \in E$, E is the set of all possible inputs. The external player input e and controller value determine outputs through the system function f, that is, $y = f(e, c)$. Without loss of generality, we assume that the MinMax strategy of the Controller is $\max_{c \in C} \{\min_{e \in E} f(e, c)\}$, while the corresponding strategy of external player is $\min_{e \in E} \{\max_{c \in C} f(e, c)\}$

1. If the MinMax strategy leads to Nash equilibrium, by the definition of Nash equilibrium, $\exists (e^*, c^*)$, such that for $\forall e \in E, c \in C, f(e^*, c) \le f(e^*, c^*) \le f(e, c^*)$. (a) Closure: when $(e, c) = (e^*, c^*)$, since Controller adopts a MinMax strategy, it will output c^*. For external players, they have no incentive to change to other strategies, because $f(e^*, c^*) \le f(e, c^*)$. (b) Convergence: when $(e, c) \ne (e^*, c^*)$, since Controller adopts a MinMax strategy, it will output c^*. For external players, the best option is to get the strategy to output e^*, because when $f(e^*, c^*) \le f(e, c^*)$, $\min_{e \in E} f(e, c^*) = f(e^*, c^*)$. Therefore, the system stabilizes.
2. If the MinMax strategy stabilize to a state with outputs (e', c'), according to definition of MinMax strategies, $\min_{e \in E} f(e, c') = f(e', c') = \max_{c \in C} f(e', c)$. Because of convergence, we have $\forall e \in E, c \in C, f(e', c) \le f(e', c') \quad \wedge \quad f(e', c') \le f(e, c')$. Therefore, (e', c') is a Nash equilibrium.

MinMax strategy and Nash equilibrium are two different solution concepts in game theory. MinMax strategy cannot guarantee Nash equilibrium, while Nash equilibrium is not necessarily derived from MinMax strategy. As far as we know, Theorem 1 is the first result to link them together via stabilization.

3.1 Case Study: Pursuit-Evasion Game (PEG)

We illustrate the MinMax strategy technique via a target capture game, one application based on our 1000+ nodes Exscal project in DARPA-NEST program [13][14]. In this game, the pursuer (controller) tries to catch the evader (external input) as soon as possible, while the evader tries to prolong the time taken to be caught (catch time) T_c, the output of this system. This is a zero-sum game. Zero-sum describes a situation in which a player's gain or loss is exactly balanced by the losses or gains of the other player(s).

Model: We denote the current time as t. The list of state variables in this game is the following:

– $L_p(t) = (x_p, y_p)$: pursuer location, $L_e(t) = (x_e, y_e)$: evader location
– V_p: the maximal pursuer speed, V_e: the maximal evader speed

When evader is caught at time t_c, the catch time is calculated by: $T_c = t_c - t$. The payoff for pursuer in this game, $\mathcal{J}_p$, is defined as: $\mathcal{J}_p = -T_c$, while the payoff for evader in this game, $\mathcal{J}_e$, is defined as : $\mathcal{J}_e = T_c$. We define the distance between the pursuer and the evader as: $dist(t) = \|L_p(t) - L_e(t)\| = \sqrt{(x_p - x_e)^2 + (y_p - y_e)^2}$.

MinMax strategies: Every Δt time interval, both the pursuer and the evader should respectively move to the location based on their own MinMax and MaxMin strategy. To design a MinMax strategy for the pursuer, we should consider the best actions of an evader. Figure 3 describes the pursuer strategy (proof details can be found in [15]). Literally, the pursuer strategy is to move towards to evader with full speed in the next time interval $[t, t + \Delta t]$.

Input: (x_e, y_e)
Output: $(x_p(t + \Delta t), y_p(t + \Delta t))$
Parameter: V_p, V_e
Internal state: (x_p, y_p)
$x_p(t + \Delta t) = x_p + \frac{x_e - x_p}{dist(t)} \cdot V_p \Delta t$
$y_p(t + \Delta t) = y_p + \frac{y_e - y_p}{dist(t)} \cdot V_p \Delta t$

(a) Pursuer strategy

Input: (x_p, y_p)
Output: $(x_e(t + \Delta t), y_e(t + \Delta t))$
Parameter: V_p, V_e
Internal state: (x_e, y_e)
$x_e(t + \Delta t) = x_e + \frac{x_e - x_p}{dist(t)} \cdot V_p \Delta t$
$y_e(t + \Delta t) = y_e + \frac{y_e - y_p}{dist(t)} \cdot V_p \Delta t$

(b) Evader strategy

Fig. 3. Pursuer strategy and Evader strategy

Stabilization of strategies: The MinMax strategy of pursuer provides robustness against uncertainty of evader strategy:

Theorem 2. *In PEG, if the pursuer follows the actions described by Figure 3, the following inequality holds even if the evader changes its strategy.*

$$T_c(t) \geq T_c(t + \varDelta t) + \varDelta t, \qquad T_c(t) = \frac{dist(t)}{V_p - V_e}$$

Proof. As shown in Figure 2(b), after time interval $\varDelta t$, the pursuer will move to B'. In this case, the best option for evader is to move to A', because only in this case does the following equation holds: $T_c(t) = T_c(t + \varDelta t) + \varDelta t$. Otherwise, assuming the evader moves to any location A_1 other than A', then by triangle inequality: $\overline{BA} + \overline{AA'} > \overline{BA_1} \Rightarrow \overline{BA} + \overline{AA'} > \overline{B_1A_1} + \overline{BB_1}$. In other words, $dist(t) > dist(t + \varDelta t) + (V_p - V_e)\varDelta t \Rightarrow T_c(t) > T_c(t + \varDelta t) + \varDelta t$.

Similarly, the following inequality also holds:

Theorem 3. *In PEG, if the evader follows the actions described by Figure 3, the following inequality holds, even if the pursuer changes its strategy.*

$$T_c(t) \leq T_c(t + \varDelta t) + \varDelta t, \qquad T_c(t) = \frac{dist(t)}{V_p - V_e}$$

The pursuer strategy and the evader strategy form a Nash equilibrium (as we will prove next) that is based on MinMax strategy, thus they achieve stabilizing optimality of the game.

Theorem 4. *In PEG, if the pursuer follows the MinMax strategy described by Figure 3, the system has stabilizing optimality.*

Proof. Firstly, we prove that the MinMax strategies of the pursuer and the evader lead to Nash equilibrium. We denote the possible strategies of pursuer and evader as $(a_p, a_e), a_p \in P, a_e \in E$, and the actions in Figure 3 as (a_p^*, a_e^*) separately.

From Theorem 2, we have: $\min_{a_e \in E} T_c(a_p^*, a_e) = T_c(t) = T_c(a_p^*, a_e^*)$. From Theorem 3, we have: $\max_{a_p \in P} T_c(a_p, a_e^*) = T_c(t) = T_c(a_p^*, a_e^*)$. Therefore, $T_c(a_p, a_e^*) \leq T_c(a_p^*, a_e^*) \leq T_c(a_p, a_e^*)$. The MinMax strategies of the pursuer and the evader thus lead to Nash equilibrium. From Theorem 1, Nash equilibrium leads to the stabilization of system.

Stabilization of implementation of strategies: When the strategy is implemented as a program, which may introduce additional states, to ensure the stabilization of the system, we must consider the stabilization of the implementation. In our case study, the state information–location of pursuer and evader can be corrupted. The optimal pursuer strategy is however based solely on the latest location information, and is thus independent of history information. If the state information is corrupted, the pursuer should continue to query for the latest location and move according to its optimal strategy. After it receives the correct location information, Nash equilibrium is reestablished. In other words, it is straightforward to implement this strategy as a program that is stabilizing.

3.2 Discussion

When MinMax strategies do not suffice to derive Nash equilibrium, mixed strategies can be applied. A mixed strategy is to choose randomly between different strategies based on calculated weighted possibilities. The celebrated Minmax theorem states that a solution for mixed MinMax strategies for two players always exists and the solution is always a Nash equilibrium. Therefore, by using mixed MinMax strategies, we can always obtain stabilizing solutions. We suggest that MinMax is probably the best available strategy for a wide range of zero-sum games. MinMax has been applied in many applications to deal with uncertainties. In [16] the MinMax approach to the design of systems that are robust with respect to modeling uncertainties is studied, and the efficacy of the methods proposed for a general game is validated for the case of problems of matched filtering, Wiener filtering, quadratic detection, and output energy filtering. We also applied MinMax to a much more complex application–asset protection game in [14].

4 Stabilizing Optimality Via Feedback Control

When external inputs are difficult to estimate or are unobservable, but outputs are measurable, one approach to designing system optimality is via output feedback. There are, however, two major issues with this approach: (a) Difficulty of determining equilibrium: As we noted before, equilibrium is not predefined in many cases. It may therefore necessitate the use of optimization procedures. These procedures may however be of high complexity or be error-prone when equilibrium varies. (b) Stability of feedback loop: This may be difficult to achieve owing to delay and uncertainty of feedback loop. Uncertainty is inherent when the system model is inaccurate or when faults occur.

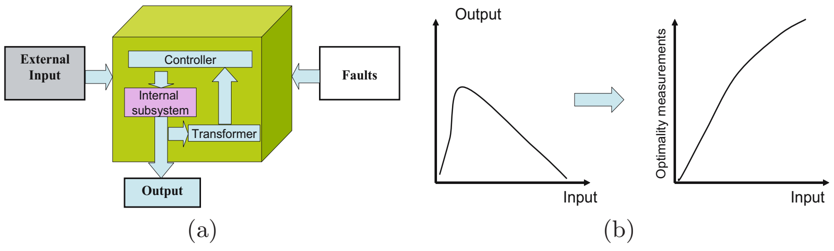


Fig. 4. (a) System stabilization via feedback control (b) Transform output into optimality measurements

To eschew the difficulty of finding equilibrium, we add a stabilizing **Output Transformer** to the feedback loop. An output transformer is a function from outputs into optimality measurements (OM). It is found in several cases that the Input-OM relationship is much simpler than Input-Output relationship, as is illustrated in Figure 4(b), which makes the design of stable feedback control based on OM simpler than that based on outputs.

The design of an output transformer depends on the particular application, but the following two conditions are sufficient for its use:

1. **Monotonicity:** The input-OM function is monotonic (increasing or decreasing). This condition deals with the possibility that when output is not optimal for a given input, the output by itself may not suffice to decide whether whether then input is less than the optimal input or more than the optimal input. Monotonicity of the Input-OM function provides definite feedback to the input.
2. **Uniqueness:** When optimality is obtained, OM is a fixed value or in a value in a narrow region, and is independent of the external inputs. This property provides robustness against varying external inputs that affect equilibrium.

Theorem 5. *If a stabilizing output transformer satisfies* **Monotonicity** *and* **Uniqueness**, *the system has stabilizing optimality.*

Proof. Let M be the Input-OM function, and $m = M(i)$ where i is the input and m is the OM. Let the fixed OM value where optimality is obtained be denoted by m^*.

Firstly, we assume that $M(i)$ is an increasing function. The simple feedback control algorithm shown in following stabilizes to the fixed value m^* (The algorithm stabilizes to a narrow region in presented in Figure 8).

```
if $(m > m^*)$      $i = i - \Delta i$;
else if $(m < m^*)$     $i = i + \Delta i$;
```

The following conditions are satisfied: (a) Closure: When the system satisfies optimality, $m = m^*$. In this case, the control algorithm leaves the input i unchanged, and so the output also stays unchanged. When $m = m^*$ is obtained, the system satisfies optimality which, by Uniqueness, is independent of the input. Therefore, optimality is closed. (b) Convergence: when the system state is not optimal, $m \neq m^*$. The control algorithm keeps changing the input i until $m = m^*$ eventually.
In the case that $M(i)$ is a decreasing function, similar control algorithms can be designed. Thus, the system can be stabilized.

We emphasize that Monotonicity and Uniqueness are not necessary conditions, so other forms of output transformers may also exist for achieving stabilizing optimality. There control algorithm would likelybe more complex.

4.1 Case Study: Duty Cycle Adaptation

We illustrate the stabilizing output transformer technique via a protocol for duty cycle adaptation. Network longevity is a key requirement for battery powered wireless sensor network. This suggests that node radios must be scheduled to switch off most of the time, i.e., to achieve as low a duty cycle as is possible while still accommodating the network traffic. Analytical results [17] indicate that different traffics require different duty cycles to achieve optimal energy efficiency. The goal of duty cycle adaptation then is to provide sufficient but minimum duty cycle for accommodating varying traffic.

Matching the duty cycle of the system to the load is a challenge problem and achieving its stability is even more difficult. If the duty cycle is lower than required, higher collision or sender buffer overflow can happen; if the duty cycle is higher than required, energy is wasted on idle listening. Changing the duty cycle may change the link reliability and thus the routing structure, which changes the traffic. However, traffic may affect duty cycle in return. Therefore, oscillation may happen in this control loop. Stabilizing duty cycle adaptation is a critical requirement for low duty cycle systems.

For the sake of presentation, let us consider a 6 node wireless sensor network example to illustrate our design. (More complex network deployment and traffic pattern can be found in [18].) All of the nodes are within communication range of each other. All 5 senders transmit with the same rate randomly when receiver is up.

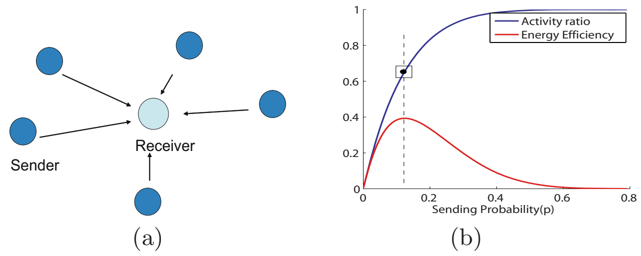


Fig. 5. (a) A wireless network with 5 senders and 1 receiver (b)Activity ratio & energy efficiency as a function of traffic level

Transformer design: The list of state variables in this protocol is the following:

– $D(t)$: Controller value: the receiver duty cycle at time t
– $I_i(t) = I(t)$: External input: traffic from sender i at time t
– $P_i(t) = P(t)$: Input to the internal system: the probability of sender i transmitting when receiver is up at time t
– $O(t)$: output traffic at time t, note that it includes only the goodput.
– $C(t)$: output collisions at time t

The input to the internal system, $P_i(t)$, is: $P_i(t) = \frac{I_i(t)}{D(t)} = \frac{I(t)}{D(t)} = P(t)$. Therefore, the output traffic is: $O(t) = 5 \cdot P(t) \cdot (1 - P(t))^4 \cdot D(t)$. Energy efficiency is defined as the ratio of output traffic and receiver duty cycle: $E_e(t) = \frac{O(t)}{D(t)} = 5 \cdot P(t) \cdot (1 - P(t))^4$. As shown in Figure 5(b), when $P(t)$ increases, energy efficiency increases before reaching a maximum, and then decreases thereafter. We define **activity ratio** as the optimality measurement of this case study.

Activity ratio is measured through output, and is defined as ratio of the total activity versus receiver duty cycle.

$$A(t) = \frac{O(t) + C(t)}{D(t)} = 1 - (1 - P(t))^5$$

Note that activity ratio is equal to the probability of the channel being non-idle. By using the activity ratio as the optimality measurement [18], we have transformed an optimization problem into a fixed point feedback control problem, as shown in Figure 5(b).

Basic feedback protocol algorithm: Before we present the algorithm, we introduce a Proposition firstly [18].

Proposition 1. *When activity ratio converges to within a small region* $[A_{min}, A_{max}]$*, optimal energy efficiency is obtained.*

In this section, we focus on providing a feedback control mechanism to ensure the activity ratio converges to within a small region $[A_{min}, A_{max}]$, wherein the optimal duty cycle is obtained.
Figure 6 presents a simple generic control protocol by using Multiple-Increasing-Multiple-Decreasing (MIMD). Let:

d_r be the receiver duty cycle, A_{max} be the maximum activity ratio,
A_{min} be the minimum activity ratio, α be duty cycle increasing rate,
β be duty cycle decreasing rate;

```
Input: $r_a, d_r(k)$    Output: $d_r(k+1)$    Parameter: $A_{min}, A_{max}, \alpha, \beta$
if $(r_a > A_{max}) \Rightarrow d_r(k+1) = d_r(k) + d_r(k) * \alpha$;
else if $(r_a < A_{min}) \Rightarrow d_r(k+1) = d_r(k) - d_r(k) * \beta$;
```

Fig. 6. Basic adaptive non-stabilizing duty cycle protocol

Although the invariant for this simple program is $r_a \in [A_{min}, A_{max}]$, the program cannot guarantee stabilization, as we show in the next section.
Stabilization of Feedback Control Protocol: Although MIMD achieves better convergence and energy efficiency, the method does not converge. Unless α and β are not chosen carefully, it is possible that the activity ratio r_a may oscillate from below A_{min} to above A_{max}, as is shown in Figure 7.

To prevent oscillation, we add stabilization into the basic feedback control protocol, as shown in Figure 8. The main idea is that when a transition from $r_a > A_{max}$ to $r_a < A_{min}$ or from $r_a < A_{min}$ to $r_a > A_{max}$ happens, the rate of receiver duty cycle change decreases. In this protocol, the variable *lstate* is used to indicate different states:

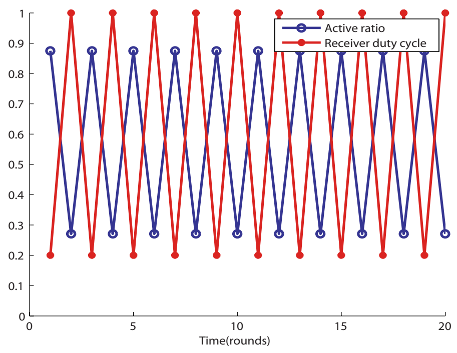


Fig. 7. Oscillation in the basic adaptive duty cycle algorithm

INC: duty cycle is increasing
DEC: duty cycle is decreasing
$OVER$: transitioning from $< A_{min}$ to $> A_{max}$
NOR: steady state
LOW: transitioning from $> A_{max}$ to $< A_{min}$

Variables δ_i and δ_d denote the step size of increasing or decreasing duty cycle. This protocol provides guarantee of stabilization:

```
Input: $r_a, d_r(k)$
Output: $d_r(k+1)$
Parameter: $A_{min}, A_{max}, \alpha, \beta$
State: $lstate, \delta_i, \delta_d$
if $(r_a > A_{max})$
    if $(lstate = INC) \| (lstate = NOR)$
        $d_r(k+1) = d_r(k) + d_r(k) * \alpha;$
        $\delta_i = d_r(k+1) - d_r(k);$
    if $(lstate = DEC) \| (lstate = OVER)$
        $\delta_d = \delta_d/2;$
        $d_r(k+1) = d_r(k) + \delta_d;$
        $lstate = OVER;$
    if $(lstate = LOW)$
        $\delta_i = \delta_i/2;$
        $d_r(k+1) = d_r(k) + \delta_i;$
else if $(r_a < A_{min})$
    if$(lstate = DEC)$
        $d_r(k+1) = d_r(k) - d_r(k) * \beta;$
        $\delta_d = d_r(k) - d_r(k+1);$
    if $(lstate = OVER)$
        $\delta_d = \delta_d/2;$
        $d_r(k+1) = d_r(k) - \delta_d;$
    if $(lstate = OVER)$
        $\delta_d = \delta_d/2;$
        $d_r(k+1) = d_r(k) - \delta_d;$
    if $(lstate = INC) \| (lstate = LOW)$
        $\delta_i = \delta_i/2;$
        $d_r(k+1) = d_r(k) - d_r(k) * \delta_i;$
        $lstate = LOW;$
    $lstate = DEC;$
else
    $lstate = NOR$
```

Fig. 8. The stabilizing optimality protocol for adaptive duty cycle

Theorem 6. *When the incoming traffic is steady, the adaptive duty cycle protocol described in Figure 8 stabilizes to an activity ratio in $[A_{min}..A_{max}]$, by which optimal efficiency is obtained.*

Proof. The invariant for this program is $r_a \in [A_{min}, A_{max}]$, the program is closed. Next, we will prove its convergence.

For a network with η senders, every node transmits with a certain duty cycle d_s^i. Note the duty cycle of different senders may be different, so we use a vector D_s to represent: $D_s = [d_s^1, d_s^2, ..., d_s^\eta]$. The relationship between the receiver activity ratio r_a and the receiver duty cycle d_r can be expressed as a function: $r_a = f(D_s, d_r)$. When D_s is fixed, function $f(D_s, d_r)$ is a decreasing function. In other words, when receiver duty cycle increases, activity ratio decreases when incoming traffic is fixed. For instance, in the case of random traffic, $r_a = f(D_s, d_r) = 1 - (1 - p_i)^\eta$, where $p_i = \max\{1, \frac{d_s^i}{d_r}\}$. We use Lyapunov theorem to prove convergence. Let the Lyapunov function be: $E_l = \min\{\|r_a - A_{min}\|, \|A_{max} - r_a\|\}$. When no transition happens, either $\|r_a - A_{min}\|$ or $\|A_{max} - r_a\|$ is a decreasing function, guaranteed by the decreasing function $r_a = f(D_s, d_r)$. When a transition happens, the protocol guarantees that an infinitely smaller step size of duty cycle is either added or subtracted. Function E_l is still a decreasing function. By the well known Lyapunov theorem, this protocol stabilizes to a static point. Given the continuity of the function $r_a = f(D_s, d_r)$, the activity ratio stabilizes to a point in $A_{min}..A_{max}$.

The algorithm described in Figure 8 is generic, so it can be applied in any system to achieve stabilization, when a transformer as described in Theorem 5 is added into system.

5 Conclusion

In this paper, we formulated optimality maintenance in dynamical system in terms of the standard notion of stabilization. We focused on three techniques —estimator-based, MinMax controllers that lead to Nash equilibrium, and transformer-based — for stabilization of dynamical systems. Each of these techniques relates to a different aspect of the system, respectively, its input, its controller, and its output.

One advantage of the formulation in terms of stabilization is the appreciation (in Lemma 1, Theorem 1, and Theorem 5) that the components designed for dynamical systems to maintain optimality should themselves be stabilizing. Likewise, the concrete implementations of these components should be stabilizing.

We illustrated the MinMax and transformer based techniques via case studies. Although these examples do include distributed computing, and indeed the latter can achieve stabilization in a network by independent local stabilization of its nodes, it is apparent that several advanced methods studied in the theory of stabilization for composition of stabilizing components can be exploited to deepen these techniques. As such, we find that these techniques deserve to be substantially further studied by the community.

Acknowledgment. This work was supported in part by NSF grants NSF-NETS/NOSS-0520222 and NSF-HDCCSR-0341703. We thank Vinodkrishnan Kulathumani as well as the referees for their helpful comments and suggestions.

References

1. Manna, Z., Pnueli, A.: Models for reactivity. Informatica, 609–678 (1993)
2. Gouda, M.G., Herman, T.: Adaptive programming. IEEE Transaction of Software Engineering 17(9), 911–921 (1991)
3. Arora, A., Gouda, M.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering 19(10), 1015–1027 (1993)
4. Kutten, S., Patt-Shamir, B.: Adaptive stabilization of reactive protocols. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 396–407. Springer, Heidelberg (2004)
5. Arora, A., Nesterenko, M.: Unifying stabilization and termination in message-passing systems. Distributed Computing 17(3), 279–290 (2005)
6. Burman, J., Kutten, S., Herman, T., Patt-Shamir, B.: Asynchronous and fully self-Stabilizing time-adaptive majority consensus. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, Springer, Heidelberg (2006)
7. Beauquier, J., Genolini, C., Kutten, S.: Optimal reactive k-stabilization: the case of mutual exclusion. In: PODC. Proceedings of the 18th ACM Symposium on Principles of Distributed Computing, pp. 209–218 (1999)
8. Theel, O.: Exploitation of Lyapunov Theory for Verifying Self-Stabilizing Algorithms. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, Springer, Heidelberg (2000)
9. Dhama, A., Oehlerking, J., Theel, O.: Verification of Orbitally Self-stabilizing Distributed Algorithms using Lyapunov Functions and Poincaré Maps. In: Proceedings of the 12th International Conference on Parallel and Distributed Systems (2006)
10. Anceaume, E., Défago, X., Gradinariu, M., Roy, M.: Towards a Theory of Self-organization. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, Springer, Heidelberg (2006)
11. Dolev, S., Tzachar, N.: Empire of Colonies: Self-stabilizing and Self-organizing Distributed Algorithms. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, Springer, Heidelberg (2006)
12. 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)
13. Arora, A., Ramnath, R., Ertin, E., Bapat, S., Naik, V., Cao, H., et al.: ExScal: Elements of an extreme wireless sensor network. In: Proceedings of the 11th International Conference on Embedded and Real-Time Computing Systems and Applications (2005)
14. Cao, H., Ertin, E., Kulathumani, V., Sridharan, M., Arora, A.: Differential Games in Large Scale Sensor Actuator Networks. In: IPSN. Proceedings of the 5th International Conference on Information Processing in Sensor Networks (2006)
15. Cao, H., Ertin, E., Arora, A.: MiniMax Equilibrium of Networked Differential Games, Technical Report OSU-CISRC-4/07 (2007)
16. Verdu, S., Poor, H.: On minimax robustness: A general approach and applications. IEEE Transactions on Information Theory IT-30, 328–340 (1984)
17. Cao, H., Parker, K.W., Arora, A.: O-MAC: a receiver centric power management protocol. In: ICNP. Proceedings of the 14th IEEE International Conference on Network Protocols, IEEE Computer Society Press, Los Alamitos (2006)
18. Cao, H., Arora, A., Parker, K.W., Lai, T.H.: Continuous asynchronous discovery with efficient synchronous communication for mobile networks, Technical Report OSU-CISRC-4/07 (2007)

Snap-Stabilizing Prefix Tree for Peer-to-Peer Systems*

Eddy Caron[1], Frédéric Desprez[1], Franck Petit[2], and Cédric Tedeschi[1]

[1] LIP Laboratory
UMR CNRS-ENS Lyon-UCB Lyon-INRIA 5668
46 Allée d'Italie, 69364 Lyon Cedex 07, France
[2] LaRIA Laboratory
CNRS-University of Picardie
5, rue du Moulin Neuf, 80000 Amiens, France

Abstract. *Resource Discovery* is a crucial issue in the deployment of computational grids over large scale peer-to-peer platforms. Because they efficiently allow range queries, *Tries* (*a.k.a., Prefix Trees*) appear to be among promising ways in the design of distributed data structures indexing resources. *Self-stabilization* is an efficient approach in the design of reliable solutions for dynamic systems. A *snap-stabilizing* algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps.

In this paper, we provide the first snap-stabilizing protocol for trie construction. We design particular tries called *Proper Greatest Common Prefix* (PGCP) Tree. The proposed algorithm arranges the n label values stored in the tree, in average, in $O(h + h')$ rounds, where h and h' are the initial and final heights of the tree, respectively. In the worst case, the algorithm requires an $O(n)$ extra space on each node, $O(n)$ rounds and $O(n^2)$ actions. However, simulations show that, using relevant data sets, this worst case is far from being reached and confirm the average complexities, making this algorithm efficient in practice.

Keywords: Peer-to-peer systems, Fault-tolerance, Self-stabilization, Snap-stabilization, Grid computing.

1 Introduction

These last few years have seen the development of large scale grids connecting distributed resources (computation resources, storage facilities, computation libraries, etc.) in a seamless way. This is now an efficient alternative to supercomputers for solving large problems such as high energy physics, bioinformatics or simulation. However, existing middleware systems always require a minimal stable centralized infrastructure and are not usable over dynamic large scale distributed platforms. To cope with the characteristics of these future platforms,

* This work was developed with financial support from the ANR (Agence Nationale de la Recherche) through the LEGO project referenced ANR-05-CIGC-11.

it has been widely suggested to use peer-to-peer technologies inside middleware [22]. Early distributed hash tables (DHT), designed for very large scale platforms, *e.g.,* to share files over the Internet, have several major drawbacks. Among them, there is the fact that they only support exact match queries. An important amount of work has recently been undertaken to allow more complex querying over peer-to-peer systems. A promising way to achieve this is the use of *tries* (*a.k.a., prefix trees*). Trie-based approaches outperform other ones by efficiently supporting range queries and easily extending to multi-criteria searches.

Unfortunately, although fault tolerance is a mandatory feature of systems aiming at being deployed at large scale (to avoid data loss and allow a correct routing of messages through the network), tries only offer a poor robustness in dynamic environment. The crash of one or several nodes leads to the loss of stored objects and to the split of the trie into several subtries. These subtries may then not reconnect correctly, making the trie invalid and thus unable to process queries. Among recent trie-based approaches, the fault-tolerance is either ignored, or handled by preventive mechanisms, intensively using replication which can be very costly in terms of computing and storage resources. Afterward, the purpose is to compute the right trade-off between the replication cost and the robustness of the system. Nevertheless, replication does not formally ensure the recovery of the system after arbitrary failures. From this point on, it remains only to use a strategy based on the *best-effort* approach. This is why we believe that such systems could take advantage of using self-stabilization techniques in order to satisfy the fault tolerance requirements.

The concept of self-stabilization [16] is a general technique to design a system tolerating arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. Thus, a self-stabilizing system does not need to be reinitialized and is able to recover from transient failures by itself.

In this paper, we propose a snap-stabilizing distributed algorithm to build a *Proper Greatest Proper Common Prefix* (GPCP) Tree starting from any labeled rooted tree. A *snap-stabilizing* [13] algorithm ensures that the system always maintains the desirable behavior and is obviously optimal in stabilization time. The property of snap-stabilization is achieved within the well-known Dijkstra's theoritical model [15] where in each computation step, each node can atomically read variables (or, registers) owned by its neighboring nodes.

The proposed algorithm arranges the n label values (each node holds a single label) stored in the tree, in average, in $O(h + h')$ rounds, where h and h' are the initial (before reconstruction) and final (after reconstruction) height of the tree, respectively. In the worst case, the algorithm requires an $O(n)$ extra space on a given node, $O(n)$ rounds and $O(n^2)$ operations. However, simulations show that, using relevant data sets, the worst case is far from being reached and confirm the average complexity. It also shows the practical efficiency of the proposed algorithm and the benefit of snap-stabilization in the design of efficient algorithms for unreliable, dynamic environments where the best-effort seems to be a valuable strategy.

In Section 2, we summarize recent peer-to-peer technologies used for resource discovery and their fault-tolerance mechanisms, followed by similar works undertaken in the field of self-stabilization. In Section 3, we describe the abstract model in which our algorithm is designed, and present what it means for a distributed algorithm to be snap-stabilizing. We also specify the PGCP Tree and related distributed data structures. In Section 4, the snap-stabilizing scheme protocol is presented, and its correctness proof and complexities discussed. Simulation process are explained and results given in Section 5. Finally, we conclude by summarizing the contribution of the paper and a brief description of next steps in this work.

2 Related Work

First peer-to-peer algorithms aiming at retrieving objects were based on the flooding of the network, overloading the network while providing non-exhaustive responses. Addressing both the scalability and the exhaustiveness issues, the distributed hash tables [25,26,29], logical hops required to route and the local state grow logarithmically with the number of nodes participating in the system. Unfortunately, DHTs present several major drawbacks. Among them, the rigidity of the requesting mechanism, only allowing exact match queries, hinders its use over distributed computational platforms that require more complex meanings of search.

A large amount of work tackles the opportunity to allow more flexibility in the retrieval process over structured peer-to-peer networks. Peer-to-peer systems users have been given the opportunity to plug different technologies on DHTs, such as the ability to retrieve resources described by semi-structured languages [5], to manage data thanks to traditional database operations [30], or to support multi-attribute range queries [1,23,27,28]. Among this last series of work supporting multi-attribute range queries, a new kind of overlay, based on tries, has emerged. Trie-structured approaches outperform others in the sense that logarithmic (or constant if we assume an upper bound on the depth of the trie) latency is achieved by parallelizing the resolution of the query in the several branches of the trie.

Prefix Hash Tree (PHT) [24] dynamically builds a trie of the given key-space as an upper layer and maps it over any DHT-like network. Obviously, the architecture of PHT results in the multiplication of the complexities of the trie and of the underlying DHT. The problem of fault tolerance is then delegated to the DHT layer. Skip Graphs, introduced in [3], are also similar to a trie, but rely on skip lists, allowing the use of their probabilistic fault tolerance. Nevertheless, a repair mechanism of the particular skip graph structure is provided. Nodewiz [6], another trie-structured overlay does not address the fault-tolerance problem by assuming the nodes reliable. Finally, P-Grid [14] tolerance is based on probabilistic replication. Initially designed for the purpose of service discovery over dynamic computational grids and aimed at solving some drawbacks of these previous approaches, we recently developed a novel architecture, based on a logical greatest common prefix tree [11]. This structure, more formally described in the following, is dynamically built as objects, *e.g.,* computational services, are declared by some servers. The fault tolerance is also addressed by replication of nodes and links of the tree. Another advantage of the technology presented in [11] is its ability to greedily take into account the heterogeneity of the underlying physical network to make a more efficient tree overlay.

To summarize, the fault-tolerance issue is mostly either ignored, delegated or replication-based. In [10], we provided a first alternative to the replication approach. The idea was to let the trie crash and to *a posteriori* reconnect and reorder the nodes. However, this protocol assumed the validity of subtries being reordered, thus limiting the field of initial configurations being handled and repaired. In the following sections, we present a new protocol able to repair any labeled rooted tree to make a valid greatest common prefix tree and thus to offer a general systematic mechanism to maintain distributed tries.

In the self-stabilizing area, some investigations take interest in maintaining distributed data structures. The solutions in [19,20,21] focus on binary heap and 2-3 trees. Several approaches have also been considered for a distributed spanning tree maintenance *e.g.,* [2,4,12,17,18]. In [18], a new model for distributed algorithms designed for large scale systems is introducted. In [7], the authors presented the first snap-stabilizing distributed solution for the Binary Search Tree (BST) problem. Their solution requires $O(n)$ rounds to build the BST, which is proved to be asymptotically optimal for this problem in the same paper.

3 Preliminaries

In this section, we first present the distributed system model used in the design of our algorithm. Then, we recall the concept of snap-stabilization and specify the distributed data structures considered.

3.1 Distributed System

The distributed algorithm presented in this paper is intended for practical *peer-to-peer* (P2P) networks. A P2P network consists of a set of asynchronous *physical* nodes with distinct IDs, communicating by message passing. Any physical node P_1 can communicate with any physical node P_2, provided P_1 knows the ID of P_2 (ignoring physical routing details). Each *physical* node maintains one or more *logical* nodes of the distributed *logical* tree. Our algorithm is run inside all these *logical* nodes. Note that the tree topology is susceptible to changes during its reconstruction. Each *logical* node of the tree has to be considered mapped on a *physical* node of the underlying network. However, the mapping process falls beyond the scope of this paper.

In order to simplify the design, proofs, and complexity analysis of our algorithm, we use the theoretical formal *state model* introduced in [15]. We apply this model on logical nodes (or simply, nodes) only. The message exchanges are modeled by the ability of a node to read the variables of some other nodes, henceforth referred to as its neighbors. A node can only write to its own variables. Each action is of the following form: $< label >:: < guard > \rightarrow < statement >$. The guard of an action in the program of 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 its guard evaluates to true. We assume that the actions are atomically executed, meaning the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The *state* of a node is defined by the values of its variables. The *state* of a system is a product of the states of all nodes. In the sequel, we refer to the state of a node and the system as a *state* and a *configuration*, respectively. Let a relation denoted by $\mapsto$, on $\mathcal{C}$ (the set of all possible configurations of the system). A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration.

A processor p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists at least an action A such that the guard of A is true in γ. We consider that any enabled node p is *neutralized* in the computation 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 node represents the following situation: At least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.) We assume an *unfair and distributed daemon*. The *unfairness* means that even if a processor p is continuously enabled, then p may never be chosen by the daemon unless p is the only enabled node. The *distributed* daemon implies that during a computation step, if one or more nodes are enabled, then the daemon chooses at least one (possibly more) of these enabled nodes to execute an action.

In order to compute the time complexity, we use the definition of *round*. This definition captures the execution rate of the slowest node in any computation. The set of all possible computations of $\mathcal{P}$ is denoted as $\mathcal{E}$. The set of possible computations of $\mathcal{P}$ starting with a given configuration $\alpha \in \mathcal{C}$ is denoted as $\mathcal{E}_\alpha$. Given a computation e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action of the protocol or the neutralization of every enabled node from the first configuration. Let e'' be the suffix of e, *i.e.*, $e = e'e''$. Then *second round* of e is the first round of e'', and so on.

3.2 Snap-Stabilization

Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$.

Definition 1 (Snap-stabilization). *The protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_\mathcal{P}$ on $\mathcal{E}$ if and only if the following condition holds: $\forall \alpha \in \mathcal{C} : \forall e \in \mathcal{E}_\alpha :: \ e \vdash \mathcal{SP}_\mathcal{P}$.*

3.3 Proper Greatest Common Prefix Tree

Let an ordered alphabet A be a finite set of letters. Denote $\prec$ an order on A. A non empty *word* w over A is a finite sequence of letters $a_1, \ldots, a_i, \ldots, a_l$, $l > 0$. The *concatenation* of two words u and v, denoted $u \circ v$ or simply uv, is equal to the word $a_1, \ldots, a_i, \ldots, a_k, b_1, \ldots, b_j, \ldots, b_l$ such that $u = a_1, \ldots, a_i, \ldots, a_k$ and $v = b_1, \ldots, b_j, \ldots, b_l$. Let ϵ be the *empty word* such that for every word w, $w\epsilon = \epsilon w = w$. The *length* of a word w, denoted by $|w|$, is equal to the number of letters of w—$|\epsilon| = 0$.

A word u is a *prefix* (respectively, *proper prefix*) of a word v if there exists a word w such that $v = uw$ (resp., $v = uw$ and $u \neq v$). The *Greatest Common Prefix* (resp., *Proper Greatest Common Prefix*) of a collection of words $w_1, w_2, \ldots, w_i, \ldots$ $(i \geq 2)$, denoted $GCP(w_1, w_2, \ldots, w_i, \ldots)$ (resp. $PGCP(w_1, w_2, \ldots, w_i, \ldots)$), is the longest prefix u shared by all of them (resp., such that $\forall i \geq 1, u \neq w_i$).

Definition 2 (PGCP Tree). *A* Proper Greatest Common Prefix Tree *is a labeled rooted tree such that each node label is the Proper Greatest Common Prefix of every pair of its children labels.*

In the design of our protocol, we also needs the relaxed form of PGCP Tree defined as follows:

Definition 3 (PrefixHeap). *A PrefixHeap is a labeled rooted tree such that each node label is the Proper Greatest Common Prefix of all its children labels.*

4 Snap-Stabilizing PGCP Tree

In this section, we present the snap-stabilizing PGCP tree maintenance. We provide a detailed explanation of how the algorithm works from initialization until the labels are arranged in the tree such that it becomes a PGCP tree. Next, the proof of snap-stabilization and complexity issues are given.

4.1 The Algorithm

The code of our solution is shown in Algorithms 1 and 2. We assume that initially, there exists a labeled rooted tree spanning the network. Every node p maintains a finite set of children $C_p = \{c_1, \ldots, c_k\}$, which contains the addresses of its children in the tree. Each node p is able to know the address of its father using the macro f_p. The uniqueness of the father is ensured by the use of the function $MinID(S)$ which returns the minimal values in the set S[1]. So, each node p can locally determine if it is either (1) the single *root* of the spanning tree (f_p is unspecified), (2) an *internal* node (f_p is specified and $C_p \neq \emptyset$), or (3) a *leaf* node ($c_p = \emptyset$). In the sequel, we denote the set of nodes in the tree rooted at p

[1] In a real P2P network, the relationship child/father is easily preserved by exchanging messages between a child node and its father.

as T_p (hereafter, also called the *tree* T_p) and the *height* of the tree rooted at p as $h(T_p)$.

Each node p holds a label l_p and a state S_p in $\{I, B, H\}$[2]—stand for $Idle$, $Broadcast$, and $Heapify$, respectively. The algorithm uses two basic functions to create and delete nodes from the tree. The **NEWNODE**(*lbl*, *st*, *chldn*) function creates a new node labeled by *lbl*, whose initial state is *st* and with a set of children initialized with *chldn*. Once the new node created by this function is integrated to a set of children, the f_p macro will ensure its father to be correctly set. Finally, the same f_p macro will set the father variable of nodes in *chldn*. The **DESTROY**(p) function is called to stop the process of a given node, (its reference should have been previously deleted from any other node).

Algorithm 1. Snap-Stabilizing PGCP Tree — Variables, Macros, and Actions

```
Variables:   $l_p$, the label of $p$
             $C_p = \{c_1, \ldots, c_k\}$
             $S_p = \{I, B\}$ if $p$ is the root, $\{I, H\}$ if $p$ is a leaf node, $\{I, B, H\}$ otherwise ($p$ is an internal node)

Macros:      $f_p \equiv MinID(\{q :\ p \in C_q\})$
             $SameLabel_p(L) \equiv \{c \in C_p |\ (l_c = L)\}$
             $SameGCP_p(L) \equiv \{c_1, c_2, \ldots, c_k \in C_p |\ GCP(c_1, c_2, \ldots, c_k) = L\}$
             $SamePGCP_p(L) \equiv SameGCP_p(L) \setminus \{c \in SameGCP_p(L) |\ l_c = L\}$

Actions:
                                         {For the root node}
    $InitBroadcast$ ::        $S_p = I \wedge (\forall c \in C_p |\ S_c = I) \longrightarrow S_p := B;$
    $InitRepair$ ::           $S_p = B \wedge (\forall c \in C_p |\ S_c = H) \longrightarrow$ HEAPIFY();REPAIR(); $S_p := I;$
                                         {For the internal nodes}
    $ForwardBroadcast$ ::     $S_p = I \wedge S_{f_p} = B \wedge (\forall c \in C_p |\ S_c = I) \longrightarrow S_p := B;$
    $BackwardHeap$ ::         $S_p = B \wedge S_{f_p} = B \wedge (\forall c \in C_p |\ S_c = H) \longrightarrow$ HEAPIFY(); $S_p := H;$
    $ForwardRepair$ ::        $S_p = H \wedge S_{f_p} = I \wedge (\forall c \in C_p |\ S_c \in \{H, I\}) \longrightarrow$ REPAIR(); $S_p := I;$
    $ErrorCorrection$ ::      $S_p = B \wedge S_{f_p} \in \{H, I\} \longrightarrow S_p := I;$
                                         {For the leaf nodes}
    $InitHeap$ ::             $S_p = I \wedge S_{f_p} = B \longrightarrow S_p := H$
    $EndRepair$ ::            $S_p = H \wedge S_{f_p} = I \longrightarrow S_p := I;$
```

The basic idea of the algorithm is derived from the fast version of the snap-stabilizing PIF in [8] and runs in three phases: The root initiates the first phase, called the *Broadcast* phase, by executing Action $InitBroadcast$. All the internal nodes in the tree participate in this phase by forwarding the broadcast message to their descendants — Action $ForwardBoradcast$. Once the broadcast phase reaches the leaves, they initiates the second phase of our scheme, called the *heapify* phase, by executing Action $InitHeap$.

During the heapify phase, a *PrefixHeap* is built — refer to Definition 3. We also ensure in this phase that for every node p, p is a single node in T_p with a value equal to l_p. The heapify phase is computed using Procedure $HEAPIFY()$, executed by all the internal — Actions $BackwardHeap$. The heapify phase eventually reaches the root which also executes Procedure $HEAPIFY()$ and initiates the third and last phase of our scheme, called the *Repair* phase — Action $InitRepair$. The aim of this phase is to correct the two following problems that can occur in the *PrefixHeap*. First, even if no node in T_p has the same label

[2] To ease the reading of the algorithm, we assume that $S_p \in \{I, B\}$ (respectively, $\{I, H\}$) if p is the root (resp., p is a leaf). We could easily avoid this assumption by adding the following guarded action for the root (resp.leaf) node: $S_p = H$ (resp. $S_p = B$) $\longrightarrow S_p := I$. Note that this correction could occur only once.

Algorithm 2. Snap-Stabilizing PGCP Tree — Procedures

```
1.01 Procedure HEAPIFY()
1.02     $C_p := C_p \cup \{\mathbf{NEWNODE}\ (l_p, H, \{\})\}$
1.03     $l_p := \mathbf{GCP}(\{l_c |\ c \in C_p\})$
1.04     for all $c \in C_p |\ l_c = l_p$ do
1.05         $C_p := C_p \cup C_c \setminus \{c\}$
1.06         DESTROY($c$)
1.07     done

2.01 Procedure REPAIR()
2.02     while $\exists(c_1, c_2) \in C_p |\ l_{c_1} = l_{c_2}$ do
2.03         $C_p := C_p \cup \{\mathbf{NEWNODE}(l_{c_1}, H, C_{s|\ s \in SameLabel(l_{c_1})})\}$
2.04         for all $c \in SameLabel_p(l_{c_1})$ do
2.05             DESTROY($c$)
2.06         done
2.07     done
2.08     while $\exists c \in C_p |\ SamePGCP_p(l_c) \neq \emptyset$ do
2.09         $C_p := C_p \cup \{\mathbf{NEWNODE}(l_c, H, C_c \cup SamePGCP_p(l_c))\}$
2.10         $C_p := C_p \setminus SamePGCP_p(l_c)$
2.11         DESTROY($c$)
2.12     done
2.13     while $\exists(c_1, c_2) \in C_p |\ |GCP(l_{c_1}, l_{c_2})| > |l_p|$ do
2.14         $C_p := C_p \cup \{\mathbf{NEWNODE}(GCP(l_{c_1}, l_{c_2}), H, SameGCP_p(GCP(l_{c_1}, l_{c_2}))\}$
2.15         $C_p := C_p \setminus SameGCP_p(GCP(l_{c_1}, l_{c_2}))$
2.16     done
```

as p, the same label may exist in different branches of the tree; Second, if each node is the greatest common prefix of its children labels, it is not necessarily the greatest common prefix of any pairs of its children labels.

The *repair* phase works similarly as in the Broadcast phase. The root and the internal nodes execute Procedure $REPAIR()$ starting from the root toward the leaves — Actions *InitRepair* and *ForwardRepair*. During this phase, for each node p, four cases can happen:

1. Several children of p have the same label. Then, all the children with the same label are merged into a single child — Lines 2.02 to 2.07;
2. The labels of some children of p are prefixed with the label of some of its brothers. In that case, the addresses of the prefixed children are moved into the corresponding brother — Lines 2.08 to 2.12;
3. The labels of some children of p are prefixed with a label which does not exist among their brothers and which are longer than the label of p. Then, for each set of children with the same prefix, p builds a new node with the corresponding prefix label and the corresponding subset of nodes as children — Lines 2.13 to 2.16.
4. If none of the previous three cases appear, nothing is done.

Finally, Phase $REPAIR()$ ends at leaf nodes by executing Action *EndRepair*. This indicates the end of the PGCP tree construction. Note that since we are considering self-stabilizing systems, the internal nodes need to correct abnormal situations due to the unpredictable initial configuration. The unique abnormal situation which could avoid the normal progress of the three phases of our scheme is the following: An internal node p is in State B (done with its broadcast phase) but its father f_p is in State H or I, indicating that it is done executing its Heapify phase or it is Idle, respectively. In that case, p executes Action *ErrorCorrection*, in the worst case, pushing down T_p the abnormal broadcast phase until reaching the leaf nodes of T_p. This guarantees the liveness of the protocol despite unpredictable initial configurations of the system.

4.2 Correctness Proof

In this section we show that the algorithm described in Subsection 4.1 is a snap-stabilizing PGCP tree algorithm. The complexities are also discussed.

Remark 1. To prove that an algorithm provides a snap-stabilizing PGCP tree algorithm, we need to show that the algorithm satisfies the following two properties: (1) starting from any configuration, the root eventually executes an initialization action; (2) Any execution, starting from this action, builds a PGCP tree.

Let us first consider the algorithm by ignoring the two procedures $HEAPIFY()$ and $REPAIR()$. In that case, the algorithm is very similar to the snap-stabilizing PIF in [8]. The only difference between both algorithms consists in the third phase. In Algorithm 1, the third phase is initiated by the root only, *after* the heapify phase terminated only, whereas in [8], the third phase can be initiated by any node once itself and its father are done with the second phase. That means that with the solution in [8], both the second and the third phase can run concurrently. That would be the case with Algorithm 1 if the guard of Action $ForwardRepair$ has been as follows: $S_p = H \wedge S_{f_p} \in \{H, I\} \wedge (\forall c \in C_p | \ S_c \in \{H, I\})$

However, it follows from the proofs in [8] that the behavior imposed by our solution is a particular behavior of the snap-stabilizing PIF algorithm. This behavior happens when all the nodes are slow to execute the action corresponding to the third phase. Since the algorithm in [8] works with an unfair daemon, the algorithm ensures that, eventually, the root initiates the third phase, leading the system to behave as Algorithm 1. Therefore, ignoring the effects of the two procedures $HEAPIFY()$ and $REPAIR()$ on the tree topology, the proof of snap-stabilization in [8] is also valid with our algorithm.

Considering the two procedures $HEAPIFY()$ and $REPAIR()$ again, since in every p, the set C_p is finite, it directly follows from the code of the two procedures in Algorithm 2 that that in every p, the set C_p is finite: every execution of Procedures $HEAPIFY()$ or $REPAIR()$ is finite.

It follows from the above discussion :

Lemma 1. *Starting from any configuration, the root node can execute Action InitBroadcost in a finite time even if the daemon is unfair.*

As a corollary of Lemma 1, the first condition of Remark 1 holds. Also, this show that every PGCP tree computation initiated by the root eventually terminates. It remains to show that the second condition of Remark 1 also holds for any node p.

Lemma 2. *After the execution of Procedure* **HEAPIFY** *by a node p, T_p is a PrefixHeap.*

Proof. We prove this by induction on $h(T_p)$. Since Procedure **HEAPIFY**() cannot be executed by a leaf node, we consider $h(T_p) \geq 1$.

1. Let $h(T_p)$ be equal to 1. So, all the children of p are leaves. Executing Lines 1.02 to 1.03, p is as a new child, itself a leaf node, labeled with l_p, while l_p contains the greatest common prefix of all its children. After the execution of Lines 1.04 to 1.07, p contains no child c such that $l_c = l_p$. Thus, l_p is a PGCP of all its children labels.
2. Assume that the lemma statement is true for any p such that $h(T_p) \leq k$ where $k \geq 1$. We will now show that the statement is also true for any p such that $h(T_p) = k+1$. By assumption, the lemma statement is true for all the children of p, *i.e.,* $\forall c \in C_p$, l_c is a proper prefix of any label in T_c, and l_c is the PGCP of all nodes in C_c. So, after executing Procedure **HEAPIFY**(), following the same reasoning as in Case 1, l_p is a PGCP of all its children, and since themselves are the root of a PrefixHeap, for every $c \in C_p$, l_p is also a proper prefix of any label in T_c. Hence, the lemma statement is also true for p.

Corollary 1. *After the system executed a complete Heapify phase, the whole tree is a PrefixHeap.*

Lemma 3. *After the execution of Procedure* **REPAIR***() by a node p such that $h(T_p) \geq 1$, then for every pair $(c_1, c_2) \in C_p$, $l_p = PGCP(c_1, c_2)$.*

Proof. Given p such that $h(T_p) \geq 1$ and that l_p is a proper prefix of any l_c for $c \in C_p$ (what we know by Lemma 2), if the tree following conditions are true for every pair $(c_1, c_2) \in C_p$, the statement $\forall (c_1, c_2) \in C_p$, $l_p = PGCP(c_1, c_2)$ is true:

1. $l_{c_1} \neq l_{c_2}$;
2. l_{c_1} (resp. l_{c_2}) is not a prefix of l_{c_2} (resp. l_{c_1});
3. $|GCP(l_{c_1}, l_{c_2})| = |l_p|$.

Clearly, after the execution of Lines 2.02 to 2.07, Lines 2.08 to 2.12, and Lines 2.13 to 2.16, Conditions 1, 2, and 3 holds, respectively.

By induction of Lemma 3 on every node of the path from the root to each leaf node, we can claim:

Corollary 2. *After the system executed a complete Repair phase, the whole tree is a PGCP tree.*

Proof. By induction of Lemma 3 on every node of the path from the root to each leaf node.

From corollaries 1 and 2, and the fact that after the root executed Action *InitBroadcast*, the three phases *Broadcast*, *Heapify*, and *Repair* proceed one after another [8], we can claim the following result:

Theorem 1. *Running under any daemon, Algorithm 1 and Algorithm 2 provide a snap-stabilizing Proper Greatest Common Prefix Tree construction.*

4.3 Complexity

Theorem 2. *The average time for the PGCP tree construction is $O(h + h')$ rounds. In the worst case, the construction requires $O(n)$ space complexity, $O(n)$ rounds and $O(n^2)$ operations, where n is the number of nodes of the tree.*

Proof. By similarity with the PIF, we can easily establish that the *broadcast* phase has reached all leaf nodes in $O(h)$ rounds, where h is the height of the tree when the *InitBroadcast* action is performed. We also easily see that the *heapify* phase reaches the root in $O(h)$. During the *repair* phase, the number of rounds required to reach all leaf nodes of the repaired tree (and thus end the cycle) is clearly $O(h')$, where h' is the height of this repaired tree (each round increment the depth by 0 or 1). The first part of the theorem is established.

When the *repair phase* is initiated, more precisely after the execution of the **HEAPIFY** macro and before the execution of the **REPAIR** macro on the root, it may happen that the tree becomes a star graph, each node being a child of the root (obviously except the root itself). This case is clearly the worst case, not only in terms of extra space required ($n - 1 = O(n)$) but also in terms of number of operations since the complexity of the **REPAIR** macro depends on the number of the root's children, *i.e.,* also $n - 1$. More precisely, the **REPAIR** macro is a combination of three operations: merging nodes, lines 2.02 to 2.06, moving nodes under other ones, lines 2.08 to 2.12 or creating a new subtree, lines 2.13 to 2.16. Among the set of possible combinations, the one that leads to the weakest parallelism is the move of $n - 2$ children of the root under a given node s, since, in the next round, s will be the only one process to work, *i.e.,* process these $n - 2$ nodes. If this worst case repeats (and the final topology is a chain), the complexity is of the following shape:

$$a \times (n - 1) + a \times (n - 2) + \ldots + a = O(n^2)$$

where a is a constant. Even if the worst case is not really attractive, we use simulations in the next section to see what we can expect in real life in terms of latency and extra space.

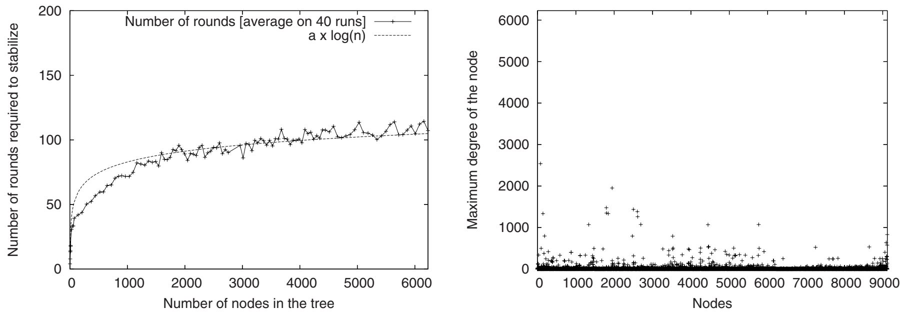
5 Simulation Results

To better capture the expected behavior of the snap-stabilizing PGCP tree, we simulated the algorithm using relevant data sets which reflect the use of computational platforms. The simulator is written in Python and contains the three following main parts:

1. It creates the tree with a set of labels of basic computational services commonly used in computation grids such as the names of routines of linear algebra libraries, the names of operating systems, the processors used in today's clusters and the nodes' addresses. The number of keys is up to 5200, creating trees up to 6228 nodes (the final tree size is the number of labels inserted plus the number of labels created to reflect the prefix patterns). For

instance, inserting two labels `DTRSM` and `DTRMM` results in a tree whose root (common father of `DTRSM` and `DTRMM`) is labeled by `DTR`.

2. It destroys the tree by moving subtrees, randomly. This is achieved by modifying the father of a randomly picked node and moving it from the set of children of its father to the set of children of a randomly chosen node. This operation is repeated up to $n/2$ nodes (meaning that, in average, approximately $n/2$ nodes are connected to a wrong father).
3. It launches the algorithms by testing for each node if the state of the node and those of its neighbors satisfy the guard of some action in the algorithm, in which case the statement of the action is executed. This step is repeated until the tree is in a stable configuration, *i.e.,* a configuration where all nodes are in state I again.



(a) Number of Rounds. (b) Highest Degrees.

Fig. 1. Simulation of the snap-stabilizing PGCP tree

We have first collected results on the latency of the algorithm. Figure 1-(a) gives the average number of rounds required to have a stable configuration, starting from 40 different bad configurations. The tree size ranges between 2 and 6228. We observe that the number of rounds required by the algorithm has a logarithmic behavior (and not linear as previously suggested by the worst case). It clearly scales according to the height of the tree, thus confirming the average complexity of the algorithm and its good scalability.

We have also collected results on the extra space required on each node. Since the tree topology undergoes changes during the reconstruction, degrees of nodes also dynamically change as nodes are created, destroyed, merged or moved. Figure 1-(b) shows the highest degree of nodes, *i.e.,* the real extra space required on each node, including nodes created and/or destroyed during the reconstruction. The final tree size is 6228; the total number of nodes, including temporary nodes, is 9120. The experiment shows that the highest of maximum degree of all nodes is 2540, and most of maximum degrees are very low (less than 50). This can be partly explained by the fact that the deepest a node is, the smaller is its degree. In other terms, during a breadth-first traversal of the tree, the topology quickly enlarges close to the root and then its breadth remains relatively stable until reaching the leaf nodes. More generally, this simulation shows that the worst case is far to be reached and that only few nodes will require an large extra space.

6 Conclusion

This paper presents the first snap-stabilizing greatest common prefix tree and a general self-stabilization algorithm for distributed tries. It provides an alternative to tree-structured peer-to-peer networks suffering from the high cost of replication mechanisms and a first step of an innovating way to reach the fault tolerance requirements over large distributed systems. Our algorithm is optimal in terms of stabilization time since we prove it to be snap-stabilizing. It requires in average a number of rounds proportional to the height of the tree, thus providing a good scalability. This result has been confirmed by simulation experiments based on relevant data sets. On the theoretical side, our future work will consist to improve the worst case complexities in terms of extra space requirements and total latency. Also, note that our model assumes that the processes can communicate with each other. In the state model, this is modeled as if every process can read the variables of all the processes of the network. However, once implemented in the message-passing model, the protocol requires communications between processes involved in the tree only. So, on the experimental side of our future works, we plan to implement this algorithm in the message-passing with a model based on that introduced in [18]. On this other hand, we also plan to implement our algorithm inside a prototype of a peer-to-peer indexing system we are currently developing, based on the JXTA toolbox. First experiments have been conducted on the Grid'5000 platform [9].

References

1. Andrzejak, A., Xu, Z.: Scalable, Efficient Range Queries for Grid Information Services. In: Peer-to-Peer Computing, pp. 33–40 (2002)
2. Arora, A., Gouda, M.G.: Distributed Reset. IEEE Transactions on Computers 43, 1026–1038 (1994)
3. Aspnes, J., Shah, G.: Skip Graphs. In: Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 384–393 (January 2003)
4. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time Optimal Self-stabilizing Synchronization. In: STOC 1993. Proceedings of the 25th Annual ACM Symposium on Theory of Computing, pp. 652–661. ACM Press, New York (1993)
5. Balazinska, M., Balakrishnan, H., Karger, D.: INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In: International Conference on Pervasive Computing 2002 (2002)
6. Basu, S., Banerjee, S., Sharma, P., Lee, S.: NodeWiz: Peer-to-Peer Resource Discovery for Grids. In: GP2PC. 5th International Workshop on Global and Peer-to-Peer Computing (May 2005)

7. Bein, D., Datta, A.K, 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)
8. Bui, A., Datta, A., Petit, F., Villain, V.: State-optimal snap-stabilizing pif in tree networks. In: IEEE (ed.) Proceedings of the 4th International Workshop on Self-Stabilizing Systems, pp. 78–85. IEEE Computer Society Press, Los Alamitos (1999)
9. Cappello, F., et al.: Grid'5000: a Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, pp. 99–106. Springer, Heidelberg (2005)
10. Caron, E., Desprez, F., Fourdrignier, C., Petit, F., Tedeschi, C.: A Repair Mechanism for Tree-structured Peer-to-peer Systems. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, Springer, Heidelberg (2006)
11. Caron, E., Desprez, F., Tedeschi, C.: A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids. In: Montresor, A., Wierzbicki, A., Shahmehri, N. (eds.) P2P2006. The Sixth IEEE International Conference on Peer-to-Peer Computing, Cambridge, September 6-8 2006, pp. 106–113. IEEE Computer Society Press, Los Alamitos (2006)
12. Chen, N.S., Yu, H.P., Huang, S.T.: A Self-stabilizing Algorithm for Constructing Spanning Trees. Information Processing Letters 39, 147–151 (1991)
13. Cournier, A., Datta, A.K., Petit, F., Villain, V.: Enabling Snap-Stabilization. In: ICDCS 2003. Proceedings of the 23rd International Conference on Distributed Computing Systems, p. 12. IEEE Computer Society Press, Washington (2003)
14. Datta, A., Hauswirth, M., John, R., Schmidt, R., Aberer, K.: Range Queries in Trie-Structured Overlays. In: The Fifth IEEE International Conference on Peer-to-Peer Computing (2005)
15. Dijkstra, E.W.: Self-stabilizing Systems in Spite of Distributed Control. Commun. ACM 17(11), 643–644 (1974)
16. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
17. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of Dynamic Systems Assuming only Read/Write Atomicity. Distributed Computing 7, 3–16 (1993)
18. Herault, T., Lemarinier, P., Peres, O., Pilard, L., Beauquier, J.: A model for large scale self-stabilization. In: I E E E Sc. (ed.) IPDPS 2007. 21th International Parallel and Distributed Processing Symposium, IEEE Computer Society Press, Los Alamitos (2007)
19. Herman, T., Pirwani, I.: A Composite Stabilizing Data Structure. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 182–197. Springer, Heidelberg (2001)
20. Herman, T., Masuzawa, T.: A Stabilizing Search Tree with Availability Properties. In: IEEE (ed.) ISADS 2001. Proceedings of the 5th International Symposium on Autonomous Decentralized Systems, pp. 398–405 (2001)
21. Herman, T., Masuzawa, T.: Available Stabilzing Heaps. Information Processing Letters 77, 115–121 (2001)
22. Iamnitchi, A., Foster, I.: On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 118–128. Springer, Heidelberg (2003)
23. Oppenheimer, D., Albrecht, J., Patterson, D., Vahdat, A.: Distributed Resource Discovery on PlanetLab with SWORD. In: WORLDS. Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (December 2004)

24. Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S.: Prefix Hash Tree An indexing Data Structure over Distributed Hash Tables. In: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, St. John's, Newfoundland, Canada (July 2004)
25. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A Scalable Content-Adressable Network. In: ACM SIGCOMM, ACM Press, New York (2001)
26. Rowstron, A., Druschel, P.: Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
27. Schmidt, C., Parashar, M.: Enabling Flexible Queries with Guarantees in P2P Systems. IEEE Internet Computing 8(3), 19–26 (2004)
28. Shu, Y., Ooi, B.C., Tan, K., Zhou, A.: Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In: Peer-to-Peer Computing, pp. 173–180 (2005)
29. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In: ACM SIGCOMM, pp. 149–160. ACM Press, New York (2001)
30. Triantafillou, P., Pitoura, T.: Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In: Aberer, K., Koubarakis, M., Kalogeraki, V. (eds.) DBISP2P 2003. LNCS, vol. 2944, pp. 169–183. Springer, Heidelberg (2004)

Decentralized, Connectivity-Preserving, and Cost-Effective Structured Overlay Maintenance

Yu Chen and Wei Chen

Microsoft Research Asia
{ychen,weic}@microsoft.com

Abstract. In this paper we present a rigorous treatment to structured overlay maintenance in decentralized peer-to-peer (P2P) systems subject to various system and network failures. We present a precise specification that requires the overlay maintenance protocols to be decentralized, preserve overlay connectivity, always converge to the desired structure whenever possible, and only maintain a small local state independent of the size of the system. We then provide a complete protocol with proof showing that it satisfies the specification. The protocol solves a number of subtle issues caused by decentralization and concurrency in the system. Our specification and the protocol overcomes a number of limitations of existing overlay maintenance protocols, such as the reliance on a centralized and continuously available bootstrap system, the assumption of a known system stabilization time, and the need to maintain large local membership lists.

Keywords: structured overlay maintenance, peer-to-peer, fault tolerance.

1 Introduction

Since their introduction, structured overlays have been used as an important substrate for many peer-to-peer applications. In a structured peer-to-peer overlay, each node maintains a partial list of other nodes in the system, and these partial lists together form an overlay topology that satisfies certain structural properties (e.g., a ring). Various system events, such as node joins, leaves and crashes, message delays and network partitions, affect overlay topology. Thus, an overlay topology should adjust itself appropriately to maintain its structural properties. Topology maintenance is crucial to the correctness and the performance of applications built on top of the overlay.

Most structured overlays are based on a logical key space, and they can be conceptually divided into two components: leafset tables and finger tables.[1] The leafset table of a node keeps its logical neighbors in a key space, while the finger table keeps relatively faraway nodes in the key space to enable fast routing along the overlay topology. The leafset tables are vital for maintaining a correct overlay topology since finger tables can be constructed efficiently from the correct leafset

[1] The term leafset is originally used in Pastry [19] while the term finger is originally used in Chord [21].

tables. Therefore, our study focuses on leafset maintenance. In particular, we focus on one-dimensional circular key space and the ring-like leafset topology in this space, similar to many studies such as [19,21].

Leafset maintenance is a continuously running protocol that needs to deal with various system events. An important criterion for leafset maintenance is convergence. That is, the leafset topology can always converge back to the desired structure after the underlying system stabilizes (but without knowing about system stabilization), no matter how adverse the system events were before system stabilization.

In this paper, we provide a rigorous treatment to leafset convergence. Our contributions are mainly twofold. First, we provide a precise specification for leafset maintenance protocols with cost effectiveness requirements. All properties of the specification are desired by applications, while together they prohibit protocols with various limitations appeared in previous work. Second, we provide a complete protocol that is proven to satisfy our specification.

There are several distinct features in our specification. First, our specification explicitly emphasizes connectivity preservation: the connectivity of the leafset topology may only be broken by adverse system events such as node crashes and network failures, but it should not be broken by the maintenance protocol itself. Some previous protocols such as Chord [14] and Pastry [19] allow runs in which the topology is broken due to protocol logic itself. Specifying the Connectivity Preservation property is not simple. We need to define a system stabilization time after which no adverse system events occur and require that the maintenance protocol no longer disconnect any nodes in the system afterwards. We dedicate a section to show that defining such a system stabilization time is subtle in that any time earlier will not guarantee connectivity preservation.

Second, we explicitly put requirements on cost effectiveness: the size of the local state maintained by the protocol in the steady state only depends on the size of its leafset table, but should not depend on the system's size. To be cost-effective, a protocol inevitably needs to remove some extra entries in the leafset (as in many existing protocols), but such removals may jeopardize the connectivity of the topology. Therefore, handling the apparent conflict between connectivity preservation and cost effectiveness is the key in our protocol design. Some existing protocols ([11,15]) rely on the maintenance of a large membership list to preserve connectivity, and thus is not cost-effective.

Third, we explicitly address how to heal topology partition by introducing an interface function $\mathsf{add}(\mathit{contacts})$. Although the overlay could be more resistant to topology partition by maintaining more entries in the routing tables [14], network partitions are still inevitable, especially when failures on major network links happen. Therefore, we believe partition healing is an indispensable part of the protocol. The interface $\mathsf{add}(\mathit{contacts})$ and its specification cleanly separates partition detection from partition healing: A separate mechanism may be used to detect topology partition, and then to call the $\mathsf{add}(\mathit{contacts})$ interface (only once) to bridge the partitioned components, while afterwards the maintenance protocol will automatically converge the topology. Our specification keeps the dependency on an external mechanism such as a bootstrap system at the minimum, while some previous protocols heavily rely on continuously available bootstrap systems to keep connectivity [7,20].

Moreover, we provide a complete protocol and prove that it satisfies our specification. As indicated already, the core of the protocol is to handle the conflict between connectivity preservation and cost effectiveness: The protocol should remove extra entries in the leafset while preserving the topology's connectivity. The protocol addresses a couple of subtle issues: one is how to nullify the effects of adverse system events without knowing when the system stabilizes, and the second is to avoid livelocks that may be caused by inopportune invocations of the add(*contacts*) interface. The correctness proof is technically involved and long, because our protocol needs to deal with system asynchrony and various system failures and events.

The correctness of our protocol is based on the availability of a dynamic failure detector that eventually can correctly detect failures of neighbors of a node. One may argue that in peer-to-peer environments, such failure detectors are unrealistic. We justify our model with the following reasons. First, studying the convergence behavior of a dynamic protocol under system failures is important to understand the correctness and the efficiency of the protocol, and to compare different protocols under the same condition. Such studies naturally assume a model in which system failures eventually stop, for which the paradigm of self stabilization is a direct example.[2] Second, the theoretical assumption that the system stabilizes after a certain time point means in practice a long enough stable period for the topology to converge. Based on our simulation study [4], we show that with some optimizations the convergence speed of our protocol is fast ($O(\log N)$ where N is the number of nodes in the system), so system stabilization assumption may not seem so unreasonable in certain settings. Third, failure detection accuracy can be greatly improved if we consider voluntary leaves, in which a leaving node notifies its neighbors before leaving the system. Therefore, the failure detection requirement in the model may be more easily achieved for a sufficiently long time than considering only node crashes.

To our knowledge, our protocol is the first one that satisfies all the properties required by the specification with a complete correctness proof. We believe that our work could compensate many system-level studies on structured overlay maintenance and provide a more formal approach to study the correctness of overlay maintenance protocols.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 defines our system model with the failure detector specification. Section 4 introduces the complete specification of convergent leafset maintenance protocols. Section 5 presents the complete leafset maintenance protocol. We conclude the paper and discuss future work in Section 6. Our full technical report [4] contains further results including complete proofs, a sample implementation of

[2] In Section 4 we will elaborate the relationship between our specification and self stabilization.

the failure detector in one partially synchronous model, and optimizations for fast convergence of the protocol.

2 Related Work

Many existing structured P2P overlay proposals mention that each node should have a leafset table. However, those such as Pastry [19], CAN [17], and SkipNet [10] only provide brief descriptions on what a correct leafset table looks like and how to fix it when the leafset table becomes incorrect because of system churns. These proposals assume that there is a correct leafset table on each node to begin with, then give methods to repair the leafset tables in response to various system events. Bamboo DHT [18], the latest Pastry improvements [2,9], and $\mathcal{DKS}$ [8] adopt practical mechanisms to improve overlay maintenance and routing correctness in a dynamic environment. These mechanisms are system-level improvements, while there are no proofs or formal studies on protocol guarantees, such as connectivity preservation and convergence.

In [14], Liben-Nowell et al. point out the topology maintenance issues of the original Chord protocol [21] and propose an "idealize" process to adjust the immediate successor of each node to improve topology maintenance. This approach is essentially a method to guarantee convergence, but the maintenance restricts itself to immediate successor data structure. Although each node stores a successor list to handle successor failures, the node does not actively maintain the list. Instead, it uses the successor list of its immediate successor to overwrite its own one, and thus it could be disconnected from other nodes in its own list. Therefore, it is only a special case of our protocol, is less robust, and is difficult to accommodate partition healing, which requires maintaining multiple links together to bridge partitioned components.

Some recent work uses the approach of self stabilization [5,6] to study overlay maintenance. The T-Man [11] and TChord [15] protocols are self-stabilizing, but they do not consider global membership changes from system churns. They require to keep an essentially full membership list on each node, so the maintenance cost increases significantly when the system is large or the membership changes over time. Authors of [7] and [20] also propose self-stabilizing overlay maintenance protocols. But their protocols and proofs depend on the existence of a *continuously available* bootstrap system. In [7], the bootstrap system needs to handle all join and repair requests, and needs to issue periodic broadcast messages for self stabilization purpose, while in [20] each node must periodically initiate look-ups to the bootstrap system. These protocols impose significant load and availability requirements on the bootstrap system. In contrast, our protocol only needs an external mechanism such as a bootstrap system when the topology is partitioned, and it only needs the bootstrap system once after system stabilization. Therefore, the load and availability requirements on the bootstrap system are minimized.

The "Linearization" method in [16] describes a self-stabilizing algorithm to transform any connected graph into a sorted list. Although a bootstrap system is not required, the algorithm does not consider node churns and asynchronous/concurrent effects in a distributed message passing environment.

Authors of Ranch [13] provide an overlay maintenance protocol with a formal proof of correctness. However, they do not consider fault tolerance: all nodes leaves are "active leave", in which case all nodes invoke a special leave protocol before getting offline. We believe silent failures must be considered in a wide area environment, and dealing with them makes the model, the specification and the protocol design significantly depart from those studied in [13].

In [1], Angluin et al. proposed a method for fast construction of an overlay network by a tree-merging process. Their protocol is not a convergent overlay maintenance protocol, because they assume that overlay construction is executed when the underlying system is known to have stabilized and they do not consider the adverse impacts of system conditions before system stabilization.

3 System Model

We consider a distributed peer-to-peer system consisting of nodes (peers) from the set $\Sigma = \{x_1, x_2, \ldots\}$. Each node has a unique numerical ID drawn from a one-dimensional circular key space $\mathcal{K}$. We use x to represent both a node $x \in \Sigma$ and its ID in $\mathcal{K}$. For convenience, we set $\mathcal{K} = [0, 1)$, all real numbers between 0 and 1. We define the following distances in key space $\mathcal{K}$: For all $x, y \in \mathcal{K}$, (a) the *clockwise distance* $d^+(x, y)$ is $y - x$ when $y \geq x$ and $1 + y - x$ when $y < x$; (b) the *counter-clockwise distance* $d^-(x, y) = d^+(y, x)$, and (c) the *circular distance* $d(x, y) = \min(d^+(x, y), d^-(x, y))$.

Throughout the paper, we use continuous global time to describe system and protocol behavior, but individual nodes do not have access to global time. Nodes have local clocks, which are used to generate increasing timestamps and periodic events on the nodes. Local clocks are not synchronized with one another. They provide an interface function getClockValue(), which is only required to return monotonically increasing time values on a node even if the node has failures between the calls to the function.

Nodes may join and leave the system or crash at any time. We treat a node leave and crash as the same type of event; that is, a node disappears from the system without notifying other nodes in the system, and we refer to such an event as a *failure* in the system. We define a *membership pattern* Π as a function from time t to a finite and nonempty subset of Σ, such that $\Pi(t)$ refers to all of the *online* nodes at time t. Nodes not in $\Pi(t)$ are considered *offline*. For the purpose of studying overlay convergence, we assume that the set of online nodes $\Pi(t)$ eventually stabilizes. That is, there is an unknown time t such that for all $t' \geq t$, $\Pi(t')$ remains the same, which we denote as $sset(\Pi)$. Let GST_N (N stands for nodes) be the *global stabilization time* of the nodes, which is the earliest time after which $\Pi(t)$ does not change any more. Henceforth, all specification properties refer to an arbitrary membership pattern Π.

Nodes communicate with one another by sending and receiving messages through asynchronous communication channels. We assume that there is a bidirectional channel between any pair of nodes. The channels cannot create or duplicate messages, but they might delay or drop messages. The channels are eventually reliable in the following sense: There exists an earliest time $GST_M \geq GST_N$ such that for any message m sent by $x \in sset(\Pi)$ to $y \in sset(\Pi)$ after time GST_M, m is eventually received by y.

To deal with failures in asynchronous environments, we assume the availability of a failure detector, which is a powerful abstraction that encapsulates all timing assumptions on message delays, processing speed, and local clock drifts [3]. Unlike the original model in [3], our failure detector is for dynamic environments, and we do not assume that the failure detector knows a priori a set of nodes to monitor. Instead, the failure detector provides an input interface $\mathsf{register}(S)$ for a node to register a set of nodes $S \subset \Sigma$ to be monitored by the failure detector. A node may invoke $\mathsf{register}(S)$ many times with a different set S to change the set to be monitored. The failure detector also provides an output interface $\mathsf{detected}(x)$ to notify a node that it detects the failure of a node $x \in \Sigma$.

Informally, the failure detector should eventually detect all failures among all registered nodes, and should eventually not make any wrong detections on nodes still online. More rigorously, it satisfies the following properties:

– *Strong Completeness*: For all $x \in sset(\Pi)$ and all $y \notin sset(\Pi)$, if x invokes $\mathsf{register}(S)$ with $y \in S$ at some time t, then there is a time $t' > t$ at which either the failure detector outputs $\mathsf{detected}(y)$ on x or x invokes $\mathsf{register}(S')$ with $y \notin S'$.
– *Eventual Strong Accuracy*: For all $x, y \in sset(\Pi)$, there is a time t such that for all $t' \geq t$, the failure detector will not output $\mathsf{detected}(y)$ on x at time t'.

Our failure detector differs from the eventually perfect failure detector $\diamond\mathcal{P}$ in the static environment [3] in that our failure detector relies on application inputs to learn the set of processes to monitor. We denote our failure detector as $\diamond\mathcal{P}_D$ (D stands for dynamic). In our protocol $\diamond\mathcal{P}_D$ is only used for each node to monitor its neighbors, so it is easier to achieve than $\diamond\mathcal{P}$ that requires monitoring all nodes in the system.

Every node in the system executes protocols by taking *steps* triggered by events, which include input events invoked by applications, message receipt events, periodic events generated by the local clock, and failure detection events $\mathsf{detected}()$. In each step, a node may change its local state, register with the failure detector, and send out a finite number of messages. For simplicity we assume that the time to execute a step is negligible, but a node might fail during the execution of a step. We also assume that there are only a finite number of steps taken during any finite time interval, and at each time point, there is at most one step taken by one node.[3]

A *run* of a leafset maintenance protocol is an infinite sequence of steps together with the increasing time points indicating when the steps occur, such that it conforms with the above assumptions on membership pattern, message delivery, and failure detection.

[3] Our results also work if each step is not instantaneous or there are multiple concurrent steps at the same time, but it would make our description and proof more cumbersome to handle these situations.

4 The Specification for Leafset Maintenance

We now specify the desired properties for a leafset maintenance protocol. Our specification always refers to an arbitrary execution of the protocol with an arbitrary membership pattern $\mathit{\Pi}$.

First, we define the function $\mathsf{leafset}(x, \mathit{set})$ as follows: We have a fixed constant $L \geq 1$, which informally means that the leafset of a node should have L closest nodes on each side of it in the circular space. Given a finite subset $\mathit{set} \subseteq \mathit{\Sigma}$ and a node x, If $|\mathit{set} \setminus \{x\}| < 2L$, then $\mathsf{leafset}(x, \mathit{set}) = \mathit{set} \setminus \{x\}$. Otherwise, sort $\mathit{set} \setminus \{x\}$ as (a) $\{x_{+1}, x_{+2}, \ldots\}$ such that $d^+(x, x_{+1}) < d^+(x, x_{+2}) < \ldots$, and (b) $\{x_{-1}, x_{-2}, \ldots\}$ such that $d^-(x, x_{-1}) < d^-(x, x_{-2}) < \ldots$ Then, we have $\mathsf{leafset}(x, \mathit{set}) = \{x_{+1}, x_{+2}, \ldots, x_{+L}\} \cup \{x_{-1}, x_{-2}, \ldots, x_{-L}\}$.

In the leafset maintenance protocol, each node x maintains a variable *neighbors*, the value of which is a finite subset of $\mathit{\Sigma}$. Informally, $x.\mathit{neighbors}$ should eventually converge onto the correct leafset, meaning $x.\mathit{neighbors} = \mathsf{leafset}(x, \mathit{sset}(\mathit{\Pi}))$, in which case the final topology resembles a ring structure.

Each node also has an interface function $\mathsf{add}(\mathit{contacts})$, where *contacts* is a finite subset of $\mathit{\Sigma}$. This function is used to bridge partitioned components. In particular, it can be used in the following situations: (a) adding initial contacts when the system is initially bootstrapped; (b) introducing contact nodes when a new node joins the system; and (c) introducing nodes in other partitioned components after the overlay is partitioned (perhaps due to transient network partitions).

To formalize our requirements, we first need to address the connectivity of the leafset topology. For any directed graph G, we say that 1) it is *strongly connected* if there is a directed path between any pair of nodes in G, 2) it is *weakly connected* (or simply *connected*) if there is an undirected path (when treating edges in G as undirected) between any pair of nodes in G, and 3) it is *disconnected* if it is not weakly connected. The *leafset topology* at time t is a directed graph $G(t) = \langle \mathit{\Pi}(t), E(t) \rangle$, where $E(t) = \{\langle x, y \rangle | x, y \in \mathit{\Pi}(t) \ \wedge \ y \in x.\mathit{neighbors}_t\}$. For any node $x \in \mathit{\Pi}(t)$, we denote $P_x(t)$ as the set of nodes in the *connected* subgraph of $G(t)$ that contains x; that is, $P_x(t)$ is the set of nodes that have undirected paths to x.

A key property we require on leafset maintenance is that the protocol should not break the connectivity of the topology. However, the topology might also be broken by underlying system behaviors out of protocol control, such as node failures and message delays. To factor out system-induced topology break-ups, we only require that the topology is not broken once the underlying system is stabilized. To do so, we first need to define the stabilization time of the system.

Let GST_D (D stands for detector) be the global stabilization time of the failure detector $\diamond\mathcal{P}_D$, which is the earliest time $t \geq GST_N$ such that $\diamond\mathcal{P}_D$ will not output $\mathsf{detected}(y)$ on any $x \in \mathit{sset}(\mathit{\Pi})$ for any $y \in \mathit{sset}(\mathit{\Pi})$ after time t. That is, GST_D is the earliest time after which the failure detector does not make wrong detections on online nodes any more. After GST_D, both the nodes and the failure detector stabilize, but nodes might still receive old messages sent before GST_D that may adversely affect the convergence of the topology. Thus, we define GST_S (S stands for system) to be the global stabilization time of the system, which is the earliest time $t \geq \max(GST_D, GST_M)$ such that all messages sent before GST_D or GST_M have been delivered by time t or are lost. Since there are only a finite number of messages that could have been sent before GST_D or GST_M, we know GST_S must be a finite value. Note that these stabilization times are defined for each run of the leafset maintenance protocol.

Our connectivity preservation property is defined based on GST_S as follows:

– *Connectivity Preservation*: For any $t \geq GST_S$, for any directed path from x to y in $G(t)$, for any time $t' > t$, there is a directed path from x to y in $G(t')$.

Connectivity Preservation is a key property to guarantee leafset convergence, but it is not explicitly addressed or enforced by previous protocols in a purely peer-to-peer environment. The following theorem shows the necessity of GST_S, meaning that no algorithm can guarantee connectivity preservation starting from a time earlier than GST_S. The proof of the theorem can be found in [4].

Theorem 1. *For any convergent leafset maintenance protocol A and any small real value $\epsilon > 0$, there exists a run in which G_t is weakly connected for some t such that $GST_S - \epsilon < t < GST_S$, but at a later time $t' \geq GST_S$, $G_{t'}$ is not weakly connected.*

By the Connectivity Preservation property, we know that the connected component $P_x(t)$ can only grow after time GST_S. Since $\Pi(t)$ does not change after GST_S and is finite, we know that $P_x(t)$ eventually stabilizes. The next property requires that the leafset of x eventually contains the correct leafset in the connected component of x:

– *Eventual Inclusion*: There is a time t such that for all $t' \geq t$ and for all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.neighbors_{t'}) = \mathsf{leafset}(x, P_x(t'))$.

If the topology becomes connected at some time after GST_S, then Eventual Inclusion together with Connectivity Preservation means that eventually $\mathsf{leafset}(x, x.neighbors_{t'}) = \mathsf{leafset}(x, sset(\Pi))$ for all $x \in sset(\Pi)$. The properties also imply that the weakly connected component $P_x(t)$ will become strongly connected eventually. Note that the Eventual Inclusion property should hold no matter if there are invocations of $\mathsf{add}()$ after GST_S.

If the topology is partitioned, an application (or even a user) should be able to use the $\mathsf{add}()$ interface to heal the partition. This is specified by the following property:

– *Partition Healing*: For any $x, y \in sset(\Pi)$, if there is an invocation of $\mathsf{add}(S)$ on x at time $t > GST_S$ with $y \in S$, then there is a time $t' > t$ such that x and y are connected in $G(t')$ (i.e., $P_x(t') = P_y(t')$).

The Partition Healing property ensures that only one invocation of $\mathsf{add}()$ on one node is necessary to bridge the partition, as long as we use an S that contains a node from every component in $\mathsf{add}(S)$. Afterwards, Eventual Inclusion and

Connectivity Preservation properties guarantee the autonomous convergence of the topology without any further help.

The following property requires that eventually the leafset maintenance protocol should only maintain the actual leafset entries, provided that the application eventually stops invoking add().

– *Eventual Cleanup*: If there is a time t after which no add() is invoked at any node in the system, then there is a time t' such that for all time $t'' \geq t'$ and all $x \in sset(\Pi)$, $\mathsf{leafset}(x, x.neighbors_{t''}) = x.neighbors_{t''}$.

We call a leafset maintenance protocol *convergent* if it satisfies Connectivity Preservation, Eventual Inclusion, Partition Healing, and Eventual Cleanup. If an external mechanism guarantees to call add() as described in Partition Healing, then the convergent protocol ensures that the topology is eventually connected and the leafset of every node is correct, i.e., $x.neighbors = \mathsf{leafset}(x, sset(\Pi))$.

One informative way to understand the specification is to see how it avoids a trivial implementation that always splits every node into a singleton, i.e., sets $x.neighbors$ to $\emptyset$ on every node x. This implementation would correctly satisfy the specification if there were no Partition Healing property. With Partition Healing, however, after GST_S the protocol is forced to reconnect nodes after add() invocations, and by Connectivity Preservation, the protocol has to keep these connections, and then by Eventual Inclusion and Eventual Cleanup, the protocol has to converge to a correct leafset structure. Thus trivially splitting nodes is prohibited by the specification.

Besides convergence, the leafset maintenance protocol should also be cost-effective in terms of the cost to maintain the *neighbors* set on the nodes. We look at the maintenance cost when the protocol reaches its *steady state*: that is, assuming that there is no more add() invoked at any node, the *neighbors* set of each online node has already included the correct leafset entries in its stabilized connected component and nothing more. The cost effectiveness is characterized by the following property:

– *Cost Effectiveness*: If there is a time t after which no add() is invoked at any node in the system, then in the steady state of the protocol, on each node the size of the local state and the number of nodes registered to the failure detector are both $O(L)$.

When counting the size, we assume that each node ID and each clock value take a constant number of bits to represent. The property specifies that in the steady state the local state and the number of nodes monitored by the failure detector on each node is linear to the size of the leafset and is not related to the system's size. The requirement of $O(L)$ nodes registered to the failure detector prevents a protocol from monitoring a large set of nodes in the steady state. The property also implies that in the steady state each node can only send messages to $O(L)$ nodes and the size of each message is at most $O(L)$.

Our specification of convergent overlay maintenance protocols is similar to self stabilization [5,6] in that we require the leafset topology to eventually converge to the desired structure (each connected component is a ring structure) no matter what the topology was before the underlying system stabilizes. Our specification differs from self stabilization in the following aspects: First, we consider an open system where applications may invoke add() to add new contact nodes at any time, while self stabilization considers a closed system without any application interference. Second, unlike in the self stabilization model, we do not assume that all system states can be arbitrarily corrupted before system stabilization (e.g., local clock values cannot go backwards).

5 Leafset Maintenance Protocol

Our leafset maintenance protocol consists of five sub-protocols: (a) the add() protocol to add new contacts supplied by the application (Fig. 1, lines 3–8); (b) the failure-handling protocol to remove the failed nodes from the leafset upon the notification of failure detector (Fig. 1, lines 9–10); (c) the invite protocol to invite closer nodes into leafset (Fig. 2); (d) the replacement protocol to replace faraway nodes that should not be in the leafset with closer nodes (Fig. 3);[4] and (e) the deloopy protocol to detect and resolve a special incorrect topology called loopy topology (Fig. 5). The replacement protocol (Fig. 3) is our key contribution, so we focus our attention on this sub-protocol while briefly explaining other sub-protocols. Even though each sub-protocol has its own functionality, they have to work together to provide the desired self-stabilizing and cost-effective features specified in the previous section.

All of these sub-protocols (except the failure-handling one) use a periodic ping-pong messaging structure. For ease of understanding, each type of ping-pong message is sent independently. In actual implementations, one can unify all periodic ping-pong messages together for efficiency.

5.1 Add New Contacts and Handle Failures

On each node, the protocol maintains a *neighbors* set as required by the specification. The protocol keeps an invariant that a node y is added into $x.neighbors$ only after x receives a pong message directly from y. This invariant verifies the liveness of any nodes to be added into the *neighbors* set and prevents different unwanted behaviors in different sub-protocols.

In the add() protocol, if the nodes were added directly into the *neighbors* set without any verification, the property *Eventual Inclusion* would not be satisfied because the application might keep inserting failed nodes via add(). To solve this problem, the add(*contacts*) protocol (Fig. 1, lines 3–8) uses a ping-pong message loop to check the liveness of the nodes being added. In this way, the add() invoked after GST_N will not add any failed nodes into the *neighbors* set of any online nodes, since the failed nodes cannot respond to the PING-CONTACT messages.

[4] Technically, the faraway nodes for a node x are those in $x.neighbors \setminus \mathsf{leafset}(x, x.neighbors)$. Whenever necessary, we use $x.var$ to denote the variable *var* on x.

```
On node $x$:
Data structure:
    $neighbors$: set of nodes intended for leafset entries, initially $\emptyset$.
add($contacts$)
    **foreach** $y \in contacts$, send PING-CONTACT to $y$
Upon receipt of PING-CONTACT from $y$:
    send PONG-CONTACT to $y$
Upon receipt of PONG-CONTACT from $y$:
    $neighbors \leftarrow neighbors \cup \{y\}$; register($neighbors$)
Upon detected($y$):
    $neighbors \leftarrow neighbors \setminus \{y\}$
```

Fig. 1. Leafset maintenance protocol, Part I: Add new contacts and handle failures

5.2 Invite Closer Nodes

The invite protocol (Fig. 2) uses a variable *cand* to store candidate nodes to be invited into the *neighbors* set. The candidate nodes are discovered by exchanging local leafset views through the PING-ASK-INV and PONG-ASK-INV messages. Once a node x discovers some new candidates, it uses the periodic PING-INVITE and PONG-INVITE message loop to invite these candidates into $x.neighbors$. The invitation is successful when the candidate y sends back the PONG-INVITE message to x and x verifies that y is indeed qualified to be in x's leafset (lines 27).The invite protocol is in principle similar to other leafset maintenance protocols (e.g. [21,18,11,20]), except that we use PING-INVITE and PONG-INVITE messages to prevent a phenomenon called *ghost entry*. A *ghost entry* is an entry of a failed node that keeps bouncing among the *neighbors* sets of two or more online nodes, as explained below.

In the above example, suppose y is a failed node with ID adjacent to x and z. We also suppose y is still in $z.neighbors$. When x sends PING-ASK-INV message to z, z returns y. Without the message loop of PING-INVITE and PONG-INVITE, x would add y into $x.neighbors$ directly. After z told x about y, its failure detector reports y's failure and y is removed from $z.neighbors$. Later z contacts x to find some nodes to be invited, and x returns y. So y is added back to $z.neighbors$. Then y could be removed from $x.neighbors$ by a failure detector notification on x, and added back again by the PONG-ASK-INV message from z.

This process can repeat forever, making y bouncing back and forth between $x.neighbors$ and $z.neighbors$. The *ghost entry* phenomenon violates the property of *Eventual Inclusion*. It could be eliminated by the PING-INVITE and PONG-INVITE message loop. With this message loop, a failed node will not be added into the *neighbors* set by the invitation protocol since it cannot send any PONG-INVITE messages. Therefore, it will not be returned to other nodes as an invitation candidate, either.

```
On node $x$:
11 Data structure:
12     $cand$: candidate nodes for $neighbors$, initially $\emptyset$.
13 Repeat periodically:
14     foreach $y \in neighbors$, send PING-ASK-INV to $y$
15 Upon receipt of PING-ASK-INV from a node $y$:
16     $view \leftarrow \mathsf{leafset}(y, neighbors)$; send (PONG-ASK-INV, $view$) to $y$
17     $cand \leftarrow cand \cup \{y\}$
18 Upon receipt of (PONG-ASK-INV, $view$) from $y$
19     $cand \leftarrow cand \cup view$
20 Repeat periodically                                  /* invite closer nodes */
21     foreach $y \in cand \setminus neighbors$
22         if $y \in \mathsf{leafset}(x, cand \cup neighbors)$ then send PING-INVITE to $y$
23     $cand \leftarrow \emptyset$
24 Upon receipt of PING-INVITE from $y$:
25     send PONG-INVITE to $y$
26 Upon receipt of PONG-INVITE from $y$:
27     if $y \in \mathsf{leafset}(x, neighbors \cup \{y\}) \setminus neighbors$ then
28             $neighbors \leftarrow neighbors \cup \{y\}$; $\mathsf{register}(neighbors)$
```

Fig. 2. Leafset maintenance protocol, Part II: Invite closer nodes in the key space

5.3 Replace Faraway Nodes

The replacement protocol (Fig. 3) is responsible for removing faraway nodes from the *neighbors* sets to keep *neighbors* sets small. This protocol is our key contribution to provide *Cost Effectiveness*, and the key differentiator from other protocols. When removing the faraway nodes, we need to ensure both safety (*Connectivity Preservation*) and liveness (*Eventual Inclusion* and *Eventual Cleanup*), in the presence of concurrent replacements and other system events.

To ensure safety, we use a closer node to replace a faraway node instead of removing it directly. The basic replacement flow consists of two ping-pong loops. Suppose a node x intends to remove a node z since z is not in $\mathsf{leafset}(x, x.neighbors)$. Node x uses the PING-ASK-REPL and PONG-ASK-REPL loop (lines 33–39) with node z to obtain a replacement node y, which is recorded by x in $x.repl[z]$. (If there does not exist a node v satisfy the condition at line 36, y is set to $\perp$ and returned to x.) Then x uses the PING-REPLACE and PONG-REPLACE message loop to verify with y about the replacement (lines 40–52). If y finds z in $y.neighbors$ at the time it receives the PING-REPLACE message from x, it acknowledges x with a PONG-REPLACE message. Only after receiving the PONG-REPLACE message from y, x may replace z with y in $x.neighbors$. This method tries to ensure that after the removal of edge $\langle x, z\rangle$ from the overlay, there is still a path from x to z via y. The first ping-pong loop tries to find an alternative path to replace $\langle x, z\rangle$. The second ping-pong loop tries to ensure y's liveness and the validity of the path.

```
On node $x$:
Data structure:
    $repl[\,]$: for each $z \in neighbors$, $repl[z]$ is a node to replace $z$, initially $\perp$
    $commit[\,]$: for each $z \in neighbors$, $commit[z]$ is the time when $x$ commits to $z$
          in a replacement task, initially 0
    /* $repl[\,]$ and $commit[\,]$ only maintains entries for nodes in $neighbors$ */
    $ts$: timestamp of the replacement task, initially 0
Repeat periodically:
    foreach $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$, send PING-ASK-REPL to $z$
Upon receipt of PING-ASK-REPL from $z$:
    $y \leftarrow v$ such that $v \in \mathsf{leafset}(x, neighbors)$ and $d(z,v) < d(z,x)$ and
       $d(z,v) = \min_{u \in \mathsf{leafset}(x,neighbors)} d(z,u)$
    send (PONG-ASK-REPL, $y$) to $z$
Upon receipt of (PONG-ASK-REPL, $y$) from $z$
    if $z \in neighbors$ then $repl[z] \leftarrow y$
Repeat periodically:
    $ts \leftarrow \mathsf{getClockValue}()$
    foreach $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$ and $repl[z] \neq \perp$
       send (PING-REPLACE, $z$, $ts$) to $repl[z]$
Upon receipt of (PING-REPLACE, $z$, $ts$) from $y$:
    if $z \in neighbors$ then
       $commit[z] \leftarrow \mathsf{getClockValue}()$; send (PONG-REPLACE, $z$, $ts$) to $y$
Upon receipt of (PONG-REPLACE, $z$, $ts$) from $y$:
    if $z \in neighbors \setminus \mathsf{leafset}(x, neighbors)$ and $y = repl[z]$ then
       $neighbors \leftarrow neighbors \cup \{y\}$
       if $commit[z] < ts$ then
          $neighbors \leftarrow neighbors \setminus \{z\}$; $commit[y] \leftarrow \mathsf{getClockValue}()$
       $\mathsf{register}(neighbors)$
```

Fig. 3. Leafset maintenance protocol, Part III: Replace faraway nodes

The above basic flow alone, however, cannot nullify the indirect effects of adverse system events before time GST_D when there are concurrent replacements, and thus the topology connectivity could still be jeopardized. For example, in Fig. 4, x replaces z with y after time GST_S when it receives the PONG-REPLACE message sent by y after time GST_D. In the meantime, there is a concurrent task in which y wants to replace z with u. After sending the PONG-REPLACE message to x, y receives the PONG-REPLACE message from u and successfully replaces z with u. However, the time that u sends the PONG-REPLACE message to y could be before GST_D. So an erroneous "$\mathsf{detected}(z)$" on u immediately after the sending of the message could remove z from $u.neighbors$. As the result, x is relying on the alternative path $x \rightarrow y \rightarrow u \rightarrow z$ to remove z from $x.neighbors$, but the path is broken since u removed z from $u.neighbors$. However, x is not aware of these concurrent events, and it still removes z after GST_S, which breaks the connectivity. This shows the indirect effect of adverse system events before GST_D. A similar danger exists when x tries to replace z and y concurrently.

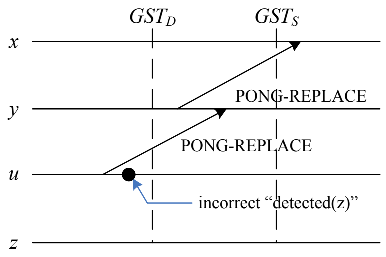


Fig. 4. Concurrent replacement tasks introduce indirect effects of adverse system events before GST_D and break topology connectivity

We introduce variables *ts* and *commit*[ ] to eliminate these dangerous concurrent replacements. Variable *ts* is a timestamp identifying the current replacement task when a node sends out PING-REPLACE messages (line 41), and its value is piggybacked with the PING-REPLACE and PONG-REPLACE messages. For each $z \in x.neighbors$, variable $x.commit[z]$ records the time when x commits to z in a replacement task, either when x verifies the replacement of z for another node y (line 46), or when x uses z to replace another node y (line 51). The key condition is that x can only successfully replace z in a replacement task whose timestamp *ts* is higher than $commit[z]$ on x (line 50). The use of *ts* and *commit*[ ] variables avoids any dangerous concurrent replacement tasks in the system. In the example of Fig. 4, after y sends the PONG-REPLACE message to confirm the replacement of z for x, $y.commit[z]$ is updated to a new timestamp that is larger than the timestamp of y's own concurrent replacement task to z. So when y receives the PONG-REPLACE from u, it will not remove z from $y.neighbors$. As shown by our proof, it is the core mechanism to satisfy the *Connectivity Preservation* property.

Next, we restrict the selection of replacement node y to guarantee the *Eventual Cleanup* property. A node y can be a replacement of z for x only when y is closer to x than z and is in z's leafset (line 36). The distance constraint avoids circular replacement, while the leafset constraint guarantees that y can successfully verify the replacement. The latter is true because our invite protocol guarantees that eventually the leafsets are mutual, so z will be in y's leafset. These two replacement selection constraints guarantee the progress of the replacement tasks, and thus the *Eventual Cleanup* property.

The mechanisms introduced so far are not enough to guarantee the *Eventual Inclusion* property, however. During the proof of an earlier version of the protocol, we uncovered the following subtle livelock scenario in which the add() invocations interfere with leafset convergence. Whenever node x wants to replace z with y, the replacement is rejected because x just committed to z in a replacement task that replaces another node u with z. The rejections can keep happening if an application keeps invoking add($\{u\}$) on x at inopportune times such that the edge from x to u is continually being added back to the topology. The inability for x to replace z with y is not an issue by itself. However, it is possible that there is a node v that should be in x's leafset, and the only way x learns about v is through z by the replacement protocol (the invite protocol

```
On node $x$:
53 Data structure:
54     $succ$: a derived variable, $succ = x$ if $neighbors = \emptyset$ else $succ = y \in neighbors$
           such that $d^+(x, y) = \min\{d^+(x, z) : z \in neighbors\}$
55 Repeat periodically:
56     if $neighbors \neq \emptyset$ and $d^+(x, 0) < d^+(x, succ)$ then
57         send (PING-DELOOPY, $x$) to $succ$
58 Upon receipt of (PING-DELOOPY, $u$) from $y$:
59     if $x = u$ then return
60     if $neighbors = \emptyset$ or $d^+(x, 0) < d^+(x, succ)$ then
61         $cand \leftarrow cand \cup \{u\}$; send PONG-DELOOPY to $u$
62     else
63         send (PING-DELOOPY, $u$) to $succ$
64 Upon receipt of PONG-DELOOPY from $y$:
65     $cand \leftarrow cand \cup \{y\}$
```

Fig. 5. Leafset maintenance protocol, Part IV: Loopy detection

will not help if all nodes in $z.neighbors$ are outside x's leafset range). In this case, x cannot replace z with y and thus will not learn about v, so the leafset convergence will not occur.

To fix this problem, we break the replacement of z with y on node x into two phases. First, x can add node y into $x.neighbors$ (line 49), without checking the constraint of $z.commit < ts$. Next, x can remove z only when the condition $z.commit < ts$ holds (lines 50–51). With this change, x can still find closer nodes through z even if x cannot replace z.

We also find another similar livelock scenario if the replacement node is selected from z's *neighbors* set rather than its leafset ($\mathsf{leafset}(z, z.neighbors)$) in line 36. The discovery of these subtle and even counter-intuitive livelock scenarios shows that a rigorous and complete proof helps us in discovering subtle concurrency issues that are otherwise difficult to discern.

5.4 Detect Loopy Structure

With the sub-protocols explained so far, the topology still might be incorrect, because it can be in a special state called the *loopy state* as defined in [14]. A node's *successor* is the closest node in its *neighbors* set according to the clockwise distance. A topology is in the loopy state if following the successor links one may traverse the entire key space more than once before coming back to the starting point. We use a deloopy protocol (Fig. 5) similar to the one in [14] to detect the loopy state and resolve it. The protocol essentially initiates a PING-DELOOPY message along the successor links to see if the message makes a complete traversal of the logical space before coming back to the initiator. If so, a loopy state is found, and the protocol puts the two end nodes of this traversal into each other's *cand* sets, so that the invite protocol is triggered to resolve the loopy state.

Our protocol is cost-effective because in the steady state each node only maintains sets *neighbors* and *cand*, mappings *repl*[ ] and *commit*[ ], which contain $O(L)$ number of nodes, and only nodes in the *neighbors* set are eventually registered with the failure detector.

Putting all sub-protocols together, we have a full protocol that satisfies all properties in our specification, as summarized by the following theorem.

Theorem 2. *The leafset maintenance protocol provided in Fig. 1, 2, 3, and 5 is both convergent and cost-effective, which means it satisfies the Connectivity Preservation, Partition Healing, Eventual Cleanup, Eventual Inclusion, and Cost Effectiveness properties.*

6 Conclusions and Future Work

In this paper, we propose a formal specification of peer-to-peer structured overlay maintenance, and introduce a complete protocol that matches the specification. The protocol is able to preserve overlay connectivity in a purely peer-to-peer manner while maintaining a small leafset, and it is able to converge any connected topology to the correct configuration.

The primary focus of this paper is the formal treatment of ring-based overlay maintenance. For a more practical implementation, a number of issues need to be addressed, which can be regarded as the future directions of our work. First, potential optimizations is possible to save the maintenance bandwidth of our protocol. Second, we may be able to weaken our model assumptions such as the availability of the dynamic failure detector $\diamond\mathcal{P}_D$ and the existence of the global stabilization time GST_S to match closer to the dynamic peer-to-peer environments, by following the similar approach in [12] for example. Another direction is to study the convergence speed of our protocol. On this front, we have conducted simulation studies with some heuristics to achieve an $O(\log N)$-level convergence time where N is the total number of nodes in the system [4]. We are looking into theoretical analysis of the fast convergence protocols. Finally, generalizing our results to other structured overlay topologies is also useful.

References

1. Angluin, D., Aspnes, J., Chen, J.: Fast construction of overlay networks. In: Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures, ACM Press, New York (2005)
2. Castro, M., Costa, M., Rowstron, A.: Performance and dependability of structured peer-to-peer overlays. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (2004)
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
4. Chen, Y., Chen, W.: Decentralized, connectivity-preserving, and cost-effective structured overlay maintenance. Technical Report MSR-TR-2007-84, Microsoft Research (2007)

5. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
6. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
7. Dolev, S., Kat, R.I.: Hypertree for self-stabilizing peer-to-peer systems. In: Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications, IEEE Computer Society Press, Los Alamitos (2004)
8. Ghodsi, A., Alima, L.O., Haridi, S.: Low-bandwidth topology maintenance for robustness in structured overlay networks. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Track 9 (2005)
9. Haeberlen, A., Hoye, J., Mislove, A., Druschel, P.: Consistent key mapping in structured overlays. Technical Report TR05-456, Rice Computer Science Department (2005)
10. Harvey, N.J.A., Jones, M.B., Saroin, S., Theimer, M., Wolman, A.: Skipnet: A scalable overlay network with practical locality properties. In: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (2003)
11. Jelasity, M., Babaoglu, O.: T-Man: Gossip-based overlay topology management. In: Brueckner, S.A., Serugendo, G.D.M., Hales, D., Zambonelli, F. (eds.) ESOA 2005. LNCS (LNAI), vol. 3910, pp. 1–15. Springer, Heidelberg (2006)
12. Keidar, I., Shraer, A.: How to choose a timing model? In: Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (2007)
13. Li, X., Misra, J., Plaxton, C.G.: Active and concurrent topology maintenance. In: Proceedings of the 18th International Symposium on Distributed Computing (2004)
14. Liben-Nowell, D., Balakrishnan, H., Karger, D.: Analysis of the evolution of peer-to-peer systems. In: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, ACM Press, New York (2002)
15. Montresor, A., Jelasity, M., Babaoglu, O.: Chord on demand. In: Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing, IEEE Computer Society Press, Los Alamitos (2005)
16. Onus, M., Richa, A., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (2007)
17. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: SIGCOMM 2001. Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (2001)
18. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: Proceedings of the USENIX Annual Technical Conference (2004)
19. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
20. Shaker, A., Reeves, D.S.: Self-stabilizing structured ring topology p2p systems. In: Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing, IEEE Computer Society Press, Los Alamitos (2005)
21. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM 2001. Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (2001)

On the Performance of Dijkstra's Third Self-stabilizing Algorithm for Mutual Exclusion

Viacheslav Chernoy[1], Mordechai Shalom[2], and Shmuel Zaks[1]

[1] Department of Computer Science, Technion, Haifa, Israel
vchernoy@tx.technion.ac.il, zaks@cs.technion.ac.il
[2] TelHai Academic College, Upper Galilee, 12210, Israel
cmshalom@telhai.ac.il

Abstract. In [7] Dijkstra introduced the notion of self-stabilizing algorithms, and presented three such algorithms for the problem of mutual exclusion on a ring of processors. The third algorithm is the most interesting of these three, but is rather non intuitive. In [8] a proof of its correctness was presented, but the question of determining its worst case complexity – that is, providing an upper bound on the number of moves of this algorithm until it stabilizes – remained open. In this paper we solve this question, and prove an upper bound of $O(n^2)$ (n being the size of the ring) for this algorithm's complexity. This complexity applies to a centralized as well as to a distributed scheduler.

1 Introduction

The notion of self stabilization was introduced by Dijkstra in [7]. He considers a system, consisting of a set of processors, and each running a program of the form: **if** *condition* **then** *statement*. A processor is termed *privileged* if its condition is satisfied. A *scheduler* chooses any privileged processor, which then executes its statement (*i.e.,* makes a move); if there are several privileged processor, the scheduler chooses any of them. Such a scheduler is termed *centralized*. A scheduler that chooses any subset of the privileged processors, that are then making their moves simultaneously, is termed *distributed*. Thus, starting from any initial configuration, we get sequences of moves (termed *executions*). The scheduler thus determines all possible executions of the system. A specific subset of the configurations is termed *legitimate*. The system is *self-stabilizing* if any possible execution will eventually get - that is, after a finite number of moves - only to legitimate configurations. The number of moves from any initial configuration until the system stabilizes is often referred to as *stabilization time* (see, *e.g.,* [2,6,12,15]).

Dijkstra studied in [7] the fundamental problem of mutual exclusion, for which the subset of legitimate configurations includes the configurations in which exactly one processor is privileged. In [7] the processors are arranged in a ring, so that each processor can communicate with its two neighbors using a shared memory, and where not all processors use the same program. Three algorithms were presented – without correctness or complexity proofs – in which each processor could be in one of $k > n$, four and three states, respectively (n being the number of processors). A centralized scheduler was assumed.

The analysis - correctness and complexity - of Dijkstra's first algorithm is rather straightforward. The correctness under a centralized scheduler is for any $k \geq n-1$, and under a distributed scheduler for any $k \geq n$. The stabilization time under a centralized scheduler is $\Theta(n^2)$ (following [4] this is also the expected number of moves). There is little in the literature regarding the second algorithm, probably since it was extended in [11] to general trees, or since more attention was devoted to the third algorithm, which is rather non-intuitive. For this latter algorithm Dijkstra presented in [8] a proof of correctness (another proof was given in [10], and a proof of correctness under a distributed scheduler was presented in [3]). Actually, it is only after [8] that an extensive study of the area of self-stabilization began, and expanded to a variety of directions (see, *e.g.,* [9,13]).

Though while dealing with proofs of correctness one can sometimes get also complexity results, this was not the case with this proof of [8]. Referring to this Dijkstra's third algorithm, the authors in [1] state that *"The complexity study of this algorithm has never been made"*; to the best of our knowledge, this statement is also true today. Moreover, the authors claim that *"Surprisingly, no exact result on worst case stabilization time has been published. The reason for this is perhaps that Dijkstra's algorithm does not monotonically converge towards a stabilized state. Some punctual bursts can momentarily lead it far from its goal"*. The authors of [1] then proceed and present an algorithm, similar to that of Dijkstra, and prove an upper bound of $5\frac{3}{4}n^2$ for the stabilization time of their algorithm. A lower bound of $\Omega(n^2)$ for this algorithm is known (see [14], and also Note 1 in Section 2).

In this paper we provide an upper bound on the stabilization time of Dijkstra's third algorithm; specifically, we prove that the number of moves from any initial configuration until the system stabilizes is $O(n^2)$. We do so by extending the proof of [8]. The result applies to a centralized scheduler as well as to a distributed one.

In Section 2 we present Dijkstra's algorithm, and outline the details of the proof of [8] needed for our discussion. In Section 3 we present observations regarding the proof of [8], and then present our proof of the upper bound.

2 Dijkstra's Algorithm

In this section we present Dijkstra's third algorithm of [7] (to which we refer throughout this paper as *Dijkstra's algorithm*, or just *the algorithm*), and to its proof of correctness of [8]. Following [8], our discussion assumes a centralized scheduler (we will get back to a distributed scheduler after Theorem 2).

In [7] there are n processors $p_0, p_1, \ldots, p_{n-1}$, that are arranged in a ring; that is, the processors adjacent to p_i are $p_{(i-1) \bmod n}$ and $p_{(i+1) \bmod n}$, for $i = 0, 1, \ldots, n-1$. Processor p_i has a local state $x_i \in \{0, 1, 2\}$. Two processors –

namely, p_0 and p_{n-1} – run special programs, while all intermediate processors p_i, $1 \leq i \leq n-2$, run the same program. The programs of the processors are as follows:

Program for processor p_0:

```
if  $x_0 + 1 = x_1$  then  $x_0 := x_0 - 1$  end.
```

Program for processor p_i, $1 \leq i \leq n-2$:

```
if  $(x_i + 1 = x_{i-1})$  or  $(x_i + 1 = x_{i+1})$  then  $x_i := x_i + 1$  end.
```

Program for processor p_{n-1}:

```
if  $(x_{n-2} = x_0)$  and  $(x_{n-1} \neq x_0 + 1)$  then  $x_{n-1} := x_0 + 1$  end.
```

Recall that the subset of legitimate configurations for this problem includes the configurations in which exactly one processor is privileged. The configuration $x_0 = \cdots = x_{n-1}$ and $x_0 = \cdots = x_i \neq x_{i+1} = \cdots = x_{n-1}$ are legitimate (see also (7), (8) and (9) of Example 2).

It is proved in [8] that this algorithm self stabilizes, and the system thus achieves mutual exclusion. In this proof the following notation is used. Given an initial configuration $x_0, x_1, \ldots, x_{n-1}$, and placing the processors on a line, consider each pair of neighbors p_{i-1} and p_i, for $i = 1, \ldots, n-1$ (note that though p_{n-1} and p_0 are neighbors on the ring, they are not considered here to be neighbors). Draw an arrow from x_i to x_{i-1} if $x_i = x_{i-1} + 1$ (termed *left arrow*), and from x_{i-1} to x_i if $x_{i-1} = x_i + 1$ (termed *right arrow*). In this paper we choose to denote a left arrow by '<', and a right arrow by '>'. Thus, for each two neighboring processors with states x_{i-1} and x_i, either $x_{i-1} = x_i$, or $x_{i-1} < x_i$, or $x_{i-1} > x_i$. Recall that $x_{i-1} < x_i$ means that x_{i-1} is smaller by 1 than x_i, and $x_{i-1} > x_i$ means that x_{i-1} is larger by 1 than x_i, where all arithmetic is modulo 3. For a given configuration $C = x_0, x_1, \ldots, x_{n-1}$, Dijkstra introduces the function

$$f(C) = \#left\ arrows + 2\#right\ arrows\ . \tag{1}$$

Example 1. For $n = 6$, a possible initial configuration C is

$$C: \quad x_0 = 1, x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 2, x_5 = 2\ .$$

This configuration will thus be denoted as

$$C: \qquad 1 \quad 1 \ > \ 0 \ < \ 1 \ < \ 2 \quad 2\ . \tag{2}$$

For this configuration we have $f(C) = 1 \times 2 + 2 \times 1 = 4$. □

Note 1. Using the set of configurations $x_0 > x_1 > \cdots > x_{n-1}$ as initial configurations one can easily derive the $\Omega(n^2)$ lower bound for this algorithm (the details are left to the reader). □

It follows immediately from (1) that for any configuration C of n processors

$$0 \quad \leq \quad f(C) \quad \leq \quad 2(n-1)\ . \tag{3}$$

Equation (1) is used in [8] for the proof of correctness, as follows. There are eight possible moves of the system: one possible move for processor p_0, five possible moves for any intermediate processor p_i, $0 < i < n-1$, and two possible moves for p_{n-1}. These eight possibilities are summarized in Table 1. In this table C_1 and C_2 denote the configurations before and after the move, respectively, and $\Delta f = f(C_2) - f(C_1)$. In the table we show only the local parts of these configurations. For example, in the first row, p_0 is privileged; therefore in C_1 we have $x_0 < x_1$, and in C_2 $x_0 > x_1$, and since one left arrow is replaced by the right arrow, $\Delta f = f(C_2) - f(C_1) = 1$. It is proved in [8] that each execution is infinite (that is, the scheduler can always find at least one privileged processor). Then it is shown that p_0 makes infinite number of moves. Then the execution is partitioned into *phases*, which start with a move of p_0 and end just before its next move. It is argued that the function f decreases by at least 1 after each phase. By (3) it follows that the algorithm terminates after at most 2(n-1) phases.

Table 1.

Case	**Processor**	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{\Delta f}$
0	p_0	$x_0 < x_1$	$x_0 > x_1$	$+1$
1	p_i	$x_{i-1} > x_i = x_{i+1}$	$x_{i-1} = x_i > x_{i+1}$	0
2	p_i	$x_{i-1} = x_i < x_{i+1}$	$x_{i-1} < x_i = x_{i+1}$	0
3	p_i	$x_{i-1} > x_i < x_{i+1}$	$x_{i-1} = x_i = x_{i+1}$	-3
4	p_i	$x_{i-1} > x_i > x_{i+1}$	$x_{i-1} = x_i < x_{i+1}$	-3
5	p_i	$x_{i-1} < x_i < x_{i+1}$	$x_{i-1} > x_i = x_{i+1}$	0
6	p_{n-1}	$x_{n-2} > x_{n-1}$	$x_{n-2} < x_{n-1}$	-1
7	p_{n-1}	$x_{n-2} = x_{n-1}$	$x_{n-2} < x_{n-1}$	$+1$

Though the function f enables the proof of correctness of the algorithm, it cannot be used for analyzing its complexity (that is, the number of moves from any configuration until reaching a legitimate configuration). The reason for this is that in three cases (cases 1, 2 and 5 in Table 1) the function f does not change (that is, $\Delta f = 0$), and therefore the change in the function cannot reflect the actual number of moves, that might be even unbounded. Indeed, the proof of [8] takes into account the moves of processors p_0 and p_{n-1} and the cases 3 and 4 of the intermediate processors p_i, but it does not consider the cases 1, 2 and 5.

This is the point one has to overcome in order to modify Dijkstra's proof so that it will also enable the estimate of the complexity of the algorithm. This is what we are doing in the next section, in which we first get more insight into the properties of the algorithm and its proof, and then introduce a new function with which we are able to measure its complexity.

3 Upper Bound Proof

In this section we present our main result for the upper bound of Dijkstra's algorithm for $n > 2$ (the case $n = 2$ is trivial). Our discussion includes three steps. We first introduce the function $\hat{f}$ that is a slight modification of the function f (of (1)), with which we are able to get more properties of the behavior of the algorithm. We then introduce a new function g, and discuss its properties; this function enables us to deal with the complexity of the algorithm. Finally we put all of these properties together and provide a proof for the upper bound. These three steps are presented in Sections 3.1, 3.2 and 3.3, respectively.

3.1 Preliminaries

We now present some consequences of the proof of [8] that we will later use in our proof of the upper bound. We use the function $\hat{f}$ defined on any configuration C as follows:

$$\hat{f}(C) = (\#left\ arrows - \#right\ arrows) \bmod 3\ . \tag{4}$$

The connection between the functions f and $\hat{f}$ is obvious, by (1): $f(C) = \#left\ \ arrows + 2\#right\ \ arrows = (\#left\ \ arrows - \#right\ \ arrows) + 3\#right\ \ arrows$, hence

$$\hat{f}(C) \equiv f(C) \pmod 3\ . \tag{5}$$

We now discuss the properties of the function $\hat{f}$ in a few lemmas and corollaries. Throughout the discussion we refer to the cases according to Table 1.

Lemma 1. *For any configuration C:*

a. $\hat{f}(C) = 0$ **iff** $x_{n-1} = x_0$.
b. Any move of processor p_i, $1 \le i \le n-2$, does not change the function $\hat{f}$ (that is, $\Delta\hat{f} = 0$).
c. p_{n-1} is privileged according to case 7 **iff** *$\hat{f}(C) = 0$ and $x_{n-2} = x_{n-1}$.*
d. p_{n-1} is privileged according to case 6 **iff** *$\hat{f}(C) = 2$ and $x_{n-2} > x_{n-1}$.*

Proof.

a. $\hat{f}(C) = 0$ **iff** the difference between the number of left arrows and right arrows is 0 modulo 3. Since '<' denotes an increase by 1 from x_{i-1} to x_i, and '>' denotes a decrease by 1, therefore this holds **iff** $x_{n-1} = x_0$.
b. Follows immediately from Table 1 and (5).
c. p_{n-1} is privileged according to case 7 **iff** $x_0 = x_{n-2} = x_{n-1}$. By (a) (of this lemma) this happens **iff** $\hat{f}(C) = 0$ and $x_{n-2} = x_{n-1}$.
d. p_{n-1} is privileged according to case 6 **iff** $x_0 = x_{n-2} > x_{n-1}$. It remains to show that $\hat{f}(C) = 2$. By considering the configuration C' of the first $n-1$ processors it follows by (a) that $\hat{f}(C') = 0$, and therefore $\hat{f}(C) = 2$. □

Corollary 1. *After processor p_{n-1} makes a move (case 6 or 7), we get to a configuration C for which $\hat{f}(C) = 1$.*

Proof. Note that by Table 1, move 6 decreases and move 7 increases $\hat{f}$ by 1. Hence after processor p_{n-1} moves, $\hat{f}(C) = 1$. □

The next corollary follows from Corollary 1 and Table 1; it is actually Lemma 0 of [8].

Corollary 2. *Starting from any configuration, in any prefix of an execution the number of moves of p_{n-1} is bounded by the number of moves of p_0 + 1.*

The following lemma and corollary extend this property as follows:

Lemma 2. *Starting from any configuration, and during any execution, the following holds:*

a. *For any two successive moves of processor p_{n-1} where the second move is of case 6, there is at least one move of processor p_0 between them.*
b. *For any two successive moves of processor p_{n-1} where the second move is of case 7, there are at least two moves of processor p_0 between them.*

Proof. By Corollary 1, after the first of these two successive moves, $\hat{f}$ becomes 1.

a. If the second move of processor p_{n-1} is of case 6, then between the two moves of p_{n-1}, $\hat{f}$ had to change from 1 to 2. Since moves 1-5 do not change $\hat{f}$, we conclude that processor p_0 moved at least once between them.
b. If the second move is of case 7, then between two moves of p_{n-1}, $\hat{f}$ had to change from 1 to 0. Since the only processor that can change the value of $\hat{f}$ is p_0, this means that p_0 had to move at least twice between them. □

By Lemma 2 it follows that

Corollary 3. *Starting from any configuration, in any prefix of an execution the number of moves of case 6 plus twice the number of moves of case 7 of p_{n-1} is bounded by the number of moves of p_0 + 1.*

We summarize the properties of the functions $\hat{f}$ in Table 2. In this table we also include the function g discussed in Section 3.2. In this table we denote the changes in the function $\hat{f}$ ($\Delta\hat{f} = \hat{f}(C_2) - \hat{f}(C_1)$) and g ($\Delta g = g(C_2) - g(C_1)$).

3.2 The Function g

We now introduce the function g. This function decreases by at least 1 during each move of any intermediate processor p_i (cases 1-5). Unfortunately, moves of processors p_0 and p_{n-1} increase g. However, by combining results of Section 3.1 and the properties of g we manage to derive the upper bound on the number of moves to reach stabilization.

Table 2.

Case	**Processor**	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{\Delta g}$	$\mathbf{\Delta \hat{f}}$
0	p_0	$x_0 < x_1$	$x_0 > x_1$	$n-2$	$+1$
1	p_i	$x_{i-1} > x_i = x_{i+1}$	$x_{i-1} = x_i > x_{i+1}$	-1	0
2	p_i	$x_{i-1} = x_i < x_{i+1}$	$x_{i-1} < x_i = x_{i+1}$	-1	0
3	p_i	$x_{i-1} > x_i < x_{i+1}$	$x_{i-1} = x_i = x_{i+1}$	$5-3n \leq -1$	0
4	p_i	$x_{i-1} > x_i > x_{i+1}$	$x_{i-1} = x_i < x_{i+1}$	$3i-3n+5 \leq -1$	0
5	p_i	$x_{i-1} < x_i < x_{i+1}$	$x_{i-1} > x_i = x_{i+1}$	$-3i+2 \leq -1$	0
6	p_{n-1}	$x_{n-2} > x_{n-1},\ \hat{f} = 2$	$x_{n-2} < x_{n-1},\ \hat{f} = 1$	$n-2$	-1
7	p_{n-1}	$x_{n-2} = x_{n-1},\ \hat{f} = 0$	$x_{n-2} < x_{n-1},\ \hat{f} = 1$	$2n-4$	$+1$

Given a configuration $C = x_0, x_1, \ldots, x_{n-1}$, we define the function $g(C)$ as follows:

$$g(C) \;=\; \sum_{\substack{1 \leq i \leq n-1 \\ x_{i-1} < x_i}} (n+i-3) \;+\; \sum_{\substack{1 \leq i \leq n-1 \\ x_{i-1} > x_i}} (2n-i-3) \tag{6}$$

Example 2.

- If in a configuration $C,\ x_0 = x_1 = \cdots = x_{n-1},$ then $g(C) = 0.$ (7)
- If in a configuration $C,\ x_0 = \cdots = x_{i-1} < x_i = \cdots = x_{n-1},$ then
  $g(C) = n+i-3.$ (8)
- If in a configuration $C,\ x_0 = \cdots = x_{n-i-1} > x_{n-i} = \cdots = x_{n-1},$ then
  $g(C) = n+i-3.$ (9)
- If in a configuration $C,\ x_0 < x_1 = \cdots = x_{n-2} > x_{n-1},$ then
  $g(C) = 2n-4.$
- If in a configuration $C,\ x_0 < x_1 < \cdots < x_{n-1},$ then
  $$g(C) = \sum_{i=1}^{n-1}(n+i-3) = \frac{3}{2}(n-1)(n-2).$$
- If n is odd and in a configuration $C,\ x_0 > \cdots > x_{\frac{n-1}{2}} < x_{\frac{n+1}{2}} < \cdots < x_{n-1},$ then
  $$g(C) = \sum_{i=1}^{\frac{n-1}{2}}(2n-i-3) + \sum_{i=\frac{n+1}{2}}^{n-1}(n+i-3) = \frac{7}{4}n^2 - 5n + \frac{13}{4}. \tag{10}$$
- If n is even and in a configuration $C,\ x_0 > \cdots > x_{\frac{n}{2}} < x_{\frac{n}{2}+1} < \cdots < x_{n-1},$ then
  $$g(C) = \sum_{i=1}^{\frac{n}{2}}(2n-i-3) + \sum_{i=\frac{n}{2}+1}^{n-1}(n+i-3) = \frac{7}{4}n^2 - 5n + \frac{12}{4}. \tag{11}$$

□

The changes in the function g in each of the eight possible moves are summarized in Table 2. These changes can be obtained by using the examples above. For example, for a move of case 0 we get by (8) and (9) that $\Delta g = (2n-4)-(n-2) = n-2$, and for a move of case 5 $\Delta g = (2n-i-3)-(2n+2i-5) = 2-3i \leq -1$.

Lemma 3. *For any configuration C, $0 \leq g(C) \leq \frac{7}{4}n^2 - 5n + \frac{13}{4}$.*

Proof. For any $1 \leq i \leq n-1$, the following holds:

$$n-2 \leq n+i-3 \leq 2(n-2),$$

$$n-2 \leq 2n-i-3 \leq 2(n-2).$$

This, together with (7), implies $\min_C g(C) = 0$.

The maximal value of g is for a configuration C in which the first half of the arrows point to the right and all others point to the left. Formally, since $n+i-3 \leq 2n-i-3 \;\Leftrightarrow\; 2i \leq n$, then:

$$\max_C g(C) \;=\; \sum_{i=1}^{n-1} \max(\,n+i-3,\; 2n-i-3\,) = \sum_{i=\lfloor \frac{n}{2} \rfloor+1}^{n-1} (n+i-3) + \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (2n-i-3)\,.$$

By (10) and (11) we conclude $\max_C g(C) = \frac{7}{4}n^2 - 5n + \frac{13}{4}$. □

3.3 Main Contribution

We now turn to present our main result. Following the proof of [8] we know that starting from any initial configuration the algorithm will get to a legitimate configuration in finite time. We are now ready to measure this time in the following theorem.

Theorem 1. *Assume the system starts from an initial configuration C, and that an execution of Dijkstra's algorithm gets into a legitimate configuration in T moves. If within these first T moves there are exactly x, y and z moves of cases 0, 6 and 7, respectively (the cases refer to Table 2), then*

$$T \;\leq\; g(C) + x(n-1) + y(n-1) + z(2n-3).$$

Proof. According to Table 2, each move of any intermediate processor p_i decreases function g at least by 1 (cases 1-5) and each of the x, y, z moves of case 0, 6, 7 increase the function g by $n-2$, $n-2$, $2n-4$, respectively.

Therefore the total number of moves performed by all the intermediate processors p_i is bounded by $g(C) + x\ (n-2) + y\ (n-2) + z\ (2n-4)$. This is true since otherwise we'll get into a configuration C' with $g(C') < 0$, which contradicts Lemma 3.

Since we assumed that the total number of moves performed by p_0 and p_{n-1} is exactly $x+y+z$, it follows that

$$\begin{aligned} T &\leq g(C) + x(n-2) + y(n-2) + z(2n-4) + (x+y+z) = \\ &= g(C) + x(n-1) + y(n-1) + z(2n-3). \end{aligned}$$

□

In the discussion in Section 2 we mentioned that the number of phases is bounded by $2(n-1)$, and therefore $x \le 2(n-1)$. In addition, by Corollary 2 it follows that $y+z \le x+1$. Therefore we get

$$\begin{aligned}
T &\le g(C) + x(n-1) + y(n-1) + z(2n-3) \le \\
&\le \max g + x(n-1) + (y+z)(\max(n-1, 2n-3)) = \\
&= \max g + x(n-1) + (x+1)(2n-3) = \\
&= \max g + x(n-1+2n-3) + (2n-3) \le \\
&\le \max g + 2(n-1)(3n-4) + 2n-3 = \\
&= \max g + 6n^2 - 12n + 5 \le \\
&\le \frac{7}{4}n^2 - 5n + \frac{13}{4} + 6n^2 - 12n + 5 = \\
&= 7\frac{3}{4}n^2 - 17n + 8\frac{1}{4} \le 7\frac{3}{4}n^2 \ .
\end{aligned}$$

Note that we achieved this upper bound by combining the properties of the function g with the original proof of [8] (in particular Corollary 2) and without considering the function $\hat{f}$.

If we also use the function $\hat{f}$, then we can apply similar argument and use Corollary 3. By this corollary we have $y + 2z \le x + 1$, and therefore:

$$\begin{aligned}
T &\le g(C) + x(n-1) + y(n-1) + z(2n-3) \le \\
&\le \max g + x(n-1) + n(y+2z) - (y+3z) \le \\
&\le \max g + x(n-1) + n(y+2z) \le \\
&\le \max g + x(n-1) + n(x+1) = \\
&= \max g + x(2n-1) + n \le \\
&\le \max g + 2(n-1)(2n-1) + n = \\
&= \max g + 4n^2 - 5n + 2 \le \\
&\le \frac{7}{4}n^2 - 5n + \frac{13}{4} + 4n^2 - 5n + 2 = \\
&= 5\frac{3}{4}n^2 - 10n + 5\frac{1}{4} \le 5\frac{3}{4}n^2 \ .
\end{aligned}$$

A more careful analysis can be used to show an upper bound of $4n^2$ ([5]). Therefore the following theorem holds:

Theorem 2. *Starting from any initial configuration, any execution of Dijkstra's algorithm gets into a legitimate configuration in at most $O(n^2)$ moves.*

Recall that the analysis assumed a centralized scheduler. We argue that the same bound applies also for a distributed scheduler. This is the case since it can be shown that any move done concurrently by $k > 1$ processors can be simulated by a sequence of exactly k moves of individual processors (see [3]); this follows also from the fact that it is not possible that all n processors are privileged simultaneously.

References

1. Beauquier, J., Debas, O.: An optimal self-stabilizing algorithm for mutual exclusion on bidirectional non uniform rings. In: Proceedings of the Second Workshop on Self-Stabilizing Systems, pp. 17.1–17.13 (1995)
2. Beauquier, J., Johnen, C., Messika, S.: Brief announcement: Computing automatically the stabilization time against the worst and the best schedules. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 543–547. Springer, Heidelberg (2006)
3. Burns, J.E., Gouda, M.G., Miller, R.E.: On relaxing interleaving assumptions. In: Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89 (1989)
4. Chang, E.J.H., Gonnet, G.H., Rotem, D.: On the costs of self-stabilization. Information Processing Letters 24, 311–316 (1987)
5. Chernoy, V., Shalom, M., Zaks, S.: Better bounds for Dijkstra's 3rd algorithm on mutual exclusion. (in preparation)
6. Cobb, J.A., Gouda, M.G.: Stabilization of general loop-free routing. Journal of Parallel and Distributed Computing 62(5), 922–944 (2002)
7. Dijkstra, E.W.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery 17(11), 643–644 (1974)
8. Dijkstra, E.W.: A belated proof of self-stabilization. Distributed Computing 1, 5–6 (1986)
9. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
10. Kessels, J.L.W.: An exercise in proving self-stabilization with a variant function. Information Processing Letters 29, 39–42 (1988)
11. Kruijer, H.S.M.: Self-stabilization (in spite of distributed control) in tree-structured systems. Information Processing Letters 8, 91–95 (1979)
12. Nakaminami, Y., Kakugawa, H., Masuzawa, T.: An advanced performance analysis of self-stabilizing protocols: stabilization time with transient faults during convergence. In: IPDPS 2006. 20th International Parallel and Distributed Processing Symposium, 25-29 April 2006. Rhodes Island, Greece, (2006)
13. Schneider, M.: Self-stabilization. ACM Computing Surveys 25, 45–67 (1993)
14. Tchuente, M.: Sur l'auto-stabilisation dans un réseau d'ordinateurs. RAIRO Informatique Theoretique 15, 47–66 (1981)
15. Tsuchiya, T., Tokuda, Y., Kikuno, T.: Computing the stabilization times of self-stabilizing systems. IEICE Transactions on Fundamentals of Electronic Communications and Computer Sciences E83A(11), 2245–2252 (2000)

Stability of the Multiple-Access Channel Under Maximum Broadcast Loads

Bogdan S. Chlebus[1], Dariusz R. Kowalski[2], and Mariusz A. Rokicki[3]

[1] Department of Computer Science and Engineering, University of Colorado at Denver and Health Sciences Center, Denver CO 80217, USA
[2] Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
[3] Centre National de la Recherche Scientifique, Université Paris Sud, 91405 Orsay Cedex, France

Abstract. We investigate deterministic broadcasting on multiple-access channels in the framework of adversarial queuing. A protocol is stable when the number of packets stays bounded, and it is fair when each packet is eventually broadcast. We address the question if stability and fairness can be achieved against the maximum injection rate of one packet per round. We study three natural classes of protocols: acknowledgment based, full sensing and fully adaptive. We show that no adaptive protocol can be both stable *and* fair for the system of at least two stations against leaky-bucket adversaries, while this is achievable against window adversaries. We study in detail small systems of exactly two and three stations attached to the channel. For two stations, we show that bounded latency can be achieved by a full-sensing protocol, while there is no stable acknowledgment-based protocol. For three stations, we show that bounded latency can be achieved by an adaptive protocol, while there is no stable full-sensing protocol. We develop an adaptive protocol that is stable for any number of stations against leaky-bucket adversaries. The protocol has $\mathcal{O}(n^2)$ packets queued simultaneously, which is proved to be best possible as an upper bound. We show that protocols that do not use queue sizes at stations in an effective way or are greedy by having stations with nonempty queues withhold the channel cannot be stable in systems of at least four stations.

Keywords: multiple-access channel, deterministic broadcast, adversarial queuing, stability, fairness, latency.

1 Introduction

Multiple access channels model distributed communication environments supporting broadcasting. The properties of a system that make it multiple-access channel are as follows. There are a number of stations attached to a transmission medium. A packet transmitted by a station reaches all the stations, including the sender. A transmission is successfully received if it does not overlap with any other transmissions. We consider a synchronous model in which stations use local clocks ticking at the same rate. A station transmits at a round determined by its clock, with a transmission filling the whole round. This means that if at least two stations transmit at a round, then no messages are received at this round.

We investigate stability, fairness and latency of broadcast protocols for multiple-access channels. Stability means that the number of packets stored in local queues is bounded at all rounds of an execution. The injection rate of one packet per round on the average is the maximum injection rate to be possibly handled in a stable manner, since at most one packet can be successfully broadcast at a round. Fairness denotes the property that each packet is eventually successfully broadcast. Latency of packets is defined as an upper bound on time spent waiting in queues.

Our results. We address the question if stability, preferably with bounded latency of packets, can be achieved against the maximum injection rate 1 on multiple-access channels. The answer turns out to depend on the following parameters: the kind of adversary, the class of protocols, and finally the number n of stations attached to the channel. We consider two standard adversarial models: window adversaries and leaky-bucket adversaries, all adversaries with injection rate 1. We study three classes of protocols: acknowledgment based, full sensing and fully adaptive.

We show that the kind of adversary matters. For leaky-bucket adversaries, achieving both stability *and* fairness is impossible, except for the trivial case of a single station. Regarding just stability with respect to leaky-bucket adversaries, this is not possible by full-sensing protocols but achievable by adaptive protocols. The stable protocol we develop has the stations store $\mathcal{O}(n^2)$ packets in queues, which is shown to be the asymptotically best possible bound in general.

For window adversaries, the situation is more complex. Impossibility of both stability *and* fairness holds when there are at least four stations in the system, as was shown in [8]. For three stations, bounded latency is achievable by adaptive protocols, but no full-sensing protocol can be stable. For two stations, bounded latency is achievable by full-sensing protocols, but no acknowledgment-based protocol is stable.

We show additionally that protocols that are stable need to have some properties and do not have others. In particular, stable protocols have to use the queue sizes at stations in a sufficiently explicit way, and cannot have stations behave greedily by withholding the channel after a successfull transmission.

Related work. Most of the previous work on dynamic broadcasting on multiple-access channels has concentrated on scenarios when packets are injected subject to statistical constraints. When a broadcast environment is randomized, the behavior of the system can be modeled as a Markov chain and stability is captured by ergodicity. The well known protocols like Aloha [1] and binary exponential backoff [17] have been proposed to handle broadcast with stochastic injection rates; Gallager [9] gives an overview of early research in this direction. For recent work, see the papers by Goldberg, Jerrum, Kannan and Paterson [11], Goldberg, MacKenzie, Paterson and Srinivasan [12], Håstad, Leighton and Rogoff [13], and Raghavan and Upfal [18].

Adversarial queuing was proposed by Borodin, Kleinberg, Raghavan, Sudan and Williamson [7] as an approach to study stability of contention-resolution protocols in store-and-forward routing. They showed, among other things, that a directed acyclic network is stable with injection rate 1, for any greedy contention-resolution protocol. Universal stability of a protocol denotes stability in any network, and universal stability of a network denotes stability of an arbitrary protocol in the network, both under constant injection rates smaller than 1. These notions were introduced by Andrews, Awerbuch, Fernández, Leighton, Liu and Kleinberg [4]; they were later studied by Gamarnick [10] and Alvarez, Blesa and Serna [3]. Bhattacharjee, Goel and Lotker [6] showed the the popular FIFO protocol can be unstable at arbitrarily low injection rates. Lotker, Patt-Shamir and Rosén [15] showed that every work-preserving contention-resolution protocol is stable if injection rate is smaller than $1/(D+1)$, where D is an upper bound on the length of any path that a packet needs to traverse. Koukopoulos, Mavronicolas, Nikoletseas and Spirakis [14] addressed the question of how structural properties of networks affect stability of contention-resolution protocols. Adaptive protocols have packets carry only their destination addresses, rather than complete routing paths; the stability of such protocols was considered by Aiello, Kushilevitz, Ostrovsky and Rosén [2].

Bender, Farach-Colton, He, Kuszmaul and Leiserson [5] studied stability of randomized backoff on multiple-access channels in the adversarial queuing model, where stability meant that throughput is as large as injection rate. They showed, among other things, that exponential backoff is unstable for rates $\rho \geq c \lg\lg n/\lg n$, for a sufficiently large constant c, where n denotes the number of stations. Stability of deterministic broadcast protocols for multiple access channels in the framework of adversarial queueing was first considered by Chlebus, Kowalski and Rokicki [8]. They defined strong stability to hold when the number of queued packets is proportional to the maximum number of packets that an adversary may inject simultaneously in an execution. They showed that no adaptive protocol for a channel with collision detection can be strongly stable for injection rates that are $\omega(\frac{1}{\log n})$ and gave a full-sensing protocol for a channel with collision detection that is both universally stable and strongly stable for injection rates at most $\frac{1}{2(\lceil \lg n \rceil+1)}$. For a channel without collision detection, they developed a full-sensing protocol that is both universally stable and strongly-stable for injection rates at most $\frac{1}{c \lg^2 n}$, for some $c > 0$. They showed existence of an acknowledgment-based protocol that is strongly stable for injection rates at most $\frac{1}{cn \lg^2 n}$, for some $c > 0$, and developed an explicit acknowledgment-based protocol that is strongly stable for injection rates at most $\frac{1}{27n^2 \ln n}$. Finally, they showed that no acknowledgement-based protocol is stable for injection rates larger than $\frac{3}{1+\lg n}$.

2 Technical Preliminaries

A multiple access channel is a broadcast system with specific additional properties that we discuss in this section. We use letter n to denote the number of stations attached to a communication medium. Each of the n stations has a unique name assigned to it, each name is an integer in the range $[1, n]$.

Multiple-access channel. What makes a broadcast system *multiple-access channel* is the property that a transmission is successfully received by all the stations if and only if the transmission does not overlap with any other transmissions. A packet successfully broadcast is said to be *heard* on the channel. We consider a synchronous channel in which executions of protocols are structured as sequences of rounds so that overlapping transmissions occur at the same round. Multiple transmissions at the same round result in a conflict for access to the channel, which is called *collision*. When no stations transmit at a round, then the feedback that the stations receive from the channel is called *silence*; we may also say about such a round that the channel or the round is *silent*. A channel may be equipped with a *collision detection* mechanism, which makes the stations able to distinguish between silence and collision at a round. If no collision detection mechanism is available, then the stations perceive collision as silence. A round during which no contents is heard as transmitted on the channel is said to be *void*; either silence or collision are merely obtained as the feedback from the channel during a void round.

Adversaries. We consider worst-case performance of protocols that need to handle traffic determined by an adversarial setting. An adversary is defined by a set of patterns of injections of packets into stations. We consider two standard classes of adversaries: window adversaries and leaky-bucket adversaries. In the context of adversarial queuing, window adversaries were first used by Borodin, et al. [7] and leaky-bucket adversaries by Andrews, et al. [4].

The letter ρ denotes the rate of packet injection, which intuitively means the average number of packets injected over a sufficiently long time interval. Let $0 < \rho \leq 1$ be an injection rate and w a positive integer called the *window size*. The *window adversary of type* (ρ, w) may inject at most ρw packets in each contiguous segment of w rounds into any set of stations. Let $0 < \rho \leq 1$ be an injection rate and b a positive integer. The *leaky-bucket adversary of type* (ρ, b) may inject at most $\rho t + b$ packets in each nonempty contiguous segment of $t > 0$ rounds into any set of stations. An adversary is said to be *of injection rate* ρ when it is either of window-type (ρ, w) or of leaky-bucket-type (ρ, b), for some w or b, respectively. The maximum number of packets that an adversary may inject at a round is called *burstiness*. The window adversary of type (ρ, w) has burstiness ρw. The leaky-bucket adversary of type (ρ, b) has burstiness $\rho + b$.

The injection rate 1 is maximum in the sense that any larger injection rate certainly allows the adversary to make the number of packets queued at stations grow unbounded, as at most one packet per round can be heard. We consider only adversaries of injection rates exactly 1. Such adversaries differ among themselves by their burstiness, which is either the window w, for a window adversary, or $b+1$, for a leaky-bucket adversary. Both models of window adversaries and leaky-bucket adversaries are equivalent for injection rates strictly smaller than 1, while the leaky-bucket adversary of injection rate 1 can generate sequences of injections not captured by any window adversary of rate 1, as was showed by Rosén [19].

It follows that a possibility-type of a result for leaky-bucket adversaries of rate 1 holds automatically for window adversaries of rate 1, while an impossibility result showed for window adversaries (of rate 1) holds automatically for leaky-bucket adversaries (of rate 1).

Protocols as automata. A protocol executed in a system of stations attached to a multiple-access channel is formally modeled as automaton in a way standard in representing distributed systems [16]. The *state* of a station is determined by the values of its local variables. Each station has a queue of pending packets, which operates in a FIFO manner. The contents of transmitted packets do not affect state transitions, in the sense that each packet is treated as an abstract token. The value of a local clock at a station is not a component of a state: the clocks are used only to determine consecutive rounds. No packets are ever discarded by dequeuing without a prior successful transmission.

An *execution* is a sequence of events occurring at consecutive rounds, as determined by synchronized clocks at stations. A *message* received from the channel consists of a transmitted packet, if any, and additional control bits, if any. The event at a round is represented by the following sequence of consecutive actions at each station:

(1) the station either performs a transmission or pauses, accordingly to its state; the packet to transmit is taken from the queue at the station.
(2) the station receives a feedback from the channel, in the form of either hearing a message or a collision signal or silence;
(3) new packets are injected into the station, if any; these packets are enqueued.
(4) a state transition occurs at the station, based on the state at the end of the previous round, the packets injected at this round, and the feedback from the channel at this round; in particular, the station that transmitted successfully dequeues the transmitted packet.

Classes of protocols. We consider three classes of deterministic protocols, as introduced in [8].

A broadcasting protocol is *adaptive* if control bits may be piggybacked on transmitted packets and also control messages without any packet may be transmitted. Feedback from the channel falls into one of the following five categories: (1) packet without an attached control message, (2) packet with a control message piggybacked on it, (3) control message without any packet, (4) silence, or finally (5) collision.

A broadcasting protocol is *full sensing* if no control bits may be used, neither attached to packets nor transmitted as separate messages. A feedback received from the channel is in one of the following three forms: (1) packet, (2) silence, or finally (3) collision.

A broadcasting protocol is defined to be *acknowledgement based* when a state transition depends only on which consecutive round it is spent to process the currently handled packet, where the numbering of rounds starts from the first round when the packet started to be processed. A feedback received from the channel, that matters for a station running an acknowledgment-based protocol, is of one of two forms: (1) the packet just transmitted by the station or (2) something else.

A protocol $\mathcal{P}$ is said to be *queue-size oblivious* if the size of queue is not used in state transitions of $\mathcal{P}$. In particular, such protocols do not maintain queue sizes, and adaptive queue-size oblivious protocols cannot include queue size among control bits attached to packets. Any acknowledgment-based protocol is queue-size oblivious. All the protocols given in [8] are queue-size oblivious; these protocols were analyzed in [8] for injection rates smaller than 1.

Quality of service. The measures of quality that we work with are to express the broadcast functionality of the system. A protocol $\mathcal{P}$ is said to be *fair* against an adversary $\mathcal{A}$ when every packet injected by $\mathcal{A}$ is eventually heard on the channel. Another property concerns stability, which occurs when the queues at stations are bounded. More precisely, a protocol $\mathcal{P}$ is said to be *stable* against an adversary $\mathcal{A}$ if, for any execution of the protocol in a system of n stations against this adversary, there is a number $s(n)$ such that the number of packets stored in the queues is at most $s(n)$ at any round. A protocol $\mathcal{P}$ is said to have *bounded latency* against an adversary $\mathcal{A}$ if, in any execution of the protocol in a system of n stations against the adversary $\mathcal{A}$, there is a number $\ell(n) > 0$ such that, for each injected packet, the time interval from the injection of the packet until the packet is heard on the channel is of length at most $\ell(n)$. If a station stores some x packets in its queue at a round of an execution of a protocol, then the latency of the protocol is at least x in this execution. It follows that a bounded-latency protocol is both fair and stable. Any protocol given in this paper, that is both fair and stable against some adversary, happens also to be of bounded latency against this adversary.

3 Two Stations

We consider two stations attached to a multiple-access channel. It turns out that, against leaky-bucket adversaries, an adaptive protocol can handle traffic in a stable way on two stations, while achieving both stability and fairness is impossible even for two stations. The situation is different with respect to window adversaries: there is a full-sensing bounded-latency protocol, while no acknowledgment-based protocol can be stable.

Theorem 1. *No protocol can be both stable and fair for the system of at least two stations against the leaky-bucket adversary with burstiness* 2.

Proof. Consider two stations p and q and (an adaptive) protocol $\mathcal{P}$ that is both stable and fair. We construct an execution of $\mathcal{P}$ that contains infinitely many void rounds, while the adversary injects at least one packet per round on the average.

We determine an execution by specifying *milestone rounds* $t_0, t_1, t_2, \ldots$, where $t_i < t_{i+1}$ for any $i \geq 0$, and what happens between these rounds. Define t_0 to be the first round of the execution; this round is void because no packets have been injected before the round. Let the adversary inject one packet at p at the first round. Suppose the execution has been defined until some round t_i, for $i \geq 0$. Next we define the round $t_{i+1} > t_i$ and what happens in the interval between t_i and t_{i+1}.

If the round $t_i + 1$ is void, then the adversary injects one packet at p at this round and we set $t_{i+1} = t_i + 1$. Suppose the round $t_i + 1$ is not void: then the adversary does not inject any packet at round $t_i + 1$. The adversary keeps injecting one packet at p per round starting from the round $t_i + 2$. Since the protocol is fair, eventually station q has an empty queue; let u be the first round after $t_i + 1$ when this is the case. When no packet is heard on the channel at some round w, where $w > u$, then p does not transmit at round w. If such a round w exists, then set $t_{i+1} = w$ and let the adversary inject two packets at p at this round. Otherwise, when w does not exist, station p keeps transmitting at every round after u. Let the adversary inject an additional packet at q at round $u + 1$. This behavior is consistent with the definition of a leaky-bucket adversary with burstiness 2 because the adversary did not inject any packet at round $t_i + 1$. This means that starting from round $u + 1$, station q has a pending packet while p keeps transmitting. Station q needs to transmit the pending packet at least once, because otherwise the protocol would not be fair. Let q transmit this packet for the first time at some round $t > u + 1$. Since p also transmits at t, there is a collision, which makes the round t void. Define t_{i+1} to be this t and let the adversary inject two packets at p at this round.

This pattern of injections is consistent with the definition of the leaky-bucket adversary of burstiness 2, while the number of packets at the queues of the two stations at round t_i, for $i \geq 0$, is at least $i + 1$. □

Theorem 2. *No full-sensing protocol can be stable for a system of at least two stations against the leaky-bucket adversary with burstiness* 2.

Window adversaries. Next we show that bounded latency can be achieved in a system of two stations by a full-sensing protocol against any window adversary. We start with a full-sensing protocol that has a positive integer number i in its code interpreted as a window; the protocol is called 2-Full-Sensing(i). The protocol is structured as a sequence of consecutive phases, with each phase consisting of exactly i rounds. Packets injected in the course of a phase are considered as *available* during the next phase. A phase is used to broadcast precisely the packets injected in the immediately preceding phase.

In the first phase of the first i rounds all stations simply wait. Consider the first round of one of the next phases. If one of the stations realizes that it contains exactly i available packets, then it spends the whole phase of i rounds transmitting these packets. Suppose that none of the stations contains exactly i available packets. In this case p starts transmitting at the first round of the phase. Node q counts the number of packets transmitted by p and starts transmitting its packets immediately after silence or when q realizes at a round that its number of available packets is equal to the number of the remaining rounds of the phase. Each station needs to maintain two counters: one is the size of the queue and the other is the consecutive round in a phase. The count of rounds is updated by incrementing the value by 1 modulo i, while the count of the queued packets is updated following insertions and successful transmissions.

Next we define an extension called 2-Full-Sensing to handle an adversary of arbitrary window. The protocol works by trying to run protocols 2-Full-Sensing(i), for consecutive values of i, starting from 2-Full-Sensing(1) to test the window $w = 1$. A transition from 2-Full-Sensing(i) to 2-Full-Sensing($i + 1$) requires a way to handle packets in the queues when 2-Full-Sensing($i+1$) is invoked; such packets are called *old*. If we started an execution with 2-Full-Sensing(i), where the window i implicit in 2-Full-Sensing(i) is the same as that of the adversary, then no collisions ever occur, by the properties of 2-Full-Sensing(i) discussed above. Otherwise a collision can possibly occur at an event. Such a collision can be detected by the stations, since both of them transmit while none can hear a packet. If a collision occurs in the course of executing 2-Full-Sensing(i), then both stations invoke 2-Full-Sensing($i + 1$), which is to run quietly in the background as follows. The action to be performed at a round is not executed but instead enqueued in an additional local queue DELAY operating in a FIFO manner. At the same time: first station p transmits all its old packets, and after a silent round station q does the same. After another silent round, both stations start executing instructions obtained by dequeuing DELAY, while simultaneously they keep enqueueing the actions of 2-Full-Sensing($i + 1$) as it is running in the background.

Theorem 3. *Protocol* 2-Full-Sensing *has bounded latency for a system of two stations against any window adversary.*

Theorem 4. *No acknowledgment-based protocol is stable in a system of two stations against the window adversary of burstiness* 2*.*

4 Three Stations

We show that there is an adaptive protocol that handles injections of a packet per round in a stable and fair way in a system of three stations, while no full-sensing protocol can provide even stability. We consider only window adversaries in this section. The three stations are named p, q and r. The stations are ordered $\langle p, q, r\rangle$ in a cyclic fashion; when we refer to the next station after a given one, we mean this cyclic ordering. If g is a station, then the station immediately following g is denoted g', and the station immediately following g' is denoted g''.

Adaptive protocol of bounded latency. We start with an adaptive protocol designed for a specific window adversary. The protocol is called 3-Adaptive-Window(i), where i is interpreted as the window of the adversary. The protocol is structured as a sequence of consecutive phases, each *phase* consists of i consecutive rounds. Since the injection rate is 1, there are at most i packets injected during a phase. A phase is used to send the packets injected in the previous phase and no other packets; the packets injected in a phase are called *available* during the next phase. Stations may attach control information to transmitted packets, so the protocol is adaptive. For instance, when a station transmits the last pending packet, then it may attach a label indicating this property of the packet, which in turn may allow some other station to take over without a delay, so that the stations can transmit back to back.

For each phase, there is a station designated to be the *last* for the phase. The protocol is structured such that the last station of the phase transmits at the last round of the phase. If the last station still has a packet to transmit during the last round of the phase, then the packet is transmitted and a control bit is attached to the packet, otherwise only a control bit is sent. The control bit is to indicate whether any packets have been injected into the last station in the current phase. In the first phase, all the stations pause through the first $i-1$ rounds. Station p is designated to be the last one for the fist phase and it transmits in the last round i of the first phase. Consider an arbitrary phase, called simply *current*, and let g be the last station of this phase. We consider two cases, depending on whether g has received any packets in the current phase to be transmitted in the *next* phase.

If g received packets in the current phase, then g starts the next phase with a sequence of transmissions of all its available packets. If g has exactly i such packets, then g is also the last station for the next phase. Otherwise, when g transmits its last available packet in the next phase, g attaches the ‘over’ control bit to the last packet. After hearing the ‘over’ signal, the stations g' and g'' know how many rounds have remained in the next phase; let t be this number. If any station among g' or g'' has received t packets in the current phase, then this station unloads all its t pending packets in the remaining rounds of the next phase, and also gains the status to be last for the next phase. Otherwise station g' starts unloading its available packets, if any, while g'' gains the status to be last for the next phase. If g' does not have any available packets, then g' simply pauses, otherwise g' attaches the ‘over’ bit to the last transmitted packet. When g'' hears either silence or the ‘over’ signal, then g'' starts unloading its available packets, if any. Finally, g'' transmits at the last round of the phase.

Next consider the case when g has not received any packets in the current phase. Station g informes the remaining stations about this fact by the transmission in the last round of the current phase. Now the situation is similar as in the previous case after g sent the ‘over’ signal, with $t = i$. If any among g' or g'' has received i packets in the current phase, then this station unloads all its i available packets in the remaining rounds of the next phase, and also gains the status to be last for the next phase. Otherwise station g' starts unloading its available packets, if any, while g'' gains the status to be last for the next phase. Station g'' takes over as soon as either silence or the ‘over’ signal is heard.

Lemma 1. *Protocol* 3-ADAPTIVE-WINDOW(w) *is of bounded latency against the adversary with window size w in a system of three stations.*

We show next that there is a stable and fair adaptive protocol for three stations that does not know the window of the adversary but relies on the mechanism of collision detection. The protocol is called 3-Adaptive-Col-Det. It is obtained by modifying protocol 3-Adaptive-Window(i) as follow. Initially protocol 3-Adaptive-Col-Det runs 3-Adaptive-Window(1) to try the window $w = 1$. When a collision occurs while running 3-Adaptive-Window(i), then 3-Adaptive-Window($i+1$) is invoked. We apply a similar approach as in Section 3 by using a local queue DELAY and having 3-Adaptive-Window($i+1$) run quietly in the background. Packets stored already in the local queues at a round when 3-Adaptive-Window($i+1$) is invoked are called *old*. First the old packets are transmitted, by the stations p, q and r, in this order. A transition to the next station is indicated either by a control bit 'over' attached to the last old packet of a station or only this bit transmitted when the station does not have any old packets. After all the old packets have been heard, stations proceed to execute 3-Adaptive-Window($i + 1$) by enqueueing each action in DELAY while executing what is obtained be dequeuing DELAY at the same round.

Lemma 2. *Protocol* 3-Adaptive-Col-Det *is of bounded latency in a system of three stations for channels with collision detection.*

Next we present the ultimate protocol for three stations for the channel without collision detection, it is called 3-Adaptive. The protocol simulates collisions by silent rounds. The underlying idea is that if a station transmits at a round and hears silence, then collision occurred at this round, since otherwise the station would here its own packet. This might occur with only two stations involved and detecting collision, while the third one would not realize that collision occurred. To cope with this, we use the property of protocol 3-Adaptive-Col-Det that the last station for a phase transmits at the last round of the phase.

The details of the simulation of collision detection in 3-Adaptive-Col-Det are as follows. Consider a phase that is second or later after an invocation of 3-Adaptive-Window(i) in 3-Adaptive-Col-Det, for $i \geq 1$. Suppose a collision occurs in this phase. This collision is detected by at least two stations. At least one of these stations is not last in the phase: this station waits till the last round of the phase and transmits a dummy message. All the stations hear silence at this round, and so all of them learn of collision, which results in invocation of 3-Adaptive-Window($i + 1$).

Theorem 5. *Protocol* Adaptive-3 *is of bounded latency for a system of three stations against window adversaries.*

Theorem 6. *No full-sensing protocol can be stable for three stations against the window adversary of burstiness* 2*.*

5 More Stations

It was shown in [8] that no protocol can be both fair *and* stable against window adversaries for a system of at least four stations. We develop an adaptive protocol that is stable for any number of stations. Next we show that stability cannot be achieved for restricted classes of adaptive protocols against window adversaries in systems with at least four stations.

Stable protocol. We describe an adaptive protocol Move-Big-To-Front(n) and show that it is stable in any system of $n > 0$ stations. The protocol schedules exactly one station to transmit at each round, so collisions never occur. This is implemented by using a logical 'token' giving the right to transmit, which is assigned in such a way that at each round exactly one station holds the token.

Every station maintains a list of all the stations in its local memory. The lists are initialized as sorted in increasing order of names of stations. Initially the first station in the list holds the token.

The protocol is executed at a given round as follows. Station p with the token broadcasts a packet, if it has any. If the station with the token does not have a pending packet, then the station does not transmit, which results in a silent round. A station considers itself *big* at a round when it has at least n pending packets in its queue. A big station attaches a control bit to indicate this status to each packet it transmits while big. After a station announces itself to be big, it is moved to the front of the list and keeps the token for the next round. After a station broadcasts and it is not big, then the token is moved immediately to the next station in the list; the token from the last station in the list is moved directly to the first one.

Theorem 7. *If protocol* Move-Big-To-Front(n) *is executed against a leaky-bucket adversary, then the number of packets stored in queues is* $\mathcal{O}(n^2)$.

Proof. Define a *pass of the token* to be a traversal of the token starting at the front of the list and ending either at a new big station or again at the front station of the list after traversing the whole list, whichever occurs first. Define a *life cycle* for a station to be a time period which starts either at the first round of the execution or at a round when the station is discovered to be big, and which ends just before the stations is discovered to be big.

We show that if burstiness is $b + 1$ then there are at most $2(n^2 + b)$ packets in queues at any round. Suppose, to the contrary, that there is a round with at least $2n^2 + 2b + 1$ packets in queues. There is a time segment T with the following properties:

(i) there are at least n^2 and at most $n^2 + b$ packets in queues at the beginning of T,
(ii) there are at least n^2 packets in queues at each round of T, and
(iii) there are at least $2n^2 + 2b + 1$ packets in queues at the end of T.

In the remaining part of the proof we restrict our attention only to the rounds in T. Consider a pass of the token. A new big station is eventually found during this pass of the token, because at least one of the stations has at least n packets in its queue, by the pigeonhole principle based on property (ii) of T.

Let C denote the set of all the stations that are discovered at least once to be big during a round in T. If a station q is not in C, then it will eventually drift through the list to be located behind all the stations in C; the token will not visit q anymore after this has happened. When a token passes through q and there are no packets at q, then this results in a silent round. We assume the worst case when each event of receiving the token by q results in a silent round. Let p be the station discovered to be big in this pass of the token. Station p is moved to front and p will never again be behind q, so p can be associated with exactly one silent round of each such a station q not in C. Since there are $|C|$ such stations p and $n-|C|$ such stations q, the total contribution of the stations that are not in C to the number of silent rounds is at most $(n-|C|)\cdot|C|$.

We claim that once a station q is discovered to be big, then q transmits a packet each time q holds a token. To show that this is the case, consider a life cycle of q. During a pass of the token, either q is discovered to be big again, which starts a new life cycle for q with at least $n-1$ packets still remaining in the queue, or one station in C located behind q is discovered as big. The latter event results in the number of stations in C behind q in the list decreasing by one. Since there are at most $|C|-1<n$ stations in C behind q in the list, station q is visited at most $|C|-1<n$ times by the token during the life cycle of q. It follows that after all the stations in C have been discovered at least once each to be big, no station in C ever has an empty queue.

It remains to estimate the number of silent rounds before a station q in C becomes big for the first time in T. Notice that until this happens, q could obtain the token with an empty queue at most $|C|-1$ times. This is because each time q has both a token and an empty queue, there is some station p from C behind q on the list such that p is discovered big in this pass of the token. The discovery results in moving p to the front of the list, so that p stays before q until q becomes big for the first time. There are $|C|$ such stations q and $|C|-1$ such stations p. Therefore the number of silent rounds contributed by all the stations in C is at most $|C|\cdot(|C|-1)$. To sum up, the total number of silent rounds in T is at most

$$(n-|C|)\cdot|C|+(|C|-1)\cdot|C|<n\cdot|C|\leq n^2 \ . \qquad (1)$$

The difference between the number of injected packets and the number of transmitted packets equals the number of void rounds plus burstiness. The difference is at most n^2+b by (1). Combine this fact with property (i) of T to obtain $(n^2+b)+(n^2+b)=2n^2+2b$ as an upper bound on the number of packets in the system at the end of T. This contradicts property (iii) defining T. □

Theorem 8. *For any protocol for n stations, the leaky-bucket adversary of burstiness* 2 *can enforce an execution such that eventually there are* $\Omega(n^2)$ *packets in queues.*

Impossibilities. A station p heard on the channel at a round t *reserves the channel for round* $t'>t$ when the packet of p carries control bits that are interpreted by all the stations so that p will be the only station transmitting at round t'. We define a transformation among queue-size oblivious protocols which, for a given queue-size oblivious protocol $\mathcal{P}$, creates a protocol $\mathcal{P}_h$ that has stations reserve the channel.

To obtain $\mathcal{P}_h$, modify $\mathcal{P}$ in terms of the transitions among states as follows. Each station maintains an array of future rounds reserved by all the stations.

These arrays are identical at all the stations. The array is indexed by the names of the stations; an entry indicates how many rounds still remain to have the round reserved for the corresponding station. During a round reserved by some station p, no station makes any state transition except for the following: (1) p transmits a packet; (2) stations have packets injected, if any. The mechanism of reservations does not affect the functionality of a queue-size oblivious protocol since both the removal of a packet from the queue at a round of the station that reserved the round or insertions of packets into queues during reserved rounds can be achieved by having the stations 'frozen' without state transitions.

Round reservations are performed as follows. When a station p transmits a packet at a round t and p has at least one more packet, then p attaches control bits to the packet to reserve some round in the future. The *first available round* at such a round t is defined to be the first smallest round after round t that is currently not reserved; the *second available round* is defined similarly as the smallest round after the first available round that is currently not reserved. When a reservation is made, then the second available round is reserved. Each station has at most one reserved round at all times.

A round t when $\mathcal{P}_h$ is executed and such that t has not been reserved is categorized as *regular*. The simulation of $\mathcal{P}$ by $\mathcal{P}_h$ proceeds in the regular rounds, in the sense that decisions inherent to $\mathcal{P}$ about broadcasting are made at regular rounds.

Lemma 3. *Let $\mathcal{P}$ be a queue-size oblivious protocol that is stable for injection rate* 1 *in a given system of n stations. Then protocol $\mathcal{P}_h$ is stable for the injection rate* 1 *in the same system of n stations.*

Theorem 9. *There is no adaptive queue-size oblivious protocol that is stable in a system of at least four stations against the window adversary of burstiness* 2.

Proof. Take a queue-size oblivious broadcast protocol $\mathcal{P}$ and consider the transformed version $\mathcal{P}_h$. Choose some four stations p, q, r, and s. The adversary will inject packets only at these four stations, so the remaining stations can be ignored. We define an execution in which the adversary injects a packet per round on the average while the set of void rounds is infinite. Suppose we have defined the execution up to a void round; we need to specify what happens starting from the next round. We consider two cases.

The first case occurs when at least two queues are nonempty at a round. If a void round is to occur when the adversary injects a packet at a certain station, then the adversary performs such an injection. Otherwise exactly one station, say p, is to transmit at the next round. The adversary chooses a station q different from p but also with a nonempty queue. The adversary keeps injecting a packet per round into q, while p keeps reserving the channel to transmit eventually all its packets. This continues until either collision occurs or the queue in station p becomes empty. In the former case we are done, while if the latter occurs, then we either repeat the same case, but with fewer nonempty queues, or proceed as in the other case.

The second case occurs when exactly one queue is nonempty at a round. Suppose that station s has a nonempty queue, while the stations p, q, and r have empty queues at some round t. Consider a conceptual experiment in which the adversary keeps injecting one packet into station p and another into station q at every other round. This means the pattern $0, 2, 0, 2, 0, \ldots$, in terms of the numbers of packets injected. If a void round occurs eventually, then we are done. Otherwise either station p or station q makes the first successful transmission at some round $t' > t$. Suppose it is station q that transmits while station p pauses. Clearly, also none among the stations s and r transmits at round t'. Consider another conceptual experiment in which station r replaces station q, that is, we consider the pair of p and r rather than p and q. Again, if a void round occurs by round t', then we are done. Otherwise, either station r pauses or transmits at round t'. In the former case, the channel is silent at round t' since both the stations p and s do not transmit as noted above; so does q, since its queue is empty. In the latter case, station r transmits successfully at round t'.

The actual action by the adversary is performed as follows. If in any of these conceptual scenarios considered above there is a void round, then the adversary mimicks the scenario until a void round occurs. Otherwise the adversary injects a packet at every other round into both q and r. These stations transmit together at round t', which results in collision.

We have considered all possible cases and showed the corresponding scenarios to cause a void round. Such scenarios can be applied continuously resulting in queues growing unbounded. Since P_h is unstable, such is also $\mathcal{P}$ by Lemma 3. $\square$

A natural paradigm to organize a broadcast protocol is for stations to be greedy by withholding the channel: once a station p transmits successfully at a round, then p keeps transmitting as long as there are pending packets in the queue of p. We say that a broadcast protocol *withholds the channel* when stations behave this way in the course of an execution of the protocol.

Theorem 10. *No protocol that withholds the channel can be stable for a system of at least four stations against the window adversary of burstiness* 2*.*

Proof. Take a broadcast protocol that withholds the channel and consider some four stations p, q, r, and s to be the only ones into which the adversary will inject packets. We define an execution in which the adversary will inject a packet per round on the average while the set of void rounds is infinite. Suppose we have defined the execution up to a void round; we specify what happens next. We consider two cases, depending on whether at least two queues are nonempty at a round or not. Consider the former case first. If a void round is to occur at the next round when the adversary injects a packet at a certain station, then the adversary performs such an injection. Otherwise exactly one station, say p, is to transmit at the next round. The adversary chooses a station q different from p but also with a nonempty queue. The adversary keeps injecting a packet per round into q, while p withholds the channel and keeps transmitting all its packets. This continues until either collision occurs or the queue in station p becomes empty. In the former case we are done, while if the latter occurs, then we either repeat the same case, but with fewer nonempty queues, or proceed as in the latter case. When the latter case occurs, in which exactly one queue is nonempty, then we proceed exactly as in the proof of Theorem 9 for the analogous case. □

References

1. Abramson, N.: Development of the Alohanet. IEEE Transactions on Information Theory 31, 119–123 (1985)
2. Aiello, W., Kushilevitz, E., Ostrovsky, R., Rosén, A.: Adaptive packet routing for bursty adersarial traffic. Journal of Computer and System Sciences 60, 482–509 (2000)
3. Alvarez, C., Blesa, M., Serna, M.: A characterization of universal stability in the adversarial queuing model. SIAM Journal on Computing 34, 41–66 (2004)
4. Andrews, M., Awerbuch, B., Fernández, A., Leighton, T., Liu, Z., Kleinberg, J.: Universal-stability results and performance bounds for greedy contention-resolution protocols. Journal of the ACM 48, 39–69 (2001)
5. Bender, M.A., Farach-Colton, M., He, S., Kuszmaul, B.C., Leiserson, C.E.: Adversarial contention resolution for simple channels. In: SPAA. Proceedings, 17th ACM Symposium on Parallel Algorithms, pp. 325–332 (2005)
6. Bhattacharjee, R., Goel, A., Lotker, Z.: Instability of FIFO at arbitrary low rates in the adversarial queuing model. SIAM Journal on Computing 34, 318–332 (2004)
7. Borodin, A., Kleinberg, J.M., Raghavan, P., Sudan, M., Williamson, D.P.: Adversarial queuing theory. Journal of the ACM 48, 13–38 (2001)
8. Chlebus, B.S., Kowalski, D.R., Rokicki, M.A.: Adversarial queuing on the multiple-access channel. In: PODC. Proceedings, 25th ACM Symposium on Principles of Distributed Computing, pp. 92–101 (2006)
9. Gallager, R.G.: A perspective on multiaccess channels. IEEE Transactions on Information Theory 31, 124–142 (1985)
10. Gamarnik, D.: Stability of adaptive and nonadaptive packet routing policies in adversarial queueing networks. SIAM Journal on Computing 32, 371–385 (2003)
11. Goldberg, L.A., Jerrum, M., Kannan, S., Paterson, M.: A bound on the capacity of backoff and acknowledgement-based protocols. SIAM Journal on Computing 33, 313–331 (2004)
12. Goldberg, L.A., MacKenzie, P., Paterson, M., Srinivasan, A.: Contention resolution with constant expected delay. Journal of the ACM 47, 1048–1096 (2000)
13. Hastad, J., Leighton, T., Rogoff, B.: Analysis of backoff protocols for multiple access channels. SIAM Journal on Computing 25, 740–774 (1996)
14. Koukopoulos, D., Mavronicolas, M., Nikoletseas, S.E., Spirakis, P.G.: The impact of network structure on the stability of greedy protocols. Theory of Computing Systems 38, 425–460 (2005)
15. Lotker, Z., Patt-Shamir, B., Rosén, A.: New stability results for adversarial queuing. SIAM Journal on Computing 33, 286–303 (2004)
16. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
17. Metcalfe, R.M., Boggs, D.R.: Ethernet: distributed packet switching for local computer networks. Communications of the ACM 19, 395–404 (1976)
18. Raghavan, P., Upfal, E.: Stochastic contention resolution with short delays. SIAM Journal on Computing 28, 709–719 (1998)
19. Rosén, A.: A note on models for non-probabilistic analysis of packet switching networks. Information Processing Letters 84, 237–240 (2002)

Stabilization of Flood Sequencing Protocols in Sensor Networks

Young-ri Choi and Mohamed G. Gouda

Department of Computer Sciences, The University of Texas at Austin,
1 University Station C0500, Austin, Texas 78712-0233, U.S.A.
{yrchoi, gouda}@cs.utexas.edu

Abstract. Flood is a communication primitive that can be used by the base station of a sensor network to send a copy of a message to every sensor in the network. When a sensor receives a flood message, the sensor needs to check whether it has received the message for the first time and so the message is fresh, or it has received the same message earlier and so the message is redundant. In this paper, we discuss a family of four flood sequencing protocols that use sequence numbers to distinguish between fresh and redundant flood messages. They are a sequencing free protocol, a linear sequencing protocol, a circular sequencing protocol, and a differentiated sequencing protocol. We analyze the self-stabilization properties of these four flood sequencing protocols. We also compare the performance of these flood sequencing protocols, using simulation, over various settings of sensor networks. We conclude that the differentiated sequencing protocol has better stabilization property and provides better performance than those of the other three protocols.

Keywords: Self-stabilization, Flood sequencing protocol, Sequence numbers, Sensor networks.

1 Introduction

Flood is a communication primitive that can be used by the base station of a sensor network to send a copy of a message to every sensor in the network. The execution of a flood starts by the base station sending a message to all its neighbors. When a sensor receives a message, the sensor needs to check whether it has received this message for the first time or not. Only if the sensor has received the message for the first time, the sensor keeps a copy of the message and may forward it to all its neighbors. Otherwise, the sensor discards the message.

To distinguish between "fresh" flood messages that a sensor should keep and "redundant" flood messages that a sensor should discard, the base station selects a sequence number and attaches it to a flood message before the base station broadcasts the message. When a sensor receives a flood message, the sensor determines based on the sequence number in the received message whether the message is fresh or redundant. The sensor accepts the message if it is fresh and discards the message if it is redundant. We call a protocol that uses sequence numbers to distinguish between fresh and redundant flood messages a *flood sequencing protocol*.

In a flood sequencing protocol, when a fault corrupts the sequence numbers stored in some sensors in a sensor network, the network can enter an illegitimate state where the sensors discard fresh flood messages and accept redundant flood messages. Therefore, a flood sequencing protocol should be designed such that if the protocol ever reaches an illegitimate state due to some fault, the protocol is guaranteed to converge back to its legitimate states where every sensor accepts every fresh flood message and discards every redundant flood message.

In this paper, we discuss a family of four flood sequencing protocols. They are a *sequencing free* protocol, a *linear sequencing* protocol, a *circular sequencing* protocol, and a *differentiated sequencing* protocol. We analyze the stabilization properties of these four protocols. For each of the protocols, we first compute an upper bound on the convergence time of the protocol from an illegitimate state to legitimate states. Second, we compute an upper bound on the number of fresh flood messages that can be discarded by each sensor during the convergence. Third, we compute an upper bound on the number of redundant flood messages that can be accepted by each sensor during the convergence.

The rest of the paper is organized as follows. In Section 2, we discuss related work and motivation of the flood sequencing protocols. In Section 3, we present a model of sensor networks. In Section 4, we give an overview of a flood protocol. We present the four flood sequencing protocols in Sections 5, 6, 7, and 8. We analyze their stabilization properties and compare them with each other in Section 9. In Section 10, we show the simulation results of these protocols. We finally make concluding remarks in Section 11.

2 Related Work and Motivation

The practice of using sequence numbers to distinguish between fresh and redundant flood messages has been adopted by most flood protocols in the literature. In other words, most flood protocols "employ" some flood sequencing protocols to distinguish between fresh and redundant flood messages. A flood sequencing protocol can be designed in various ways, depending on several design decisions such as how the next sequence number is selected by the base station, how each sensor determines based on the sequence number in a received message whether the received message is fresh or redundant, and what information the base station and each sensor stores in its local memory. Unfortunately, flood sequencing protocols have been used without full investigation of their design decisions in the literature.

The flood protocols discussed in [1,2,3,4] assume that when a sensor receives a flood message, the sensor can figure out whether the sensor has received this message for the first time or not, without specifying any mechanism to achieve this. In [5,6], it was suggested to associate a sequence number with each flood message, but any details on how sequence numbers are used by sensors (i.e. the design decisions of their flood sequencing protocols) were not specified. The flood protocols discussed in [7,8] propose to attach a unique identifier to each flood message and make each sensor maintain a list of identifiers that the sensor has received recently. Similarly, it was suggested in [9] that each sensor maintains a list of flood messages received by the sensor recently. However, any details such as how many identifiers or messages each sensor maintains and when a sensor deletes an identifier or a message from the list were not discussed.

A flood sequencing protocol is important, since the fault tolerance property of a sensor network is affected by a flood sequencing protocol used in the network. When a fault corrupts the sequence number stored in some sensor in the network, the sensor may discard fresh flood messages and accept redundant flood messages. The number of fresh flood messages discarded by the sensor and the number of redundant flood messages accepted by the sensor, before the network reaches a legitimate state, are different depending on which flood sequencing protocol is used in the network. Therefore, we need to study various flood sequencing protocols and analyze the stabilization properties of these protocols. The stabilization properties of the flood sequencing protocols are useful for sensor network designers or developers to select a proper flood sequencing protocol that satisfies the needs of a target sensor network.

In practice, a flood sequencing protocol is used with a flood protocol that may use other techniques to improve the performance of flood such as reliability or efficiency. In this paper, each of the flood sequencing protocols is described focusing on how sequence numbers are used by sensors, and it is not described as a specific flood protocol. Note that the stabilization property of a flood protocol is affected by that of a flood sequencing protocol used in the flood protocol. If the flood protocol does not maintain any extra state such that it is based on probability [2,5], the stabilization property of the flood protocol is the same as that of the used flood sequencing protocol. If the flood protocol maintains extra state such that it is based on neighbor information [1,5], the stabilization property of the flood protocol also depends on how the extra state in each sensor is stabilized.

3 Model of Sensor Networks

In this section, we describe a formal model of the execution of a sensor network, which was introduced first in [10]. This model accommodates several characteristics of sensor networks such as unavoidable local broadcast, probabilistic message transmission, asymmetric communication, message collision, and timeout actions and randomization steps. We use the model to specify our flood sequencing protocols, verify the stabilization properties of these protocols, and develop our simulation of these protocols.

The *topology* of a sensor network is a directed graph where each node represents a distinct sensor in the network and where each directed edge is labeled with some probability. A directed edge (u,v), from a sensor u to a sensor v, that is labeled with probability p (where $p > 0$) indicates that if sensor u sends a message, then this message arrives at sensor v with probability p (provided that neither sensor v nor any "neighboring sensor" of v sends another message at the same time). If the topology of a sensor network has a directed edge from a sensor u to a sensor v, then u is called an *in-neighbor* of v and v is called an *out-neighbor* of u.

We assume that during the execution of a sensor network, the real-time passes through discrete instants: instant 1, instant 2, instant 3, and so on. The time periods between consecutive instants are equal. The different activities that constitute the execution of a sensor network occur only at the time instants, and not in the time periods between the instants. We refer to the time period between two consecutive instants t and $t+1$ as a *time unit* $(t,\, t+1)$. (The value of a time unit is not critical to our current presentation of a sensor network model, but we estimate that the value of the time unit is around 100 milliseconds.)

A sensor is specified as a program that has global constants, local variables, one timeout action, and one receiving action.

At a time instant t, if the timeout of a sensor u expires, then u executes its timeout action at t. Executing the timeout action of sensor u at t causes u to update its local variables, and to send at most one message at t. It also causes u to execute the statement "timeout-after <expression>" which causes the timeout of u to expire (again) after k time units, where k is the value of <expression> at the time unit $(t,\, t+1)$. The timeout action of sensor u is of the following form:

```
timeout-expires ->
    <update local variables of u>;
    <send at most one message>;
    <execute timeout-after <expression>>
```

To keep track of its timeout, each sensor u has an implicit variable named "timer.u". In each time unit between two consecutive instants, timer.u has a fixed positive integer value. If the value of timer.u is k, where $k > 1$, in a time unit $(t-1,\, t)$, then the value of timer.u is $k-1$ in the time unit $(t,\, t+1)$. On the other hand, if the value of timer.u is 1 in a time unit $(t-1,\, t)$, then sensor u executes its timeout action at instant t. Moreover, since sensor u executes the statement "timeout-after <expression>" as part of executing its timeout action, the value of timer.u in the time unit $(t,\, t+1)$ is the value of <expression> in the same time unit.

If a sensor u executes its timeout action and sends a message at an instant t, then an out-neighbor v of u receives a copy of the message at t, provided that the following three conditions hold.

i. A random integer number is uniformly selected in the range 0 .. 99, and this selected number is less than $100 * p$, where p is the probability label of edge (u,v) in the network topology.
ii. Sensor v does not send any message at instant t.
iii. For each in-neighbor w of v, other than u, if w sends a message at t, then a random integer number is uniformly selected in the range 0 .. 99, and this selected number is at least $100 * p'$, where p' is the probability label of edge (w,v) in the network topology.

If v sends a message at t, or if w sends a message at t and for v, selects a random number that is less than $100{*}p'$, then this message *collides* with the message sent by u with the net result that v receives no message at t.

If a sensor u receives a message at instant t, then u executes its receiving action at t. Executing the receiving action of sensor u causes u to update its own local variables. It may also cause u to execute the statement "timeout-after <expression>". The receiving action of sensor u is of the following form:

```
rcv <msg> ->
    <update local variables of u>;
    <may execute timeout-after <expression>>
```

A *state* of a sensor network protocol is defined by a value for each variable and timer.u for each sensor u in the protocol. We use the notation <var>.u to denote the value of variable <var> at some sensor u.

During the execution of a sensor network protocol, several faults can occur, resulting in corrupting the state of the protocol arbitrarily. Examples of these faults are wrong initialization, memory corruption, message corruption, and sensor failure and recovery. We assume that these faults do not continuously occur in the network.

4 Overview of a Flood Protocol

In this section, we give an overview of a flood protocol that is used with our flood sequencing protocols. Consider a network that has n sensors. In this network, sensor 0 is the base station and can initiate floods over the network. To initiate the flood of a message, sensor 0 sends a message of the form data($hmax$), where $hmax$ is the maximum number of hops to be made by this data message in the network.

If sensor 0 initiates one flood and shortly after initiates another flood, some forwarded messages from these two floods can collide with one another causing many sensors in the network not to receive the message of either flood, or (even worse) not to receive the messages of both floods.

To prevent message collision across consecutive flood messages, once sensor 0 broadcasts a message, it needs to wait enough time until this message is no longer forwarded in the network, before broadcasting the next message. The time period that sensor 0 needs to wait after broadcasting a message and before broadcasting the next message is called the *flood period*. The flood period consists of f time units. A lower bound on the value of f is computed as $(hmax - 1) * tmax + 1$. (Due to space limit, this bound is presented without proof. We refer the reader to [11] for proof.) Thus, after sensor 0 broadcasts a message, it sets its timeout to expire after f time units in order to broadcast the next message.

When a sensor receives a data(h) message, the sensor decides whether the sensor accepts the message and forwards it as a data($h - 1$) message, provided $h > 1$. To reduce the probability of message collision, any sensor u, that decides to forward a message, chooses a random period whose length is chosen uniformly from the range 1..$tmax$, and sets its timeout to expire after the chosen random period, so that u can forward the received message at the end of the random period. This random time period is called the *forwarding period.*

To analyze each of the four flood sequencing protocols, we use the following value for the flood period f:

$$f = hmax * tmax + 1.$$

(We choose this value for f, instead of the minimum value $(hmax-1)*tmax+1$, to keep our proofs of the stabilization properties simple.)

Note that the above flood period is computed to guarantee that no two consecutive flood messages ever collide with each other. In a typical execution of the protocol, each sensor chooses its forwarding period at random in the range $1..tmax$, and so most sensors likely receive the flood messages within $(hmax-1)*tmax/2$ time units, instead of $(hmax-1)*tmax$ time units. Therefore, the half (or even less) of the flood period may be used without significantly degrading the stabilization property and performance of a flood sequencing protocol.

5 First Protocol: Sequencing Free

In this section, we discuss a first flood sequencing protocol where no sequence number is attached to each flood message, and so a sensor cannot distinguish between fresh and redundant flood messages, resulting that the sensor accepts every received message. This protocol is called the *sequencing free* protocol.

To initiate the flood of a new message, sensor 0 sends a data($hmax$) message, and then sets its timeout to expire after f time units to broadcast the next message.

Each sensor u that is not sensor 0 maintains a variable called new. The value of new is true only when u is in the forwarding period (i.e. u has a flood message that has been received earlier but has not been forwarded yet). When sensor u receives a data(h) message, u always accepts the message. Sensor u forwards the message as data($h-1$), if $h > 1$ in the received message and $new = false$ in u. (A formal specification of the sequencing free protocol can be found in [11].)

Note that in all the flood sequencing protocols presented in this paper, the value of timer.0 is at most f time units, and the value of timer.u is at most $tmax$. This is maintained by the executions of all the protocols.

A state S of the sequencing free protocol is *legitimate* iff either S is a state where the predicate

(timer.0= 1) $\wedge$ (for all u, $u \neq 0$, new.u=$false$)

holds or S is a state that is reachable from a state, where this predicate holds, by some execution of the protocol.

It follows from this definition that if the protocol is executed starting from a legitimate state, then every time sensor 0 initiates a new flood, previous flood messages (whether initiated by sensor 0 legitimately or other sensors illegitimately due to some fault) are no longer forwarded in the network.

6 Second Protocol: Linear Sequencing

In this section, we discuss a second flood sequencing protocol where each flood message carries a unique sequence number that is linearly increased, and so a sensor accepts a flood message that has a sequence number larger than the last sequence number accepted by the sensor. This protocol is called the *linear sequencing* protocol.

```
sensor 0                                   {base station}
const  hmax    : integer,                  {max hop count}
       f       : integer                   {flood period}
var    slast   : integer                   {last seq number}
begin
    timeout-expires →  {generate new msg}
                       $slast := slast + 1$;
                       send data(hmax, $slast$);
                       timeout-after f
end
```

Fig. 1. A specification of sensor 0 in the linear sequencing protocol

Each flood message in this protocol is of the form data(h,s), where field h is the remaining number of hops to be made by this message, and field s is the unique sequence number of this message.

Whenever sensor 0 broadcasts a new message, sensor 0 increases the sequence number of the last message by one, and attaches the increased sequence number to the message. A formal specification of sensor 0 is given in Fig. 1.

Each sensor u that is not sensor 0 keeps track of the last sequence number accepted by u in a variable called *slast*. When sensor u receives a data(h, s) message, sensor u accepts the message if $s > slast$, and forwards it if $h > 1$. A formal specification of sensor u is given in Fig. 2. (Each sensor u also maintains a received data message that u will forward later, even though this is not explicitly specified in the specification.)

A state S of the linear sequencing protocol is *legitimate* iff either S is a state where the predicate

$$(\text{timer}.0{=}\ 1) \wedge (\text{for all } u,\ u \neq 0,\ \text{new}.u{=}false \wedge \text{slast}.u \leq \text{slast}.0)$$

holds or S is a state that is reachable from a state, where this predicate holds, by some execution of the protocol.

It follows from this definition that if the protocol is executed starting from a legitimate state, then every time sensor 0 initiates a new flood, previous flood messages are no longer forwarded in the network, and the new flood message has a sequence number that is larger than every slast.u in the network, so that every u accepts the message.

```
sensor $u$:1 .. $n-1$
const hmax       : integer,     {max hop count}
      tmax       : integer      {max forwarding period}
var   $h$,$hlast$   : 1 .. hmax,   {rcvd,last hop count}
      $s$, $slast$  : integer,     {rcvd,last seq number}
      $new$      : boolean      {true if u has msg to forward}
begin
   timeout-expires → if $new$ →      $new$ := false;
                                   send data($hlast$, $slast$)
                     [] ¬ $new$ →    skip
                     fi; timeout-after random(1,tmax)

   [] rcv data($h$, $s$) →  if $s > slast$ →{accept msg} $slast := s$;
                        if h>1 →   $new$ := true;
                                   $hlast := h - 1$
                        [] $h \leq 1$ →  skip
                        fi
                     [] $s \leq slast$ →   {discard msg} skip
                     fi
end
```

Fig. 2. A specification of sensor u in the linear sequencing protocol

7 Third Protocol: Circular Sequencing

In this section, we discuss a third flood sequencing protocol where each flood message carries a sequence number that is circularly increased within a limited range, and so a sensor accepts a flood message that has a sequence number "logically" larger than the last sequence number accepted by the sensor. This protocol is called the *circular sequencing* protocol.

Each flood message is augmented with a sequence number that has a value in the range 0 .. $smax$, where $smax > 1$. We assume that $smax$ is an even number (to keep our presentation simple).

Whenever sensor 0 broadcasts a new message, sensor 0 increases the sequence number of the last message by one circularly within the range 0 .. $smax$, i.e. $slast$:= $(slast + 1)$ mod $(smax+1)$, and attaches the increased sequence number to the message.

From the viewpoint of each sequence number s in the range 0 .. $smax$, the range can be divided into two subranges, where one subrange consists of the sequence numbers that are logically "smaller" than s, and the other subrange consists of the sequence numbers that are logically "larger" than s. Thus, sequence number s has $\frac{smax}{2}$ numbers logically smaller than it and $\frac{smax}{2}$ numbers logically larger than it. For example, if $smax = 8$, number 0 is logically smaller than 1, 2, 3, and 4, and is logically larger than 5, 6, 7, and 8.

When a sensor u receives a data(h, s) message, sensor u checks if s is logically larger than $slast$. Sensor u calls the function "Larger$(s, slast)$" that returns true if s is logically larger than $slast$, and otherwise returns false. Sensor u accepts the message if Larger$(s, slast)$ returns true, and forwards it if $h > 1$. Otherwise, sensor u discards the message.

To prove the stabilization property of the circular sequencing protocol, we make an assumption of bounded message loss as follows:

> *Bounded message loss*: Starting from any state, if sensor 0 broadcasts $\frac{smax}{2}$ consecutive flood messages, then every sensor in the network receives at least one of those flood messages.

Two explanations concerning the above assumption are in order. First, the protocol may not be self-stabilizing without any bound on message loss. For example, consider a scenario where $smax$=8. Assume that sensor 0 sends a flood message with sequence number 0 and a sensor u accepts the message. If sensor u does not receive the next 4 (i.e. $\frac{smax}{2}$) consecutive messages with sequence numbers 1, 2, 3 and 4, and later receives a fresh message with sequence number 5, it discards the message since sequence number 5 is not logically larger than sequence number 0. Sensor u also discards the next flood messages with sequence numbers 6, 7, 8, and 0, if it receives them. In this scenario, if sensor u does not receive flood messages with sequence numbers 1, 2, 3 and 4, it keeps discarding fresh flood messages. Thus, some assumption of bounded message loss is necessary for the stabilization property of the protocol.

Second, the above assumption becomes acceptable if the value of $smax$ is reasonably large enough for a given network setting. Selecting an appropriate value for $smax$ depends on the size of the network, the topology of the network, and a flood sequencing protocol used in the network. (In Section 10, we show how different values are selected for $smax$ depending on these factors.)

A state S of the circular sequencing protocol is *legitimate* iff either S is a state where the predicate

```
(timer.0=1) $\wedge$
(for all $u$, $u \neq 0$,
        (new.$u$=false) $\wedge$
        (slast.$u$ = slast.0 $\vee$
         slast.$u$ = (slast.0$-$1) mod ($smax$+1) $\vee$
         ...
         slast.$u$ = (slast.0$-\frac{smax}{2}$+1) mod ($smax$+1)
        )
) $\wedge$
(sensor 0 has already initiated at least $\frac{smax}{2} + 2$ floods)
```

holds or S is a state that is reachable from a state, where this predicate holds, by some execution of the protocol.

It follows from this definition that if the protocol is executed starting from a legitimate state, then every time sensor 0 initiates a new flood, previous flood messages are no longer forwarded in the network, and the new flood message has a sequence number that is logically larger than every slast.u in the network, so that every u accepts the message.

8 Fourth Protocol: Differentiated Sequencing

In this section, we discuss the last flood sequencing protocol where the sequence numbers of flood messages are in a limited range, similar to the circular sequencing protocol. However, in this protocol, a sensor accepts a flood message if the sequence number of the message is different from the last sequence number accepted by the sensor. This protocol is called the *differentiated sequencing* protocol.

Each flood message is augmented with a sequence number that has a value in the range 0 .. $smax$, where $smax > 0$. We assume that $smax$ is an even number (to keep our presentation simple).

Sensor 0 in this protocol is identical to the one in the circular sequencing protocol. However, when a sensor u receives a data(h, s) message, sensor u accepts the message if s is different from $slast$, i.e. $s \neq slast$, and forwards the message if $h > 1$. Otherwise, sensor u discards the message.

Similar to the circular sequencing protocol, if a sensor does not receive a large number of consecutive flood messages, the differentiated sequencing protocol may not be self-stabilizing. Thus, the proofs of the stabilization property of this protocol are based on the assumption of bounded message loss described in Section 7.

A state S of the differentiated sequencing protocol is *legitimate* iff either S is a state where the predicate

```
(timer.0=1) $\wedge$
(for all $u$, $u \neq 0$,
        (new.$u$=false) $\wedge$
        (slast.$u$ = slast.0 $\vee$
         slast.$u$ = (slast.0$-$1) mod ($smax$+1) $\vee$
         ...
         slast.$u$ = (slast.0$-\frac{smax}{2}$+1) mod ($smax$+1)
        )
)
```

holds or S is a state that is reachable from a state, where this predicate holds, by some execution of the protocol.

It follows from this definition that if the protocol is executed starting from a legitimate state, then every time sensor 0 initiates a new flood, previous flood messages are no longer forwarded in the network, and the new flood message has a sequence number that is different from every slast.u in the network, so that every u accepts the message.

9 Stabilization of the Protocols

In this section, we analyze the stabilization properties of the four flood sequencing protocols. For each of the protocols, we first compute an upper bound on the

Table 1. Stabilization properties of the flood sequencing protocols

	Convergence time (time units)	Max # of fresh msgs discarded by u until convergence	Max # of redundant msg accepted by u until convergence	Stabilization property
free	$2 * f$	0	$2 * f$	good
lin	unbounded	unbounded	$n - 1$	bad
cir	$(smax + 2) * f$	$(smax + 2) * f$	$f + 1$	good
dif	$(\frac{smax}{2} + 2) * f$	$(\frac{smax}{2} + 2) * f$	$f + 1$	good

convergence time of the protocol from an illegitimate state to legitimate states. Second, we compute an upper bound on the number of fresh flood messages that can be discarded by each sensor during the convergence. Third, we compute an upper bound on the number of redundant flood messages that can be accepted by each sensor during the convergence.

The stabilization properties of the four protocols are shown in Table 1. (Due to space limit, we present the stabilization properties without proof. We refer the reader to [11] for proof.) We also analyze the properties of the protocols after convergence (or starting from a legitimate state) in Table 2. We call these properties the *stable properties* of the protocols. In these tables, “free”, “lin”, “cir”, and “dif” represent the sequencing free, linear sequencing, circular sequencing, and differentiated sequencing protocols, respectively. Note that the properties of the circular sequencing and differentiated sequencing protocols are analyzed under the assumption of bounded message loss.

Starting from an illegitimate state, the sequencing free protocol converges to legitimate states faster than the other three protocols do. However, even starting from any legitimate state, a sensor cannot distinguish between fresh and redundant messages, and so the sensor accepts every received message. The number of redundant copies of the same message accepted by a sensor depends on the value of $hmax$ and the network topology. In worst case, the sensor can accept a redundant copy of the same message at each time instant during the flood period of the message. Thus, starting from any legitimate state, every sensor accepts at most f redundant copies of the same message.

In the linear sequencing protocol, sensors are required to use unbounded sequence numbers. Thus, this protocol is very expensive to implement for sensor networks that have limited resources. However, once the protocol starts its execution from any legitimate state, every sensor accepts every fresh message and discards every redundant message under any degree of message loss. On the other hand, in the circular sequencing and differentiated sequencing protocols, sensors use bounded sequence numbers. Thus, starting from any legitimate state, every sensor accepts every fresh message and discards every redundant message under the assumption of bounded message loss.

From the above results, we conclude that overall the differentiated sequencing protocol has better stabilization and stable properties than those of the other three protocols.

Table 2. Stable properties of the flood sequencing protocols

	Max # of fresh msgs discarded by u after convergence	Max # of redundant copies of the same msg accepted by u after convergence	Stable property
free	0	f	bad
lin	0	0	good
cir	0	0	good
dif	0	0	good

10 Simulation Results

We have developed a simulator that can simulate the execution of the four flood sequencing protocols, based on our model described in Section 3. In this simulator, a network is an $N * N$ grid where N is the number of sensors in each side of the grid, and the distance between a sensor (i, j) and each of $(i + 1, j)$, $(i, j + 1)$, $(i - 1, j)$, and $(i, j - 1)$, if it exists, where $0 \leq i, j < N$, is 1.

For the purpose of simulation, sensor 0 is (0,0) which is located at the left-bottom conner in a grid, and the following two types of topologies that have different network density were used.

– A topology for a sparse network: The edge probability between two sensors is labeled with a high probability 0.95 if their distance is at most 1, and with a low probability 0.5 if their distance is larger than 1 and less than 2. Otherwise, there is no edge between the two sensors. In this topology, each sensor (i,j) that is not on or near the boundary of the grid generally has 8 neighbors.
– A topology for a dense network: The edge probability between two sensors is labeled with probability 0.95 if their distance is at most 1.5, and with probability 0.5 if their distance is larger than 1.5 and less than 3. Otherwise, there is no edge between the two sensors. In this topology, each sensor (i,j) that is not on or near the boundary of the grid generally has 24 neighbors.

(The used probabilities, 0.95 and 0.5, were chosen based on some experiments on sensors. We refer the reader to [10] for details.)

The performance of a flood sequencing protocol can be measured by the following two metrics:

i. *Reach:* The percentage of sensors that receive a message sent by sensor 0.
ii. *Communication:* The total number of messages forwarded by all sensors in the network.

We ran simulations of the four flood sequencing protocols, and measured the above two metrics in 10*10 and 20*20 grids for both sparse and dense network topologies. In our simulations, we do not consider other techniques that can improve the performance of a flood protocol based on extra information such as probability, location, and neighbor information.

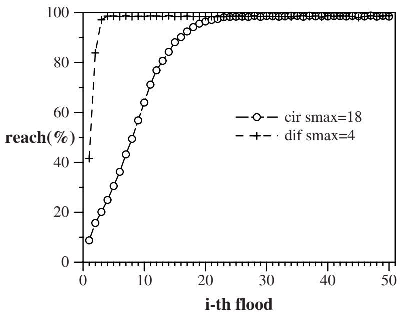
Table 3. Performance of the sequencing free and linear sequencing protocols

	sparse 10*10			sparse 20*20			dense 10*10			dense 20*20		
	hmax	Reach	Com.	hmax	Reach	Com.	hmax	Reach	Com.	hmax	Reach	Com.
free	13	99%	351.3	27	99.2%	2885.7	7	99.8%	200.5	13	99%	1262
lin	15	98.5%	97.8	28	98.5%	390.3	7	98.5%	87.5	14	98.8%	376.4

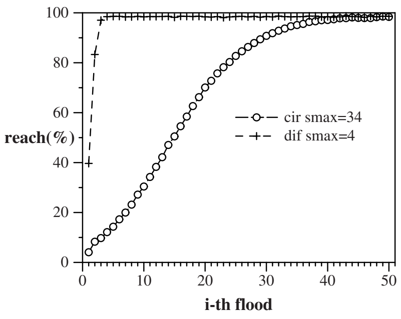
First, we studied the performance of the sequencing free protocol and the linear sequencing protocol starting from a legitimate state. The result of each simulation in this study represents the average value over the simulations of 100,000 floods.

Table 3 shows the reach and communication of the sequencing free and linear sequencing protocols in sparse and dense networks. In these simulations, $tmax = 6$ was used for a sparse network, and $tmax = 7$ was used for a dense network. From the above results, one can observe that the sequencing free protocol requires the sensors to send much more messages than those that the linear sequencing protocol does. Note that when the value of *smax* is reasonably large for a given network setting, the performance of the circular sequencing and differentiated sequencing protocols becomes same as that of the linear sequencing protocol.

Next, we studied the stabilization properties of the circular sequencing and differentiated sequencing protocols, and their performance while stabilizing. We simulated the sequences of floods starting from 1000 different illegitimate states, and computed the average reach for each i-th flood. We attempted to select an appropriate value for *smax* for each network setting such that the assumption of bounded message loss becomes acceptable, while the convergence time of each protocol is minimized.



(a) A 10*10 network



(b) A 20*20 network

Fig. 3. Reach of the circular and differentiated sequencing protocols starting from an illegitimate state in a sparse network

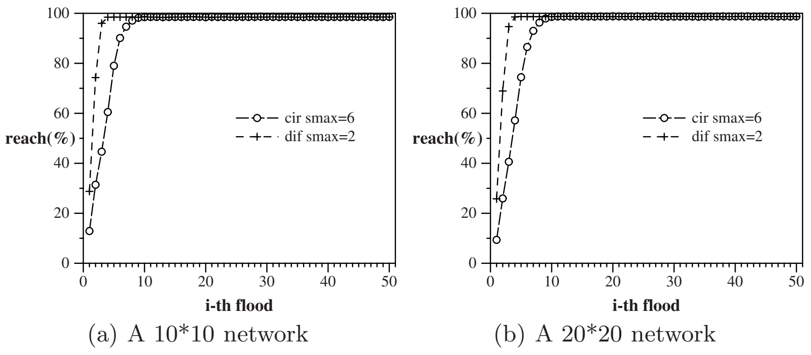


Fig. 4. Reach of the circular and differentiated sequencing protocols starting from an illegitimate state in a dense network

Figures 3 and 4 show the reach of the circular sequencing and differentiated sequencing protocols starting from an illegitimate state in a sparse network and in a dense network, respectively. During the convergence time, each sensor has a higher probability to accept a received fresh message in the differentiated sequencing protocol than that in the circular sequencing protocol. Thus, in all simulated network settings, the differentiated sequencing protocol reaches a legitimate state faster than the circular sequencing protocol does.

In summary, starting from a legitimate state, the performance of any flood sequencing protocol that attaches a sequence number to a flood message is better than that of the sequencing free protocol in terms of communication. Starting from an illegitimate state, the differentiated sequencing protocol converges to a legitimate state quickly in all simulated network settings.

11 Concluding Remarks

In this paper, we discussed a family of the four flood sequencing protocols, namely the sequencing free, linear sequencing, circular sequencing, and differentiated sequencing protocols. We analyzed the stabilization and stable properties of these four protocols, and also studied their performance, using simulation, over various settings of sensor networks. We concluded that the differentiated sequencing protocol has better overall performance in terms of communication and stabilization and stable properties compared to those of the other three protocols.

Acknowledgment

This work was supported in part by the US National Science Foundation under Grant No. 0520250.

References

1. Stojmenovic, I., Seddigh, M., Zunic, J.: Dominating Sets and Neighbor Elimination-Based Broadcasting Algorithms in Wireless Networks. IEEE Transactions on Parallel and Distributed Systems 13(1), 14–25 (2002)
2. Sasson, Y., Cavin, D., Schiper, A.: Probabilistic Broadcast for Flooding in Wireless Mobile Ad hoc Networks. In: WCNC 2003. Proceedings of IEEE Wireless Communications and Networking Conference, pp. 1124–1130. IEEE Computer Society Press, Los Alamitos (2003)
3. Li, J., Mohapatra, P.: A Novel Mechanism for Flooding Based Route Discovery in Ad Hoc Networks. In: GLOBECOM. Proceedings of the IEEE Global Telecommunications Conference, pp. 692–696 (2003)
4. Ganesan, D., Krishnamurthy, B., Woo, A., Culler, D., Estrin, D., Wicker, S.: An Empirical Study of Epidemic Algorithms in Large Scale Multihop Wireless Networks. IRP-TR-02-003 (2002)
5. Ni, S., Tseng, Y., Chen, Y., Sheu, J.: The Broadcast Storm Problem in a Mobile Ad Hoc Network. In: MOBICOM. Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking, pp. 151–162. IEEE Computer Society Press, Los Alamitos (1999)
6. Heissenbttel, M., Braun, T., Waelchli, M., Bernoulli, T.: Optimized Stateless Broadcasting in Wireless Multi-hop Networks. In: IEEE INFOCOM, pp. 1–12 (2006)
7. Williams, B., Camp, T.: Comparison of Broadcasting Techniques for Mobile Ad Hoc Networks. In: MOBIHOC. Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 194–205. ACM Press, New York (2002)
8. Johnson, D.B., Maltz, D.A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: Imielinski, T., Korth, H. (eds.) Mobile Computing, vol. 353, pp. 153–181. Kluwer Academic Publishers, Dordrecht (1996)
9. Sun, M., Feng, W., Lai, T.: Location Aided Broadcast in Wireless Ad Hoc Networks. In: Proceedings of the IEEE GLOBECOM 2001, pp. 2842–2846. IEEE Computer Society Press, Los Alamitos (2001)
10. Gouda, M., Choi, Y.: A State-based Model of Sensor Protocols. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 246–260. Springer, Heidelberg (2006)
11. Choi, Y., Gouda, M.: Stabilization of Flood Sequencing Protocols in Sensor Networks. Technical Report TR-06-58, Department of Computer Sciences, The University of Texas at Austin (2006)

Stabilization of Loop-Free Redundant Routing

Jorge A. Cobb

Department of Computer Science, The University of Texas at Dallas, U.S.A
cobb@utdallas.edu

Abstract. Consider a network of processes that exchange messages via FIFO communication channels. Each process chooses a subset of its neighboring processes to be its successors. Furthermore, there is a distinguished process, called root, that may be reached from any other process by following the successor relation at each hop. Thus, under the successor relation, the processes are arranged as a directed acyclic graph that converges on the root process, i.e., a converging DAG (c-DAG). We present a network where each process may dynamically change its choice of successors, and during this change, the following two nice properties are satisfied. First, if the initial state of the network forms a c-DAG, then a c-DAG is preserved at all times. Second, if the protocol is started from an arbitrary state (i.e., where each variable has an arbitrary value), then a c-DAG is automatically restored.

1 Introduction

A network consists of a set of processes that exchange messages via FIFO communication channels. A common task in a network is the construction of a spanning tree. To build a spanning tree, each process chooses ones of its neighbors as its parent on the tree. The parent is also known as the successor of the process.

Spanning trees have multiple uses. Two of the most common are unicast and broadcast routing of data messages. In unicast routing [1,2], a spanning tree is built with the destination as the root of the tree. When a process receives a message addressed to the destination, the message is forwarded to the parent on the tree. In broadcast routing [3,4], when a process receives a broadcast message from a neighbor on the spanning tree, it forwards the message to all other neighbors that are also on this tree.

In both unicast and broadcast routing, the spanning tree is required to adapt to network conditions, such as congestion, and modify its structure. In doing so, temporary loops may be introduced, and processes may become disconnected from the tree. This is undesirable, since it reduces routing performance. Thus, loop-free spanning trees were developed [5,6,7]. These ensure that, even while the spanning tree is modifying its structure, no temporary loops are introduced, and no process is disconnected from the tree. Maintaining loop-freedom is of particular importance in ad-hoc networks, due to the frequent changes in network connectivity and low network bandwidth [8,9,10].

An alternative approach is to maintain *multiple* successors at each node. A single process, called, root, has no successors, and all processes lead to the root. Thus, rather than maintaining a tree, a converging directed acyclic graph (c-DAG), is maintained, where all paths converge on the root process. This graph is used in unicast routing to provide multiple paths to the destination, i.e., to the root [11,12,13]. In broadcast routing, it provides alternative paths in the event of link failures.

All the works above assume a fail-safe model of fault-tolerance: if a process or channel fails, it simply stops functioning. This, however, does not cover some failures that are hard to detect. These include: transient hardware or software faults at lower layers, undetected corrupted messages, improper initialization of a node, or temporary disruptions from a network intruder. A broader fault-tolerant model that captures all of these transient faults is known as stabilization.

A network of processes is said to be *stabilizing* iff, starting from any arbitrary state (such as the state after an undetected fault), the network converges to a normal operating state within finite time. Stabilizing protocols are desirable due to their high degree of fault-tolerance [14]. They have the advantage of not requiring a global initialization, plus they tolerate all types of transient faults.

Multiple techniques to build loop-free and stabilizing spanning trees exist in the literature [15,16,17,18]. All of these techniques assume a shared memory model.

To our knowledge, only a single technique for constructing a loop-free and stabilizing c-DAG has been presented in the literature [19]. However, it suffers from the following drawbacks: a) a shared memory model is assumed, b) when a process chooses to change its successor set, this is restricted to occur only during a diffusing computation initiated by the root, and c) a temporary loop may be created in the event of a channel failure, even though the failed channel is not part of the c-DAG.

In this paper, we present a technique that solves the above problems. Processes exchange information via message passing, which is a more practical model than shared memory. A process is free to change its successor set without having to coordinate with the root process. Finally, loops are never introduced, even if channels fail.

We present our network of processes in three steps. First, we present processes that avoid loops at all times. However, the choice of successors for each process is limited. Then, we enhance our processes to have freedom in choosing their successors. Finally, we further enhance our processes to be stabilizing.

2 A Converging DAG of Processes

In this section, we present a general overview of the problem. We begin with some notation.

A *network* consists of a set of processes interconnected via communication channels. Two processes are *neighbors* if they are joined by a pair of channels. A *network path* is a sequence of processes where for each pair (u, v) of consecutive processes in the path, v is a neighbor of u.

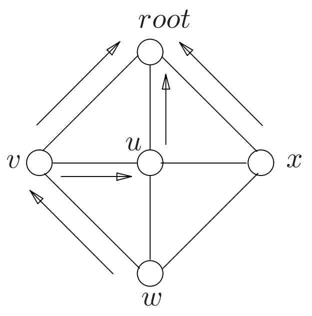


Fig. 1. Converging DAG of processes

Each process u maintains a variable, $u.S$, where it stores the identifiers of a subset of its neighboring processes. If process $v \in u.S$, then v is said to be a *successor* of u and u is said to be a *predecessor* of v.

A path is *active* when, for each pair (u, v) of consecutive processes in the path, v is a successor of u. Process v is *reachable* from a process u when there is an active path from u to v.

For example, consider Figure 1. In this figure, the neighbor relation is denoted by lines, and the successor relation is denoted by arrows. Thus, all processes are neighbors of u, v has two successors, i.e., $v.S = \{u, \mathit{root}\}$, and w has only one successor, $w.S = \{v\}$.

We require that all active paths be simple paths, i.e., loop-free. In consequence, the successor relation forms a directed acyclic graph.

We assume that there exists a distinguished process, which we call *root* (see Figure 1). In addition, we require that for every non-root process, there must exist an active path from the process to the root. This implies that the successor set of all non-root processes is non-empty. Also, the root process becomes a convergence point for the digraph, and hence, we refer to this structure as a converging DAG (c-DAG).

Contrary to earlier work [15,19,16,17,18], our processes do not choose which neighbors should be added to the successor set. We assume this is guided by a higher-layer application that chooses a particular structure. To capture the behavior of the application without imposing any restrictions, our processes simply choose non-deterministically whether or not to add a neighbor to the successor set. Our processes ensure that the requirements presented above are met at all times. Furthermore, if these requirements are not met initially, then the processes automatically converge to a state where the requirements are met.

We conclude this section with some path notation.

$$\begin{aligned}
|P| &: \text{ number of processes in path } P \\
P_j &: j^{\text{th}} \text{ process in } P,\ 1 \le j \le |P| \\
L &: \text{ maximum length of a simple path} \\
\mathit{active}(P) &: (\forall\, j \,:\, 1 < j \le |P| \,:\, P_j \in P_{j-1}.S) \\
\mathit{below}(u, v, x) &: (\exists\, P \,:\, \mathit{active}(P) \wedge |P| \le x \,:\, P_1 = u \wedge P_{|P|} = v)
\end{aligned}$$

3 Process Notation

Before presenting our processes, we first give a short overview of the notation that we use in specifying their behavior. This is similar to the notation introduced in [20]. Processes communicate with each other via the exchange of messages over FIFO channels. We use the following notation when referring to channels and messages.

$Ch(u, v)$	: channel from u to v
$m(u, v)$	: message of type m from u to v
$m(u, v).f$	: value of field f in message $m(u, v)$
$neigh(u, v)$	: function returning **true** iff both $Ch(u, v)$ and $Ch(v, u)$ exist.

Without loss of generality, for every pair of distinct processes u and v, either both $Ch(u, v)$ and $Ch(v, u)$ exist or neither of these two channels exist.

Each process is specified by a set of inputs, a set of variables, a parameter, and a set of actions. In general, a process is specified as shown below.

```
process <process name>
inp
   <input name>       : <type>,
         ...
var
   <variable name>    : <type>,
         ...
par
   <parameter name> : <type>
begin
   <action>
[]
         ...
[]
   <action>
end
```

The inputs declared in a process can be read, but not written, by the actions of that process. The variables declared in a process can be read and written by the actions of that process. The parameter is discussed below. To distinguish between variables of different processes, we prefix the variable names with the process name. For example, variable $u.r$ corresponds to variable r in process u. If a variable does not have a process prefix, the process is understood from the context.

Every action in a process is of the form: <guard> $\rightarrow$ <statement>. The <guard> can be of three types: local, receiving, and timeout.

A local guard is a boolean expression over the inputs, variables, and parameter declared in the process. A receiving guard at process u is of the form

rcv m **from** v

where v is a neighbor of u. Finally, a timeout guard is of the form

$$\textbf{timeout}\ m \notin Ch(u,v) \wedge m' \notin Ch(v,u)$$

where v is a neighbor of u.

The <statement> is a sequence of message send statements or conditional statements. Conditional statements are of the following form.

<variable> := <expression> **if** <boolean expression>

If <boolean expression> is true before the conditional statement is executed, then <variable> is assigned the current value of <expression>.

The parameter declared in a process is used to write a set of actions as one action, with one action for each possible value of the parameter. For example, if we have the following parameter definition,

```
par
  $g : 1\ ..\ 2$
```

then the following action

$$x = g \rightarrow x := x + g$$

is a shorthand notation for the following two actions.

$$\begin{array}{l} x = 1 \rightarrow x := x + 1 \\ [] \\ x = 2 \rightarrow x := x + 2 \end{array}$$

An execution step of a protocol consists of choosing an action whose guard evaluates to true and executing the statement of this action. We assume all executions of a protocol are weakly fair, that is, an action whose guard is continuously true must be eventually executed.

We often refer to each element in an array variable A. With some abuse of notation, the expression $A = x$ is equivalent to $(\forall i :: A[i] = x)$. Similarly, the assignment statement $A := x$ assigns the value x to each element of A.

4 Ranked Processes

In this section we present a network of simple processes that maintain a c-DAG. We assume such structure exists in the initial state. Thus, this network is not stabilizing.

Active paths are maintained loop-free as follows. Each process is assigned a rank value. We denote by R the set of all possible rank values, and by $u.r$ the rank of process u. Whenever process u adds a new successor to $u.S$, the new successor must have a rank greater than that of u. In consequence, for every pair of processes u and v, where $v \in u.S$, $v.r$ is greater than $u.r$.

The reason all active paths are loop-free is simply as follows. Let P be an active path with a loop, that is, $P_1 = u = P_{|P|}$. Then, because ranks increase from each process to its successor, this implies that $u.r$ is greater than $u.r$, which is not possible.

We next formalize process ranks and the relation on rank values. We are given a relation $\preceq$ on ranks. This relation satisfies the following properties.

i. Transitive:
For every r, r', and r'',
$(r \preceq r' \wedge r' \preceq r'') \Rightarrow (r \preceq r'')$

ii. Antisymmetric:
For every r and r',
$r \preceq r' \wedge r' \preceq r \Rightarrow r = r'$

iii. Bounded:
There exists a value $\top$ (top) such that, for all r, $r \preceq \top$, and a value $\bot$ (bottom) such that, for all r, $\bot \preceq r$.

We denote the reflexive reduction of $\preceq$ as $\prec$.

The above general definition of rank allows for many possible choices of R and $\preceq$. For example, R could simply be the set of natural numbers, $\preceq$ be $\leq$, and $\prec$ be $<$. In addition, ranks could be based on the model of maximizable metrics introduced in [21,22]

Ranks could be independent of the application that chooses the successor set. In this case, ranks would simply be used to prevent the application from violating the requirements on active paths. On the other hand, ranks could be intimately related to the application. E.g., assume the c-DAG is used for datagram routing in computer networks. Then, the successor set may be chosen to be those neighbors that provide the lowest metric to the root process. The metric could be as simple as the hop count to the root, or it could be a more complex metric, such as bottleneck bandwidth, queuing delay, or a combination of all of these. The rank in this case would simply be the metric used by the application.

We next present the processes in this network. The rank given to each process is fixed. However, we relax this requirement in the next section. We first show the specification of a non-root process u.

```
process $u$
inp
    $G$    : set of pid's,   {neighbor set}
    $r$    : element of $R$ {rank}
var
    $S$    : subset of $G$   {successor set}
par
    $g$    : element of $G$ {any neighbor}
begin
    true →                  $upd.r := r$;
                            send $upd$ to $g$
[]
    rcv $upd$ from $g$ →    $S := S \bigcup \{g\}$    if $r \prec upd.r \wedge$ any
[]
    any $\wedge |S| > 1$ →  $S := S - \{g\}$
end
```

Each process periodically sends an *upd* (update) message to each of its neighbors. The *upd* message contains the rank of the process.

Each non-root process contains three actions. In the first action, process u sends an update to neighbor g, and includes its rank in this update.

In the second action, process u receives an *upd* message from neighbor g. If the rank of g is greater than that of u, then u adds g to its successor set. We model the application's choice of adding g to the successor set by including the operator **any** in the statement's condition. The operator **any** nondeterministically returns **true** or **false**.

In the third action, process u removes a neighbor g from its successor set. This, however, is done only if the successor set of u does not become empty. Again, we represent the choice of removing g from the successor set of u by including the operator **any** in the guard of the action.

The specification of the root process is given below.

```
process root
inp
  G     :  set of pid's,   {neighbor set}
const
  S     :  $\emptyset$               {successor set}
  r     :  $\top$               {rank}
par
  g     :  element of G  {any neighbor}
begin
  true →               upd.r := r;
                       send upd to g
[]
  rcv upd from g →     skip
end
```

The root process consists of two actions. In the first action, the root sends an update message to a neighbor. In the second action, the root receives an update message from a neighbor. Since the root is not allowed to have successors, it simply discards the message. Note that the successor set and the rank are constant values, which are the empty set and the top rank, respectively.

5 Dynamically-Ranked Processes

Having a fixed rank at each process restricts significantly the set of neighbors from which the process can choose successors. In consequence, the overall structure of the c-DAG is also restricted. To allow a dynamic structure, the rank of each process must also be dynamic. We address dynamic ranks in this section, and show how loops are avoided. Our technique has some similarities with earlier non-stabilizing loop-free protocols [5,6,7].

In the previous section, loops were avoided by ensuring the following two conditions.

1. The rank of every process is less than the rank of each of its successors.
2. When a process adds a new successor, the rank of the new successor is greater than that of the process.

However, these conditions are stronger than necessary, and are a consequence of processes having a fixed rank. To support dynamic ranks, we replace the above conditions with the following.

Definition 1. (Loop-Avoidance Conditions)

1. *When a process adds a new successor, the rank of the new successor is greater than the rank of the process.*
2. *When a process u adds a new successor, all processes below u must have a rank at most the rank of u.*
3. *A process cannot increase its rank to a value greater than the rank of any of its successors.*
4. *If the rank of a process is greater than that of any of its successors, then the process must reduce its rank to be at most the rank of all of its successors.* □

Note that the above conditions allow a process to reduce its rank at any time and by any amount.

The first three conditions imply that a new successor cannot be below the process, and thus loops are avoided. That is, if process u adds a new successor, the rank of the successor is greater than that of u, but at the same time all processes below u have a rank at most that of u. Hence, the new successor cannot be below u. The fourth condition aids in the implementation of the second condition, as will be shown later in this section.

As an example, consider Figure 2(a). The structure is the same as that in Figure 1, and each process is labeled with its rank. The rank of each process is an integer, and $\prec$ is simply $<$.

Assume u attempts to add x to its successor set. Since the rank of u is greater than that of x, from the perspective of u, x may be below u. To determine if this is the case, u decreases its rank to be less than the rank of x, as shown in Figure 2(b). This in turn causes all processes below u to decrease their ranks, as shown in Figure 2(c). Once this operation completes, if the rank of x is still greater than that of u, then x is not below u, and u can add x to its successor set. This is shown in Figure 2(d).

We next consider each of the first three loop avoidance conditions. For each, we show how violating the condition may result in a loop.

Consider the first condition, and consider Figure 2(a). If u adds a successor with lesser rank, namely w, then a loop is formed. Consider the second condition, and consider Figure 2(b). If u adds a successor, again w, before the rank of w has been decreased to be less than that of u, then a loop is formed, even though the rank of w is greater than that of u. Finally, consider the third condition and Figure 2(c). If w were to increase its rank to a value greater than the rank of v, its rank would be greater than the rank of u. This would allow u to add w to its successors and cause a loop.

We next address how to implement the conditions above. In particular, each process must lower its rank to be at most the rank of each of its successors. In addition, each process must be able to determine that each process below it has a rank no greater than its own. We consider each of these in turn.

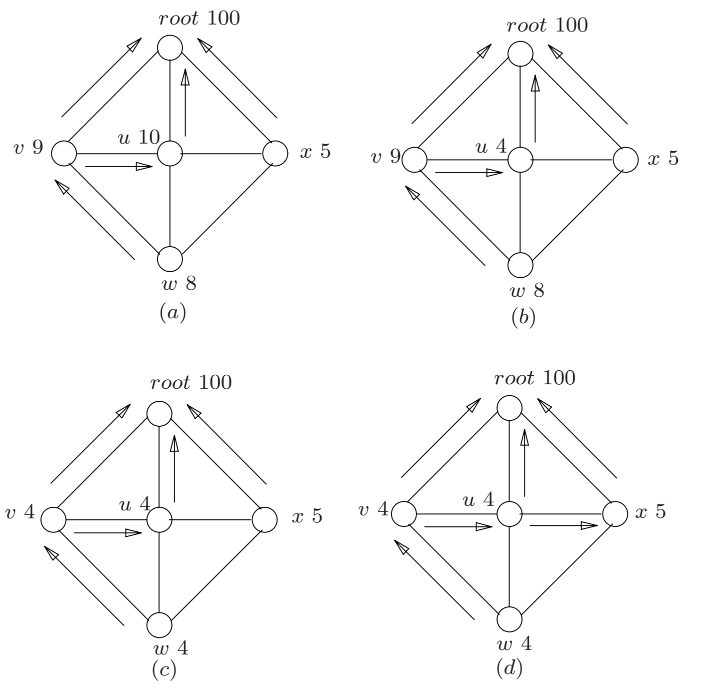


Fig. 2. Avoiding a loop while decreasing the rank

As in the previous section, each process u periodically sends an *upd* message to all its neighbors. The message contains the rank of the process. Process u maintains two additional variables, $u.\widetilde{r}$ and $u.\widetilde{S}$. Variable $u.\widetilde{S}$ is a set containing those neighbors from whom u has received an *upd* message. Variable $u.\widetilde{r}$ contains a lower bound on the ranks of the successors of u from whom u has received an *upd* message, i.e., from successors in $u.\widetilde{S}$. When u has received an *upd* message from all successors, i.e., $u.S \subseteq u.\widetilde{S}$, $\widetilde{r}$ contains a lower bound on the rank of all successors. At this time, $u.\widetilde{r}$ is assigned to $u.r$. Furthermore, to prepare for another round of *upd* messages from each neighbor, $u.\widetilde{S}$ is set to the empty set, and $u.\widetilde{r}$ is assigned the top rank.

We next address how a process can determine that the ranks of all processes below it are at most its own rank. Each process maintains an array D with the depth of rank ordering. That is, D has an entry per neighbor, and the value of the entry is in the range $0 \mathinner{..} L$. Let g be a predecessor of u. If $u.D[g] = i$, then all processes that are below both u and g up to i hops below u have a rank that is at most the rank of u.

More formally, we have the following rank ordering property.

Definition 2. (Rank Ordering Property)
Consider any active path P, where: $t = P_1, g = P_{|P|-1}, u = P_{|P|}$, and $2 \leq |P| \leq u.D[g]$. Then, the following holds:

$$t.r \preceq u.r \wedge (t.\widetilde{r} \preceq u.r \vee P_2 \notin t.\widetilde{S}) \wedge (\forall x : neigh(x,t) : upd(t,x).r \preceq u.r)$$

In addition, if $u.D[g] = 1$*, then* $upd(u,g).r \preceq u.r$*.* □

Note that when $u.D[g] = L$, the rank of all processes below u is at most the rank of u, and u is free to add a new successor.

Finally, consider how D should be updated. When a neighbor g receives an *upd* message from process u, g returns an *ack* message to u. This *ack* message contains two values. The first value, $ack(g,u).r$, is the current rank of g. The second value, $ack(g,u).d$, contains the minimum of all the elements in array D at g. This indicates to u the depth at which processes below g have a rank at most that of g. However, if u is not a successor of g, then $ack(g,u).r = \bot$ and $ack(g,u).d = L$.

When process u receives an *ack* message from neighbor g, it checks the rank of the message and its own rank. If the rank of g is at most the rank of u, then the depth along g is increased by one. That is, $u.D[g] := max(u.D[g], ack(g,u).d{+}1)$.

We have yet to address when the value of $u.D[g]$ is decreased. Note that as long as $u.r$ increases, then the value of $u.D[g]$ need not decrease, since the rank ordering property is not violated. However, if $u.r$ decreases, this property may no longer hold. Thus, whenever $u.r$ is decreased, $u.D[g]$ is assigned zero for all g.

We are now ready to present the specification of a network with dynamic rank. Below, we present the specification of a non-root process u.

```
process $u$
inp
  $G$      :  set of pid's          {neighbor set}
  $L$      :  integer               {max. path length}
var
  $S, \widetilde{S}$   :  subset of $G$          {successor set and its iteration set}
  $r, \widetilde{r}$   :  element of $R$         {rank and its iteration value}
  $D$      :  array [$G$] of 0 .. $L$   {rank depth}
par
  $g$      :  element of $G$         {any neighbor}


begin
  timeout $upd \notin Ch(u,g) \wedge ack \notin Ch(g,u)$ $\rightarrow$
                      $D[g] := max(1, D[g])$;
                      $upd.r := r$;
                      send $upd$ to $g$
[]
  rcv $upd$ from $g$ $\rightarrow$    $\widetilde{S} := \widetilde{S} \bigcup \{g\}$;
                      $S := S \bigcup \{g\}$   if $r \prec upd.r \wedge D = L \wedge$ any;
                      $\widetilde{r} := any\{x \mid x \preceq min(\widetilde{r}, upd.r)\}$   if $g \in S$;
                      $reply(g)$
[]
  rcv $ack$ from $g$ $\rightarrow$    $D[g] := max(D[g], ack.d + 1)$
                           if $ack.r \preceq r \wedge D[g] > 0$
[]
```

```
    $S \subseteq \widetilde{S}$ →              $D := 0$ **if** $\widetilde{r} \prec r$;
                               $r, \widetilde{r}, \widetilde{S} := \widetilde{r}, \top, \emptyset$
  $[\!]$
    **any** $\wedge$ $|S| > 1$ →        $S := S - \{g\}$
  **end**
```

The process consists of five actions. In the first action, a new *upd* message is sent to a neighbor g. The message is sent only if the previous *upd* message has been received (or is lost) and its corresponding *ack* has been received (or is lost). Furthermore, since $upd(u, g).r = u.r$, we can safely assign a value of at least one to $u.D[g]$.

In the second action, an *upd* message is received from a neighbor g. Neighbor g is added as a successor if the rank ordering property is not violated, and in addition, the higher layer application chooses g as a successor. We represent this by the operator **any**, which nondeterministically returns **true** or **false**. In this action, $reply(g)$ is a shorthand for the following sequence of statements.

```
$ack.r, ack.d := r, min\{D\}$ **if** $g \in S$;
$ack.r, ack.d := \bot, L$       **if** $g \notin S$;
**send** $ack$ **to** $g$
```

In the third action, an *ack* is received from a neighbor g. Variable $u.D[g]$ is increased provided the rank of g is at most the rank of u and $u.D[g] > 0$. The reason for $u.D[g] > 0$ is as follows. If $u.D[g] = 0$, then is possible that the *ack* received is in response to an *upd* message sent *before* $u.r$ was decreased. This would cause synchronization problems between u and g, and the rank ordering property may be violated.

In the fourth action, process u has finished receiving an *upd* message from all neighbors. It then updates $u.r$, $u.\widetilde{r}$, and $u.\widetilde{S}$ as discussed earlier.

In the fifth action, process u removes neighbor g from its successor set, provided the successor set does not become empty, and provided that the higher-layer application, which we model by the operator **any**, chooses to remove g.

We present below the specification of the root process.

```
**process** $root$
**inp**
  $G$      : set of pid's          {neighbor set}
  $L$      : integer               {max. path length}
**const**
  $S, \widetilde{S}$   : $\emptyset, \emptyset$              {successor set and its iteration set}
  $r, \widetilde{r}$   : $\top, \top$              {rank and its iteration value}
  $D$      : array $[G]$ of $L$        {rank depth}
**par**
  $g$      : element of $G$          {any neighbor}
**begin**
  **timeout** $upd \notin Ch(u, g) \wedge ack \notin Ch(g, u) = 0$ →
                           $upd.r := r$;
                           **send** $upd$ **to** $g$
[]
  **rcv** *upd* **from** $g$ $\rightarrow$   $reply(g)$
[]
  **rcv** *ack* **from** $g$ $\rightarrow$   **skip**
**end**
```

The root process consists of three simple actions. In the second action, $reply(g)$ is a shorthand for the following sequence of statements.

```
$ack.r, ack.d := \top, L$;
**send** *ack* **to** $g$
```

Notice that the value of $root.D$ is always L, and that the value of $root.r$ is always $\top$. This is because the root does not need to decrease its rank, since it has no successors.

6 c-DAG Restoration

The processes in the previous section ensure that the network is maintained loop-free at all times. However, they are not stabilizing. In particular, if a loop exists at the initial state of the execution, then the loop may be maintained throughout the execution. In this section, we enhance our processes with the ability of automatically breaking any existing loop, and restoring the integrity of the c-DAG. Loops are detected using an extension of the spanning-tree technique we presented in [16].

Although the dynamically-ranked processes of the previous section are not stabilizing, they have an interesting property. Starting from any arbitrary state, the rank ordering property will eventually hold and continue to hold. That is, the processes stabilize to the rank-ordering property. Therefore, even though loops that exist at the initial state may not be broken, there is a point in the execution after which no new loops may be created.

Given that the rank-ordering property is stabilizing, the main obstacle in the stabilization of our processes is the removal of existing loops. Thus, processes must be able to detect the presence of a loop. In addition, the loop must be broken, and any processes that become separated from the c-DAG must rejoin it.

To detect loops, each process maintains an estimate of the number of hops to the root process. This estimate is maintained in variable $u.h$. Each *upd* message from u now contains two values: the rank $u.r$ and hop count $u.h$. Process u assigns to $u.h$ the largest hop count of each of its successors plus one.

To collect the hop counts from each neighbor, process u maintains a variable $u.\widetilde{h}$. This variable contains the maximum hop count (plus one) of every neighbor in $u.\widetilde{S}$, i.e., of every neighbor from whom an *upd* message has been received. When an *upd* has been received from every neighbor, $u.\widetilde{h}$ is assigned to $u.h$ and $u.\widetilde{h}$ is assigned zero.

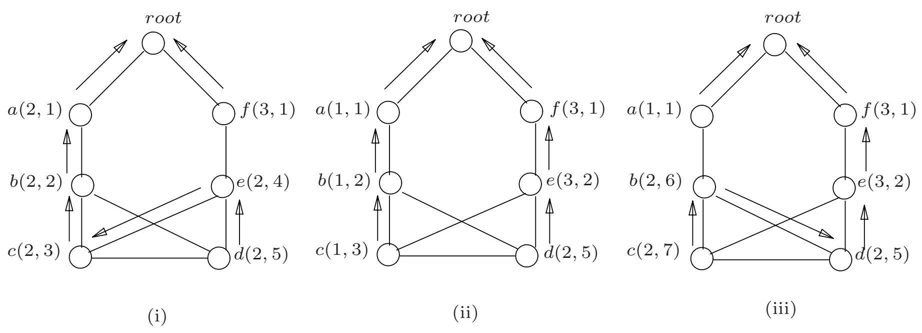


Fig. 3. Incorrect loop detection

Since the maximum length of a simple network path is L, we expect the value of $u.h$ to never increase beyond $L - 1$. Thus, a straightforward way to detect a loop is to check if $u.h \geq L$. If so, process u concludes that it is involved in a loop. However, this is not accurate due to the dynamic nature of the network, as we demonstrate below.

Consider the network in Figure 3. The rank of each process is an integer, and $\prec$ is simply $<$. In this network, $L = 7$. Alongside each process are its rank and its distance, in that order. The initial state of the network is given in Figure 3(i).

Assume process e adds f to its successor set, and then removes c from its successor set. For the moment, assume the channel from e to d is slow, so d does not update its values from those of e for some time. After e changes successors, the rank of a drops to one, and this new rank is propagated to b and c. Still, d has not updated its values from those of e. This is shown in Figure 3(ii). Next, assume process b chooses d as a successor, and then removes a from its successor set. The new rank of b is then propagated to c. Still, d has not updated its values from those of e. This is shown in Figure 3(iii).

Note that in Figure 3(iii), $c.h = 7 = L$. Thus, $c.h$ indicates the presence of a loop, even though none exists. (The scenario in Figure 3 can be extended further to show that $c.h$ grows without bound even though a loop is never present.) Therefore, a simple hop count cannot be used as a method of loop detection.

The above problem of erroneous loop detection is due to the flexibility in adding and removing successors. These operations need to be restricted, but not to the extent of making the structure inflexible. We choose to restrict them as follows.

Definition 3. (Loop Detection Conditions)

1. *A process u cannot add a neighbor v to its successor set if $v.h \geq L$.*
2. *A process u, where $u.h \geq L$, cannot add nor remove neighbors from its successor set unless all of processes v below it have $v.h \geq L$.*
3. *A process u cannot decrease $u.h$ to less than L until all processes v below it have $v.h \geq L$.* □

From the above restrictions, when process u reaches a hop count of at least L, it stops adding or removing successors. In addition, no process will choose u as its successor. Then, a hop count of at least L propagates to all descendants of u. In this way, the structure below u will cease to change. Thus, since the maximum length of a simple path is L, no process below u should obtain a hop count of $2L$ unless a loop exists.

When $u.h \geq 2L$, process u assumes that either itself or a process above it is part of a loop. Process u will empty its successor set (thus breaking the loop) and then choose as a successor the first neighbor which indicates that its hop count is less than $2L$. As in the previous section, process u ensures that the new successor is not below u, and thus, no new loops are be formed.

What remains to be addressed is the method by which process u determines that all its descendants have a hop count of at least L. We present a property similar to the rank ordering property of the previous section. Previously, $u.D = i$ implied that all processes at most i hops below u have a rank that is at most $u.r$. We now strengthen the meaning of $u.D = i$ to also imply that, if $u.h \geq L$, then all processes at most i hops below u have a hop count of at least L.

We refer to the pair of values $(u.r, u.h)$ as the *extended-rank* of u. For terseness, we will write this pair as $u.(r, h)$. Below, we define a relation $\preceq$ on extended-ranks[1]. The loop-avoidance conditions (Definition 1) of the previous section also hold for extended-ranks.

We define $\preceq$ on extended-ranks as follows: $(r, h) \preceq (r', h')$ iff

$$r \preceq r' \wedge (h' \geq L \Rightarrow h \geq L)$$

We define $(r, h) \prec (r', h')$ similarly, except that $r \preceq r'$ is replaced by $r \prec r'$.

The rank-ordering property of the previous section (Definition 2) can now be replaced by the following extended-rank-ordering property.

Definition 4. (Extended-Rank Ordering Property)
Consider any active path P, where: $t = P_1, g = P_{|P|-1}, u = P_{|P|}$, and $2 \leq |P| \leq u.D[g]$. Then, the following holds:

$$\begin{aligned} t.(r, h) \preceq u.(r, h) \wedge (t.(\widetilde{r}, \widetilde{h}) \preceq u.(r, h) \vee P_2 \notin t.\widetilde{S}) \wedge \\ (\forall x : neigh(x, t) : upd(t, x).(r, h) \preceq u.(r, h)) \end{aligned}$$

In addition, if $u.D[g] = 1$, then $upd(u, g).(r, h) \preceq u.(r, h)$. □

Using the above property, each process u can deduce that, if $u.D = L \wedge u.h \geq L$, then all processes below u have a hop count of at least L. Once this happens, u, can reduce its hop count to less than L (if allowed by the hop counts of its successors) and make changes to its successor set.

We may now present the specification of a non-root process u in the c-DAG-forming network of processes.

[1] We overload the symbol $\succeq$ to be a relation on ranks and a relation on extended-ranks. Which of these two meanings is appropriate is evident from the context.

```
process $u$
inp
  $G$          : set of pid's              {neighbor set}
  $L$          : integer                   {max. path length}
var
  $S, \widetilde{S}$       : subset of $G$              {successor set and its iteration set}
  $r, \widetilde{r}$       : element of $R$             {rank and its iteration value}
  $h, \widetilde{h}$       : $0 .. 2L$                  {hop count and its iteration value}
  $D$          : array $[G]$ of $0 .. L$    {rank depth}
par
  $g$          : element of $G$             {any neighbor}
begin
  timeout $upd \notin Ch(u,g) \wedge ack \notin Ch(g,u)$ $\rightarrow$
                          $D[g] := max(1, D[g])$;
                          $upd.r, upd.h := r, h$;
                          send $upd$ to $g$
[]
  rcv $upd$ from $g$ $\rightarrow$     $\widetilde{S} := \widetilde{S} \bigcup \{g\}$;
                          $S := S \bigcup \{g\}$   if $new\_succ(g)$;
                          $\widetilde{r} := any\{x \mid x \preceq min(\widetilde{r}, upd.r)\}$
                                  if $g \in S$;
                          $\widetilde{h} := max\{\widetilde{h}, upd.h + 1\}$
                                  if $g \in S$;
                          $S, \widetilde{r}, \widetilde{h} := \{g\}, upd.r, upd.h + 1$
                                  if $break(g)$;
                          $reply(g)$
[]
  rcv $ack$ from $g$ $\rightarrow$     $D[g] := max(D[g], ack.d + 1)$
                                  if $ack.(r,h) \preceq (r,h) \wedge D[g] > 0$

[]
  $S \subseteq \widetilde{S}$ $\rightarrow$                $\widetilde{h} := max(\widetilde{h}, L)$    if $max(h, \widetilde{h}) \geq L \wedge D < L$;
                          $\widetilde{r} = \perp$ if $\widetilde{h} = 2L$;
                          $D := 0$             if $(\widetilde{r}, \widetilde{h}) \prec (r, h)$;
                          $r, h, \widetilde{r}, \widetilde{h}, \widetilde{S} := \widetilde{r}, \widetilde{h}, \top, 0, \emptyset$
[]
  any $\wedge$ $|S| > 1$ $\rightarrow$       $S := S - \{g\}$       if $\neg(max(h, \widetilde{h}) \geq L \wedge D < L)$
end
```

Process u consists of five actions. In the first action, process u sends an *upd* message to a neighbor g. This action is the same as before except that the message also contains the hop count.

In the second action, an *upd* message is received from a neighbor g. The first two statements are similar to those in the previous section. In this action, $new\_succ(g)$ is equivalent to the following.

$$(r,h) \prec upd.(r,h) \wedge D = L \wedge upd.h < L$$

Thus, the only difference from before is that extended-ranks are used when comparing the values of u against those of the received message, and furthermore, $upd.h < L$ is necessary to satisfy the loop-detection conditions.

The next two statements in the action remain the same. The fifth statement breaks away from a loop. Here, $break(g)$ is defined as follows.

$$(r,h) = (\perp, 2L) \wedge D = L \wedge (r,h) \prec upd.(r,h)$$

That is, if $h = 2L$, then u is involved in a loop, and it may choose g as its sole successor (thus breaking the loop) provided the loop avoidance conditions are not violated, i.e., the extended-rank of g is greater than that of u and $D = L$. The reason we chose $r = \perp$ is explained below.

Finally, $reply(g)$ in the second action is a shorthand for the following sequence of statements.

```
$ack.r, ack.h := r, h$;
$ack.d := L$            **if** $g \notin S$;
$ack.d := min\{D\}$ **if** $g \in S$;
**send** $ack$ **to** $g$
```

In the third action, an ack message is received from a neighbor g, and $D[g]$ is increased. The only difference between this action and that of the previous section is that the decision to increase $D[g]$ is based on process extended-ranks.

In the fourth action, r and h are updated from $\widetilde{r}$ and $\widetilde{h}$ after an upd message has been received from every neighbor. The action differs from the previous section by not allowing h to be reduced below L until all descendants of u have a hop count of at least L. This is necessary to satisfy the loop detection conditions. In addition, if $h = 2L$, i.e., if a loop is detected, the rank is set to the bottom value. This is done to "poison" all the descendants of u also with a bottom rank, and thus the successor which will be used to break the loop must have a rank higher than the bottom value.

In the fifth action, a successor is removed. This operation is not allowed if the hop count of u is at least L and there are still neighbors whose hop count is less than L. This is also necessary to satisfy the loop detection conditions.

The specification of the root process is shown below.

```
**process** $root$
**inp**
    $G$        :  set of pid's            {neighbor set}
    $L$        :  integer                 {max. path length}
**const**
    $S, \widetilde{S}$    :  $\emptyset, \emptyset$                  {successor set and its iteration set}
    $r, \widetilde{r}$    :  $\top, \top$                  {rank and its iteration value}
    $h, \widetilde{h}$    :  $0, 0$                  {hop count and its iteration value}
    $D$        :  array $[G]$ of $L$        {rank depth}
**par**
```

```
    $g$       : element of $G$          {any neighbor}
  begin
    timeout $upd \notin Ch(u,g) \wedge ack \notin Ch(g,u)$ →
                          $upd.r, upd.h := r, h$;
                          send $upd$ to $g$
  []
    rcv $upd$ from $g$ →    $reply(g)$
  []
    rcv $ack$ from $g$ →    skip
  end
```

The process consists of four simple actions. In the first action, the root sends an *upd* message to a neighbor g. In the second action, the root receives an *upd* message and it returns an *ack* message. In this action, *reply*(g) is a shorthand for the following sequence of statements.

$ack.r, ack.h, ack.d := \top, 0, L$;
send ack **to** g

In the third action, the root receives (and discards) an *ack*.

7 Protocol Correctness

Due to space restrictions, we present the proof of correctness in [23]. Here, we very briefly outline the proof for the interested reader.

A network stabilizes to a predicate Z iff every computation of the network contains a suffix where each state of the computation satisfies Z [14]. Thus, the system will reach a state after which it will continuously satisfy Z.

Starting from any initial state, the first property that is restored automatically is the rank-ordering property (in the dynamic-rank network), and the extended-rank-ordering property (in the c-DAG-forming network). Since the structure of the proof is similar for both networks, in [23], we abstract both of these proofs into a single one by introducing a network of abstract processes, where each abstract process has a general behavior that captures the behavior of both the dynamically-ranked processes and the stabilizing processes.

Theorem 1. (Restoring Ranks)

– *The rank-ordering property (Definition 2) is stabilizing in the dynamic-rank network of processes (Section 4).*
– *The extended-rank-ordering property (Definition 4) is stabilizing in the c-DAG-forming network (Section 6).* □

Once ranks between nodes have the correct relationship, new loops cannot be formed, and we have the following.

Theorem 2. (Loop-Freedom)
The c-DAG-forming network stabilizes to the following:

$$(\forall u :: \neg(\exists P :: active(P) \wedge (P_1 = u) \wedge loop(P))) \qquad \Box$$

That is, loops are broken when the hop-count of processes reaches $2 \cdot L$, and no new loops are formed due to the label-ordering property. Finally, the desired state is then reached.

Theorem 3. (c-DAG Restoration)
The c-DAG-forming network stabilizes to the following:

$$(\forall\, u \;::\; (\exists\, P \;::\; active(P) \wedge P_1 = u \wedge P_{|P|} = root) \qquad \Box$$

We therefore have that all active paths are loop-free, and each node has at least one active path to the root, i.e., a c-DAG is restored and maintained.

References

1. Hedrick, C.: Routing information protocol. RFC 1058 (1988)
2. Moy, J.: Ospf version 2. RFC 1247 (August 1991)
3. Cobb, J.A., Gouda, M.G.: The request-reply family of group routing protocols. IEEE Transactions on Computers 46(6), 659–672 (1997)
4. Deering, S., Cheriton, D.: Multicast routing in datagram networks and extended lans. ACM Transactions on Computer Systems 8(2) (May 1990)
5. Garcia-Luna-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Transactions on Networking 1(1) (February 1993)
6. Garcia-Luna-Aceves, J.J., S., M.: A path-finding algorithm for loop-free routing. IEEE/ACM Transactions on Networking 5(1) (February 1997)
7. Segall, A.: Distributed network protocols. IEEE Transactions on Information Theory IT-29(1), 23–35 (1983)
8. Garcia-Luna-Aceves, J., Soumya, R.: On-demand loop-free routing with link vectors. In: Proceedings of the 12th IEEE International Conference on Network Protocols, IEEE Computer Society Press, Los Alamitos (2004)
9. Johnson, D.B., Maltz, D.A., Hu, Y.C.: The dynamic source routing protocol for mobile ad hoc networks (dsr). draft-ietf-manet-dsr-09.txt (work in progress)
10. Perkins, C.E., Royer, E.M.: Ad hoc on-demand distance vector routing. In: Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, pp. 90–100. IEEE Computer Society Press, Los Alamitos (1999)
11. Vutukury, S., Garcia-Luna-Aceves, J.J.: An algorithm for multi-path computation using distance-vectors with predecessor information. In: Proceedings of the ICCCN Conference (1999)
12. Vutukury, S., Garcia-Luna-Aceves, J.J.: A distributed algorithm for multi-path computation. In: Proceedings of the IEEE GLOBECOM Conference, IEEE Computer Society Press, Los Alamitos (1999)
13. Zaumen, W., Garcia-Luna-Aceves, J.J.: Loop-free multi-path routing using generalized diffusing computations. In: Proc. of the INFOCOM Conference (1998)
14. Gouda, M.G.: The triumph and tribulation of system stabilization. In: Helary, J.-M., Raynal, M. (eds.) WDAG 1995. LNCS, vol. 972, pp. 1–18. Springer, Heidelberg (1995)
15. Arora, A., Gouda, M.G., Herman, T.: Composite routing protocols. In: Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos (1990)

16. Cobb, J.A., Gouda, M.G.: Stabilization of general loop-free routing. Journal of Parallel and Distributed Computing 62, 922–944 (2002)
17. Cobb, J.A., Waris, M.: Propagated timestamps: A scheme for the stabilization of maximum-flow routing protocols. In: Proceedings of the Third Workshop on Self-Stabilizing Systems, pp. 185–200 (1997)
18. Gouda, M.G., Schneider, M.: Maximum flow routing. In: Proceedings of the Second Workshop on Self-Stabilizing Systems (1995)
19. Cobb, J.A.: Convergent multipath routing. In: Proceedings of the International Conference on Network Protocols (2002)
20. Gouda, M.: The Elements of Network Protocols. Wyley (1997)
21. Gouda, M.G., Schneider, M.: Maximizable routing metrics. In: Proceedings of the IEEE International Conference on Network Protocols, pp. 71–78. IEEE Computer Society Press, Los Alamitos (1998)
22. Gouda, M., Schneider, M.: Stabilization of maximal metric trees. In: Proceedings of the Workshop on Self-Stabilizing Systems at the International Conference on Distributed Computing Systems (June 1999)
23. Cobb, J.A.: Stabilization of loop-free redundant routing. The University of Texas at Dallas technical report (2007)

Secure Failure Detection in TrustedPals[⋆]

Roberto Cortiñas[1], Felix C. Freiling[2], Marjan Ghajar-Azadanlou[3], Alberto Lafuente[1], Mikel Larrea[1], Lucia Draque Penso[2], and Iratxe Soraluze[1]

[1] The University of the Basque Country, San Sebastián, Spain
[2] Department of Computer Science, University of Mannheim, Germany
[3] Department of Computer Science, RWTH Aachen University, Germany

Abstract. This paper presents a modular redesign of TrustedPals, a smartcard-based security framework for solving secure multiparty computation (SMC). TrustedPals allows to reduce SMC to the problem of fault-tolerant consensus between smartcards. Within the redesign we investigate the problem of solving consensus in a general omission failure model augmented with failure detectors. To this end, we give novel definitions of both consensus and the class of $\diamond\mathcal{P}$ failure detectors in the omission model and show how to implement $\diamond\mathcal{P}$ and have consensus in such a system with some weak synchrony assumptions. The integration of failure detection into the TrustedPals framework uses tools from privacy enhancing techniques such as message padding and dummy traffic.

1 Introduction

Consider a set of parties who wish to correctly compute some common function F of their local inputs, while keeping their local data as private as possible, but who do not trust each other, nor the channels by which they communicate. This is the problem of *Secure Multi-party Computation* (SMC) [22]. SMC is a very general security problem, i.e., it can be used to solve various real-life problems such as distributed voting, private bidding and auctions like Ebay, sharing of signature or decryption functions and so on. Unfortunately, solving SMC is—without extra assumptions—very expensive both in terms of communication (number of messages) and time (number of synchronous rounds).

TrustedPals [3] is a smartcard-based implementation of SMC which allows much more efficient solutions to the problem. Conceptually, TrustedPals considers a distributed system in which processes are locally equipped with tamper proof security modules (see Fig. 1). In practice, processes are implemented as a Java desktop application and security modules are realized using Java Card Technology enabled smartcards [5]. Roughly speaking, solving SMC between processes is achieved by having the security modules jointly simulate a *trusted third party* (TTP), as we now explain.

[⋆] Work by the Spanish authors was supported by the Spanish Research Council, under grant HA2005-0078. Work by the German authors was supported by DAAD PPP Programme *Acciones Integradas Hispano Alemanas*.

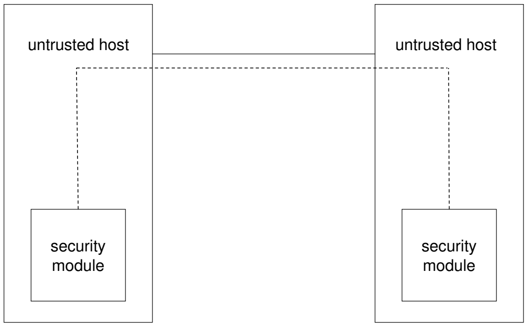


Fig. 1. Processes with tamper proof security modules

To solve SMC in the TrustedPals framework, the function F is coded as a Java function and is distributed within the network in an initial setup phase. Then processes hand their input value to their security module and the framework accomplishes the secure distribution of the input values. Finally, all security modules compute F and return the result to their process. The network of security modules sets up confidential and authenticated channels between each other and operates as a *secure overlay* within the distribution phase. Within this secure overlay, arbitrary and malicious behavior of an attacker is reduced to rather benign faulty behavior (process crashes and message omissions). TrustedPals therefore allows to reduce the security problem of SMC to a fault-tolerant synchronization problem [3], namely that of *consensus*.

To date, TrustedPals assumed a *synchronous* network setting, i.e., a setting in which all important timing parameters of the network are known and bounded. This makes TrustedPals sensitive to unforeseen variations in network delay and therefore not very suitable for deployment in networks like the Internet. In this paper, we explore how to make TrustedPals applicable in environments with less synchrony. More precisely, we explore the possibilities to implement TrustedPals in a modular fashion inspired by results in fault-tolerant distributed computing: We use an *asynchronous* consensus algorithm and encapsulate (some weak) timing assumptions within a device known as a *failure detector* [4].

The concept of a failure detector has been investigated in quite some detail in systems with merely crash faults [13]. In such systems, correct processes (i.e., processes which do not crash) must eventually permanently suspect crashing processes. There is very little work on failure detection and consensus in message omissions environments. In fact, it is not clear what a sensible definition of a failure detector (and consensus) is in such environments because the notion of a correct process can have several different meanings (e.g., a process with no failures whatsoever or a process which just does not crash but omits messages).

Related Work. Delporte, Fauconnier and Freiling [8] were the first to investigate non-synchronous settings in the TrustedPals context. Following the approach of

Chandra and Toueg [4] (and similar to this paper) they separate the trusted system into an asynchronous consensus layer and a partially synchronous failure detection layer. They assume that transient omissions are masked by a piggybacking scheme. The main difference however is that they solve a *different version of consensus* than we do: Roughly speaking, message omissions can cause processes to communicate only indirectly, i.e., some processes have to relay messages for other processes. Delporte, Fauconnier and Freiling [8] only guarantee that all processes that can communicate directly with each other solve consensus. In contrast, we allow also those processes which can only communicate indirectly to successfully participate in the consensus. As a minor difference, we focus on the class $\diamond\mathcal{P}$ of eventually perfect failure detectors whereas Delporte, Fauconnier and Freiling [8] implement the less general class Ω. Furthermore, Delporte, Fauconnier and Freiling [8] do not describe how to integrate failure detection within the TrustedPals framework: A realistic adversary who is able to selectively influence the algorithms for failure detection and consensus can cause their consensus algorithm to fail.

Apart from Delporte, Fauconnier and Freiling [8], other authors also investigated solving consensus in systems with omission faults. Unpublished work by Dolev et al. [10,9] also follows the failure detector approach to solve consensus, however they focus on the class $\diamond\mathcal{S}(om)$ of failure detectors. Babaoglu, Davoli and Montresor [19] also follow the path of $\diamond\mathcal{S}$ to solve consensus in partitionable systems.

Recently, solving SMC *without* security modules has received some attention focusing on two-party protocols [17,18]. In systems *with* security modules, Avoine and Vaudenay [2] examined the approach of jointly simulating a TTP. This approach was later extended by Avoine et al. [1] who show that in a system with security modules fair exchange can be reduced to a special form of consensus. They derive a solution to fair exchange in a modular way so that the agreement abstraction can be implemented in diverse manners. Benenson et al. [3] extended this idea to the general problem of SMC and showed that the use of security modules cannot improve the resilience of SMC but enables more efficient solutions for SMC problems. All these papers assume a *synchronous* network model.

Correia et al. [6] present a system which employs a real-time distributed security kernel to solve SMC. The architecture is very similar to that of TrustedPals as it also uses the notion of architectural hybridization [21]. However, the adversary model of Correia et al. [6] assumes that the attacker only has remote access to the system while TrustedPals allows the owner of a security module to be the attacker. Like other previous work [3,2,1] Correia et al. [6] also assume a synchronous network model at least in a part of the system.

Our work on TrustedPals can also be regarded as building failure detectors for arbitrary (*Byzantine*) failures which has been investigated previously (see for example Kihlstrom, Moser and Melliar-Smith [15] and Doudou, Garbinato and Guerraoui [11]). In contrast to previous work on Byzantine failure detectors, we use security modules to avoid the tar pits of this area.

Contributions. In this paper we present a modular redesign of TrustedPals using consensus and failure detection as modules. More specifically, we make the following technical contributions:

- We give a novel definition of $\diamond\mathcal{P}$ in the omission model and we show how to implement $\diamond\mathcal{P}$ in a system with weak synchrony assumptions in the spirit of partial synchrony [12].
- We give a novel definition of consensus in the omission model and give an algorithm which uses the class $\diamond\mathcal{P}$ to solve consensus. The algorithm is an adaptation of the classic algorithm by Chandra and Toueg [4] for the crash model.
- We integrate failure detection and consensus securely in TrustedPals by employing message padding and dummy traffic, tools known from the area of privacy enhancing techniques.

Paper Outline. This paper is structured as follows: In Sect. 2 we give an overview over and motivate the system model of TrustedPals. In Sect. 3 we define and implement the failure detector $\diamond\mathcal{P}$ in the omission failure model. We then use this failure detector to solve consensus in Sect. 4. In Sect. 5 we describe how to integrate failure detection and consensus securely in the TrustedPals framework. For lack of space, the correctness proofs of the algorithms as well as more details on the security evaluation can be found elsewhere [7].

2 System Model and Architecture

2.1 Untrusted and Trusted System

To be able to precisely reason about algorithms and their properties in the TrustedPals system we now formalize the system assumptions within a hybrid model, i.e., the model is divided into two parts (see Fig. 2). The upper part consists of n processes which represent the *untrusted hosts*. The lower part equally consists of n processes which represent the security modules. Because of the lack of mutual trust between untrusted hosts, we call the former part the *untrusted system*. Since the security modules trust each other we call the latter part the *trusted system*. Each host is connected to exactly one security module by a direct communication link.

Summarizing, there are two different types of processes: processes in the untrusted system and processes in the trusted system. For brevity, we will use the unqualified term *process* if the type of process is clear from the context.

Within the untrusted system each pair of hosts is connected by a pair of unidirectional communication links, one in each direction. Since the security modules also must use these links to communicate, the trusted system can be considered as an overlay network which is a network that is built on top of another network. Nodes in the overlay network can be thought of as being connected by virtual or

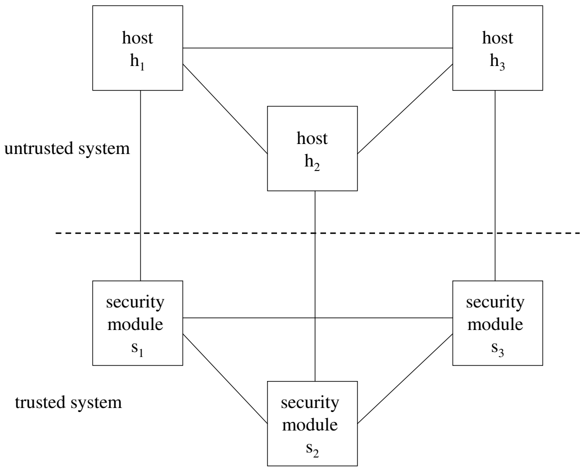


Fig. 2. The untrusted and trusted system

logical links. In practice, for example, smartcards could form the overlay network which runs on top of the Internet modeled by the untrusted processes. Within the trusted system we assume the existence of a public key infrastructure, which enables two communicating parties to establish confidentiality, message integrity and user authentication without having to exchange any secret information in advance.

We assume reliable channels, i.e., every message inserted to the channel is eventually delivered at the destination. We assume no particular ordering relation on channels.

2.2 Timing Assumptions

We assume that a local clock is available to each host, but clocks are not synchronized within the network. Security modules do not have any clock, they just have a simple step counter, whereby a step consists of receiving a message from other security modules, executing a local computation, and sending a message to other security modules. Passing of time is checked by counting the number of steps executed.

Since trusted and untrusted system operate over the same physical communication channel, we assume the same timing behavior for both systems. Both systems are assumed to be *partially synchronous* meaning that eventually bounds on all important network parameters (processing speed differences, message delivery delay) hold. The model is a variant of the partial synchrony model of Dwork, Lynch and Stockmeyer [12]. The difference is that we assume reliable channels.

We say that a message is *received timely* if it is received after the bounds on the timing parameters hold. Omission of such a message can be reliably detected using timeout-based reasoning.

2.3 Failure Assumptions

The model is hybrid because we have distinct failure assumptions for both systems. The failure model we assume in the untrusted system is the *Byzantine failure model* [16]. A Byzantine process can behave arbitrarily. In the trusted system we assume the failure model of *general omission*, which we now explain.

The concept of *omission* faults, meaning that a process drops a message either while sending (*send* omission) or while receiving it (*receive* omission), was introduced by Hadzilacos [14] and later generalized by Perry and Toueg [20]. The failure model used for the trusted system is that of *general omission*, in which processes can crash and experience either send-omissions or receive omissions. We allow the possibility of *transient* omissions, i.e., a process may temporarily drop messages and later on reliably deliver messages again.

A process (untrusted host or security module) is *correct* if it does not fail. A process is *faulty* if it is not correct. We assume a majority of processes to be correct both in the untrusted and in the trusted system. Note that a faulty security module implies a faulty host but a faulty host not necessarily implies a faulty security module.

The motivation behind this hybrid approach is that the system runs in an environment prone to attacks, but the assumptions on the security modules and the possibility to establish secure channels reduce the options of the attacker in the trusted system to attacks on the liveness of the system, i.e., destruction of the security module or interception of messages on the channel.

2.4 Classes of Processes in the Trusted System

The omission model in the trusted system implies the possibility of both transient send omissions and receive omissions. Given two processes, p and q, if a single message m sent from p to q is not delivered by q, the following question arises: has p suffered a send omission, or has q suffered a receive omission? Formally, one of the two processes is incorrect, but it is not possible to determine which one. Observe that considering both processes p and q incorrect can be too restrictive. This leads us to reconsider the different classes of processes in the omission model with respect to the common correct/incorrect classification. In particular, processes suffering a limited number of omissions, e.g., processes that do not suffer omissions with some correct process, will be considered as *good*, since they can still participate in a distributed protocol like consensus.

On the basis of this motivation, we consider the following two classes of processes:

Definition 1. *A process p is* in-connected *if and only if:*

(1) p is a correct process, or
(2) p does not crash and there exists a process q such that q is in-connected and all messages sent by q to p are eventually received timely by p (i.e., q does not suffer any send-omission with p, and p does not suffer any receive-omission with q).

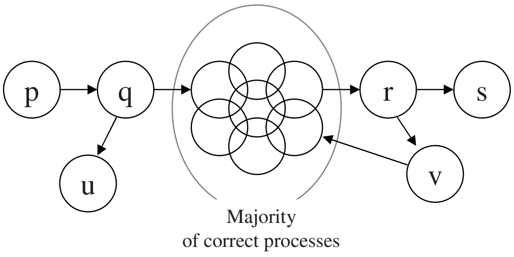


Fig. 3. Examples for classes of processes

Definition 2. *A process p is* out-connected *if and only if:*

(1) p is a correct process, or
(2) p does not crash and there exists a process q such that q is out-connected and all messages sent by p to q are eventually received timely by q (i.e., p does not suffer any send-omission with q, and q does not suffer any receive-omission with p).

Observe that correct processes are both in-connected and out-connected. Observe also that the definitions of in-connected and out-connected processes are recursive. Intuitively, there is a timely path with no omissions from every correct process to every in-connected process. Also, there is a timely path with no omissions from every out-connected process to every correct process, and hence to every in-connected process.

Fig. 3 shows an example. In the figure, arcs represent timely links with no omissions (they are not shown for the majority of correct processes). Processes p and q are out-connected, while process s is in-connected, and processes r

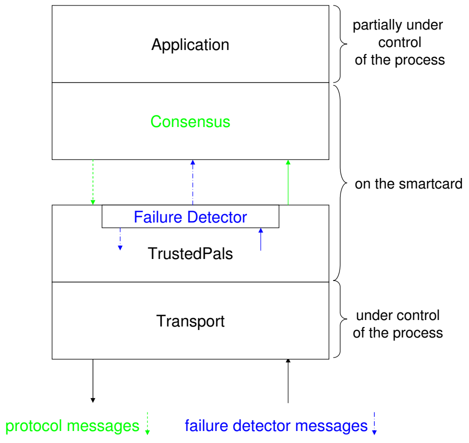

Application
partially under control of the process
Consensus
on the smartcard
Failure Detector
TrustedPals
Transport
under control of the process
protocol messages
failure detector messages

Fig. 4. The architecture of our system

and v are both in-connected and out-connected. Finally, process u is neither in-connected nor out-connected.

2.5 The TrustedPals Architecture

Fig. 4 shows the layers and interfaces of the proposed modular architecture for TrustedPals. A message exchange is performed on the transport layer, which is under control of the untrusted host. The failure detector and the security mechanisms for message encryption etc. run in the TrustedPals layer. In the consensus layer runs the consensus algorithm. On the application layer, which again is under the control of the untrusted host, protocols like fair exchange operate.

3 Failure Detection in TrustedPals

Based on the two new classes of processes defined in the previous section, we redefine now the properties that $\diamond\mathcal{P}$ must satisfy in the omission model. While the common correct/faulty classification of processes is well addressed by means of a list of suspected processes, in the omission model we will consider two lists of processes, one for the in-connected processes and the other one for the out-connected processes. If a process p has a process q in its list of in-connected (out-connected) processes, we say that *p considers q as in-connected* (*out-connected*). The $\diamond\mathcal{P}$ class of failure detectors in the omission model satisfies the following properties:

– *Strong Completeness.* Eventually every process that is not out-connected will be permanently considered as not out-connected by every in-connected process.
– *Eventual Strong Accuracy.* Eventually every process that is out-connected will be permanently considered as out-connected by every in-connected process.
– *In-connectivity.* Eventually every process that is in-connected will permanently consider itself as in-connected.

Figs. 5, 6 and 7 present an algorithm implementing $\diamond\mathcal{P}$. The algorithm provides to every process p a list of in-connected processes, $InConnected_p$, and another list of out-connected processes, $OutConnected_p$. For every in-connected process p, these lists will have the information required to satisfy the properties of $\diamond\mathcal{P}$. In particular, the list $OutConnected_p$ will eventually and permanently contain exactly all the out-connected processes. Regarding the $InConnected_p$ list, it will eventually and permanently contain p itself.

In order to detect message omissions, messages carry a sequence number. Besides, every process p uses a matrix M_p of $n \times n$ elements. In the beginning, all processes are supposed to be correct, so every element in the matrix has a value of 1. If all messages sent from a process q to a process p are received timely by p, $M_p[p][q]$ will be maintained to 1. Otherwise, process p will set $M_p[p][q]$ to

```
Procedure main()
    $InConnected_p \leftarrow \Pi$
    $OutConnected_p \leftarrow \Pi$
    forall $q \in \Pi - \{p\}$ do
        $\Delta_p(q) \leftarrow$ default time-out interval    {$\Delta_p(q)$ denotes the duration of $p$'s time-out interval for $q$}
        $next\_send_p[q] \leftarrow 1$    {sequence number of the next message sent to $q$}
        $next\_receive_p[q] \leftarrow 1$    {sequence number of the next message expected from $q$}
        $Buffer_p[q] \leftarrow \emptyset$

    forall $q \in \Pi$ do
        forall $u \in \Pi$ do
            $M_p[q][u] \leftarrow 1$    {$M_p[q][u] = 0$ means that $q$ has not received at least one message from $u$}
        $Version_p[q] \leftarrow 0$    {$Version_p$ contains the version number for every row of $M_p$}

    $UpdateVersion \leftarrow$ false

    || Task 1: repeat periodically
        if $UpdateVersion$ then    {$p$'s row has changed}
            $Version_p[p] \leftarrow Version_p[p] + 1$
            $UpdateVersion \leftarrow$ false
        forall $q \in \Pi - \{p\}$ do
            send ($ALIVE$, $p$, $next\_send_p[q]$, $M_p$, $Version_p$) to $q$    {sends a heartbeat}
            $next\_send_p[q] \leftarrow next\_send_p[q] + 1$    {$p$ updates its sequence number for $q$}

    || Task 2: repeat periodically
        if ( $p$ did not receive ($ALIVE$, $q$, $next\_receive_p[q]$, $M_q$, $Version_q$)
             from $q \neq p$ during the last $\Delta_p(q)$ ticks of $p$'s clock ) then
        {the next message in the sequence has not been received timely}
            $\Delta_p(q) \leftarrow \Delta_p(q) + 1$
            if $M_p[p][q] = 1$ then
                $M_p[p][q] \leftarrow 0$    {the potential omission is reflected in $M_p$}
                $UpdateVersion \leftarrow$ true
                call update_In_Out_Connected_lists()

    || Task 3: when receive ($ALIVE$, $q$, $c$, $M_q$, $Version_q$) for some $q$
        if $c = next\_receive_p[q]$ then    {it is the next message expected from $q$}
            call deliver_next_message($q$, $M_q$, $Version_q$)    {the message is delivered}
            $next\_receive_p[q] \leftarrow next\_receive_p[q] + 1$
            while ($ALIVE$, $q$, $next\_receive_p[q]$, $M_q$, $Version_q$) $\in Buffer_p[q]$ do
                call deliver_next_message($q$, $M_q$, $Version_q$)
                remove ($ALIVE$, $q$, $M_q$, $next\_receive_p[q]$, $Version_q$) from $Buffer_p[q]$
                $next\_receive_p[q] \leftarrow next\_receive_p[q] + 1$
            if $Buffer_p[q] = \emptyset$ then
                $M_p[p][q] \leftarrow 1$    {so far $p$ has received all messages from $q$}
                $UpdateVersion \leftarrow$ true
            if $M_p$ has changed then
                call update_In_Out_Connected_lists()
        else
            insert ($ALIVE$, $q$, $c$, $M_q$, $Version_q$) into $Buffer_p[q]$
```

Fig. 5. $\diamond\mathcal{P}$ in the omission model: main algorithm

0. In this way, the matrix will have the information needed to calculate the lists of in-connected and out-connected processes.

Actually, M represents the transposed adjacency matrix of a directed graph, where the value of the element $M[p][q]$ shows if there is an arc from q to p. We can derive from powers of the adjacency matrix if there is a path with no

```
Result: $InConnected_p$ and $OutConnected_p$ lists
Procedure update_In_Out_Connected_lists()
    $A_p \leftarrow (M_p)^n$                                   {$A_p$ is the $n$-th power of the $M_p$ matrix}
    forall $u, v \in \Pi$ do
        if $A_p[u][v] > 0$ then
            $A_p[u][v] \leftarrow 1$
    $In \leftarrow \emptyset$
    $Out \leftarrow \emptyset$
    forall $q \in \Pi$ do
        if $(\sum_{i=0}^{n-1} A_p[q][i] \geq \lceil \frac{(n+1)}{2} \rceil)$ then
            $In \leftarrow In \cup \{q\}$
        if $(\sum_{i=0}^{n-1} A_p[i][q] \geq \lceil \frac{(n+1)}{2} \rceil)$ then
            $Out \leftarrow Out \cup \{q\}$
    $InConnected_p \leftarrow In$
    $OutConnected_p \leftarrow Out$
```

Fig. 6. $\diamond\mathcal{P}$ in the omission model: procedure update_In_Out_Connected_lists()

```
Input: $q$: process from which the message has been received; $M_q$: $q$'s knowledge about the
       system; $Version_q$: version number of each row of $M_q$
Result: update of $M_p$ matrix and $Version_p$ vector
Procedure deliver_next_message()
    forall $v \in \Pi$ do                          {$q$'s row of $M_q$ is systematically copied into $M_p$}
        $M_p[q][v] \leftarrow M_q[q][v]$
    forall $u \in \Pi - \{p, q\}$ do
        if $Version_q[u] > Version_p[u]$ then      {$q$'s information about $u$ is more recent than
        $p$'s}
            forall $v \in \Pi$ do
                $M_p[u][v] \leftarrow M_q[u][v]$
            $Version_p[u] \leftarrow Version_q[u]$
```

Fig. 7. $\diamond\mathcal{P}$ in the omission model: procedure deliver_next_message()

omission of any length between every pair of processes. Observe that in the given algorithm a process does not monitor itself and, as a consequence, the elements of the main diagonal of the matrix are always set to 1. Taking this into account, the n-th power of the adjacency matrix, $A_p = (M_p)^n$, gives us the information we need to obtain the sets of in-connected and out-connected processes. A process p is in-connected if it is able to receive all the messages (either directly or indirectly) from at least $\lceil \frac{(n+1)}{2} \rceil$ processes. Similarly, a process p is out-connected if at least $\lceil \frac{(n+1)}{2} \rceil$ processes are able to receive (either directly or indirectly) all the messages sent by p. The lists of in-connected and out-connected processes are computed in the *update_In_Out_Connected_lists*() procedure, which is called every time a value of the matrix M_p is changed.

In Task 1 (line 14), a process p periodically sends a heartbeat message to the rest of processes. When a message is sent, the sequence number associated to the destination is incremented. Observe that the matrix M_p is sent in the heartbeat messages.

In Task 2 (line 21), if a process p does not receive the next expected message from a process q in the expected time, the value of $M_p[p][q]$ is set to 0.

In Task 3 (line 28), received messages are processed. The messages a process p receives from another process q are delivered following the sequence number $next\_receive_p[q]$. Every process p has a buffer for every other process q to store unordered messages received from q. If p receives a message from q with a sequence number different from the expected one, this message is inserted in $Buffer_p[q]$ and the message is not delivered yet (line 42). A message is delivered when it is the next expected message, either because it has been just received (line 30) or it is inside the buffer (line 33). If the delivered message was in the buffer, it is removed from there. Having delivered the next expected message from a process q, if the buffer is empty it means that there is no message left from q, so $M_p[p][q]$ is set to 1. This way, process p fills its corresponding row in the matrix indicating if all the messages it expected from every other process have been received timely.

The procedure *deliver_next_message*() is used to update the adjacency matrix M_p using the information carried by the message. In the procedure, process p copies into M_p the row q of the matrix M_q received from q. This way, p learns about q's input connectivity. With respect to every other process u, a mechanism based on version numbers is used to avoid copying old information about u's input connectivity. Process p will only copy into M_p the row u of M_q if its version number is higher.

4 $\Diamond\mathcal{P}$-Based Consensus in TrustedPals

In the *consensus* problem, every process proposes a value, and correct processes must eventually decide on some common value that has been proposed. In the crash model, every *correct process* is required to eventually decide some value. This is called the *Termination* property of consensus. In order to adapt consensus to the omission model, we argue that only the Termination property has to be redefined. This property involves now every in-connected process, since, despite they can suffer some omissions, in-connected processes are those that will be able to decide.

The properties of consensus in the omission model are the following:

– *Termination.* Every *in-connected* process eventually decides some value.
– *Integrity.* Every process decides at most once.
– *Uniform agreement.* No two processes decide differently.
– *Validity.* If a process decides v, then v was proposed by some process.

Figs. 8 and 9 present an algorithm solving consensus using $\Diamond\mathcal{P}$ in the omission model. It is an adaptation of the well-known Chandra-Toueg consensus algorithm. Instead of explaining the algorithm from scratch, we just comment on the modifications required to adapt the original algorithm:

– In Phase 2, the current coordinator waits for a majority of estimates while it considers itself as in-connected in order not to block. Only in case it receives a majority of estimates a valid estimate is sent to all. If it is not the case,

```
     {Every process p executes the following}
(1)  Procedure propose($v_p$)
(2)      $estimate_p \leftarrow v_p$                {$estimate_p$ is p's estimate of the decision value}
(3)      $state_p \leftarrow undecided$
(4)      $r_p \leftarrow 0$                         {$r_p$ is p's current round number}
(5)      $ts_p \leftarrow 0$                        {$ts_p$ is the last round in which p updated $estimate_p$, initially 0}

         {Rotate through coordinators until decision is reached}
(6)      while $state_p = undecided$ do
(7)          $r_p \leftarrow r_p + 1$
(8)          $c_p \leftarrow (r_p \bmod n) + 1$     {$c_p$ is the current coordinator}
(9)          Phase 1: {All processes p send $estimate_p$ to the current coordinator}
(10)             send $(p, r_p, estimate_p, ts_p)$ to $c_p$

(11)         Phase 2:
             {The current coordinator tries to gather $\lceil\frac{(n+1)}{2}\rceil$ estimates. If it succeeds,
              it proposes a new estimate. Otherwise, it sends a $NEXT$ message to all}
(12)             if $p = c_p$ then
(13)                 wait until
                     ( ($p \in \Pi - InConnected_p$) or
                       (for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$ from q) )
(14)                 if for $\lceil\frac{(n+1)}{2}\rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$ from q then
(15)                     $success_p \leftarrow TRUE$
(16)                     $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received
                         $(q, r_p, estimate_q, ts_q)$ from $q\}$
(17)                     $t \leftarrow$ largest $ts_q$ such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
(18)                     $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t)$
                         $\in msgs_p[r_p]$
(19)                     send $(p, r_p, estimate_p)$ to all
(20)                 else
(21)                     $success_p \leftarrow FALSE$
(22)                     send $(p, r_p, NEXT)$ to all

(23)         Phase 3: {All processes wait for the new estimate proposed by the coordinator}
(24)             wait until
                 ( ($p \in \Pi - InConnected_p$) or
                   received [ $(c_p, r_p, estimate_{c_p})$ or $(c_p, r_p, NEXT)$ ] from $c_p$ or
                   ($c_p \in \Pi - OutConnected_p$) )
(25)             if received $(c_p, r_p, estimate_{c_p})$ from $c_p$ then
(26)                 $estimate_p \leftarrow estimate_{c_p}$
(27)                 $ts_p \leftarrow r_p$
(28)                 send $(p, r_p, ack)$ to $c_p$
(29)             else
(30)                 send $(p, r_p, nack)$ to $c_p$

(31)         Phase 4:
             {If the current coordinator sent a valid estimate in Phase 2, it waits for replies of
              out-connected processes while it considers itself as in-connected. If $\lceil\frac{(n+1)}{2}\rceil$
              processes replied with ack, the coordinator R-broadcasts a decide message}
(32)             if ($p = c_p$) and ($success_p = TRUE$) then
(33)                 wait until [ ($p \in \Pi - InConnected_p$) or
                                  for all process q: ( received $(q, r_p, ack)$ or
                                                       received $(q, r_p, nack)$ or
                                                       $q \in \Pi - OutConnected_p$ ) ]
(34)                 if for $\lceil\frac{(n+1)}{2}\rceil$ processes q:received $(q, r_p, ack)$ then
(35)                     R-broadcast$(p, r_p, estimate_p, decide)$
```

Fig. 8. Solving consensus in the omission model using $\diamond\mathcal{P}$: main algorithm

```
{If p R-delivers a decide message, p decides accordingly}
(36) when R-deliver($q, r_q, estimate_q, decide$) do
(37)     if $state_p = undecided$ then
(38)         $decide(estimate_q)$
(39)         $state_p \leftarrow decided$
```

Fig. 9. Solving consensus in the omission model using $\diamond\mathcal{P}$: adopting the decision

the coordinator sends a *NEXT* message indicating that the current round cannot be successful.

– In Phase 3, every process p waits for the new estimate proposed by the current coordinator while p considers itself as in-connected and the coordinator as out-connected in order not to block. Also, p can receive a *NEXT* message indicating that the current round cannot be successful. In case p receives a valid estimate, it replies with a *ack* message. Otherwise, p sends a *nack* message to the current coordinator.
– In Phase 4, if the current coordinator sent a valid estimate in Phase 2, it waits for replies of out-connected processes while it considers itself as in-connected in order not to block. If a majority of processes replied with *ack*, the coordinator R-broadcasts a decide message.

When a process p sends a consensus message m to another process q, the following approach is assumed: (1) p sends m to all processes, including q, except p itself, and (2) whenever p receives for the first time a message m whose destination is another process q different from p, p forwards m to all processes (except the process from which p has received m and p itself). Clearly, this approach can take advantage of the underlying all-to-all implementation of the $\diamond\mathcal{P}$ failure detector.

5 Integrating Failure Detection and Consensus Securely

As depicted in Fig. 4, the TrustedPals layer receives messages from the consensus protocol and from the failure detector. If an untrusted host could distinguish protocol messages from failure detector messages he could intercept all former messages while leaving the latter untouched. This would result in a failure detector working properly but a consensus protocol to block forever. In order to prevent such malicious actions we piggyback the protocol messages on the failure detector messages, which are sent in regular time intervals. To make sure that the adversary can not distinguish the packets with the protocol message piggybacked from the ones without protocol message, packets will have the same size, i.e., failure detector messages are padded and protocol messages are divided into a predefined length. It might be inefficient for small messages to be padded or large packets split up in order to get a message of the desired size. However, it is necessary to find an acceptable tradeoff between security and performance such that a message size provides better security in expense of worse performance.

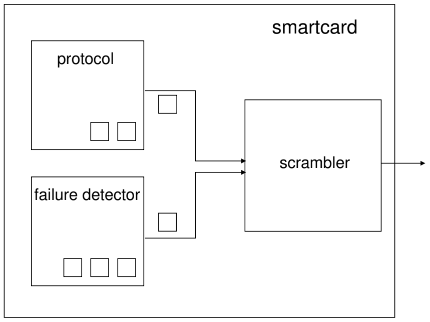


Fig. 10. Smartcard with scrambler

We assume a *scrambler* which receives the protocol and failure detector messages and outputs equal looking messages of the same size in regular time intervals (see Fig. 10). It proceeds as follows. Whenever a protocol message has to be sent, it will be piggybacked on the failure detector message. If there is no protocol message ready to be sent, the packet's payload will be filled with random bits. In order to be efficient, the predefined size of the messages sent will be kept as small as possible. If a protocol message is too big, it will be divided, using a fragmentation mechanism, and piggybacked into multiple failure detector messages. Since the protocol is asynchronous, even long delays can be tolerated as long as the failure detector works correctly.

Cryptography is applied to prevent and detect cheating and other malicious activities. We use a public key cryptosystem for encryption. Each message m in our model will be signed and then encrypted in order to reach authenticity, confidentiality, integrity, and non-repudiation.

The source and destination address are encrypted because this enables the receiver of a message to check whether the received message was intended for it or not and who the sender was. Thus, a malicious process cannot change the destination address in the header of a message from its security module and send it to an arbitrary destination without being detected. To detect a message deletion or loss, each message which is sent gets an identification number, where the fragment offset field determines the place of a particular fragment in the original message with same identification number.

As an example for the scrambler's function, consider the situation where the scrambler takes a protocol message m, whose size is three times the size of a failure detector message, from the queue of protocol messages to be sent. The scrambler divides the protocol message in three parts and assigns the next available sequence number to each part. Also each part gets a fragment offset. The first message part gets the fragment offset 1, the second message part gets the fragment offset 2, and the last message part gets the fragment offset 3. Next, the sequence number, all other fields, and the first message part all together are signed with the private key of the sender. After that, the signature is encrypted. Then, the next failure detector message is taken from the queue of failure detector messages to be sent and the encrypted message part is inserted into the failure detector message payload. Now, the first message part is ready to be sent in the next upcoming interval. The same is applied to the second and third part of the protocol message.

References

1. Avoine, G., Gärtner, F., Guerraoui, R., Vukolic, M.: Gracefully degrading fair exchange with security modules. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 55–71. Springer, Heidelberg (2005)
2. Avoine, G., Vaudenay, S.: Optimal fair exchange with guardian angels. In: Chae, K.J., Yung, M. (eds.) Information Security Applications. LNCS, vol. 2908, pp. 188–202. Springer, Heidelberg (2004)
3. Benenson, Z., Fort, M., Freiling, F., Kesdogan, D., Penso, L.D.: Trustedpals: Secure multiparty computation implemented with smartcards. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 306–314. Springer, Heidelberg (2006)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
5. Chen, Z.: Java Card Technology for Smart Cards, 1st edn. Addison-Wesley, Reading (2000)
6. Correia, M., Veríssimo, P., Neves, N.F.: The design of a COTS-Real-time distributed security kernel. In: Bondavalli, A., Thévenod-Fosse, P. (eds.) Dependable Computing EDCC-4. LNCS, vol. 2485, pp. 234–252. Springer, Heidelberg (2002)
7. Cortiñas, R., Freiling, F.C., Ghajar-Azadanlou, M., Lafuente, A., Larrea, M., Penso, L.D., Soraluze, I.: Secure Failure Detection in TrustedPals. Technical Report EHU-KAT-IK-07-07, The University of the Basque Country, (July 2007), Available at http://www.sc.ehu.es/acwlaalm/
8. Delporte-Gallet, C., Fauconnier, H., Freiling, F.C.: Revisiting failure detection and consensus in omission failure environments. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, Springer, Heidelberg (2005)
9. Dolev, D., Friedman, R., Keidar, I., Malkhi, D.: Failure detectors in omission failure environments. Technical Report TR96-1608, Cornell University, Computer Science Department (September 1996)
10. Dolev, D., Friedman, R., Keidar, I., Malkhi, D.: Failure detectors in omission failure environments. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing, p. 286 (1997)
11. Doudou, A., Garbinato, B., Guerraoui, R.: Encapsulating failure detection: from crash to Byzantine failures. In: Proceedings of the Int. Conference on Reliable Software Technologies, Vienna (May 2002)
12. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM 35(2), 288–323 (1988)
13. Freiling, F.C., Guerraoui, R., Kouznetsov, P.: The failure detector abstraction. Technical report, Department for Mathematics and Computer Science, University of Mannheim (2006)
14. Hadzilacos, V.: Issues of Fault Tolerance in Concurrent Computations. PhD thesis, Harvard University 1984, also published as Technical Report TR11-84
15. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Byzantine fault detectors for solving consensus. The Computer Journal 46(1) (2003)

16. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
17. MacKenzie, P., Oprea, A., Reiter, M.: Automatic generation of two-party computations. In: ACM SIGSAC. SIGSAC: 10th ACM Conference on Computer and Communications Security, ACM Press, New York (2003)
18. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — A secure two-party computation system. In: USENIX. Proceedings of the 13th USENIX Security Symposium (August 2004)
19. Babaoglu, Ö., Davoli, R., Montresor, A.: Group communication in partitionable systems: Specification and algorithms. IEEE Trans. Softw. Eng. 27(4), 308–336 (2001)
20. Perry, K.J., Toueg, S.: Distributed agreement in the presence of processor and communication faults. IEEE Transactions on Software Engineering 12(3), 477–482 (1986)
21. Sousa, P., Neves, N.F., Veríssimo, P.: Proactive resilience through architectural hybridization. In: Proceedings of the 2006 ACM Symposium on Applied Computing, pp. 686–690 (2006)
22. Yao, A.C.: Protocols for secure computations. In: Proceedings of the Twenty-Third Annual Symposium on Foundations of Computer Science, pp. 160–164 (1982)

Probabilistic Fault-Containment

Anurag Dasgupta[1], Sukumar Ghosh[2], and Xin Xiao[3]

[1] University of Iowa, USA
adasgupt@cs.uiowa.edu
[2] University of Iowa, USA
ghosh@cs.uiowa.edu
[3] University of Iowa, USA
xinxiao@cs.uiowa.edu

Abstract. Research on fine tuning stabilization properties has received attention for nearly a decade. This paper presents a probabilistic algorithm for fault-containment, that confines the effect of any single fault to the immediate neighborhood of the faulty process, with an expected recovery time of $O(\Delta^3)$. The most significant aspect of the algorithm is that the *fault-gap*, defined as the smallest interval after which the system is ready to handle the next single fault with the same efficiency, depends only on Δ, and is independent of the network size. We argue that a small fault-gap increases the availability of the fault-free system.

1 Introduction

A distributed system is *self-stabilizing*, when starting from an arbitrary initial configuration, the system returns to a *legitimate configuration* in a bounded number of steps, and remains in that configuration thereafter. Such arbitrary configurations may be caused by transient failures that can corrupt the system state. However, in most well-designed systems, the possibility of a massive failure is miniscule, and single failures are much more likely to occur. To increase the efficiency of fault-tolerance, it is important to guarantee a much faster recovery from all single failures, while also guaranteeing eventual recovery from more major failures.

The problem of containing the effect of minor failures is becoming important not only because they are more likely to occur, but also due to the dramatic growth of network sizes. In most self-stabilizing systems, a single transient failure can potentially contaminate a large portion of the system. The tight containment of the effect of single failures depends on the context: *containment in time* implies that all observable variables of the system recover to their legitimate values in $O(1)$ time, whereas *containment in space* means that the processes at $O(1)$ distance from the faulty process make observable changes. For optimal performance, both of these properties should hold.

An important issue in system design is the mean time between failures, commonly termed as MTBF. Once a fault-containing system recovers from a single failure in constant time, how much time will elapse before the system becomes ready to recover from the next single failure with the same efficiency, is an important metric, and is called the *fault-gap* [6]. If the next failure hits the system sooner than this period, then the guarantee of $O(1)$ time recovery falls apart. Thus fault-gap determines the availability of the fault-free system, and a low fault-gap reflects better availability. The growth of the network size increases the probability of the occurrence of failures. So scale-free fault-gap is an important design goal.

1.1 Our Contributions

In this paper, we present a probabilistic solution to the fault-containment problem for a class of distributed systems. The solution transforms a self-stabilizing distributed system S into a system T that is both weakly fault-containing and self-stabilizing, with a randomized scheduler. *Weak fault-containment* means that from all single failures, the expected recovery time of the transformed version is dependent only on the degree of the nodes, and independent of the network size. Furthermore, observable changes are confined to only the immediate neighbors of the faulty processes with a high probability. In addition to the weak fault-containment property, T recovers from all k-faulty $(k > 1)$ configurations in $O(k.n)$ steps.

Why should anyone care about an algorithm that allows the neighbors to be contaminated, when better solutions are available? The answer lies in the small *fault-gap*. The dramatic growth of the network size raises the probability of failures, and most solutions to fault-containment that we know of achieve a fault-gap of $O(n)$ or worse. As a result, when two single faults occur relatively quickly, the system fails to provide the guarantee of efficient recovery, and in fact the second single failure may require $O(n)$ (or higher) time for recovery. This seriously undermines the availability of the fault-free system. Our solution guarantees that the fault-gap depends only on the degree of the nodes, and is independent of the size of the network. This will significantly increase the *availability* of the fault-free system. In fact, our solution handles the recovery from simultaneous failures at nodes that are distance-3 apart with the same efficiency, as long as such failures occur at intervals of $O(\Delta^3)$ or higher.

1.2 Related Work

Kutten and Peleg [11] introduced a protocol for fault-mending that corrects the systems from minor failures, but provides no guarantee for stabilization. Ghosh et al [6] [7] demonstrated how containment can be combined with stabilization, and analyzed the cost of it. Dolev and Herman introduced *superstabilizing* protocols [4], that, in addition to being stabilizing, guarantee that during convergence from configurations that arise from legitimate states by small-scale topology changes, certain passage predicates are satisfied.

For specific problems, self-stabilizing protocols that exhibit certain fault-containment properties have been studied: [5] solves the leader election problem on a ring, [8] addresses the spanning tree construction problem, and [9] demonstrates BFS tree construction.

Herman's self-stabilizing protocol [10] for mutual exclusion on a ring contains the effect of any spurious token that may have been generated by a single-process fault. Kutten and Patt-Shamir [12] proposed an asynchronous stabilizing algorithm for the persistent-bit problem – their solution leads to recovery in $O(k)$ time from any k-faulty configuration. A similar protocol for mutual exclusion appears in [1]. Recently, Beauquier et al investigated the 1-strong property [2] that guarantees strong spatial confinement, and time-adaptive recovery.

1.3 Organization of the Paper

This paper has seven sections. Section 2 describes the model of computation. Section 3 presents the main algorithm for fault-containment. Section 4 presents the important results and their proofs. Section 5 presents a bounded version of the unbounded solution of Section 3. The experimental results are summarized in Section 6. Section 7 contains some concluding remarks. Finally, the proof of the main theorem of Section 3 appears in the Appendix.

2 The Model of Computation

Let $G = (V, E)$ denote the topology of a distributed system, where V represents the set of processes $\{0, 1, \cdots, n-1\}$, and each edge $(i, j) \in E$ represents a bidirectional link between processes i and j. We use the notation N_i to represent the neighbors of i: thus $(i, j) \in E \Leftrightarrow j \in N_i$. Processes communicate with their immediate neighbors (also called the distance-1 neighbors) using the shared memory model. Each process i executes a program that consists of one or more guarded actions $g \rightarrow A$, where g is a predicate involving the variables of i and those of its immediate neighbors, and A is an action that updates one or more variables of i. A central demon serializes all guarded actions. The global state consists of the local states of all the processes. A *computation* of the system is a finite or infinite sequence of global states that satisfies two properties: (a) if s and s' are two consecutive states in the sequence, then there exists a process i such that i has an enabled guard in s and execution of the corresponding action results in the state s', and (b) if the sequence is finite, then in the last state of the sequence, no process has an enabled guard. We assume a randomized scheduler, where the central demon *randomly chooses* an action with an enabled guard with uniform probability. We focus on a class of systems for which actions are *reversible*, i.e. if there exists an action that changes the state from s to s', then there exists another action that changes the state from s' to s.

A stabilizing system converges to a legitimate configuration L that is ordinarily defined in terms of the observable or *primary* variables. However, in most cases, fault-containment requires the use of auxiliary or *secondary variables* too. Define the local state of each process i as an ordered pair $\langle p_i, s_i \rangle$, where p_i denotes the primary variables, and s_i denotes the secondary variables. Correspondingly, we write the global state as an ordered pair $\langle p, s \rangle$, where p is the collection all primary variables and s is the collection of all secondary variables. For a legitimate configuration L, $\langle p, s\rangle \in L \Rightarrow p \in L_p$ and $s \in L_s$. The *containment time* T_c is the time needed to establish L_p from any 1-faulty configuration, and ideally it should be $O(1)$. The *stabilization time* T_s is the maximum time needed to establish L from an arbitrary initial configuration. After the system recovers from a single fault in $O(1)$ time, and before it is ready to recover from the next single fault in $O(1)$ time, both p and s need to stabilize. The maximum time between these two events is the *fault gap*. Thus, it is the worst case time, starting from any 1-faulty state, to reach a state in L. Finally, the *contamination number* is the maximum number of processes that change the primary part of their local states during recovery from any 1-faulty configuration.

3 Probabilistic Algorithms for Fault-Containment

For the sake of exposition, we consider the *persistent-bit* protocol, in which a set of processes maintains the value of a replicated bit $v \in \{0, 1\}$ across a connected network. Thus $L \equiv \forall i, j : v(i) = v(j)$, and there are two distinct legal configurations. Using a randomized demon, the following protocol is trivially stabilizing:

Algorithm 1. Program for process i

```
**do** $\exists j \in N_i : v(j) \neq v(i) \rightarrow v(i) := v(j)$ **od**
```

We begin with the following lemma about Algorithm 1:

Lemma 1. *The persistent bit protocol is not fault-containing with a randomized demon.*

Proof. Consider a linear array of processes numbered $0, 1, \ldots, n-1$ from left to right, and assume that initially $\forall i : v(i) = 1$ holds. Let a failure of process 0 change $v(0)$ to 0. With this as the starting state, the computation can be reduced to a run of *gambler's ruin*[1]: whenever a process with $v = 0$ executes an action, the boundary between the dissimilar values of v shifts to the left, and whenever a process with $v = 1$ executes an action, the boundary moves to the right. The game is over when the system reaches L, and per [3] the expected number of moves needed is $(1 \times n - 1)$, *i.e.* $O(n)$. Thus, the protocol is *not* fault-containing. □

Note. With a deterministic demon, the protocol is not even stabilizing. (End of note)

[1] The original study is by Coolidge[3] in 1909, where he showed that if two gambler start with capitals of x and $N - x$, and each fair coin toss transfers a dollar from one to the other depending on the outcome of the toss, then the expected number of steps to finish the game is $x.(N - x)$.

To make the protocol fault-containing, we add to each process i a secondary variable $x(i)$ whose domain is the set of non-negative integers. In a way, $x(i)$ will reflect the priority of process i in executing an action to update $v(i)$. Process i will update $v(i)$, when the following three conditions hold:

1. The randomized scheduler chooses i,
2. $\exists j \in N_i : v(j) \neq v(i)$, and
3. $\forall j \in N_i : x(i) \geq x(j)$.

After updating $v(i)$, process i will increase $x(i)$ to $max\ \{x(j) : j \in N_i\} + m$, where $m > 0$. In case only the first two conditions hold, but not the third, process i will increment the value of $x(i)$ by 1, and leave $v(i)$ unchanged. Algorithm 2 shows the modified protocol:

Algorithm 2. Program for process i

```
do {action 1} $\exists j \in N_i : v(j) \neq v(i) \wedge \forall k \in N_i : x(i) \geq x(k) \rightarrow$
                    $v(i) := v(j); x(i) := max\{x(k) : k \in N_i\} + m$
$\Box$ {action 2} $\exists j \in N_i : v(j) \neq v(i) \wedge \exists\ k \in N_i : x(i) < x(k) \rightarrow$
                    $x(i) := x(i) + 1$
$\Box$ {action 3} $\forall j \in N_i : v(j) \neq v(i) \rightarrow v(i) := v(j)$
od
```

Observe that once a process i updates $v(i)$, it becomes difficult for its neighbors to change their v-values, since their x-values will lag behind that of i. The larger is the value of m, the greater is the difficulty. A neighbor j of i will be able to update $v(j)$ only if it is chosen by the random scheduler m times, without choosing i even once. On the other hand, it becomes easier for i to update $v(i)$ again in the near future.

Failures can not only corrupt v, but also corrupt x. Assume that $L \equiv \forall j : v(j) = 1$, a single failure at process i changes $v(i)$ to 0, and $x(i)$ to some unknown value. If $\forall j \in N_i : x(i) > x(j)$ then process i is likely to change its $v(i)$ soon again. As a result, the fault is contained in a small number of steps, and the contamination number is one. However, a smart adversary injecting the failure at process i is likely to set $x(i)$ to the smallest value (i.e. 0). This makes the neighbors of process i better candidates for changing their v, before process i executes a move to complete the recovery. However, it also raises the x-values of these neighbors of i above those of *their* neighbors. In order that the fault percolates to a node at distance-2 from the faulty process i, such a distance-2 node has to be chosen by the scheduler at least m times, without choosing its neighboring distance-1 node even once. With a large value of m, the probability of such an event is very low. This explains the mechanism of containment. In the mean time, the condition $\forall j \in N_i : v(j) = 1$ is likely to hold several times. If on one such occasion the faulty process is chosen by the random scheduler (the third action, note that its guard does not depend on x) then $v(i)$ will change to 1, and the recovery will be complete.

4 Results

We begin with an analysis of the spatial containment. Assume that all nodes have a degree Δ. Then the following theorem holds:

Theorem 1. *When m is large, the effect of a single failure is restricted to only the immediate neighbors of the faulty process.*

Proof. Suppose the faulty process has n_1 neighbors and the contaminated process has n_2 neighbors that are distance-2 neighbors of the faulty process. The probability that a distance-2 neighbor is contaminated is largest, when only one distance-1 process is contaminated, then only one neighbor of that contaminated distance-1 process (which is a distance-2 neighbor of the faulty process) is contaminated. The probability of one distance-1 neighbor being contaminated is $\frac{n_1}{n_1+1}$. To contaminate a distance-2 neighbor, the scheduler must select the specific process m times. So the probability of one distance-2 neighbor being contaminated is $\frac{1}{(n_1+n_2+1)^m}$. Therefore, after a node becomes faulty, the probability that some distance-2 neighbor of the faulty process becomes contaminated is $\frac{n_1}{n_1+1} \times n_2 \times \frac{1}{(n_1+n_2+1)^m}$. By choosing a large value of m, this probability can be made as small as possible. □

Theorem 2. *If $\Delta << m$ then the expected number of steps needed to contain a single fault is $O(\Delta^3)$.*

Proof. The proof appears in the Appendix.

Theorem 2 paints a pessimistic picture about the containment time when the graph is dense, *i.e.* $\Delta = O(n)$. The containment time is not $O(1)$ anymore. However, spatial containment property still holds in as much as the contamination number is between 1 and Δ w.h.p. The more dense the graph is, the smaller is the contamination number. Below, we separately analyze the extreme case of a dense topology: a completely connected graph.

Theorem 3. *For a completely connected graph, if $m >> 1$ then the contamination number is 1 with high probability.*

Proof. At least one neighbor j of the faulty process i is likely to update $v(j)$, and raise $x(j)$ at least m steps above the x-values of the rest. To prevent a second neighbor k from updating $v(k)$, the system must recover to L before the scheduler chooses the neighbor k at least m times, without choosing j even once. We use $P\{n_0, n_1, n_2, \ldots, n_i, n_{i+1}, \ldots, n_{\Delta-1}\}$ to denote the probability that $\forall i, 1 \le i \le \Delta$, node i is chosen n_i times. So the probability of node k being chosen m times before j being chosen even once is:

$$P\{n_0, n_1, n_2, \ldots, n_{j-1}, n_j = 0, n_{j+1}, \ldots, n_k = m, \ldots, n_{\Delta-1}\} \qquad (1)$$
$$\forall i, 0 \le i \le \Delta - 1 \wedge i \ne j, n_i \le m.$$

With increasing $n_i, 1 \le i \le \Delta \wedge i \ne j$ (1) is decreasing (see Lemma 2 in the Appendix for a proof). So the above probability is maximum when node k is consecutively chosen m times, and other nodes are never chosen. The maximum probability is:

$$P\{0, 0, \ldots, 0, m, 0, \ldots, 0, 0\} = \frac{1}{\Delta^m} \tag{2}$$

Since $n_i, 0 \leq i \leq \Delta - 1 \wedge i \neq j$ can be any value between 0 and m, and there are totally $m^{\Delta-2}$ possible situations, we apply the maximum estimate to each such case. As a result, the probability of the system having two contaminated processes is no larger than $\frac{m^{\Delta-2}}{\Delta^m}$, which approaches 0 as m approaches ∞. □

The mechanism will reveal that a high clustering coefficient limits the probability of contamination to only a small fraction of the distance-1 neighbors. The completely connected graph exhibits an extreme form of this property.

4.1 Computing the Availability

An interesting aspect of the proposed algorithm is that $L_s = true$, thus there is no overhead for stabilizing the secondary variables. So, L holds as soon as L_p holds. This leads to the following theorem:

Theorem 4. *For single failures, the fault gap equals the containment time.*

As a consequence of this, within an expected time of $O(\Delta^3)$ after each single failure, the system is ready to withstand the next single failure with the same efficiency. Furthermore, since only the distance-1 neighbors are contaminated with high probability, the proposed algorithm enables the system to recover from all concurrent failures of nodes that are distance-3 or more apart with the same efficiency. This significantly increases the availability of the system compared to existing solutions that we know of.

5 A Bounded Solution

A drawback of the proposed solution is that the x-variables grow in an unbounded manner, and it affects the implementability of the protocol. To address this, we will now transform the solution into one that relies on bounded variables only.

In Algorithm 2, when a process i executes action 1, it raises the value of $x(i)$ so that $\forall j \in N_i : x(j) < x(i)$ holds. This makes process i a local leader, and when the local leader is chosen by the scheduler, it immediately executes its action to update $v(i)$. Let us set an upper bound $M - 1$ for x, where M is an odd integer and $M > 1$ Furthermore, we let actions 1 and 3 increment x *mod* M. To make $x(j)$ "less than" $x(i)$, we have to define the *l*ess than operation $\prec$ appropriately. We define it as follows:

```
if $x(j) \in \{x(i) + 1 \; mod \; M, x(i) + 2 \; mod \; M, \ldots, x(i) + \frac{M-1}{2} \; mod \; M\}$
then $x(i) \prec x(j)$ else $x(j) \prec x(i)$
```

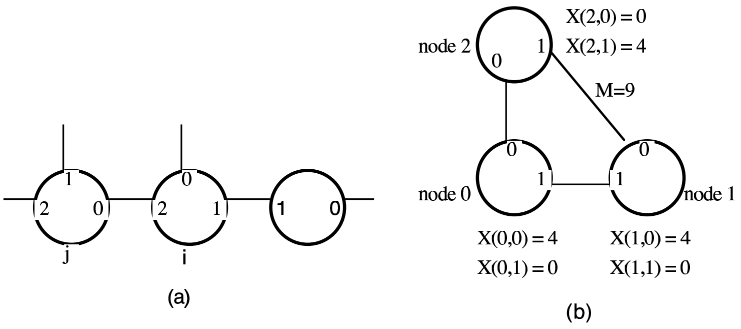


Fig. 1. (a) Identifying the ports of a node, (b) With $M = 9$, there is no local leader here

Clearly, $\prec$ is not transitive. In order that the condition $\forall j \in N_i : x(j) \prec x(i)$ holds, we will treat each x as a vector (and denote it henceforth by X) with Δ elements $0, 1, \ldots, \Delta - 1$. Let the k^{th} port of process i be connected to the l^{th} port of process j (Fig. 2(a)). We denote this by $((i,k),(j,l)) \in E$. A process i will execute action 1 when

$\exists j \in N_i : v(j) \neq v(i), \wedge$
$\forall k \in N_i : ((i,u),(k,w)) \in E,\ X(k,w) \prec X(i,u).$

We also modify the last part of action 1 as:

$\forall k \in N_i : ((i,u),(j,w)) \in E,\ X(i,u) := X(j,w) + \frac{M-1}{2}\ mod\ M$

The above modification explicitly forces the condition $X(k,w) \prec X(i,u)$ across each edge (i,k) of node i connecting to a neighbor, and establishes process i as a local leader, by setting the components of X at a maximum distance "above" those of its neighbors. Algorithm 3 shows the bounded version of Algorithm 2. The purpose of the second action is to let the non-leaders gradually "catch up" with a neighboring local leader, and is an adaptation od action 2 in Algorithm 2.

Algorithm 3. The bounded solution: Program for process i

```
do {action 1} $\exists j \in N_i : v(j) \neq v(i) \wedge$
              $\forall k \in N_i : ((i,u),(k,w)) \in E,\ X(k,w) \prec X(i,u) \rightarrow$
              $v(i) := v(j);\ \forall k \in N_i : ((i,u),(j,w)) \in E,$
              $X(i,u) := X(j,w) + \frac{M-1}{2}\ mod\ M$
□ {action 2}  $\exists j \in N_i : v(j) \neq v(i) \wedge \exists k \in N_i : ((i,u),(k,w)) \in E \wedge$
              $X(i,u) \prec X(k,w) \rightarrow X(i,u) := X(i,u) + 1\ mod\ M$
□ {action 3}  $\forall j \in N_i : v(j) \neq v(i) \rightarrow v(i) := v(j)$
od
```

Using the modified interpretation of the "less than" relation $\prec$, and by replacing m by $\frac{M-1}{2}$, Algorithm 3 becomes semantically equivalent to Algorithm 2.

However, by converting x into a vector X, there is no guarantee that there will be always be a local leader (Fig. 2(b)) ready to execute action 1. If the initial configuration is 1-faulty, and the scheduler chooses the faulty process, then this is not a concern, since action 3 does not rely on the x values at all. However, this may be an issue when the system starts from a k-faulty configuration and $k > 1$ We close our arguments by discussing the impossibility of deadlock in Algorithm 3.

Theorem 5. *Algorithm 3 guarantees that starting from any initial configuration, eventually some node is elected as a local leader.*

Proof. Assume that the system starts with a configuration where there is no local leader eligible to execute action 1. The possibility of some non-leader becoming a local leader requires that the scheduler chooses it $\frac{M-1}{2}$ times without choosing a neighbor even once. The probability of this event is $2^{-\frac{M-1}{2}}$. □

Once a local leader is elected, there always exists at least one local leader until the recovery is complete. Thus the time $2^{\frac{M-1}{2}}$ is an additional start-up cost that needs to be added to the stabilization time.

Theorem 6. *On an array of processes, the expected number of moves needed to stabilize from a failure of k contiguous nodes is $O(k.n)$.*

Proof. On an array of processes numbered 0 through $n-1$ from left to right, assume that initially $\forall i : v(i) = 1$ holds. Let a failure change the values of $v(0)$ through $v(k-1)$ $(1 < k < n)$ to 0. Whenever the node to the left of the boundary (between 0 and 1) executes move to update its v, the boundary moves one place to the left. Similarly, when the node to the right of the boundary makes a move to update its v, the boundary moves to the right. The system will stabilize when all v become identical.

The probability of the boundary moving to the right (or to the left) is $2^{-\frac{M-1}{2}}$, and the balance reflects the probability of the boundary remaining unchanged. We reduce this computation to a version of gambler's ruin where to win or lose a dollar (call it a *step*), not one, but $\frac{M-1}{2}$ consecutive heads or tails of a fair coin will be necessary[2]. It follows from [3] that, the expected number of *steps* needed to finish the game is $k \times (n-k)$. Since each *step* here costs an expected number of $2^{\frac{M-1}{2}}$ moves, the expected number of moves needed to finish the game is $2^{\frac{M-1}{2}}.k.(n-k)$. □

To validate this result on a general topology, we need to analyze gambler's ruin in multiple dimensions. This is beyond the scope of the current paper. We leave this as a conjecture. Based on the fact that a legal configuration is reachable, the following theorem is a weaker version of the result.

Theorem 7. *Algorithm 2 is stabilizing.*

[2] One extra step will be needed whenever the fortune changes from one player to the other, but with large M, we ignore the impact of this, and map the computation to a Markov process.

6 Experimental Results

We ran simulation experiments to study the containment property for graphs with various degrees, and different values of m. In the first experiment, the topology is a multidimensional torus with each node having a degree of Δ, and in each dimension there are 9 nodes (so $n = 9^{\Delta}$). The initial values of x were randomly chosen. For a given Δ, after a single fault is injected, the fraction of cases where the fault is successfully contained increases as m increases. This is consistent with the analysis – the value of m represents the effectiveness of the "fence" around the faulty zone.

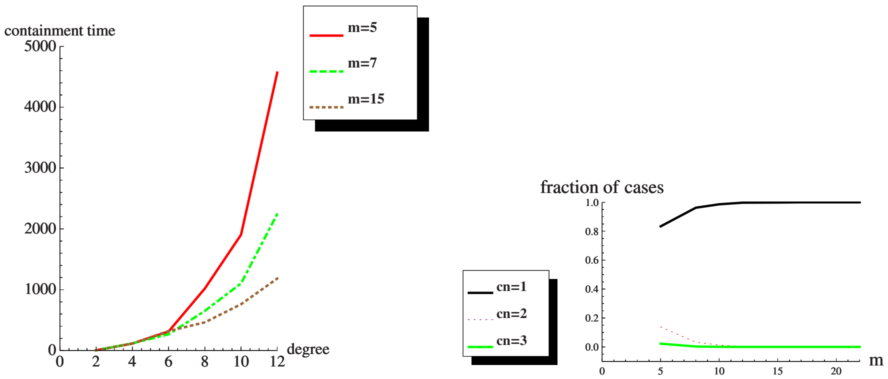


Fig. 2. (a) Stabilization time for various values of m with Δ as a parameter. (b) Fraction of nodes that are contaminated at distances 1 and 2 from the faulty node for various values of m.

Also, the *stabilization time* increases for smaller values of m, since the fault contaminates a small number of nodes beyond the distance-1 neighbors, and recovery from multiple failures is inefficient. (This explains the existence of a knee in the curves). When Δ increases (but the fault is successfully contained), for a long time, the system may be in a "stuttering mode" since neither the fault propagates beyond the distance-1 neighbors, nor the recovery is complete.

A second experiment conducted on a (30×30) grid measured the fraction of cases where the fault was not contained after 900 moves. When $m = 5$, the fault is successfully contained in 96% of the cases, and for $m = 10$ the number exceeds 99% (Fig. 2(b)).

7 Conclusion

The proposed algorithm allows the immediate neighbors of the faulty process to be contaminated. However with high probability, the failure does not propagate to the distance-2 neighbors and beyond. One can use m (or M for the bounded version of the solution) as a tuning parameter to tune the performance of the protocol. A lower value of m allows faster recovery at the expense of an increase in the contamination number, and the recovery time from multiple failures.

The major advantages of the proposed technique is that the fault-gap is independent of the network size. This increases the availability of the system by restoring the system's readiness to efficiently tolerate the next single fault within a short time. This is where our algorithm is different from other algorithms, where the fault-gap is $O(n)$ or worse.

As the degree of the nodes increases, the stabilization time increases fairly rapidly. Therefore the algorithm is suitable for sparse topologies. When the topology is dense or the clustering coefficient is large, although the stabilization time increases, a smaller fraction of the distance-1 neighbors is contaminated.

We believe that the solution to the persistent bit problem can be utilized to find fault-containing algorithms for more general problems. The task is nontrivial, and the details will be the topic for a future work.

References

1. Beauquier, J., Genolini, C., Kutten, S.: Optimal reactive k-stabilization the case of mutual exclusion. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing, pp. 209–218. ACM Press, New York (1999)
2. Beauquier, J., Delaet, S., Haddad, S.: A 1-Strong Self-Stabilizing Transformer. In: Proceedings of the Eighth Symposium on Self-Stabilizing Systems (2006)
3. Coolidge, J.L.: The Gambler's Ruin. The Annals of Mathematics 10(4), 181–192 (1909)
4. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. In: Proceedings of the Second Workshop on Self-Stabilizing Systems, pp. 3.1–3.15 (1995)
5. Ghosh, S., Gupta, A.: An exercise in fault-containment: Self- stabilizing leader election. Informat. Process. Lett. 5(59), 281–288 (1996)
6. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing, 53–73 (June 2007)
7. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault- containing self-stabilizing distributed algorithms. In: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, pp. 45–54. ACM Press, New York (1996)
8. Ghosh, S., Gupta, A., Pemmaraju, S.V.: A fault-containing self-stabilizing algorithm for spanning trees. J. Comput. Informat. 2, 322–338 (1996)
9. Ghosh, S., Gupta, A., Pemmaraju, S.V.: Fault-containing network protocols. In: Proceedings of 12th Annual ACM Symposium on Applied Computing, ACM Press, New York (1997)
10. Herman, T.: Superstabilizing mutual exclusion. In: Proceedings of 1st International Conference on Parallel and Distributed Processing: Techniques and Applications (1995)
11. Kutten, S., Peleg, D.: Fault-local distributed mending. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 20–27. ACM Press, New York (1995)
12. Kutten, S., Patt-Shamir, B.: Stabilizing time-adaptive protocols. Theor. Comput. Sci. 220, 93–111 (1999)

Appendix

Proof of Theorem 3

Assume m (or M for the bounded solution) is very large, so the error is unlikely to propagate to the distance-2 neighbors of the faulty process. In such a scenario, it is sufficient to consider the case when the error propagates to the immediate neighbors of the 1-faulty process. The state transfer diagram showing the transitions among various faulty configurations is shown in Fig. 3.

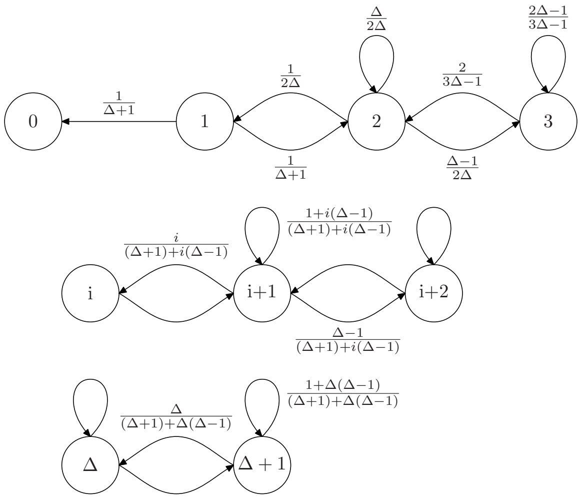

0
1
2
3
$\frac{1}{\Delta+1}$
$\frac{1}{2\Delta}$
$\frac{1}{\Delta+1}$
$\frac{\Delta}{2\Delta}$
$\frac{2}{3\Delta-1}$
$\frac{\Delta-1}{2\Delta}$
$\frac{2\Delta-1}{3\Delta-1}$
i
i+1
i+2
$\frac{i}{(\Delta+1)+i(\Delta-1)}$
$\frac{1+i(\Delta-1)}{(\Delta+1)+i(\Delta-1)}$
$\frac{\Delta-1}{(\Delta+1)+i(\Delta-1)}$
Δ
$\Delta+1$
$\frac{\Delta}{(\Delta+1)+\Delta(\Delta-1)}$
$\frac{1+\Delta(\Delta-1)}{(\Delta+1)+\Delta(\Delta-1)}$

Fig. 3. Each node is a state corresponding to the size of the faulty region, and the label on each edge represents the probability of the corresponding state transition

As the error will propagate at most to the immediate neighbors, the system can have at most Δ+1 errors. We use $p_{i,j}$ to denote the probability that the number of the errors changes from i to j. So the probabilities are listed below:

$$p_{1,0} = \frac{1}{\Delta + 1} \tag{3}$$

$$p_{2,1} = \frac{1}{2\Delta} \tag{4}$$

$$p_{2,2} = \frac{\Delta}{2\Delta} \tag{5}$$

$$p_{1,2} = \frac{1}{\Delta + 1} \tag{6}$$

and for $3 \leq i \leq \Delta - 1$

$$p_{i,i+1} = \frac{i}{(\Delta+1) + i(\Delta-1)} \tag{7}$$

$$p_{i+1,i+1} = \frac{1 + i(\Delta-1)}{(\Delta+1) + i(\Delta-1)} \tag{8}$$

$$p_{i+1,i+2} = \frac{\Delta - 1}{(\Delta+1) + i(\Delta-1)} \tag{9}$$

and

$$p_{\Delta+1,\Delta} = \frac{\Delta - 1}{(\Delta+1) + \Delta(\Delta-1)} \tag{10}$$

$$p_{\Delta+1,\Delta+1} = \frac{1 + \Delta(\Delta-1)}{(\Delta+1) + \Delta(\Delta-1)} \tag{11}$$

We use $P[X]$ to denote the probability that the system recovers using x moves. The expected number of moves needed is

$$\begin{aligned} E &= 1 \times P[1] + 2 \times P[2] + 3 \times P[3] + \cdots \\ &= \sum_{x=1}^{\infty} XP[X]. \end{aligned}$$

We can calculate $P[X]$ as following using (3), (6), (4):

$$P[1] = p_{1,0} = \frac{1}{\Delta+1} \tag{12}$$

$$P[2] = 0 \tag{13}$$

$$P[3] = p_{1,2}p_{2,1}p_{1,0} = \frac{1}{\Delta+1}\frac{1}{2\Delta}\frac{1}{\Delta+1} = \frac{1}{2\Delta(\Delta+1)^2} \tag{14}$$

$$P[4] = p_{1,2}p_{2,2}p_{2,1}p_{1,0} = \frac{1}{\Delta+1}\frac{\Delta}{2\Delta}\frac{1}{2\Delta}\frac{1}{\Delta+1} \tag{15}$$

and for $n \geq 1$, we get the following recursive function of $P[2n+2]$ using $P[2n+1]$:

$$P[2n+2] = 2\sum_{j=2}^{m} p_{j,j} P[2n+1] \tag{16}$$

If $n+1 < \Delta+1$ then m=n. If $n+1 > \Delta+1$ then $m = \Delta+1$. This is because if $n+1 < \Delta+1$, using $2n+2$ steps, the system can reach at most the $n+1$-th state. So the repeated moves can happen in any state within the n-th state. But if $n+1 > \Delta+1$, the system can move through all the states, so the repeated moves can happen anywhere within the $\Delta+1$ states.

We can also write $P[2n+3]$ using $P[2n+1]$ as:

$$P[2n+3] = 2\sum_{j=2}^{m}\sum_{i=2}^{m} p_{j,j}p_{i,i}P[2n+1] + 2\sum_{k=2}^{m'} p_{i,i+1}p_{i+1,i}P[2n+1] \qquad (17)$$

The values of m and m' are the same as in (16). The system can have either two additional moves that do not change the current state, or the system can first reach a state in one step and come back in the next step. So we substitute $P[1], P[2], P[3], P[4]$ using (12), (13), (14), (15), (16), (17) in (12) and let $p_{max} = max\{p_{i,j}, \forall i, \forall j\}$:

$$\begin{aligned}
E &= 1\times P[1] + 2\times P[2] + 3\times P[3] + 4\times P[4] \\
&\quad + \sum_{n=2}^{\infty}\{(2n+1)P[2n+1] + (2n+2)P[2n+2] + (2n+3)P[2n+3]\} \\
&= 1\times\frac{1}{\Delta+1} + 2\times 0 + 3\times\frac{1}{2\Delta(\Delta+1)^2} + 4\times\frac{1}{4\Delta(\Delta+1)^2} \\
&\quad + \sum_{n=2}^{\infty}\{(2n+1) + 2(2n+2)\sum_{j=2}^{m}\sum_{i=2}^{m} p_{i,i}p_{j,j} + (2n+3)(2\sum_{i=2}^{m'} p_{i,i+1}p_{i+1,i} \\
&\quad + 2\sum_{j=2}^{m''}\sum_{i=2}^{m''} p_{i,i}p_{j,j})\}\times P[2n+1] \\
&< O(\frac{1}{\Delta^2}) + \sum_{n=2}^{\infty}\{(2n+1)p_{max}^{2n+1} + 2(2n+2)\Delta^2 p_{max}^{2n+2} + (2n+3)(2\Delta+2\Delta^2)p_{max}^{2n+3}\} \\
&= \sum_{n=2}^{\infty}(2n+1)p_{max}^{2n+1} + 2\Delta^2\sum_{n=2}^{\infty}(2n+2)p_{max}^{2n+2} + (2\Delta+2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3}.
\end{aligned}$$

Let $T_1 = \sum_{n=2}^{\infty}(2n+1)p_{max}^{2n+1}, T_2 = 2\Delta^2\sum_{n=2}^{\infty}(2n+2)p_{max}^{2n+2}, T_3 = (2\Delta + 2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3}$. As T_3's order is the same as T_2 and larger than T_1, we just need to calculate T_3.

$$\begin{aligned}
T_3 &= (2\Delta+2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3} \\
&= (2\Delta+2\Delta^2)(2p_{max}^3\sum_{n=2}^{\infty} np_{max}^{2n} + 3p_{max}\sum_{n=2}^{\infty} p_{max}^{2n}) \\
&= (2\Delta+2\Delta^2)[2p_{max}^3(\frac{2p_{max}^4}{1-p_{max}^2} + p_{max}^6) + 3p_{max}^3\frac{1}{1-p_{max}^2}] \\
&= (2\Delta+2\Delta^2)\frac{4p_{max}^7 + 2p_{max}^3p_{max}^6(1-p_{max}^2) + 3p_{max}^3}{1-p_{max}^2}.
\end{aligned}$$

Let $p_{max} = \frac{a}{\Delta+a}, a = 1 + \Delta(\Delta - 1)$:

$$\begin{aligned} T_3 &= (2\Delta + 2\Delta^2)\frac{4a^7(\Delta+a)^2 + 2a^9 + 3a^3(\Delta+a)^6}{(\Delta^2+2\Delta a)(\Delta+a^7)} \\ &= O(\Delta^2) \times \frac{O(a^9)}{O(\Delta^3)O(a^7)} \\ &= \frac{O(a^2)}{O(\Delta)} \\ &= O(\Delta^3). \end{aligned}$$

So we get

$$E = O(\frac{1}{\Delta^2}) + O(\Delta^3) = O(\Delta^3) \tag{18}$$

□

Lemma 2. *The probability that node k is chosen m times before node j is chosen once is maximum when node k is consecutively chosen m times and other nodes are never chosen*

Proof. The probability that node k is chosen m times before node j is chosen once is:

$$\begin{aligned} P\{n_k = m, n_j = 0\} &= \sum_{i=1 \wedge i \neq j \wedge i \neq k}^{\Delta} \sum_{n_i=0}^{m} P\{n_1, n_2, \ldots, n_j = 0, \ldots, n_k = m, \ldots, n_\Delta\} \\ &= \binom{\sum_{i=1}^{\Delta} n_i}{n_1}\binom{\sum_{i=2}^{\Delta} n_i}{n_2} \cdots \binom{\sum_{i=j}^{\Delta} n_i}{0} \cdots \\ &\times \binom{\sum_{i=k}^{\Delta} n_i}{m} \cdots \binom{\sum_{i=\Delta}^{\Delta} n_i}{n_\Delta}\left(\frac{1}{\Delta}\right)^{\sum_{i=1}^{\Delta} n_i} \\ &= l \times \left(\frac{1}{\Delta}\right)^q. \end{aligned}$$

If we increase any $n_i, 1 \leq i \leq \Delta \wedge i \neq j \wedge i \neq k$ to $n_i + 1$, we can see this selection process as follows: first do the selection in the same way as before we increase n_i, then choose any one of the eligible process. There are in all $\Delta - 1$ processes that can be chosen for the last step, and the probability of choosing any one of them is $\frac{1}{\Delta}$, so the probability will be $l \times (\Delta - 1) \times \left(\frac{1}{\Delta}\right)^{q+1}$ and this is smaller than $l \times \left(\frac{1}{\Delta}\right)^q$. So the probability that node k is chosen m times before choosing node j chosen once will decrease as n_i increases. □

Self* Minimum Connected Covers of Query Regions in Sensor Networks

A.K. Datta[1], M. Gradinariu Potop-Butucaru[2], R. Patel[1], and A. Yamazaki[1]

[1] School of Computer Science, University of Nevada Las Vegas
[2] LIP6, Université Pierre et Marie Curie (Paris 6), CNRS-INRIA, France

Abstract. Sensor networks are mainly used to gather strategic information in various monitored areas. Sensors may be deployed in zones where their internal memory, or the sensors themselves, can be corrupted. Since deployed sensors cannot be easily replaced, network persistence and robustness are the two main issues that have to be addressed while efficiently deploying large scale sensor networks. The goal of forming a Minimum[1] Connected Cover of a query region in sensor networks is to select a subset of nodes that entirely covers a particular monitored area, which is strongly connected, and which does not contain a subset with the same properties.

In this paper, we consider the general case, wherein every sensor has a different sensing and communication radius. We propose two novel and robust solutions to the minimum connected cover problem that can cope with both transient faults (corruptions of the internal memory of sensors) and sensor crash/join. Also, our proposal includes extended versions which use multi-hop information. Our algorithms use small atomicity (i.e., each sensor reads variables of only one of its neighbors at a time). Our solutions are self* (self-configuration, self-stabilization, and self-healing). Via simulations, we conclude that our solutions provide better performance, in terms of coverage, than pre-existing self-stabilizing solutions. Moreover, we observe that multi-hop solutions produce a better approximation to an optimal cover set.

1 Introduction

Recent advances in technology have enabled the production of tiny networked sensors which will revolutionize information gathering and processing in both urban environments and inhospitable terrain. These wireless ad hoc sensor networks consist of a large number of tiny sensing devices with very limited resources that must coordinate amongst themselves to gather, process, and communicate information about their environments. The information to be gathered by a sensor network may need to be collected only within a particular region of a monitored area. Therefore, replies for queries should only be made by the nodes monitoring this particular area. Also, because these sensors are often densely

[1] Selecting a minimal number of connected sensors is an NP hard problem. In our work, we address the minimality in terms of inclusion.

deployed, some sensors within a network may fail or merely exhaust their energy supply. However, it may be impossible or infeasible to recharge sensors once they have been deployed, especially if they have been deployed in an inhospitable or physically unreachable terrain. Therefore, since the fundamental constraint on a networked sensor is its energy consumption, only a subset of sensors, those for which the union of their sensing regions *cover* the particular sensing region (or *query region*) should be in an active state. Our research is focused on designing a reliable, self-organizing, self-healing query-response system. To allow networked sensors to collaborate while detecting events and delivering the collected data to the sink of a query, the sensors must be able to communicate with each other, either directly or indirectly. Therefore, the sensor cover should also be strongly connected.

Related Work. In this paper, we propose novel strategies for the computation of a minimum connected cover of a query region in sensor networks.

The first strategy computes a minimum connected cover in a distributed manner using almost local (two hop neighbors) information. The second strategy is an extension of the first strategy that uses more than 2-hop information to compute a better minimum connected cover.

The problem of computing a minimum connected cover of a query region was first introduced in [9]. Two self-organizing solutions were presented in [9]. Both solutions follow a greedy strategy; however, none of the solutions is localized. The first solution is centralized — a fixed leader chooses the nodes to be part of the cover. In the second solution, a particular sensor node (which is not always the same node) behaves as the coordinator or leader. This special node collects global information in order to select the nodes to include in the final cover set.

The issues of coverage and connectivity, and the relationship between them, were analyzed in a unified framework in [13]. The CCP protocol [13] can be used to provide different degrees of coverage. It was shown in [13,17] that if the communication range is at least twice the sensing range, then complete coverage implies connectivity. When the above condition does not hold, CCP was integrated with SPAN [2] to provide both coverage and connectivity.

SPAN is a connectivity maintenance protocol in which a node volunteers to be a coordinator when it finds that two of its neighbors cannot communicate with each other directly or indirectly. To reduce the number of redundant coordinators, after a certain delay, only a single node announces its decision to be a coordinator. A similar approach was discussed in ASCENT [1]. ASCENT nodes use the number of active neighbors and message losses to decide if they should be active or passive. However, this protocol does not guarantee complete coverage of the query region.

Probabilistic studies related to coverage and connectivity in unreliable sensor networks were done in [11]. A sensor grid network of unit area was considered. This work includes a necessary and sufficient condition for network coverage and connectivity, which is based upon the probability of a node to be active (i.e., not failed) and the transmission radius of a node. Some optimal conditions for coverage were established in [17]. An algorithm for coverage was proposed based on those optimal conditions. However, that result is valid only when complete coverage implies connectivity (as discussed above). A coverage protocol in which nodes use a random delay to announce their decision to turn off was proposed in [12]. The issue of connectivity was not addressed in [12].

The GAF protocol [14] uses GPS to reduce redundant nodes when maintaining routing paths in ad-hoc networks. A randomized probing-based density control algorithm was used to maintain coverage despite node failures in the PEAS protocol [15]. In this algorithm, probing range can be changed to provide different degrees of coverage. Although these solutions are efficient in fault free environments, they are neither fault-tolerant nor self-stabilizing. Since this algorithm must be re-executed in order to repair overlay, in this scheme, every member of the network must be notified of any corruption and of the need to re-execute the algorithm.

Very recent solutions to the connected cover problem address fault-tolerance issues by reinforcing the degree of coverage and connectivity. In [18], the problem of k-coverage was addressed; the protocol ensures that any sensor is covered by k other sensors. This work is further extended in [19] to the k-coverage and k-connectivity problem. The proposed solution involves the computation of a Voronoi diagram for independent sensor nodes. However, the implementation of local Voronoi diagrams is not addressed, nor are transient faults.

To the best of our knowledge, in [4,5] we proposed the first totally decentralized, self-stabilizing, and fault-tolerant algorithms for the minimum connected covering of a query region in sensor networks. The first solution in [5], based on a greedy strategy, needs only one bit per node. However, it requests additional knowledge — the distance to the center of the query region. That is, the region is covered in successive waves from outside to inside. The coverage stops once the wave reaches the center of the monitored area. The second solution proposed in the same paper uses a pruning strategy — the elimination of redundant nodes from the final cover. Nodes were removed if their removal did not disconnect their respective neighborhoods, and if their sensing regions were completely covered by their chosen neighbors. Furthermore, in [4] we proposed another pruning-based algorithm that outperformed the solutions presented in [5]. Nodes were considered redundant if their sensing regions were covered by other chosen nodes, and if their chosen neighbors were connected through a connection path. However, this solution has a major drawback — its stabilization time is longer than the solutions presented in [5] due to the overhead introduced by the detection of communication paths between connected neighboring nodes.

Contributions. In this paper, we propose fully distributed and robust solutions to the minimum connected cover problem of query regions. Our solutions can cope with both transient faults (corruptions of the internal memory of sensors) and sensor crash/join. That is, our algorithms are self-stabilizing, fault-tolerant, and outperform the solutions presented in [4,5]. Note that we work in the general model wherein every sensor has a different sensing and communication radius. The strategy we use in our algorithms is similar to pruning used in the computation of connected dominating sets [3,8,10]. A dominating set is a set of vertices such that every vertex in the graph is either in the dominating set, or adjacent to a vertex in the dominating set. A connected dominating set is a dominating set which is also a connected subgraph. The main difference between a connected dominating set and a query region connected cover stems from the selection of dominators. In the former case, a node is a dominator of another node if the dominated node is in the transmission range of the dominator. In the latter case, a dominated node is a node that communicate with at least one dominator in its neighborhood, and whose sensing region is completely covered by dominators. Obviously, these problems are not equivalent, although a connected dominating set is also a good preliminary coverage pattern for a query region that can be extended to fully cover a query region.

Outline of the paper. In Section 2, we define the model used in this paper and specify the connected sensor cover problem. In Section 3, we present self-stabilizing solutions to the problem and their informal description. In Sections 4, we present extended versions of the algorithm. Simulation results and their discussion are included in Section 5. Due to the lack of space, the proofs of the self* aspects of our solutions, and the full simulation results are presented in the extended version of our paper [6]. Finally, in Section 6, we present some concluding remarks and propose ideas to extend this research.

2 Preliminaries and Model

Sensor Network. In this research, we consider *sensor networks* [9,13] consisting of a large number of sensors (also referred to, in this paper, as *sensor nodes* or, simply as *nodes*) which are randomly distributed in a geographical region. We model the sensor network as a *directed* communication graph $G(V, E)$, where each node in V represents a sensor, and each edge $(i, j) \in E$, called *communication edge*, indicates that j is a *neighbor* of i.

For a sensor i, there is a region, called a *sensing region*, which signifies the area in which sensor i can sense a given physical phenomenon at a desired confidence level. The sensing regions are of any convex shape. For the sake of simplicity, especially, for showing examples, the sensing regions are assumed to be circular. The *sensing range* of a sensor i indicates the maximum distance between sensor i and any point p in the sensing region of sensor i. A point p is *covered* (or monitored) by a sensor node i if the Euclidean distance between p and i is less than the sensing range of sensor i.

The *communication region* of sensor i (also called the *transmission region*) defines the area in which sensor i can communicate directly (i.e., in single hop) with other sensor nodes. The maximum distance between node i and any other node j, where j is in the communication region of i, is called the *communication range* of sensor i. Node i can communicate with node j (i.e., i can send a message to j) if the Euclidean distance between them is less than the communication range of i. Then i is called a neighbor of j, and this relation is represented by a directed edge (i, j). The set of neighbors of i is represented by N_i. Two nodes i and j can communicate directly with each other only if $i \in N_j \wedge j \in N_i$, i.e., they are neighbors of each other. If i and j are neighbors of each other, then there are two edges between them: (i, j) and (j, i).

Fault Model. This research deals with the following types of faults: (i) The state or configuration of the system may be arbitrarily corrupted. However, the program (or code) of the algorithm cannot be corrupted. (ii) Nodes may crash. That is, faults can fail-stop nodes. (iii) Nodes may recover or join the network. The topology of the network may change due to these faults. Faults may occur in any finite number, in any order, at any frequency, and at any time.

Problem Specification. In this section, we formally define the problem Connected Cover of a Query Region in sensor networks.

Definition 1 (Connected Sensor Cover). *Consider a sensor network G consisting of n sensors $I_1, I_2, \ldots, I_n$. Let S_i be the sensing region associated with sensor I_i. Given a query Q over a region R_Q in the sensor network, a set of sensors $\mathcal{S}C_Q = I_{i_1}, I_{i_2}, \ldots, I_{i_m}$ is called a* connected sensor cover *for Q if the following two conditions are satisfied:* **(a) Coverage:** $R_Q \subseteq (S_{i_1} \cup S_{i_2} \cup \ldots S_{i_m})$, *where S_{i_j} is the sensing region of sensor S_i;* **(b) Connectivity:** *The subgraph induced by $\mathcal{S}C_Q$ is strongly connected in the sense that any two sensors in this set can communicate with each other directly or indirectly.*

Definition 2 (Minimum Connected Sensor Coverage Problem). *Given a sensor network and a query over the network, the connected sensor coverage problem is to find the minimal connected sensor cover. A cover is considered minimum if it does not include another connected cover.*

Additionally, we require the algorithm (solving the above problem) to be self-organizing, self-stabilizing and self-healing [7,16]. That is, regardless of the initial state (wrong initialization of the local variables, memory or program counter corruptions) nodes <u>self-configure/self-organize</u> using only local information in order to make the system <u>self-stabilize</u> to a *legitimate state*. The legitimate state is defined with respect to a minimum connected cover formed out of the nodes that can communicate with each other either directly or indirectly. The nodes in this set are the only nodes that remain active. Moreover, under various perturbations, such as node joins, failures (due to crash or energy loss), state corruptions, or weakening of power, the minimal connected cover should be able to <u>self-heal</u> without any external intervention and the impact should be confined within a tightly bounded region around the perturbed area.

The following assumptions are made for our solutions:

Work hypothesis 1

(i) The communication radius may not be equal the sensing radius for the sensors.

(ii) The sensing radii and the communication radii of all sensors may not be equal.

(iii) There always exist a sufficient number of sensors in the network with sufficient density to cover the query region if all of them are deployed.
(iv) There exist a lot of redundant sensors which are either boundary or interior sensors with respect to the query region.

Data Structures and Notations. The data structure $Info$ has fields UID, $Status$, *Position*, *Rc*, *Rs*, S, and $MinUID$. UID represents the unique identifier (UID) of a sensor, which is a positive integer. *Status* represents the status of a sensor. The status of a sensor may be *unchosen*, *undecided*, or *chosen*. A node with the status *chosen* is part of the connected cover. *Position* represents a geometric location or coordinate of a sensor. *Rc* and *Rs* represents a communication radius and a sensing radius of a sensor, respectively, and S represents a sensing region of a sensor. Finally, *MinUID* represents the minimum UID amongst all of a sensor's neighbors' UIDs. All sensors that have the minimum UID, within a particular chosen sensor's neighborhood, are needed to ensure connectivity.

Sensor i has three shared variables: $Self\_Info_i$, N_i and N_Info_i. $Self\_Info_i$ is a data type of $Info$ which contains Sensor i's own information. N_i is a set of sensors within the communication range of Sensor i. Since we assume that each sensor has a different communication radius, we include only those sensors which have bi-directional communication with Sensor i to be within the neighbor set N_i. From this point onward, the term *neighbors* means communication neighbors and refers to only those sensors which have bi-directional communication. We use the term *sensing neighbors* to represent the sensors that are located within each others' sensing disks. *Sensing neighbors* may or may not be communication neighbors. N_Info_i is a set of δ $Info$ structures containing $Self\_Info_j$ of all sensors j in N_i.

The local variables of Sensor i are $2N\_Info_i$, which is a set of δ^2 $Info$ structures containing all 2-hop neighbors j's $Self\_Info_i$, and NN_i, which is a set of N_j for Sensor i's all neighbors j. That is, NN_i is a set of sensors located at most two hops away from Sensor i.

Macro. We introduce the macro *Read*(j) to gather sensor i's neighbors' information using small atomicity. That is, Sensor i reads only one of its neighbor's shared variables, instead of reading all neighbors' shared variables, in one atomic step.

When there is a timeout, a *timeout* action is enabled and the macro *Read*(j) reads one of Sensor i's neighbors j's variables. It reads Sensor j's $Self\_Info_j$, and includes it in its N_Info_i. If there is a duplicate, i.e., Sensor i reads Sensor j for a second time, then $Self\_Info_j$ overwrites the old data. Also, Sensor i needs to gather information from sensors located 2-hops away by reading its neighbor j's neighbor information N_Info_j. Thus, NN_i and $2N\_Info_i$, will be also updated. We assume that there is a function $F(j, NN_i)$, $j \in Ni$, that returns N_j. Then the $Next(N_i)$ function updates the pointer to the next neighbor.

3 Single Hop Self* Query Region Connected Cover

3.1 UID-Based Query Region Connected Cover

The solution we present in this section is given as Algorithm 3.1 (referred in this paper as $\mathcal{SHID}$). Although this algorithm uses 2-hop information to compute the redundant cover, messages are exchanged only within a single hop.

The steps of the algorithm are as follows:

1. The algorithm marks an *unchosen* sensor whose sensing region intersects with any portion of the query region (RQ) as *undecided*, if one of the following is true: 1) It does not have *chosen* neighbor which is also a sensing neighbor. 2) Its UID is greater than a a UID of *chosen* neighbor which is also a sensing neighbor. 3) Among the neighbors of *chosen* sensor, it has the minimum UID.
2. $MCSCNode(i)$ checks if Sensor i's status is *undecided*, and if one of the following is true: 1) It does not have any *chosen* neighbor which is also a sensing neighbor. 2) Its UID is greater than a *chosen* neighbor's UID. 3) It is the minimum UID neighbor of a *chosen* sensor. 4) A part of the sensing disk of Sensor i is not covered by a *chosen* sensor. In this case, the sensing disk of Sensor i is needed in the final cover set, so Sensor i changes its status to *chosen*.
3. $Redundant(i)$ removes any *undecided* or *chosen* sensor that has a smaller UID than a *chosen* neighbor's UID and is not a minimum UID neighbor of a *chosen* sensor, if its entire sensing disk is covered by *chosen* sensors and all *chosen* neighbors of this sensor are connected through a second path. In this case, the status of such a sensor is changed to *unchosen* (rule $\mathcal{A}_1$).
4. Rule $\mathcal{A}_1$ also ensures that any sensor whose sensing disk does not intersect with the query region has its status changed to *unchosen*.
5. All *chosen* sensors are in the final query region connected cover.

3.2 Sensing-Based Query Region Connected Cover

In the previous algorithm, the UID is used to solve the contention of sensors and remove the redundant sensors. That is, between two neighboring sensors, the one with the greater UID remains in the final cover set, and among the neighbors of a *chosen* sensor, the one with the minimum UID also remains to maintain the connectivity. In this section, we show the modification of Algorithm $\mathcal{SHID}$, which uses the sensing region instead of the UID for the above purpose, and we refer this algorithm as $\mathcal{SHRS}$(Algorithm 3.2). The idea behind this is that since every sensor has a different sensing radius, keeping the sensors which have larger coverage region results in a smaller number of sensors in the final set. Note that although the algorithm uses the sensing region instead of the UID, in the case of two or more sensors having the same sensing region, the UID is still needed as a deciding factor. The followings are necessary changes.

Algorithm 3.1 Query Region Connected Sensor Cover Algorithm for Sensor i ($\mathcal{SHID}$)

```
Constants:
    $R_Q$:: Query region;

Structure:
    $Info${
            $UID$ :: Unique user identification number
            $Status \in \{unchosen, undecided, chosen\}$ :: Status of a sensor
            $Position$ :: Geometric location or coordinate of a sensor
            $R_C$ :: Communication radius of a sensor
            $R_S$ :: Sensing radius of a sensor
            $S$ :: Sensing region of a sensor
            $MinUID$ :: minimum UID amongst all of a sensor's neighbors' UIDs
           }

Shared Variables:
    $Info\ Self\_Info_i$ :: One structure that contains information for $Sensor_i$
    $Set\ N\_Info_i$ :: Set of $\delta$ structures that contain all neighbors' information
    $Set\ N_i$ :: $\{j \in V | Dist(i,j) \le R_{C_i} \wedge Dist(i,j) \le R_{C_j}\}$

Local Variables:
    $Set\ 2N\_Info_i$ :: Set of $\delta_i + \sum \delta_{j \in N_i}$ structures that contain all 2-hop neighbors' information
    $Set\ NN_i$:: Set of $N_j$, $\forall j \in N_i$

Macro:
    $Read(j)$
            $N\_Info_i = N\_Info_i \bigcup Self\_Info_j$
            $NN_i = NN_i \bigcup N_j$
            $2N\_Info_i = 2N\_Info_i \bigcup N\_Info_j$
            $j = Next(N_i)$

Predicates:
  $QryRgnIntrsctn(i) \equiv Self\_Info_i.S \cap R_Q \ne \emptyset$;
    $\equiv$ sensing disk of Sensor $i$ intersects with some portion of query region;

  $Dist(i,j) \equiv$ Returns the Euclidean distance between Sensor $i$ and Sensor $j$;

  $SnsngNgbr(i,j) \equiv (\forall j : \mathrm{Dist(i,j)} \le min(Self\_Info_i.R_S, N\_Info_i.Self\_Info_j.R_S))$;
   $\equiv$ Sensor $i$ and Sensor $j$ are located within each others' sensing disks;

  $CvrSnsngByChsn(i) \equiv (\exists A : \forall j,k \in A, j \in N_i \wedge k \in N_j$
          $\wedge N\_Info_i.Self\_Info_j.Status = 2N\_Info_i.N\_Info_j.Self\_Info_k.Status = chosen$
          $\wedge\ Self\_Info_i.S \subset \bigcup_{j,k \in A} N\_Info_i.Self\_Info_j.S, 2N\_Info_i.N\_Info_j.Self\_Info_k.S)$;
   $\equiv$ Sensing region of Sensor $i$ is covered by a subset of $chosen$ sensors that are located
    no farther than two communication hops from Sensor $i$;

  $NeighborsConnectivity(i) \equiv$
            $(\forall j,t \in N_i, N\_Info_i.Self\_Info_j.Status = N\_Info_i.Self\_Info_t.Status = chosen,$
          $\exists k \ne i, 2N\_Info_i.N\_Info_j.Self\_Info_k.Status = chosen \wedge j,t \in N_k)$;
   $\equiv$ All $chosen$ pairs of neighbors of Sensor $i$ are connected by a $chosen$ node;

  $LstUIDNgbr(i,j) \equiv i \in N_j \wedge (Self\_Info_i.UID = N\_Info_i.Self\_Info_j.MinUID)$;
   $\equiv$ Sensor $i$ is a neighbor of Sensor $j$, and is also the neighbor of Sensor $j$ having the least UID;

  $GrtrLstOrNotNgbrOfChsn(i) \equiv (\forall j : i \in N_j, N\_Info_i.Self\_Info_j.Status \ne chosen \vee$
          $\neg SnsngNgbr(i,j) \vee Self\_Info_i.UID > N\_Info_i.Self\_Info_j.UID \vee LstUIDNgbr(i,j))$;
   $\equiv$ Sensor $i$ is not the communication neighbor, nor the sensing neighbor,
   of a $chosen$ sensor whose UID is greater than its own unless it is the "least UID"
    neighbor of this $chosen$ sensor;

  $SensorCover(i) \equiv Self\_Info_i.Status = unchosen \wedge QryRgnIntrsctn(i) \wedge GrtrLstOrNotNgbrOfChsn(i)$;
    $\equiv$ status of Sensor $i$ is $unchosen$, sensing disk of Sensor $i$ intersects with some portion of query region,
    and Sensor $i$ is not the communication neighbor, nor the sensing neighbor, of a $chosen$ sensor
    whose UID is greater than its own unless it is the "least UID" neighbor of this $chosen$ sensor;

  $MCSCNode(i) \equiv Self\_Info_i.Status = undecided$
          $\wedge(GrtrLstOrNotNgbrOfChsn(i) \vee \neg CvrSnsngByChsn(i))$;
   $\equiv$ Sensor $i$ is an $undecided$ sensor and is not the communication neighbor, nor the sensing neighbor,
   of a $chosen$ sensor whose UID is greater than its own unless it is the "least UID" neighbor of this
    $chosen$ sensor, or a part of the sensing disk of Sensor $i$ is not covered by a $chosen$ sensor;

  $Redundant(i) \equiv (Self\_Info_i.Status = undecided \vee Self\_Info_i.Status = chosen) \wedge$
          $\neg GrtrLstOrNotNgbrOfChsn(i) \wedge CvrSnsngByChsn(i) \wedge NeighborsConnectivity(i)$;
   $\equiv$ Sensor $i$ is an $undecided$ or a $chosen$ sensor and is the "lesser" communication
    and sensing neighbor of a $chosen$ sensor, but is not the neighbor of this
    sensor that has the smallest UID, the entire sensing disk of Sensor $i$ is covered
    by $chosen$ sensors, and $chosen$ neighbors of Sensor $i$ are connected through a second path

Actions:
  $\mathcal{A}_1$ :: $\neg QryRgnIntrsctn(i) \vee Redundant(i) \longrightarrow Self\_Info_i.Status = unchosen$;

  $\mathcal{A}_2$ :: $SensorCover(i) \longrightarrow Self\_Info_i.Status = undecided$;

  $\mathcal{A}_3$ :: $MCSCNode(i) \longrightarrow Self\_Info_i.Status = chosen$;

  $\mathcal{A}_4$ :: $Timeout \wedge j \in N_i \longrightarrow Read(j)$;
```

- The data structure *Info* includes the variable $MaxR_SUID$ instead of *MinUID*. It represents the UID of the sensor with the maximum R_S amongst all of a sensor's neighbors. If several sensors have the same sensing radius, then the one with the greatest UID will be selected.
- The predicate $LstUIDNgbr(i,j)$ changes to $GrtstRsNgbr(i,j)$. To maintain connectivity in Algorithm $\mathcal{SHRS}$, the predicate $GrtstRsNgbr(i,j)$ selects the sensor which has the greatest R_S among the neighbors of a chosen sensor.
- The predicate $GrtrLstOrNotNgbrOfChsn(i)$ changes to $GrtrGrtstOrNotNgbrOfChsn(i)$. Sensor i evaluates this predicate as true if one of the followings is true: 1) Sensor i does not have any *chosen* neighbor which is also a sensing neighbor. 2) Sensor i's R_S is greater than a *chosen* sensing neighbor's R_S. 3) Sensor i has the greatest R_S among the neighbors of a *chosen* sensor.

Algorithm $\mathcal{SHRS}$below shows only the modified parts while the rest remains the same.

Algorithm 3.2 Query Region Connected Sensor Cover Algorithm for Sensor i ($\mathcal{SHRS}$)

```
Changed Structure:
  Info{
        UID :: Unique user identification number
        Status ∈ {unchosen, undecided, chosen} :: Status of a sensor
        Position :: Geometric location or coordinate of a sensor
        $R_C$ :: Communication radius of a sensor
        $R_S$ :: Sensing radius of a sensor
        S :: Sensing region of a sensor
        $MaxR_SUID$ :: UID of sensor with maximum $R_S$ amongst all of a sensor's neighbors; in case of a tie,
                    sensor with greatest UID is selected
      }

Changed Predicates:
  $GrtstRsNgbr(i,j) \equiv i \in N_j \wedge (Self\_Info_i.R = N\_Info_i.Self\_Info_j.MaxR_SUID)$;
   ≡ Sensor i is a neighbor of Sensor j, and is also the neighbor of Sensor j having the greatest sensing radius;

  $GrtrGrtstOrNotNgbrOfChsn(i) \equiv (\forall j : i \in N_j, N\_Info_i.Self\_Info_j.Status \neq chosen \vee$
        $\neg SnsngNgbr(i,j) \vee Self\_Info_i.R_S > N\_Info_i.Self\_Info_j.R_S \vee GrtstRsNgbr(i,j))$;
   ≡ Sensor i is not the communication neighbor, nor the sensing neighbor, of a chosen sensor
   whose sensing radius is greater than its own unless it is the "$MaxR''_S$ neighbor of this chosen sensor;

  $SensorCover(i) \equiv Self\_Info_i.Status = unchosen \wedge QryRgnIntrsctn(i) \wedge$
        $GrtrGrtstOrNotNgbrOfChsn(i)$;
   ≡ status of Sensor i is unchosen, sensing disk of Sensor i intersects with some portion of query region,
   and Sensor i is not the communication neighbor, nor the sensing neighbor, of a chosen sensor whose sensing
   radius is greater than its own unless it is the "$MaxR''_S$ neighbor of this chosen sensor;

  $MCSCNode(i) \equiv Self\_Info_i.Status = undecided \wedge (GrtrGrtstOrNotNgbrOfChsn(i) \vee$
        $\neg CvrSnsngByChsn(i))$;
   ≡ Sensor i is an undecided sensor and is not the communication neighbor, nor the sensing neighbor,
    of a chosen sensor whose sensing radius is greater than its own unless it is the "$MaxR''_S$ neighbor of
   this chosen sensor, or a part of the sensing disk of Sensor i is not covered by a chosen sensor;

  $Redundant(i) \equiv (Self\_Info_i.Status = undecided \vee Self\_Info_i.Status = chosen) \wedge$
        $\neg GrtrGrtstOrNotNgbrOfChsn(i) \wedge CvrSnsngByChsn(i) \wedge NeighborsConnectivity(i)$;
   ≡ Sensor i is an undecided or a chosen sensor and is the communication and sensing neighbor of a chosen
   sensor that has a greater $R_S$ than its own, but is not the "$MaxR''_S$ neighbor of this sensor, the entire sensing
   disk of Sensor i is covered by chosen sensors, and chosen neighbors of Sensor i are connected through
    a second path
```

3.3 Faults and Recovery of Algorithms $\mathcal{SHID}$ and $\mathcal{SHRS}$

In this section, we focus on the fault handling features of the proposed algorithms $\mathcal{SHID}$and $\mathcal{SHRS}$. There are seven variables in the *Info* structure used in the solutions for a Sensor i: *UID*, *Status*, *Position*, R_C, R_S, S, and *MinUID* (or $MaxR_SUID$). By hypothesis only *status* can be corrupted. So, we need to show that our solutions can cope with all possible corruptions associated with this variable. In the following, we will show how they are dealt with in algorithms $\mathcal{SHID}$**(1) Wrong initialization of the *Status$_i$* variable.** All sensors, if properly initialized, start as *unchosen*. *(a) Sensor i is initialized to* ***undecided*** . Assume that Sensor i is initialized to *undecided*. If i is not a redundant node, then i remains *undecided*, and subsequently changes to *chosen*. (see Actions $\mathcal{A}_2$ and $\mathcal{A}_3$). That is, no correction is necessary. If i is redundant, then it will satisfy the predicate $Redundant(i)$ and will change to *unchosen*. *(b) Sensor i is initialized to* ***chosen*** . If the sensing disk of Sensor i does not intersect with the query region, then, by executing $\mathcal{A}_1$, Sensor i will change to *unchosen*. So, no correction is necessary. If Sensor i is redundant, then it will satisfy the predicate *Redundant*, and will change to *unchosen*. If it is non redundant then Sensor i is necessary, either to ensure coverage or connectivity, and should not be unmarked. **(2) Weakening or Failure of sensors, both in terms of communication and sensing ability.** The weakening or failure of sensors will affect their sensing and communication range. Change of R_S or R_C may change the values of $Redundant(i)$, $SensorCover(i)$, and $MCSCNode(i)$. All these changes will be reflected in the change of values of the guards of the corresponding actions. So, eventually, the status of the affected nodes will change due to the execution of these actions. These changes will affect the execution of these actions by the neighbors of the affected nodes. Therefore, any changes in the $Status_i$ variable of the affected nodes will be handled as mentioned earlier.

4 Multi-hop Self* Query Region Connected Cover

In Algorithms $\mathcal{SHID}$ and $\mathcal{SHRS}$ (Algorithms 3.1, 3.2), a sensor uses only 2-hop information to check if its entire sensing region is covered by a subset of *chosen* sensors and to check if every pair of its *chosen* neighbors has an alternate communication path. If the sensing radius is greater than the communication radius, then 2-hop information is not enough to verify this coverage condition. Since we did not assume any limitation for the sensing radius, it is possible that the sensing region of Sensor i is covered by the sensors that are located several hops away from Sensor i. Similarly, it is possible that *chosen* neighbors are connected to each other via paths of more than 2 hops. Thus, to obtain a better approximation, multi-hop information is required. Therefore, in Algorithms $\mathcal{MHID}$ and $\mathcal{MHRS}$ (the multihop versions of Algorithms $\mathcal{SHID}$and $\mathcal{SHRS}$), our goal is to further reduce the number of nodes in the final set by increasing the available information in exchange for an extra communication cost. We would like to collect upto a maximum of H-hop count information, where H is a constant.

The information required to compute the coverage condition is the UID, S, and coordinates of all *chosen* sensors within H-hops of Sensor i. Also, the hop count should be recorded. To collect this information, flooding is too expensive, and there are many redundant messages. So, we assume that there is a self-stabilizing BFS tree construction running in the background. Each *chosen* sensor maintains its own BFS tree, with height of H, rooted to itself. The information of all *chosen* sensors within H-hops from Sensor i will be passed along using these BFS trees and Sensor i receives it only from its parent sensor. Therefore, each sensor has to maintain the set of parent pointers P_i. The number of parent pointers per node is less than or equal to the number of *chosen* sensors within H-hops. This approach is studied in both ID-based and sensing-based models for breaking the network symmetry.

To gather information from *chosen* sensors located more than 2-hops away, the following change has been made in the data structure and the read action of Algorithms $\mathcal{SHID}$ and $\mathcal{SHRS}$. The data structure *Root*, which contains the root node's UID, S, *Position*, and *Hop*, was added. Also, there are extra shared variables $Root\_Info_i$ and C_i. $Root\_Info_i$ is a set of *Root* structures, and C_i is a set of *chosen* sensors within H-hop distance. A set of parent pointers is kept as a local variable. There are two separate read macros dependent upon whether or not Sensor i's neighbor j is i's parent. If j is not i's parent, then the read action is the same as that of Algorithms $\mathcal{SHID}$ and $\mathcal{SHRS}$. If j is the parent of i, when timeout occurs, i reads the root information, as well as j's information and j's neighbors' (i's 2-hop neighbors') information. After Sensor i reads $Root\_Info_j$ from j, i increments *Hop* count in $Root\_Info_i.Root_r$. If it is greater than $H - 1$, then this data is discarded, and r is removed from C_i.

This multi-hop information is applied to the predicates $CvrSnsngByChsn(i)$ and $NeighborsConnectivity(i)$. $CvrSnsngByChsn(i)$ is evaluated as true if Sensor i's sensing disk is covered by a subset of *chosen* sensors that are located no farther than H communication hops from Sensor i. For the predicate $NeighborsConnectivity(i)$ to check the connectivity of Sensor i's chosen neighbors, the new predicate $Cycle(x, y)$ is added, which is evaluated as true if there exists a cycle such that Sensors x, i, and y are vertices in the cycle, and all other vertices in this cycle are *chosen* sensors. Hence, the predicate $NeighborsConnectivity(i)$ is true when all pairs of *chosen* neighbors of Sensor i are connected by a path of *chosen* sensors located within H-hops from Sensor i.

5 Simulation and Results

Algorithms $\mathcal{SHID}$, $\mathcal{SHRS}$, $\mathcal{MHID}$ and $\mathcal{MHRS}$ compute a minimum connected sensor cover for the query region. Moreover, all proposed algorithms are fault-tolerant in terms of the self-$*$ feature (see [6] for the analytical proofs).

In our simulations, we assumed that nodes are randomly deployed on a grid of size 500×500 (350,000 nodes). Similar to [9,11,17], we considered the sensing region associated with a sensor to be a circular region centered around the sensor itself. We considered a network of 350,000 nodes in which sensors had both sensing radii and transmission radii that varied in size from 0 to 8 units. However, in some of the cases tested in our simulations, we restricted the sizes of sensors' sensing and transmission radii to be within a certain range.

The query region used in our simulations was 15 x 15 square graph units. We measured the number of sensors in the final minimum connected cover set, the number of query region sensors covered per MCSC sensor, the average number of sensors within a sensor's sensing disk, and the stabilization times for Algorithms $\mathcal{SHID}$, $\mathcal{SHRS}$, $\mathcal{MHID}$ and $\mathcal{MHRS}$, Algorithm $MCSC$ [5], and Rule k [3] [2]. We also computed an approximation ratio for each algorithm. This approximation ratio was used as a measure of optimality for each algorithm and was computed as the ratio between the number of query region sensors covered per MCSC sensor, and the average number of sensors within a sensor's sensing disk. The smaller this ratio is, the closer the final cover set chosen by a particular algorithm is to an optimal minimum connected cover set.

We used varying relative sizes of R_C and R_S for our simulations. Cases tested include $R_C \geq R_S$ (all sensors had the same size of radii of communication), $R_C \geq R_S$ (all sensors had the same size of sensing radii), $R_C \geq R_S$ (sensors had different sizes of R_C and R_S), $R_C > R_S$ (all sensors had the same size of radii of communication), $R_C > R_S$ (all sensors had the same size of sensing radii), $R_C = R_S$ (all sensors had the same size of R_C and R_S), $R_C < R_S$ (all sensors had the same size of radii of communication), $R_C < R_S$ (all sensors had the same size of sensing radii), $R_C < R_S$ (sensors had different sizes of R_C and R_S), all sensors had equal sizes of radii of communication but unequal sizes of sensing radii, all sensors had equal sizes of sensing radii but unequal sizes of radii of communication, and finally, all sensors had unequal sizes of radii of communication and unequal sizes of sensing radii.

In the following we will partially discuss the results related to the approximation ratio of the studied algorithms[3].

For the UID based algorithm, multi-hop coverage does significantly improve the algorithm's approximation ratio. Also, in most cases tested, for a R_S based algorithm, multi-hop coverage significantly improves the algorithm's approximation ratio. This implies that an algorithm that uses multi-hop coverage and connectivity versus two-hop coverage and connectivity, produces a final cover set that is closer to an optimal minimum connected cover set.

Algorithm $\mathcal{SHRS}$ or the R_S based algorithm that uses two-hop coverage and connectivity, had a better approximation ratio than the UID based algorithm that uses two-hop coverage and connectivity, when the size of R_C was less than R_S. This is due to the fact that when $R_C < R_S$, the SnsngNgbr(i,j) predicate will evaluate to true when Sensor j is a neighbor of Sensor i. Thus, $\neg$SnsngNgbr(i,j) will evaluate to false in the predicate GrtrGrtstOrNotNgbrOfChsn(i), and the sensor with the greatest R_S, within Sensor i's neighborhood, will evaluate this predicate to true and be marked as chosen by $\mathcal{A}_2$. This allows the query region to be covered with fewer nodes. However, in nearly all cases, if R_C is greater than or equal to R_S, Algorithm $\mathcal{SHRS}$ has a worse approximation ratio than Algorithm $\mathcal{SHID}$. This is due to the fact that a sensor with a sensing radius that

[2] In order to perform a fair comparison between our solutions and Rule k algorithm we have implemented the self-stabilizing version of this algorithm.

[3] Tables and detailed description of simulations are provided in [6].

is smaller than R_C can evaluate ¬SnsngNgbr(i,j) to true in GrtrGrtstOrNotNgbrOfChsn(i), even though it may have a small R_S. Thus, it can evaluate this predicate as true and change to *chosen*, even though there may be more suitable sensors (those with greater R_S') outside Sensor i's neighborhood.

As an improvement, an R_S based algorithm using multi-hop coverage (Algorithm $\mathcal{MHRS}$) produced a better cover set than all of our other algorithms when $R_C > R_S$ (all R_C equal) and produced one of the lowest approximation ratios obtained by our algorithms. This is due to the fact that the CvrSnsngByChsn(i) and NeighborsConnectivity(i) predicates have a greater chance of being evaluated to true as sensors further than 2-hops from Sensor i are considered. Subsequently, as a greater number of sensors are marked as *chosen*, a sensor, that may not have been the most suitable to be marked, may evaluate GrtrGrtstOrNotNgbrOfChsn(i) as false, evaluate Redundant(i) as true, and unmark itself. This leads to fewer nodes in the final cover set.

The approximation ratio for the multi-hop, R_S based algorithm ($R_C > R_S$ and all R_C equal) is equal to that of Algorithm $MCSC$ ($R_C = R_S$ and all R_C and R_S equal). However, Algorithm $\mathcal{MHRS}$ can also produce a cover set that is connected and that completely covers R_Q at <u>*all*</u> ranges of size of sensing and transmission radii.

Also, our Algorithm $\mathcal{MHRS}$ produces better approximation ratios than Rule k for most cases, when $R_C \geq R_S$ or $R_C > R_S$. This improvement may be attributed to the fact that a greater number of nodes were unmarked by Algorithm $\mathcal{MHRS}$'s redundancy predicate. Since the CvrSnsngByChsn(i) predicate in this algorithm considers nodes that can be located further than two hops from Sensor i, there is a greater chance that a node evaluate this predicate to true and be unmarked by Redundant(i) in Algorithm $\mathcal{MHRS}$, than a node evaluate the redundancy predicate of Rule k to true. As a result, a greater number of nodes will be unmarked by Redundant(i) in Algorithm $\mathcal{MHRS}$. Algorithm $\mathcal{MHID}$ also produces better approximation ratios than Rule k for most cases, when $R_C \geq R_S$ or $R_C > R_S$. In addition to this, when $R_C > R_S$ and all R_S are equal, Algorithm $\mathcal{SHID}$ and $\mathcal{SHRS}$ produce better approximation ratios than Rule k. In contrast to Rule k, our algorithms can also produce a cover set that is connected and that completely covers R_Q at <u>*all*</u> ranges of size of sensing and transmission radii.

Our experiments lead us to believe that the ability to produce a better approximation to an optimal cover set, combined with the ability to completely cover and produce a connected cover set for <u>*all*</u> ranges of sizes of the radius of communication and sensing radius, justify the increase in stabilization time and message complexity required for multi-hop coverage and connectivity.

6 Conclusion and Future Work

We presented two local, and two multi-hop, distributed, scalable, self-$*$ solutions to the minimum query region connected cover problem and showed how these solutions are self-organizing and self-healing as well. The algorithms are also self* contained, meaning that after a fault (transient or definitive) occurs in the system, and after stabilization, only nodes within the locality of faulty nodes change their status.

We proved the self* properties of our solutions through analytical analysis. Moreover, we have conducted extended simulations using the following measures: stabilization time, number of nodes in the final cover, the number of query region sensors covered per MCSC sensor, and the average number of sensors within a sensor's sensing disk. Due to space restrictions, the correctness proofs of our solutions and a summary of our experiments are provided in [6]. Our experiments demonstrate that the proposed algorithms perform better than the self-stabilizing algorithms proposed in [4,5,3], under certain conditions, and also produce a cover set that is connected and that completely covers the query region at all ranges of size of sensing and transmission radii.

The area of study concerned with connected coverage still raises a broad class of challenges. The generalization of this problem to the k-connected k-coverage problem has been studied in fault-free environments in [18,19]. However, an interesting open issue would be to address this problem in fault prone environments, which can be considered, in its generalized form, as the self-stabilizing, k-coverage k-connectivity problem of query regions. Another interesting research direction is the analytical evaluation of the approximation ratio and complexity of the proposed algorithms.

References

1. Cerpa, A., Estrin, D.: Ascent: Adaptive self-configuring sensor networks topologies. In: INFOCOM 2002. Proceedings of the Conference on Computer Communications (June 2002)
2. Chen, B., Jamieson, K., Balakrishnan, H., Morris, R.: Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In: MobiCom 2002. Proceedings of the Seventh Annual Inernational Conference on Mobile Computing and Networking, pp. 85–96 (July 2001)
3. Dai, F., Wu, J.: Distributed dominant pruning in ad hoc networks. In:Proceedings of ICC 2003 (2003)
4. Datta, A.K., Gradinariu, M., Patel, R.: Distributed self-$*$ minimum connected covering of a query region in sensor networks. In: ISPAN 2005, pp. 448–453 (2005)
5. Datta, A.K., Linga, P., Gradinariu, M., Raipin-Parvedy, P.: Self-* distributed query region covering in sensor networks. In: SRDS 2005. 24th IEEE Symposium on Reliable Distributed Systems, pp. 50–59 (October 2005)
6. Datta, A.K., Potop-Butucaru, M.G., Patel, R., Yamazaki, A.: Self* minimum connected covers of sensing regions in sensor networks. Technical report, INRIA, France (2007)
7. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
8. Drabkin, V., Friedman, R., Gradinariu, M.: Self-stabilizing wireless connected overlays. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 425–439. Springer, Heidelberg (2006)
9. Gupta, H., Das, S.R., Gu, Q.: Connected sensor cover: Self-organization of sensor networks for efficient query execution. In: MobiHoc 2003. Proceedings of the Fourth ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 189–200 (2003)

10. Jiang, Z., Kline, R., Wu, J., Dai, F.: A practical method to form energy efficient connected -coverage in wireless sensor networks. In: ICDCS Workshops, p. 81 (2006)
11. Shakkottai, S., Srikant, R., Shroff, N.: Unreliable sensor grids: Coverage, connectivity and diameter. In: INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 2, pp. 1073–1083 (April 2003)
12. Tian, D., Georganas, N.D.: A coverage-preserving node scheduling scheme for large wireless sensor networks. In: WSNA 2002. Proceedings of the First Workshop on Sensor Networks and Applications, pp. 32–41 (September 2002)
13. Wang, X., Xing, G., Zhang, Y., Lu, C., Pless, R., Gill, C.: Integrated coverage and connectivity configuration in wireless sensor networks. In: ACM SenSys 2003. Proceedings of the First International Conference on Embedded Networked Sensor Systems, pp. 28–39 (November 2003)
14. Xu, Y., Heidemann, J., Estrin, D.: Geography-informed energy conservation for ad hoc routing. In: MobiCom 2002. Proceedings of the Seventh Annual Inernational Conference on Mobile Computing and Networking, pp. 70–84 (2001)
15. Ye, F., Zhong, G., Cheng, J., Lu, S., Zhang, L.: PEAS: A robust energy conserving protocol for long-lived sensor networks. In: ICDCS 2003. Proceedings of the 23rd International Conference on Distributed Computing Systems, pp. 1–10 (2003)
16. Zhang, H., Arora, A.: GS3: Scalable self-configuring and self-healing in wireless networks. In: PODC 2002. Proceedings of the Twentyfirst Annual ACM Symposium on Principles of Distributed Computing (2002)
17. Zhang, H., Hou, J.C.: Maintaining sensing coverage and connectivity in large sensor networks. Technical Report UIUCDCS-R-2003-2351, University of Illinois at Urbana Champaign, (June 2003)
18. Zhou, Z., Das, S.R., Gupta, H.: Connected k-coverage problem in sensor networks. In: ICCCN, pp. 373–378 (2004)
19. Zhou, Z., Das, S.R., Gupta, H.: Fault tolerant connected sensor cover with variable sensing and transmission ranges. In: SECON (2005)

Robust Stabilizing Leader Election*

Carole Delporte-Gallet[1], Stéphane Devismes[2], and Hugues Fauconnier[1]

[1] LIAFA, Université D. Diderot (France)
{cd,hf}@liafa.jussieu.fr
[2] LaRIA, Université de Picardie Jules Verne (France)
stephane.devismes@u-picardie.fr

Abstract. We mix two approaches of the fault-tolerance: *robustness* and *stabilization*. Using these approaches, we propose leader election algorithms that tolerate both *transient* and *crash failures*. Our goal is to show the implementability of the robust self- and/or pseudo- stabilizing leader election in various systems with weak reliability and synchrony assumptions. We try to propose, when it is possible, *communication-efficient* implementations. Also, we exhibit some assumptions required to obtain robust stabilizing leader election algorithms. Our results show that the gap between robustness and stabilizing robustness is not really significant when we consider fix-point problems such as leader election.

1 Introduction

Two kinds of faults are usually considered: the *transient* and *crash* failures. The *stabilization* introduced by Dijkstra in 1974 [2] is a general technique to design algorithms tolerating transient failures. However, such stabilizing algorithms are usually not *robust*: they do not withstand crash failures. Conversely, *robust* algorithms are usually not designed to go through transient failures (*n.b.*, some robust algorithms tolerate the loss of messages, *e.g.*, [3]). There is some papers that deal with both stabilization and crash failures, *e.g.*, [4,5,6,7]. In [4], Gopal and Perry propose techniques for transforming robust protocols in a synchronous network into robust self-stabilizing versions. Beauquier and Kekkonen-Moneta introduce in [6] the first self-stabilizing failure detector implementation in a synchronous system. In [5], authors prove that robust self-stabilization cannot be achieved in asynchronous networks for a wide range of problems including leader election even when self-stabilization or robustness alone can be done.

We are interested in designing leader election algorithms that tolerate transient and crash failures. Actually, we focus on finding stabilizing solutions in message-passing with the possibility of some process crashes. The impossibility results in [8,5] constraints us to make some assumptions on the link and process synchrony. So, we look for the weakest assumptions allowing to obtain stabilizing leader election algorithm in a system where some processes may crash.

Leader election has been extensively studied in both stabilizing (*e.g.*, [9,10]) and robust (*e.g.*, [11,12]) areas. In particular, note that in the robust systems,

* The full version of this paper in available on the HAL server, see [1].

leader election is also considered as a failure detector. Such a failure detector, called Ω, is important because it has been shown in [13] that it is the weakest failure detector with which one can solve the consensus.

The notion of stabilization appears with the concept of *self-stabilization*: a *self-stabilizing algorithm*, regardless of the initial configuration of the system, guarantees that the system reaches in a finite time a configuration from which it *cannot* deviate from its intended behavior. In [14], Burns *et al* introduced the more general notion of *pseudo-stabilization*. A pseudo-stabilizing algorithm, regardless of the initial configuration of the system, guarantees that the system reaches in a finite time a configuration from which it *does not* deviate from its intended behavior. These two notions guarantee the *convergence* to a correct behavior. However, the self-stabilization also guarantees that since the system recovers a *legitimate* configuration, it remains in a *legitimate* configuration forever: the *closure* property. In contrast, a pseudo-stabilizing algorithm only guarantees an *ultimate closure*: the system can move from a *legitimate* configuration to an *illegitimate* one but eventually it remains in a *legitimate* configuration forever.

We study the problem of implementing robust self- and/or pseudo- stabilizing leader election in various systems with weak reliability and synchrony assumptions. We try to propose, when it is possible, *communication-efficient* implementations: an algorithm is *communication-efficient* if it eventually only uses $n-1$ unidirectionnal links (where n is the number of processes), which is optimal [15]. Communication-efficiency is quite challenging in the stabilizing area because stabilizing implementations often require the use of heartbeats which are heavy in terms of communication. In this paper, we first show that the notions of immediate synchrony and eventually synchrony are in some sense equivalent concerning the stabilization. Hence, we only consider synchrony properties that are immediate. In the systems we study: (1) all the processes are synchronous and can communicate with each other but some of them may crash and, (2) some links may have some synchrony or reliability properties. Our starting point is a full synchronous system noted $\mathcal{S}_5$. We show that a self-stabilizing leader election can be communication-efficiently done in such a system. We then show that such strong synchrony assumptions are required in the systems we consider to obtain a self-stabilizing communication-efficient leader election. Nevertheless, we also show that a self-stabilizing leader election that is not communication-efficient can be obtained in some weaker systems: any system where there exists at least one path of synchronous links between each pair of alive processes ($\mathcal{S}_3$) and any system having a *timely bi-source*[1] ($\mathcal{S}_4$). In addition, we show that we cannot implement any self-stabilizing leader election without these assumptions. Hence, we then consider the pseudo-stabilization. We show that communication-efficient pseudo-stabilizing leader election can be done in various weak models: any system $\mathcal{S}_4$ and any system having a *timely source*[2] and *fair* links ($\mathcal{S}_2$). Using a previous result of Aguilera *et al* ([3]), we recall that communication-efficiency

[1] A timely bi-source is a synchronous process having all its links that are synchronous.

[2] A timely source is a synchronous process having all its output links that are synchronous.

Table 1. Implementability of the robust stabilizing leader election

	$\mathcal{S}_5$	$\mathcal{S}_4$	$\mathcal{S}_3$	$\mathcal{S}_2$	$\mathcal{S}_1$	$\mathcal{S}_0$
Communication-Efficient Self-Stabilization	Yes	No	No	No	No	No
Self-Stabilization	Yes	Yes	Yes	No	No	No
Communication-Efficient Pseudo-Stabilization	Yes	Yes	?	Yes	No	No
Pseudo-Stabilization	Yes	Yes	Yes	Yes	Yes	No

cannot be done if we consider now systems having at least *one timely source* but where the *fairness* of all the links is not required ($\mathcal{S}_1$). However, we show that a non-communication-efficient pseudo-stabilizing solution can be implemented in such systems. Finally, we conclude with the basic system where all links can be asynchronous and lossy ($\mathcal{S}_0$): the leader election can be neither pseudo- nor self-stabilized in such a system ([8,5]). Table 1 summarizes our results.

It is important to note that the solutions we propose are essentially adapted from previous existing robust algorithms provided, in particular, in [11,3]. Actually, the motivation of the paper is not to propose new algorithms. Our goal is merely to show some required assumptions to obtain self- or pseudo- stabilizing leader election algorithms in systems where some processes may crash. In particular, we focus on the borderline assumptions where we go from the possibility to have self-stabilization to the possibility to have pseudo-stabilization only. Another interesting aspect of adaptating previous existing robust algorithms is to show that, for fix-point problems[3] such as leader election, the gap between robustness and stabilizing robustness is not really significant: in such problems, adding the stabilizing property is quite easy.

Roadmap. In the next section, we present the model for our systems. We then consider the problem of the robust stabilizing leader election in various kinds of systems (Sections 3 to 8). We conclude with the future works in Section 9.

2 Preliminaries

2.1 Distributed Systems

We consider *distributed systems* where each process can communicate with each other through *directed links*: there is a directed link from each process to all the others. We denote the *communication network* by the digraph $G = (V, E)$ where $V = \{1,...,n\}$ is the set of n processes ($n > 1$) and E the set of directed links. A collection of distributed *algorithms* run on the system. These algorithms can be seen as automata that enable processes to coordinate their activities. We modelize the *executions* of a distributed *algorithm* $\mathcal{A}$ in the system $\mathcal{S}$ by the pair $(\mathcal{C},\mapsto)$ where $\mathcal{C}$ is the set of configurations and $\mapsto$ is a collection of binary transition relations on $\mathcal{C}$ such that for each transition $\gamma_{i-1} \mapsto \gamma_i$ we have $\gamma_{i-1} \neq \gamma_i$. A configuration consists in the state of each process and the collection of messages

[3] Roughly speaking, a problem is a fix-point problem if the problem can be expressed by some invariant properties of some variables.

in transit at a given time. The state of a process is defined by the values of its variables. An *execution* of $\mathcal{A}$ is a *maximal* sequence $\gamma_0,\tau_0,\gamma_1,\tau_1,\ldots,\gamma_{i-1},\tau_{i-1},\gamma_i,\ldots$ such that $\forall i \geq 1$, $\gamma_{i-1} \mapsto \gamma_i$ and the transition $\gamma_{i-1} \mapsto \gamma_i$ occurs after time elapse τ_{i-1} time units ($\tau_{i-1} \in \mathbb{R}^+$). For each configuration γ in any execution e, we denote by $\overrightarrow{e_\gamma}$ the suffix of e starting in γ, $\overleftarrow{e_\gamma}$ denotes the associated prefix. We call *specification* a particular set of executions.

2.2 Self- and Pseudo- Stabilization

Definition 1 (Self-Stabilization [2]). *An algorithm $\mathcal{A}$ is* self-stabilizing *for a specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if in any execution of $\mathcal{A}$ in $\mathcal{S}$, there exists a configuration γ such that any suffix starting from γ is in $\mathcal{F}$.*

Definition 2 (Pseudo-Stabilization [14]). *An algorithm $\mathcal{A}$ is* pseudo-stabilizing *for a specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if in any execution of $\mathcal{A}$ in $\mathcal{S}$, there exists a suffix that is in $\mathcal{F}$.*

Robust Stabilization. Here, we not only consider the transient failures: our systems may go through transient and crash failures. We assume that some processes may be crashed in the initial configuration. We also assume that the links are not necessary reliable during the execution. In the following, we will show that despite these constraints, it is possible (under some assumptions) to design stabilizing algorithms. Note that the fact that we only consider initial crashes is not a restriction (but rather an assumption to simplify the proofs) because we focus on the leader election which is a fix-point problem: in such problems, the safety properties do not concern the whole execution but only a suffix.

2.3 Informal Model

Processes. Processes execute by taking steps. In a step a process executes two actions in sequence: (1) either it tries to receive one message from another process, or sends a message to another process, or does nothing, and then (2) changes its state. A step need not to be instantaneous, but we assume that each action of a step takes effect at some instantaneous moment during the step. The configuration of the system changes each time some steps take effect: if there is some steps that take effect at time t_i, then the system moves from a configuration γ_{i-1} to another configuration γ_i ($\gamma_{i-1} \mapsto \gamma_i$) where γ_{i-1} was the configuration of the system during some time interval $[t_{i-1}, t_i[$ and γ_i is the configuration obtained by applying on γ_{i-1} all actions of the steps that take effect at time t_i.

A process can fail by permanently crashing, in which case it definitively stops to take steps. A process is *alive at time t* if it is not crashed at time t. Here, we consider that all processes that are alive in the initial configuration are alive forever. An alive process executes infinitely many steps. We consider that any subset of processes may be crashed in the initial configuration.

We assume that the execution rate of any process cannot increase indefinitively: there exists a non-null lower bound on the time required by the alive processes to execute a step[4]. Also, every alive process is assumed to be *timely*: it satisfies a non-null upper bound on the time it requires to execute each step. Finally, our algorithms are structured as a *repeat forever* loop with a bounded number of steps in each loop iteration. So, each alive process satisfies a lower and an upper bound, resp. noted α and β, on the time it needs to execute an iteration of its *repeat forever* loop. We assume that each process knows α and β.

Links. Processes can send messages over a set of directed links. There is a directed link from each process to all the others. A message m carries a *type* T in addition to its *data* D: $m = (T,D) \in \{0,1\}^* \times \{0,1\}^*$. For each incoming link (q,p) and each type T, the process p has a message buffer, $\texttt{Buffer}_p[q,T]$, that can hold at most one *single* message of type T. $\texttt{Buffer}_p[q,T] =\perp$ when it holds no message. If q sends a message m to p and the link (q,p) does not lose m, then $\texttt{Buffer}_p[q,T]$ is eventually set to m. When it happens, we say that *message m is delivered to p from q* (*n.b.*, we make no assumption on the delivrance order). If $\texttt{Buffer}_p[q,T]$ was set to some previous message, this message is then overwritten. When p takes a step, it may choose a process q and a type T to read the contents of $\texttt{Buffer}_p[q,T]$. If $\texttt{Buffer}_p[q,T]$ contains a message m (*i.e.*, $\texttt{Buffer}_p[q,T] \neq\perp$), then we say that *$p$ receives m from q* and $\texttt{Buffer}_p[q,T]$ is reset to $\perp$.

A link (p,q) is *timely* if there exists a constant δ such that, for every execution and every time t, each message m sent to q by p at time t is delivered to q from p within time $t + \delta$ (any message that is initially in a timely link is delivered within time δ). A link (p,q) is *eventually timely* if there exists a constant δ for which every execution satisfies: there is a time t such that every message m that p sends to q at time $t' \geq t$ is delivered to q from p by time $t' + \delta$ (any message that is already in an eventually timely link at time t is delivered within time $t + \delta$). We assume that every process knows δ. We also assume that $\delta > \beta$. A link which is neither timely nor eventually timely can be arbitrary slow, or can lose messages. A *fair* link (p,q) satisfies: for each type of message T, if p sends infinitely many messages of type T to q, then infinitely many messages of type T are delivered to q from p. A link (p,q) is *reliable* if every message sent by p to q is eventually delivered to q from p.

Particular Caracteristics. A *timely source* (resp. an *eventually timely source*) [3] is an alive process having all its *output links* that are *timely* (resp. *eventually timely*). A *timely bi-source* (resp. an *eventually timely bi-source*) [16] is an alive process having all its (input and output) *links* that are *timely* (resp. *eventually timely*). We call *timely routing overlay* (resp. *eventually timely routing overlay*) any strongly connected graph $G' = (V',E')$ where V' is the subset of all alive processes and E' a subset of *timely* (resp. *eventually timely*) links.

Finally, note that the notions of *timeliness* and *eventually timeliness* are "equivalent" in (pseudo- or self-) stabilization in a sense that every stabilizing algorithm in a system $\mathcal{S}$ having some timely links is also stabilizing in the system $\mathcal{S}'$ where $\mathcal{S}'$ is the same system as $\mathcal{S}$ except that all the timely links in $\mathcal{S}$ are eventually timely in $\mathcal{S}'$, and reciprocally (see Theorems 1 and 2).

[4] Except for the first step that we allow to not satisfy this lower bound.

Theorem 1. *Let $\mathcal{S}$ be a system having some timely links. Let $\mathcal{S}'$ be the same system as $\mathcal{S}$ except that all the timely links in $\mathcal{S}$ are eventually timely in $\mathcal{S}'$. An algorithm $\mathcal{A}$ is pseudo-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if $\mathcal{A}$ is pseudo-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}'$.*

Proof. By definition, a timely link is also an eventually timely link. Hence, we trivially have: if $\mathcal{A}$ is pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}'$, then $\mathcal{A}$ is also pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}$.

Assume now that $\mathcal{A}$ is pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}$ but not pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}'$. Then, there exists an execution e of $\mathcal{A}$ in $\mathcal{S}'$ such that no suffix of e is in $\mathcal{F}$. Let γ be the configuration of e from which all the eventually timely links of $\mathcal{S}'$ are timely. As no suffix of e is in $\mathcal{F}$, no suffix of $\overrightarrow{e_\gamma}$ is in $\mathcal{F}$ too. Now, $\overrightarrow{e_\gamma}$ is a possible execution of $\mathcal{A}$ in $\mathcal{S}$ because (1) γ is a possible initial configuration of $\mathcal{S}$ ($\mathcal{S}$ and $\mathcal{S}'$ have the same set of configurations and any configuration can be initial) and (2) every eventually timely link of $\mathcal{S}'$ is timely in $\overrightarrow{e_\gamma}$. Hence, as no suffix of $\overrightarrow{e_\gamma}$ is in $\mathcal{F}$, $\mathcal{A}$ is not pseudo-stabilizing for $\mathcal{F}$ in $\mathcal{S}$ — a contradiction. □

Following a proof similar to the one of Theorem 1, we have:

Theorem 2. *Let $\mathcal{S}$ be a system having some timely links. Let $\mathcal{S}'$ be the same system as $\mathcal{S}$ except that all the timely links in $\mathcal{S}$ are eventually timely in $\mathcal{S}'$. An algorithm $\mathcal{A}$ is self-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}$ if and only if $\mathcal{A}$ is self-stabilizing for the specification $\mathcal{F}$ in the system $\mathcal{S}'$.*

Communication-Efficiency. An algorithm is *communication-efficient* [11] if there is a time from which it uses only $n-1$ unidirectional links.

Systems. We consider six systems denoted by $\mathcal{S}_i$, $i \in [0...5]$ (see Figure 1). All these systems satisfy: (1) the value of the variables of every alive process can be arbitrary in the initial configuration, (2) every link can initially contain a finite but unbounded number of messages, and (3) except if we explicitly state, each link between two alive processes is neither fair nor timely (we just assume that the messages cannot be corrupted). The system $\mathcal{S}_0$ corresponds to the basic system where no further assumptions are made: in $\mathcal{S}_0$, the links can be arbitrary slow or lossy. In $\mathcal{S}_1$, we assume that there exists at least one timely source (whose identity is unknown). In $\mathcal{S}_2$, we assume that there exists at least one timely source (whose identity is unknown) and every link is fair. In $\mathcal{S}_3$, we assume that there exists a timely routing overlay. In $\mathcal{S}_4$, we assume that there exists at least one timely bi-source (whose identity is unknown). In $\mathcal{S}_5$, all links are timely.

2.4 Robust Stabilizing Leader Election

In the leader election, each process p has a variable $Leader_p$ that holds the identity of a process. Intuitively, eventually all alive processes should hold the identity of the same process forever and this process should be alive. Formally, there exists an alive process l and a time t such that at any time $t' \geq t$, every alive process p satisfies $Leader_p = l$.

System	Properties
$\mathcal{S}_0$	*Links*: arbitrary slow, lossy, and initially not necessary empty *Processes*: can be initially crashed, timely forever otherwise *Variables*: initially arbitrary assigned
$\mathcal{S}_1$	$\mathcal{S}_0$ with at least one *timely source*
$\mathcal{S}_2$	$\mathcal{S}_0$ with at least one *timely source* and every link is *fair*
$\mathcal{S}_3$	$\mathcal{S}_0$ with a *timely routing overlay*
$\mathcal{S}_4$	$\mathcal{S}_0$ with at least one *timely bi-source*
$\mathcal{S}_5$	$\mathcal{S}_0$ except that all links are *timely*

$$\begin{tikzcd}
 & S_0 \arrow[dl] \arrow[dr] & \\
S_1 \arrow[d] \arrow[drr] & & S_3 \arrow[d] \\
S_2 \arrow[dr] & & S_4 \arrow[dl] \\
 & S_5 &
\end{tikzcd}$$

Fig. 1. Systems considered in this paper ($\mathcal{S} \to \mathcal{S}'$ means $\mathcal{S}' \subset \mathcal{S}$)

3 Communication-Efficient Self-Stabilizing Leader Election in $\mathcal{S}_5$

We first seek a communication-efficient self-stabilizing leader election algorithm in system $\mathcal{S}_5$. To get the communication-efficiency, we proceed as follows: Each process p periodically sends ALIVE to all other processes *only if it thinks to be the leader*, *i.e.*, *only if* $Leader_p = p$ (Lines 16-18 of Algorithm 1).

Any process p such that $Leader_p \neq p$ always chooses as leader the process from which it receives ALIVE the most recently (Lines 6-13). When a process p such that $Leader_p = p$ receives ALIVE from q, it sets $Leader_p$ to q if $q < p$ (Lines 6-13). By this method, there eventually exists at most one alive process p such that $Leader_p = p$.

Finally, every process p such that $Leader_p \neq p$ uses a *counter* that is incremented at each loop iteration to detect if there is no alive process q such that $Leader_q = q$ (Lines 21-27). When the counter becomes greater than a well-chosen value, p can deduce that there is no alive process q such that $Leader_q = q$. In this case, p simply elects itself by setting $Leader_p$ to p (Line 24) in order to guarantee the liveness of the election: in order to ensure that there eventually exists at least one process q such that $Leader_q = q$.

To apply the previously described method, Algorithm 1 uses only one message type: ALIVE and two *counters*: $SendTimer_p$ and $ReceiveTimer_p$. Any process p such that $Leader_p = p$ uses the counter $SendTimer_p$ to periodically send ALIVE to the other processes. $ReceiveTimer_p$ is used by each process p to detect when there is no alive process q such that $Leader_q = q$. These counters are incremented at each iteration of the *repeat forever* loop in order to evaluate a particular time elapse. Using the lower and upper bound on the time to execute an iteration of this loop (*i.e.*, α and β), each process p knows how many iterations it must execute before a given time elapse passed. For instance, a process p must count $\lceil \delta/\alpha \rceil$ loop iterations to wait at least δ times.

Theorem 3. *Algorithm 1 implements a communication-efficient self-stabilizing leader election in system* $\mathcal{S}_5$.

Algorithm 1. Communication-Efficient Self-Stabilizing Leader Election on $\mathcal{S}_5$

```
CODE FOR EACH PROCESS $p$:
variables:
    $Leader_p \in \{1,\ldots,n\}$
    $SendTimer_p$, $ReceiveTimer_p$: non-negative integers

repeat forever
    for all $q \in V \setminus \{p\}$ do
        if receive(ALIVE) from $q$ then
            if $(Leader_p \neq p) \vee (q < p)$ then
                $Leader_p \leftarrow q$
            end if
            $ReceiveTimer_p \leftarrow 0$
        end if
    end for
    $SendTimer_p \leftarrow SendTimer_p + 1$
    if $SendTimer_p \geq \lfloor \delta/\beta \rfloor$ then
        if $Leader_p = p$ then
            send(ALIVE) to every process except $p$
        end if
        $SendTimer_p \leftarrow 0$
    end if
    $ReceiveTimer_p \leftarrow ReceiveTimer_p + 1$
    if $ReceiveTimer_p > 8\lceil \delta/\alpha \rceil$ then
        if $Leader_p \neq p$ then
            $Leader_p \leftarrow p$
        end if
        $ReceiveTimer_p \leftarrow 0$
    end if
end repeat
```

4 Self-Stabilizing Leader Election in $\mathcal{S}_4$

We first prove that we cannot implement any communication-efficient self-stabilizing leader election algorithm in $\mathcal{S}_2$ and $\mathcal{S}_4$. To that goal, we show that it is impossible to implement such an algorithm in a stronger system: $\mathcal{S}_5^-$ where $\mathcal{S}_5^-$ is any system $\mathcal{S}_0$ having (1) all its links that are reliable and (2) all its links that are timely except at most one which can be neither timely nor eventually timely.

Lemma 1. *Let $\mathcal{A}$ be any self-stabilizing leader election algorithm in $\mathcal{S}_5^-$. In any execution of $\mathcal{A}$, any alive process p satisfies: from any configuration where $Leader_p \neq p$, $\exists k \in \mathbb{R}^+$ such that p modifies $Leader_p$ if it receives no message during k times.*

Proof. Assume, by the contradiction, that there exists an execution e where there is a configuration γ from which a process p satisfies $Leader_p = q$ forever with $q \neq p$ while p does not receive a message anymore. As $\mathcal{A}$ is self-stabilizing, it can start from any configuration. So, $\overrightarrow{e_\gamma}$ is a possible execution. Let γ' be a configuration which is identical to γ except that q is crashed in γ'. Consider any execution $e_{\gamma'}$ starting from γ' where p did not receive a message anymore. As p cannot distinguish $\overrightarrow{e_\gamma}$ and $e_{\gamma'}$, it behaves in $e_{\gamma'}$ as in $\overrightarrow{e_\gamma}$: it keeps q as leader while q is crashed — a contradiction. □

Theorem 4. *There is no communication-efficient self-stabilizing leader election algorithm in system $\mathcal{S}_5^-$.*

Proof. Assume, by the contradiction, that there exists a communication-efficient self-stabilizing leader election algorithm $\mathcal{A}$ in system $\mathcal{S}_5^-$.

Consider any execution e where no process crashes and all the links behave as timely. By Definition 1 and Lemma 1, there exists a configuration γ in e such that in any suffix starting from γ: (1) there exists an alive process l such that any alive process p satisfies $Leader_p = l$ forever, and (2) messages are received infinitely often through at least one input link of each alive process except perhaps l.

Communication-efficiency and (2) implies that messages are received infinitely often in $\overrightarrow{e_\gamma}$ through exactly $n-1$ links of the form (q,p) with $p \neq l$. Let $E' \subset E$ be the subset containing the $n-1$ links where messages transit infinitely often in $\overrightarrow{e_\gamma}$.

Consider now any execution e' identical to e except that there is a time after which a certain link $(q,p) \in E'$ arbitrary delays the messages. (q,p) can behave as a timely link an arbitrary long time, so, e and e' can have an arbitrary large common prefix. In particular, e' can begin with any prefix of e of the form $\overleftarrow{e_\gamma}e''$ with e'' a non-empty prefix of $\overrightarrow{e_\gamma}$. Now, after any prefix $\overleftarrow{e_\gamma}e''$, (q,p) can start to arbitrary delay the messages and, in this case, p eventually changes its leader by Lemma 1. Hence, for any prefix $\overleftarrow{e_\gamma}e''$, there is a possible suffix of execution in $\mathcal{S}_5^-$ where p changes its leader: for some executions of $\mathcal{A}$ in $\mathcal{S}_5^-$ there is no guarantee that from a certain configuration the leader does not change anymore. Hence, $\mathcal{A}$ is not self-stabilizing in $\mathcal{S}_5^-$ — a contradiction. □

By definition, any system $\mathcal{S}_5^-$ is also a system $\mathcal{S}_2$ and any system $\mathcal{S}_5^-$ having $n \geq 3$ processes is a particular case of system $\mathcal{S}_4$. Hence:

Corollary 1. *There is no communication-efficient self-stabilizing leader election algorithm in systems $\mathcal{S}_2$ and $\mathcal{S}_4$ with $n \geq 3$ processes.*

Since $\mathcal{S}_4$ is a particular case of systems $\mathcal{S}_3$, Corollary 1 also holds for $\mathcal{S}_3$. However, a (non-communication-efficient) self-stabilizing leader election algorithm can be trivially implemented for $\mathcal{S}_3$, henceforth for $\mathcal{S}_4$ too, as explained afterwards. Any system $\mathcal{S}_3$ is characterized by the existence of a timely routing overlay. Using this characteristic, our solution works as follows: (1) every process p periodically sends an (ALIVE,1,p) message through all its links; (2) when receiving an (ALIVE,k,r) message from a process q, a process p retransmits an (ALIVE,$k+1$,r) message to all the other processes except q if $k < n-1$. Using this method, we have the guarantee that, any alive p periodically receives an (ALIVE,$-$,q) message for each other alive process q. Each process can then locally compute in an *Alives* set the list of all alive processes. Once the list is known by each alive process, designate a leader is easy: each alive process just outputs the smallest process of its *Alives* set.

5 Pseudo-Stabilizing Communication-Efficient Leader Election in $\mathcal{S}_4$

We now show that, contrary to self-stabilizing leader election, pseudo-stabilizing leader election can be communication-efficiently done in $\mathcal{S}_4$. To that goal, we

Algorithm 2. Communication-Efficient Pseudo-Stabilizing Leader Election on $\mathcal{S}_4$

```
CODE FOR EACH PROCESS p:
 1: variables:
 2:     $Leader_p \in \{1,...,n\}$
 3:     $SendTimer_p$, $ReceiveTimer_p$, $Round_p$: non-negative integers
 4:
 5: procedure $StartRound(s)$
 6:     if $p \neq (s \bmod n+1)$ then
 7:         send(START,$s$) to $s \bmod n+1$
 8:     end if
 9:     $Round_p \leftarrow s$
10:     $SendTimer_p \leftarrow \lfloor \delta/\beta \rfloor$
11: end procedure
12:
13: repeat forever
14:     for all $q \in V \setminus \{p\}$ do
15:         if receive (ALIVE,$k$) or (START,$k$) from $q$ then
16:             if $Round_p > k$ then
17:                 send(START,$Round_p$) to $q$
18:             else
19:                 if $Round_p < k$ then
20:                     $StartRound(k)$
21:                 end if
22:                 $ReceiveTimer_p \leftarrow 0$
23:             end if
24:         end if
25:     end for
26:     $ReceiveTimer_p \leftarrow ReceiveTimer_p + 1$
27:     if $ReceiveTimer_p > 8\lceil \delta/\alpha \rceil$ then
28:         if $p \neq (Round_p \bmod n+1)$ then
29:             $StartRound(Round_p + 1)$
30:         end if
31:         $ReceiveTimer_p \leftarrow 0$
32:     end if
33:     $SendTimer_p \leftarrow SendTimer_p + 1$
34:     if $SendTimer_p \geq \lfloor \delta/\beta \rfloor$ then
35:         if $p = (Round_p \bmod n+1)$ then
36:             send(ALIVE,$Round_p$) to every process except $p$
37:         end if
38:         $Leader_p \leftarrow (Round_p \bmod n+1)$
39:         $SendTimer_p \leftarrow 0$
40:     end if
41: end repeat
```

study an algorithm provided in [11]. In this algorithm (Algorithm 2), each process p executes in rounds $Round_p = 0, 1, 2, \ldots$, where the variable $Round_p$ keeps p's current round. For each round r a unique process, $l_r = r \bmod n + 1$, is distinguished: l_r is called the *leader of the round*. The goal here is to make all alive processes converge to a round value having an alive process as leader.

When starting a new round k, a process p (1) informs the leader of the round, l_k, by sending it a (START,k) message if $p \neq l_k$ (Line 6-8), (2) sets $Round_p$ to k (Line 9), and (3) forces $SendTimer_p$ to $\lfloor \delta/\beta \rfloor$ (Line 10) so that (a) p sends (ALIVE,k) to all other processes if $p = l_k$ (Lines 35-37) and (b) p updates $Leader_p$ (Line 38). While in the round r, the leader of the round l_r periodically sends (ALIVE,r) to all other processes (Lines 33-40). A process p modifies $Round_p$ only in two cases: (i) if p receives an ALIVE or START message with a round value bigger than its own (Lines 19-20), or (ii) if p does not recently receive an ALIVE message from its round leader $q \neq p$ (Lines 26-32). In case (i),

p adopts the round value in the message. In case (*ii*), p starts the next round (Line 29). Case (*ii*) allows a process to eventually choose as leader a process that correctly communicates. Case (*i*) allows the round values to converge. Intuitively, the algorithm is pseudo-stabilizing because, the processes with the upper values of rounds eventually designates as leader an alive process that correctly communicates forever (perhaps the bi-source) thanks to (*ii*) and, then, the other processes eventually adopt this leader thanks to (*i*).

Theorem 5. *Algorithm 2 implements a communication-efficient pseudo-stabilizing leader election in system $\mathcal{S}_4$.*

6 Impossibility of Self-Stabilizing Leader Election in $\mathcal{S}_2$

To prove that we cannot implement any self-stabilizing leader election algorithm in $\mathcal{S}_2$, we show that it is impossible to implement such an algorithm in a particular case of $\mathcal{S}_2$: let $\mathcal{S}_3^-$ be any system S_2 having all its links that are reliable but containing no eventually timely overlay.

Let m be any message sent at a given time t. We say that a message *m'* is *older* than m if and only if *m'* was initially in a link or *m'* was sent at a time t' such that $t' < t$. We call *causal sequence* any sequence p_0,m_1,...,m_i,p_i,m_{i+1},...,p_{k-1},m_k such that: (1) $\forall i$, $0 \leq i < k$, p_i is a process and m_{i+1} is a message, (2) $\forall i$, $1 \leq i < k$, p_i receives m_i from p_{i-1}, and (3) $\forall i$, $1 \leq i < k$, p_i sends m_{i+1} after the reception of m_i. By extension, we say that m_k *causally depends on* p_0. Also, we say that m_k is a *new* message that causally depends on p_0 after the message $m_{k'}$ if and only if there exists two causal sequences p_0,m_1,...,p_{k-1},m_k and p_0,$m_{1'}$,...,$p_{k'-1}$,$m_{k'}$ such that $m_{1'}$ is *older* than m_1.

Lemma 2. *Let $\mathcal{A}$ be any self-stabilizing leader election algorithm in S_3^-. In every execution of $\mathcal{A}$, any alive process p satisfies: from any configuration where $Leader_p \neq p$, $\exists k \in \mathbb{R}^+$ such that p changes its leader if it receives no* new *message that causally depends on $Leader_p$ during k times.*

Proof. Assume, by the contradiction, that there exists an execution e where there is a configuration γ from which a process satisfies $Leader_p = q$ forever with $q \neq p$ while from γ p does not receive anymore a *new* message that causally depends on q. As $\mathcal{A}$ is self-stabilizing, it can start from any configuration. So, $\overrightarrow{e_\gamma}$ is a possible execution of $\mathcal{A}$. Let γ' be a configuration that is identical to γ except that q is crashed in γ'. As p only received messages that do not causally depend on q in $\overrightarrow{e_\gamma}$ (otherwise, this means that from γ, p eventually receives at least one *new* message that causally depends on q in e), there exists a possible execution $\overrightarrow{e_{\gamma'}}$ starting from γ' where p received exactly the same messages as in $\overrightarrow{e_\gamma}$ (the fact that q is crashed just prevents p from receiving the messages that causally depend on q). Hence, p cannot distinguish $\overrightarrow{e_\gamma}$ and $\overrightarrow{e_{\gamma'}}$ and p behaves in $\overrightarrow{e_{\gamma'}}$ as in $\overrightarrow{e_\gamma}$: it keeps q as leader forever while q is crashed: $\mathcal{A}$ is not a self-stabilizing leader election algorithm — a contradiction. □

Theorem 6. *There is no self-stabilizing leader election algorithm in system S_3^-.*

Proof. Assume, by the contradiction, that there exists a self-stabilizing leader election algorithm $\mathcal{A}$ in system S_3^-. By Definition 1, in any execution of $\mathcal{A}$, there exists a configuration γ such that in any suffix starting from γ there exists a unique leader and this leader no more changes. Let e be an execution of $\mathcal{A}$ where no process crashes and every link is timely. Let l be the alive process which is eventually elected in e. Consider now any execution e' identical to e except that there is a time after which there is at least one link in each path from l to some process p that arbitrary delays messages. Then, e and e' can have an arbitrary large common prefix. Hence, we can construct executions of $\mathcal{A}$ beginning with any prefix of e where l is eventually elected (during this prefix, every link behaves as a timely link) but in the associated suffix, any causal sequence of messages from l to p is arbitrary delayed and, by Lemma 2, p eventually changes its leader to a process $q \neq l$. Thus, for any prefix $\overleftarrow{e}$ of e where a process is eventually elected, there exists a possible execution having $\overleftarrow{e}$ as prefix and an associated suffix $\overrightarrow{e}$ in which the leader eventually changes. Hence, for some executions of $\mathcal{A}$, we cannot guarantee that from a certain configuration the leader will no more change: $\mathcal{A}$ is not self-stabilizing — a contradiction. □

Intuitively, Theorem 6 means that self-stabilization is impossible for a weaker system than S_3, in particular, $\mathcal{S}_2$. Hence:

Corollary 2. *There is no self-stabilizing leader election algorithm in system $\mathcal{S}_2$.*

7 Communication-Efficient Pseudo-Stabilizing Leader Election in $\mathcal{S}_2$

From Corollary 2, we know that there does not exist any self-stabilizing leader election algorithm in $\mathcal{S}_2$. We now show that pseudo-stabilizing leader elections exist in $\mathcal{S}_2$. Furthermore we can achieve communication-efficiency. The solution we propose is an adaptation of an algorithm provided in [3].

To obtain communication-efficiency, Algorithm 3 uses the same principle as Algorithm 1: Each process p periodically sends ALIVE to all other processes *only if it thinks it is the leader*. However, this principle cannot be directly applied in $\mathcal{S}_2$: if the *only* source happens to be a process with a large ID, the leadership can oscillate among some other alive processes infinitely often because these processes can be alternatively considered as crashed or alive.

To fix the problem, Aguilera *et al* propose in [3] that each process p stores in an accusation counter, $Counter_p[p]$, how many time it was previously suspected to be crashed. Then, if p thinks that it is the leader, it periodically sends ALIVE messages with its current value of $Counter_p[p]$ (Lines 23-29). Any process stores in an *Actives* set its own ID and that of each process it recently received an ALIVE message (Lines 8 and 12-16). Also, each process keeps the most up-to-date value of accusation counter of any process from which it receives an ALIVE message. Finally, any process q periodically chooses as leader the process having

Algorithm 3. Communication-Efficient Pseudo-Stabilizing Leader Election on $\mathcal{S}_2$

```
CODE FOR EACH PROCESS $p$:
 1: variables:
 2:     $Leader_p \in \{1,\ldots,n\}$, $OldLeader_p \in \{1,\ldots,n\}$
 3:     $SendTimer_p$, $ReceiveTimer_p$: non-negative integers
 4:     $Counter_p[1\ldots n]$, $Phase_p[1\ldots n]$: arrays of non-negative integers
 5:     $Collect_p$, $OtherActives_p$: sets of non-negative integers
 6:
 7: macros:
 8:     $Actives_p = OtherActives_p \cup \{p\}$
 9:
10: repeat forever
11:     for all $q \in V \setminus \{p\}$ do
12:         if receive(ALIVE,$qcnt$,$qph$) from $q$ then
13:             $Collect_p \leftarrow Collect_p \cup \{q\}$
14:             $Counter_p[q] \leftarrow qcnt$
15:             $Phase_p[q] \leftarrow qph$
16:         end if
17:         if receive(ACCUSATION,$ph$) from $q$ then
18:             if $ph = Phase_p[p]$ then
19:                 $Counter_p[p] \leftarrow Counter_p[p] + 1$
20:             end if
21:         end if
22:     end for
23:     $SendTimer_p \leftarrow SendTimer_p + 1$
24:     if $SendTimer_p \geq \lfloor \delta/\beta \rfloor$ then
25:         if $Leader_p = p$ then
26:             send(ALIVE,$Counter_p[p]$,$Phase_p[p]$) to every process except $p$
27:         end if
28:         $SendTimer_p \leftarrow 0$
29:     end if
30:     $ReceiveTimer_p \leftarrow ReceiveTimer_p + 1$
31:     if $ReceiveTimer_p > 5\lceil \delta/\alpha \rceil$ then
32:         $OtherActives_p \leftarrow Collect_p$
33:         if $Leader_p \notin Actives_p$ then
34:             send(ACCUSATION,$Phase_p[Leader_p]$) to $Leader_p$
35:         end if
36:         $OldLeader_p \leftarrow Leader_p$
37:         $Leader_p \leftarrow r$ such that $(Counter_p[r],r) = \min\{(Counter_p[q],q) : q \in Actives_p\}$
38:         if $(OldLeader_p = p) \wedge (Leader_p \neq p)$ then
39:             $Phase_p[p] \leftarrow Phase_p[p] + 1$
40:         end if
41:         $Collect_p \leftarrow \emptyset$
42:         $ReceiveTimer_p \leftarrow 0$
43:     end if
44: end repeat
```

the smallest accusation value among the processes in its $Actives_q$ set (IDs are used to break ties). After choosing a leader, if the leader of q changes, q sends an ACCUSATION message to its previous leader (Lines 33-35). The hope is that the counter of each source remains bounded, and, as a consequence, the source with the smallest counter is eventually elected.

However, the accusation counter of any source may increase infinitely often. Indeed, a source s can stop to consider itself as the leader: when s selects another process p as its leader. In this case, the source volontary stops sending ALIVE messages (for the communication efficiency), each other process that considered s as its leader eventually suspects s, and sends ACCUSATION messages to s. These messages cause incrementations of s'accusation counter. Later, due to the quality of the output links of p, p can also increase its accusation counter and then the source may obtain the leadership again.

Aguilera *et al* add a mechanism so that a source increments its own accusation counter only a finite number of times. A process now increments its accusation counter only if it receives a "legitimate" accusation: an accusation due to the delay or the loss of one of its ALIVE message. To detect if an accusation is legitimate, each process p saves in $Phase_p[p]$ the number of times it loses the leadership in the past and includes this value in each of its ALIVE messages (Line 26). When a process q receives an ALIVE message from p, it also saves the phase value sent by p in $Phase_q[p]$ (Line 15). Hence, when q wants to accuse p, it now includes its own view of p's phase number in the ACCUSATION message it sends to p (Line 34). This ACCUSATION message will be considered as legitimate by p only if the phase number it contains matches the current phase value of p (Lines 18-20). Moreover, whenever p loses the leadership and stops sending ALIVE message voluntary, p increments $Phase_p[p]$ and does not send the new value to any other process (Line 38-40): this effectively causes p to ignore all the spurious ACCUSATION messages that result from its voluntary silence.

Theorem 7. *Algorithm 3 implements a communication-efficient pseudo-stabilizing leader election in system $\mathcal{S}_2$.*

8 Pseudo-Stabilizing Leader Election in $\mathcal{S}_1$

Let $\mathcal{S}_1^-$ be any system $\mathcal{S}_0$ with an eventually timely source and $n \geq 3$ processes. In [3], Aguilera *et al* show that there is no communication-efficient leader election algorithm in system $\mathcal{S}_1^-$. Now, any pseudo-stabilizing leader election algorithm in $\mathcal{S}_1$ is also a pseudo-stabilizing leader election algorithm in $\mathcal{S}_1^-$ by Theorem 2.

Theorem 8. *There is no communication-efficient pseudo-stabilizing leader election algorithm in system $\mathcal{S}_1$ with $n \geq 3$ processes.*

By Theorem 8, there is no communication-efficient pseudo-stabilizing leader election algorithm in system $\mathcal{S}_1$ with $n \geq 3$ processes. However, using similar techniques as those previously used in the paper, we can adapt the robust but non communication-efficient algorithm for $\mathcal{S}_1^-$ given in [?] to obtain a pseudo-stabilizing but non communication-efficient leader election algorithm for $\mathcal{S}_1$.

9 Future Works

There is some possible extensions to this work. First, getting a communication-efficient leader election in a system having a *timely routing overlay* remains an open question. Then, we can study robust stabilizing leader election in systems where only a given number of processes may crash. It could be interesting to extend these algorithms and results to other models like those in [18,12] and other communication topologies. Finally, we can study the implementability of robust stabilizing decision problems.

Acknowledgements. We are grateful to Ajoy K. Datta for his interesting remarks.

References

1. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. Technical report, LIAFA (2007) available at the following address, http://hal.archives-ouvertes.fr/hal-00167935/fr/
2. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery 17, 643–644 (1974)
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. In: PODC 2003, pp. 306–314 (2003)
4. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance (preliminary version). In: PODC, pp. 195–206 (1993)
5. Anagnostou, E., Hadzilacos, V.: Tolerating transient and permanent failures (extended abstract). In: Schiper, A. (ed.) WDAG 1993. LNCS, vol. 725, pp. 174–188. Springer, Heidelberg (1993)
6. Beauquier, J., Kekkonen-Moneta, S.: Fault-tolerance and self-stabilization: Impossibility results and solutions using failure detectors. Int. J of Systems Science 28(11), 1177–1187 (1997)
7. Hutle, M., Widder, J.: Self-stabilizing failure detector algorithms. In: Parallel and Distributed Computing and Networks, pp. 485–490 (2005)
8. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: PODC, pp. 328–337 (2004)
9. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems 8(4), 424–440 (1997)
10. Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. In: PODC 1999, pp. 199–207. ACM Press, New York (1999)
11. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 108–122. Springer, Heidelberg (2001)
12. Malkhi, D., Oprea, F., Zhou, L.: Omega meets paxos: Leader election and stability without eventual timely links. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 199–213. Springer, Heidelberg (2005)
13. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
14. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. Distrib. Comput. 7(1), 35–42 (1993)
15. Larrea, M., Fernández, A., Arévalo, S.: Optimal implementation of the weakest failure detector for solving consensus. In: SRDS, pp. 52–59 (2000)
16. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with byzantine failures and little system synchrony. In: DSN, pp. 147–155 (2006)
17. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. journal version of [3](Unpublished)
18. Hutle, M., Malkhi, D., Schmid, U., Zhou, L.: Brief announcement: Chasing the weakest system model for implementing omega and consensus. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 576–577. Springer, Heidelberg (2006)

Byzantine Self-stabilizing Pulse in a Bounded-Delay Model

Danny Dolev[*] and Ezra N. Hoch

School of Engineering and Computer Science,
The Hebrew University of Jerusalem, Israel
{dolev,ezraho}@cs.huji.ac.il

Abstract. "Pulse Synchronization" intends to invoke a recurring distributed event at the different nodes, of a distributed system as simultaneously as possible and with a frequency that matches a predetermined regularity. This paper shows how to achieve that goal when the system is facing both transient and permanent (*Byzantine*) failures.

Byzantine nodes might incessantly try to de-synchronize the correct nodes. Transient failures might throw the system into an arbitrary state in which correct nodes have no common notion what-so-ever, such as time or round numbers, and thus cannot use any aspect of their own local states to infer anything about the states of other correct nodes. The algorithm we present here guarantees that eventually all correct nodes will invoke their pulses within a very short time interval of each other and will do so regularly.

The problem of pulse synchronization was recently solved in a system in which there exists an outside beat system that synchronously signals all nodes at once. In this paper we present a solution for a bounded-delay system. When the system in a steady state, a message sent by a correct node arrives and is processed by all correct nodes within a bounded time, say d time units, where at steady state the number of Byzantine nodes, f, should obey the $n > 3f$ inequality, for a network of n nodes.

1 Introduction

When constructing distributed systems, fault tolerance is a major consideration. Will the system fail if part of the memory has been corrupted (e.g. by a buffer overrun)? Will it withstand message losses? Will it overcome network disconnections? To build distributed systems that are fault tolerant to different types of faults, two main paradigms have been used: The Byzantine model and the self-stabilizing model

The *Byzantine* fault paradigm assumes that up to some fraction of the nodes in the system (typically one-third) may behave arbitrarily. Moreover, these nodes can collude in order to try and bring the system down (for more on *Byzantine* faults, see [1]).

[*] Part of the work was done while the author visited Cornell university. The work was funded in part by ISF, ISOC, NSF, CCR, and AFOSR.

The self-stabilization model assumes that the system might be thrown out of its assumed working conditions for some period of time. Once the system is back to its normal boundaries, all nodes should converge to the desired solution. For example, starting from any memory state, after a finite time, all nodes should have the same clock value (for more on self-stabilization, see [2]).

The strength of self-stabilizing systems emerges from their ability to continue functioning after recovering from a massive disruption of their assumed working conditions. The advantage of *Byzantine* tolerant systems comes from being able to withstand any kind of faults while the system operates in its known boundaries. By combining these two fault models, a distributed system can continue operating properly in the presence of faults as long as "everything is going well"; however, if "things aren't going well", the system will be able to recover once the conditions hold again, and the ratio of *Byzantine* nodes hold.

Clock synchronization is a fundamental building block in many distributed systems; hence, creating a self-stabilizing *Byzantine* tolerant clock synchronization is a desirable goal. Once such an algorithm exists, one can stabilize *Byzantine* tolerant algorithms that were not designed for self-stabilization (see [3]). Clock synchronization can be created upon a PULSEing algorithm (see [4]), which is the main motivation behind the current paper.

The main contribution of the current paper is to develop a pulse synchronization algorithm that converges once the communication network resumes delivering messages within bounded, say d, time units, and the number of Byzantine nodes, f, obeys the $n > 3f$ inequality, for a network of n nodes. The attained pulse synchronization tightness is $3d$ with a deterministic convergence time of a constant number of pulse cycles (each containing $O(f)$ communication rounds).

1.1 Related Work

Algorithms combining self-stabilization and *Byzantine* faults, can be divided into two classes. The first consists of problems in which the state of each node is determined locally (see [5,6,7]). The other class contains problems such that a node's state requires global knowledge - for example, clock synchronization such that every two nodes' clocks have a bounded difference that is independent of the diameter of the network (see [8,9,4]). The current paper is of the latter class.

The current paper makes use of the self-stabilizing *Byzantine* agreement algorithm (ss-BYZ-AGREE) presented in [10]. The above work operates in exactly the same model as the current paper, and its construction will be used as the basic building block in our current solution. Appendix A lists the main properties of this building block.

When discussing clock synchronization, it is common to represent the clocks as an integer value that progresses linearly in time (see [11]). This was previously termed digital clock synchronization ([12,13,14,15]) or "synchronization of phase-clocks" ([16]). In the current paper we provide a PULSEing algorithm; however, when comparing it to other results, we consider the digital clock synchronization algorithm that can be built upon it (as in [4]).

The first ever algorithm to address self-stabilizing *Byzantine* tolerant clock synchronization is presented in [8]. [8] discusses two models; one is synchronous, that is, all nodes are connected to some global "tick" system that produces "ticks" that reach all nodes at the same time, and messages sent at any given tick reach their destination before the following tick. The second model is a bounded-delay network, in which there is no common tick system, but messages have a bounded delay on their delivery time. There is no reason to consider an asynchronous model, since even a single fail-stop failure can't be overcome (see [17]). Note that the bounded-delay model contains the first (synchronous) one. [8] gives two solutions, one for each model, both of which converge in expected exponential time; both algorithms support $f < \frac{n}{3}$.

In [9] clock synchronization is reached in deterministic linear time. However, [9] addresses only the synchronous model, and supports only up to $f < \frac{n}{4}$. In [18], a PULSEing algorithm that operates in the synchronous model is presented, which converges in deterministic linear time, and supports $f < \frac{n}{3}$, matching the lower bounds both in the maximal number of *Byzantine* nodes, and in the convergence time (see [19] for lower bounds). In [20] a very complicated pulse synchronization protocol, in the same model as the current paper, was presented.

The current paper presents a PULSE-synchronization algorithm, which has deterministic linear convergence time, supports $f < \frac{n}{3}$, and operates in a bounded-delay model.

2 Model and Problem Definition

The model used in this paper consists of n nodes that can communicate via message passing. Each message has a bounded delivery time, and a bounded processing time at each node; in addition the message sender's identity can be validated. The network is not required to support broadcast.

Each node has a local clock. Local clocks might show different readings at different nodes, but all clocks advance at approximately the real-time rate.

Nodes may be subject to transient faults, and at any time a constant fraction of nodes may be *Byzantine*, where f, the number of *Byzantine* nodes satisfies $f < \frac{n}{3}$.

Definition 1. *A node is* **non-faulty** *if it follows its algorithm, processes messages in no more than π time units and has a bounded drift on its internal clock. A node that is not* **non-faulty** *is considered* **faulty** *(or Byzantine). A node is* **correct** *if it has been* **non-faulty** *for Δ_{node} time units.*[1]

Definition 2. *A communication network is* **non-faulty** *if messages arrive at their destinations within δ time units, and the content of the message, as well as the identity of the sender, arrive intact. A communication network is* **correct** *if it has been* **non-faulty** *for Δ_{net} time units.*[2]

[1] The value of Δ_{node} will be stated later.
[2] The value of Δ_{net} is stated below.

The value of Δ_{net} is chosen so if at time t_1 the communication network is non-faulty and stays so until $t_1 + \Delta_{net}$, then only messages sent *after* t_1 are received by non-faulty nodes.

Definition 3. *A system is* **coherent** *if the network is* **correct** *and there are at least $n - f$* **correct nodes***.*

Once the system is coherent, a message between two correct nodes is sent, received and processed within d time units, where d is the sum of δ, π and the upper bound on the potential drift of correct local timers during such a period. Δ_{net} should be chosen in such a way as to satisfy $\Delta_{net} \geq d$.. Since d includes the drift factor, and since all the intervals of time will be represented as a function of d, we will not explicitly refer to the drift factors in the rest of the paper.

2.1 Self-stabilizing Byzantine Pulse-Synchronization

Intuitively, the PULSE synchronization problem consists of synchronizing the correct nodes so they invoke their pulses together *Cycle* time apart. That is, all correct nodes should invoke pulses within a short interval, then not invoke a pulse for approximately *Cycle* time, then invoke pulses again within a short interval, and so on. Adding "Self-stabilizing *Byzantine*" to the PULSE synchronization problem, means that starting from any memory state and in spite of ongoing *Byzantine* faults, the correct nodes should eventually invoke pulses together *Cycle* time apart.

Since message transmission time varies and also due to the *Byzantine* presence, one cannot require the correct nodes to invoke pulses exactly *Cycle* time apart. Instead, $cycle_{min}$ and $cycle_{max}$ are values that define the bounds on the actual CYCLE length in a correct behavior. The protocol presented in this paper achieves $cycle_{min} = Cycle \leq \text{CYCLE} \leq Cycle + 12d = cycle_{max}$.

To formally define the PULSE synchronization problem, a notion of "PULSEing together" needs to be addressed.

Definition 4. *A correct node p invokes a pulse* **near** *time unit t if it invokes a pulse in the time interval $[t - \frac{3}{2} \cdot d, t + \frac{3}{2} \cdot d]$. Time unit t is a* **pulsing point** *if every correct node invokes a pulse near t.*

Definition 5. *A system is in a* **synchronized_pulsing_state** *in the time interval $[r_1, r_2]$ if*

1. *there is some pulsing point $t_0 \in [r_1, r_1 + cycle_{\max}]$;*
2. *for every pulsing point $t_i \leq r_2 - cycle_{\max}$ there is another pulsing point t_{i+1}, $t_{i+1} \in [t_i + cycle_{\min}, t_i + cycle_{\max}]$;*
3. *for any other pulsing point $\bar{t} \in [r_1, r_2]$, there exists i, such that $|t_i - \bar{t}| \leq \frac{3}{2} \cdot d$.*

Intuitively, the above definition says that in the interval $[r_1, r_2]$ there are pulsing points that are spaced at least $cycle_{min}$ apart and no more than $cycle_{max}$ apart.

Definition 6. *Given a coherent system,* The Self-Stabilizing Pulse Synchronization Problem *requires that:*

Convergence: *Starting from an arbitrary system state, the system reaches a synchronized_pulsing_state within a finite amount of time.*
Closure: *If the system is in a synchronized_pulsing_state in some interval* $[t_1, t_2]$ *(s.t.* $t_2 > t_1 + cycle_{\max}$*), then it is also in a synchronized_pulsing_state in the interval* $[t_1, t]$ *for any* $t > t_2$*.*

3 Solution Overview

The main algorithm, Erratic-Pulser, assumes a self-stabilizing, *Byzantine* tolerant, distributed agreement primitive, $\mathcal{Q}$, which is defined in the following section. A protocol providing the requirements of $\mathcal{Q}$ is presented in Section 5.

Using $\mathcal{Q}$, the Erratic-Pulser algorithm produces agreement among the correct nodes on different points in time at which they invoke pulses; and these points become sparse enough. Using this basic point-in-time agreement, a full pulse algorithm is built, named Balanced-Pulser. By using the basic pulseing pattern produced by Erratic-Pulser, Balanced-Pulser manages to solve the pulse-synchronization problem.

During the rest of this paper, the constants *Cycle*, $cycle_{min}$ and $cycle_{max}$ are used freely. However, it is important to note that *Cycle* must be chosen such that it is large enough. The exact limitations on the possible values of *Cycle* will be stated later. An explanation on how to create a pulseing algorithm with an arbitrary *Cycle* value is presented in Section 9.

4 The $\mathcal{Q}$ Primitive

$\mathcal{Q}$ is a primitive executed by all the nodes in the system. However, each invocation of a specific $\mathcal{Q}$ is associated with some node, p, hence a specific invocation will sometimes be referred to as $\mathcal{Q}(p)$. That is, $\mathcal{Q}(p)$ is a distributed algorithm, executed by all nodes, and triggered by p (p's special role in the execution of $\mathcal{Q}(p)$ will be elaborated upon later). In the following discussion several instances of $\mathcal{Q}(p)$ may coexist, but it will be clear from the context to which instance $\mathcal{Q}(p)$ refers. Each instance is a separate copy of the protocol and each node executes each instance separately.

$\mathcal{Q}(p)$ is a "consensus primitive", that is, each node q has an input value v_q, and upon completing the execution of $\mathcal{Q}(p)$ it produces some output value V_q. The input values and output values are boolean, i.e., $v_q, V_q \in \{0, 1\}$. Denote by τ_q the local-time at node q at which V_q is defined; that is τ_q is the local time at node q at which q determines the value of V_q (and terminates $\mathcal{Q}(p)$).

The un-synchronized and distributed nature of $\mathcal{Q}(p)$ requires distinguishing between two stages. The first stage is when p attempts to invoke $\mathcal{Q}(p)$; this attempted invocation involves exchanging messages among the nodes. The second stage is when enough correct nodes agree to join p's invocation of $\mathcal{Q}(p)$, and hence start executing $\mathcal{Q}(p)$. When p is correct, the first stage and the second stage are close to each other; however, when p is faulty, no a priori bound can be set on the time difference between the first and the second stages. Note that p itself joins the instance of $\mathcal{Q}(p)$ only after the preliminary invocation stage.

Informally, $join_q$ is the time at which q agrees to join the instance of $\mathcal{Q}(p)$ (which is also the time at which q determines its input value v_q.) Following this stage it actively participates in determining the output value of $\mathcal{Q}(p)$. The implementation of $\mathcal{Q}(p)$ needs to explicitly instruct a node when to determine its input value.

In the following discussion, rt_{invoke} will denote the time at which p invoked $\mathcal{Q}(p)$ and $join_{first}$ will denote the time value at which the first correct node joins the execution of $\mathcal{Q}(p)$; $join_{last}$ will denote the time value at which the last correct node joins p in executing $\mathcal{Q}(p)$. That is, $join_{first} = \min_{\text{correct } q}\{join_q\}$ and $join_{last} = \max_{\text{correct } q}\{join_q\}$.

$\mathcal{Q}(p)$ is self-stabilizing, and its properties hold once the system executing it is coherent for at least $\Delta_{\mathcal{Q}}$ time. In other words, no matter what the initial values in the nodes' memory may be, after the system has been coherent for $\Delta_{\mathcal{Q}}$ time, the properties of $\mathcal{Q}(p)$ will hold.[3]

$\mathcal{Q}(p)$'s properties follow. Observe that there are different requirements, depending on whether p is a correct node or not.

1. For any node p invoking $\mathcal{Q}(p)$, the following holds:
   (a) *Agreement:* all correct nodes that have terminated have the same output value. That is, for any pair of correct nodes, q and q', which have completed $\mathcal{Q}(p)$, $V_q = V_{q'}$. V denotes this common output value.
   (b) *Validity:* if all correct nodes have the same input value ν then $V = \nu$.
   (c) *Termination:* if some correct node joins $\mathcal{Q}(p)$ then all correct nodes terminate within Δ_{max} time units from $join_{first}$ but no quicker than Δ_{min}. That is, for a correct q, $rt(\tau_q) \in [join_{first} + \Delta_{min}, join_{first} + \Delta_{max}]$, where τ_q is the local time at which q determines the value of V_q, and $rt(\tau_q)$ is the time at which this takes place.
   (d) *Tightness:* if a correct node terminates, then for any correct nodes q, q': $|rt(\tau_q) - rt(\tau_{q'})| \leq 3 \cdot d$.
   (e) *Collaboration:* if one correct node joins the execution of $\mathcal{Q}(p)$, then all correct nodes join the execution of $\mathcal{Q}(p)$ within $3 \cdot d$ of each other; that is, $|join_{last} - join_{first}| \leq 3 \cdot d$.
2. For a correct node p, starting the execution of $\mathcal{Q}(p)$ at time rt_{invoke}, the following holds:
   (a) *Strong Termination:* $join_{first} \leq rt_{invoke} + 3 \cdot d$. That is, the first correct node to join p in executing $\mathcal{Q}(p)$ does so within $3 \cdot d$ time from p's invocation of $\mathcal{Q}(p)$. Combined with *termination*, this property means that all correct nodes terminate by $rt_{invoke} + 3 \cdot d + \Delta_{max}$.
   (b) *Separation:* p does not start $\mathcal{Q}(p)$ more than once every $3 \cdot \Delta_{max}$ time units.

[3] $\Delta_{\mathcal{Q}}$ is defined below.

3. The following holds for a faulty p, invoking $\mathcal{Q}(p)$:
   (a) *Separation:* if a correct node q assigns an output value for $\mathcal{Q}(p)$ at some time t_1, then it does not assign an output value for $\mathcal{Q}(p)$ again before $t_1 + 2 \cdot \Delta_{min}$.

Remark 1. According to "termination" if $join_{first}$ is not defined, all correct nodes do not terminate. This implies that all correct nodes terminate if and only if some correct node joins p in executing $\mathcal{Q}(p)$.

Note that p may require the invocation of several $\mathcal{Q}(p)$ instances concurrently. To differentiate between these instances, they are marked with an additional index, e.g $\mathcal{Q}_1(p), \mathcal{Q}_2(p)$, etc. Each such instance has its own memory space, and hence is independent of other instances. According to the *separation* property, a correct node does not execute the same instance of $\mathcal{Q}(p)$ too often. That is, $\mathcal{Q}_1(p)$ is not executed until the previous $\mathcal{Q}_1(p)$ has terminated. A faulty node p may try to invoke $\mathcal{Q}_1(p)$ as often as it likes, however correct nodes will ignore the multiple executions.

5 Implementing $\mathcal{Q}(p)$, the ss-Byz-Q Algorithm

The implementation of $\mathcal{Q}(p)$ makes use of ss-Byz-Agree ([10]). The properties of ss-Byz-Agree and its guarantees are listed in Appendix A. In ss-Byz-Agree, when a node p wants to start an agreement on some value, it sends (*Initiator*, p, v_p) to all other nodes. Nodes receiving this message, initiate the ss-Byz-Agree algorithm, and start participating in the agreement. Other nodes, that have not received the (*Initiator*, p, v_p) message (in case p is *Byzantine*), join the ss-Byz-Agree algorithm once they are "convinced" that enough correct nodes are already executing ss-Byz-Agree on p's value.

This leads to the following insight. If a correct node q ignores an (*Initiator*, p, v_p) message sent by a *Byzantine* node (for any reason), it does not change the properties of ss-Byz-Agree. Since due to p's *Byzantine* nature, if p would have not sent this specific message to q, ss-Byz-Agree's properties would still hold. Hence, whether p sends the message and a correct node ignores it, or p doesn't send the message at all, the properties of ss-Byz-Agree remain the same. Note that this is true only if p is *Byzantine*. In what follows, when a node *rejects* a message it ignores it, and when it *accepts* a message it continues to execute the protocol as instructed.

Figure 1 presents an algorithm that implements the $\mathcal{Q}$ primitive. If node p wants to invoke $\mathcal{Q}(p)$, it does so by executing ss-Byz-Agree (p, *start_Q*) (this is the *Init* stage), which means it sends (*Initiator*, p, *start_Q*) messages to other nodes. This action triggers the *prolog* stage of the protocol. If this stage completes successfully, each correct node peforms a timing test to determine whether to join the computation of the primitive $\mathcal{Q}(p)$. The algorithm is executed in the background continuously, and it responds to messages / events that are triggered by p's execution of ss-Byz-Agree (p, *start_Q*).

```
Algorithm ss-Byz-Q
(implementing $\mathcal{Q}(p)$)                                   /* executed at node q */

Init: If $p = q$ invoke ss-Byz-Agree $(p, start\_Q)$);
                         /* by sending (Initiator, p, start_Q)) message to all */

Prolog: On receiving $(Initiator, p, start\_Q)$ message from $p$
  if $local_q > last_q[p] + 2 \cdot \Delta_{max}$ then accept the message;
    else ignore the message;

The Primitive $\mathcal{Q}(p)$:

 1. On returning from ss-Byz-Agree for $p$ with value "$start\_Q$" do
      if $local_q > last_q[p] + \Delta_{max} + 3 \cdot d$ then
        begin
            determine the input value $v_q$;      /* this is when q joins $\mathcal{Q}(p)$ */
            $start_q[p] := local_q$;
            reset $val_q[p, \_]$;
            wait for $3 \cdot d$ and then invoke ss-Byz-Agree $(q, (p, v_q))$;
                         /* by sending (Initiator, q, $(p, v_q)$) message to all */
        end
      $last_q[p] := local_q$;
 2. On receiving $(Initiator, p', (p, v_{p'}))$
      if $local_q \leq start_q[p] + 7 \cdot d$ then accept the message;
        else ignore the message;
 3. On returning from ss-Byz-Agree for $p'$ with value $(p, v_{p'})$ do
      $val_q[p, p'] = v_{p'}$;
 4. At time $local_q = start_q[p] + \Delta + 17 \cdot d$
      for all $p'$, check the agreement values $val_q[p, p']$
        if there are $n - f$ 1's, then set $V_q[p] := 1$, otherwise set $V_q[p] := 0$;
      return $V_q[p]$ as $\mathcal{Q}(p)$'s output value;

Cleanup:
  for any $p$: if $last_q[p] > local_q$ then $last_q[p] := local_q$
  for any $p$: if $start_q[p] > local_q$ then $start_q[p] := local_q$
```

Fig. 1. An algorithm that implements $\mathcal{Q}(p)$

The values of the constants for the ss-Byz-Q algorithm are: $\Delta_{max} := \Delta + 20 \cdot d$ and $\Delta_{min} := \Delta_{max} - 3 \cdot d$, where Δ represents the maximal time required to complete ss-Byz-Agree ($\Delta := 7(2f + 3)d$, see Appendix A).

In the $\mathcal{Q}(p)$ protocol in Figure 1, $local_q$ represents the local time at each node q; in addition there are two arrays of values: $start_q, last_q$. These arrays hold local-time values (per node p) of events regarding $\mathcal{Q}(p)$'s execution at q. $last_q[p]$ is used to ensure that q doesn't participate in $\mathcal{Q}(p)$ too often. $start_q[p]$ is used so that all correct nodes know when to stop collecting values of other nodes (regarding $\mathcal{Q}(p)$'s instance); these values are stored in $val_q[p, p']$.

Remark 2. In the protocols, all the comparisons of the value of $local_q$ to some other value, always compare values that are at most some bounded range apart, say D. To deal with the possible wraparound of the counter $local_q$, it is enough that the range of values of $local_q$ will be $D' > 2D$. The "cleanup" stage of the protocol (See Figure 1) ensures that comparisons over a circle of size D' are uniquely determined.

Note that the protocol parameters n, f and *Cycle* (as well as the system characteristic d) are fixed constants and thus considered part of the incorruptible correct code.[4] Thus we assume that non-faulty nodes do not hold arbitrary values of these constants.

The value of Δ_{node} is crucial for the following claims. Δ_{node} is used to ensure that non-faulty nodes "run" for some time before they become correct. In the context of this paper, a non-faulty node should not be considered correct when it executes ss-Byz-Q that it might have joined before it was non-faulty. Moreover, since ss-Byz-Q uses $ss-$ Byz-Agree which has its own requirements for a node's correctness, we set $\Delta_{node} := \Delta_{node\text{-}ss\text{-}byz\text{-}agree} + \Delta_{max} + 3 \cdot d$.[5]

Lemma 1. *Once the system is coherent, if all correct nodes pass the condition in Line 1 during a time interval* $[t_1, t_2]$ *s.t.*

1. $t_2 - t_1 \leq 3 \cdot d$, *and*
2. *at* t_1 *for any correct node* q *it holds that* $local_q > start_q[p] + \Delta_{max}$,

then Agreement, Validity, Termination, Tightness *and* Collaboration *hold.*

Proof. First we show that no $ss-$ Byz-Agree$(p', (p, v_{p'}))$ that was initiated before t_1, terminates after t_1. By assumption, at time t_1, each correct node q has $local_q > start_q[p] + \Delta_{max}$, which means that no correct node has accepted (*Initiator*, p', $(p, v_{p'})$) in the time interval $[t_1 - \Delta_{max} + 7 \cdot d, t_1]$. In the protocol, any correct node that accepts (*Initiator*, p', $(p, v_{p'})$) before $t_1 - \Delta_{max} + 7 \cdot d$, must have terminated the ss-Byz-Agree no later than $t_1 - \Delta_{max} + 7 \cdot d + \Delta + 7 \cdot d$, and hence all correct nodes must have terminated the ss-Byz-Agree no later than $t_1 - \Delta_{max} + 17 \cdot d + \Delta$. Since $\Delta_{max} := \Delta + 20 \cdot d$, we conclude that any ss-Byz-Agree that was invoked before t_1 terminated before t_1.

In addition, due to setting of $last_q[p]$ in Line 1, no correct node will pass the condition in Line 1 again, before $t_1 + \Delta_{max} + 3 \cdot d$. Hence, during the interval $[t_2, t_2 + \Delta_{max}]$ no correct node passes the condition in Line 1. Note that each correct node passes the condition in Line 1 exactly once in the interval $[t_1, t_2]$. Hence, all correct nodes reset $val_q[p, \_]$ in the interval $[t_1, t_2]$ and never do so again before $t_2 + \Delta_{max}$. In a sense, the above means that all correct nodes join p in the interval $[t_1, t_2]$ and do not join p again, until after time $t_2 + \Delta_{max}$.

From the lemma's condition; for any two correct nodes q, q', it holds that $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$. At this stage, each correct node joins p's execution of $\mathcal{Q}(p)$ and hence *Collaboration* holds.

For any pair of correct nodes, q, q', $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$. Moreover, q sends its (*Initiator*, q, (p, v_q)) message $3 \cdot d$ after its $start_q[p]$. Since $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$, q' has already set its $start_{q'}[p]$ value when it receives q's (*Initiator*, q, (p, v_q)) message. Similarly, q' receives q's (*Initiator*, q, (p, v_q)) message within $7 \cdot d$ of $start_{q'}[p]$ ($3d$ is the waiting of $3d$ in Line 1 of the

[4] A system cannot self-stabilize if the entire code space can be perturbed, see [21].

[5] $\Delta_{node\text{-}ss\text{-}byz\text{-}agree} := 14(2 \cdot f + 3) \cdot d + 10 \cdot d$ (see [10]).

protocol, additional $3d$ is the time difference in $start_q$, and d is the uncertainty in message delivery), and thus does not ignore it.

This last argument implies that for every correct node q, any other correct node q' accepts its (*Initiator*, q, (p, v_q)) message, and hence finishes ss-Byz-Agree (q, v_q) before time $start_{q'} + \Delta + 7 \cdot d$. Therefore, every correct node "hears" every other correct node's value. That is, for any triplet of correct nodes, q, q', q'' it holds that $val_q[p, q''] = val_{q'}[p, q'']$.

Consider a *Byzantine* node q. If some correct node q' has accepted its (*Initiator*, q, (p, v_q)) message, then according to point 3 of the "*Timeliness-Agreement*" property (see Appendix A), ss-Byz-Agree will terminate within $\Delta + 7 \cdot d$ time units. Hence, any other correct node q'' will terminate within $3 \cdot d$. Since $|start_{q'} - start_{q''}| \leq 3 \cdot d$, node q'' will have accepted the same value no later than $start_{q''} + \Delta + 16 \cdot d$ ($7d$ come from above, $3d$ come from the difference in $start_q$, $3d$ come from the difference in the termination of ss-Byz-Agree and another $3d$ from the waiting after setting $start_q[p]$; all together $7d + 3d + 3d + 3d = 16d$). Note that this proof holds even though correct nodes may ignore an (*Initiator*, q, (p, v_q)) message sent by a *Byzantine* node q. Since no ss-Byz-Agree that was invoked before t_1 is accepted after t_1, it holds that $val_{q'}[p, q] = val_{q''}[p, q]$. As a result all correct nodes have the same set of values when they consider the output value $v_q[p]$, hence they all agree on the same output value. In addition, if all correct nodes started with "0", they will see at most f "1"s, and hence decide $V = 0$. Moreover, if all correct nodes started with "1", then all correct nodes will decide 1. Thus *Agreement* and *Validity* hold.

Each correct node terminates within $\Delta + 17 \cdot d$ of returning from p's invocation of ss-Byz-Agree (which is the joining point of each correct node to $\mathcal{Q}(p)$), and they all terminate within $3 \cdot d$ time units of each other (since $|rt(start_q) - rt(start_{q'})| \leq 3 \cdot d$). Hence *Termination* and *Tightness* hold. □

The following shows that ss-Byz-Q converges in $\Delta_{\mathcal{Q}} := 4 \cdot \Delta_{max} + \Delta_{ss-\text{Byz-Agree}}$. For ss-Byz-Q to operate correctly, ss-Byz-Agree must converge as well. Hence, in the following, we will assume that $\Delta_{ss-\text{Byz-Agree}}$ time has already passed.[6]

Lemma 2. *Once the system has been coherent for $4 \cdot \Delta_{max}$ time units, then for a faulty p, the properties of $\mathcal{Q}(p)$ hold for* ss-Byz-Q.

Proof. Note that once the system is coherent, $start_q[p], last_q[p] \leq local_q$. Notice that $start_q[p]$ can only be updated at Line 1.

Consider the first $2 \cdot \Delta_{max}$ time units following the time at which the system became coherent. If some correct node terminates ss-Byz-Agree $(p, start\_\mathcal{Q})$ during this period, then all correct nodes do so within $3 \cdot d$ of each other. Hence, they all set their $last_q[p]$ variable within $3 \cdot d$ time units of each other. That is, the values $rt(last_q[p])$ are at most $3 \cdot d$ units apart from each other. If no correct node terminates ss-Byz-Agree $(p, start\_\mathcal{Q})$ for $2 \cdot \Delta_{max}$, then all $last_q[p]$ haven't been updated for $2 \cdot \Delta_{max}$ and hence all $last_q[p] + 2 \cdot \Delta_{max} < local_q$ for every correct node q.

[6] $\Delta_{ss-\text{Byz-Agree}} := 2\Delta + 10d$ (see [10]).

Thus, we conclude that after $2 \cdot \Delta_{max}$ time units either all correct nodes have $rt(last_q[p])$ within $3 \cdot d$ of each other or all correct nodes have $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Note that this state continues to hold as long as no correct node enters Line 1 since $last_q[p]$ is not updated at any correct node. If some correct node does update $last_q[p]$ at Line 1, then all correct nodes do so $3 \cdot d$ time units apart.

Now consider the period between $2 \cdot \Delta_{max}$ and $4 \cdot \Delta_{max}$ time units following the time the system became coherent. If no correct node terminated ss-Byz-Agree (p, $start\_\mathcal{Q}$), then each correct node, q, has $local_q > start_q[p] + \Delta_{max}$ (since $start_q[p]$ had not been updated for at least $2 \cdot \Delta_{max}$ time units). Otherwise, if some correct node q' has terminated ss-Byz-Agree (p, $start\_\mathcal{Q}$), it means that there exists some correct node $\bar{q}$ that accepted (*Initiator*, p, $start\_\mathcal{Q}$) at the *Prolog* stage. Thus, $local_{\bar{q}} > last_{\bar{q}}[p] + 2 \cdot \Delta_{max}$, which means that until $last_{\bar{q}}[p]$ is reset, $local_{\bar{q}} > last_{\bar{q}}[p] + \Delta_{max} + 3 \cdot d$. Remember that either the $rt(last_q[p])$ of each correct node $\bar{q}$ is within $3 \cdot d$ time units of all other correct nodes, or each correct node has $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Therefore, we have that for all correct nodes, until $last_q[p]$ is reset, $local_q > last_q[p] + \Delta_{max} + 3 \cdot d$; which means that when q terminates ss-Byz-Agree (p, $start\_\mathcal{Q}$) it passes the condition of Line 1, along with all other correct nodes (within a $3 \cdot d$ interval). Therefore, the $rt(start_q[p])$ of all correct nodes are within $3 \cdot d$ time units.

Thus, after $4 \cdot \Delta_{max}$, the $rt(start_q[p])$ of all correct nodes are within $3 \cdot d$ time units *and* $rt(last_q[p])$ within $3 \cdot d$ time units of each other or all correct nodes have $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Note that the next time p invokes ss-Byz-Agree, all correct nodes values of $start_q[p]$ will be greater than their $local_q$ by at least $2 \cdot \Delta_{max}$. Hence, if p invokes ss-Byz-Agree and some correct node terminates that instance of ss-Byz-Agree, then all correct nodes pass the condition of Line 1 within $3 \cdot d$ of each other, and each correct node has $local_q > start_q[p] + \Delta_{max}$. Hence, by Lemma 1 all properties except for *Separation* hold.

To show that *Separation* holds, notice that once a correct node has passed Line 1, it won't do so again for at least $2 \cdot \Delta_{max} - 3 \cdot d$ time units. In addition, it will terminate the current instance of $\mathcal{Q}$ within Δ_{max}. Hence, the next invocation of $\mathcal{Q}$ cannot terminate before $2 \cdot \Delta_{min}$. And *Separation* holds. □

Lemma 3. *Once the system has been coherent for* $4 \cdot \Delta_{max}$ *time units, then for a correct* p*, the properties of* $\mathcal{Q}(p)$ *hold for* ss-Byz-Q*, given that* p *does not initiate ss-*Byz-Agree *(p, start_Q) earlier than* $3 \cdot \Delta_{max}$ *time units following its previous invocation.*

Proof. Since *Agreement, Validity, Termination, Tightness* and *Collaboration* were proven to hold even if p is faulty (under the lemma's conditions), they clearly hold if p is correct. Hence, we still need to prove *Strong Termination* and *Separation.* To prove *Strong Termination*, note that if p is correct, and it has not invoked $\mathcal{Q}(p)$ for $3 \cdot \Delta_{max}$ time units, then when it does invoke $\mathcal{Q}(p)$, all correct nodes will accept the message (*Initiator*, p, $start\_\mathcal{Q}$) and hence, according to item 2 of the "*Timeliness-Agreement*" property of ss-Byz-Agree (see Appendix A), all correct nodes will terminate within $3 \cdot d$ time units following p's invocation of ss-Byz-Agree $(p, start\_\mathcal{Q})$ and join the execution of $\mathcal{Q}(p)$. *Separation* follows from the conditions of the lemma. □

From the above lemmas, we conclude that after $4 \cdot \Delta_{max} + \Delta_{ss-\text{BYZ-AGREE}}$ time units, SS-BYZ-Q behaves according to $\mathcal{Q}$'s properties. Setting $\Delta_{\mathcal{Q}} := 4 \cdot \Delta_{max} + \Delta_{ss-\text{BYZ-AGREE}}$ satisfies the claim that if the system has been coherent for $\Delta_{\mathcal{Q}}$ time units, then the properties of $\mathcal{Q}$ hold.

Since SS-BYZ-Q implements $\mathcal{Q}$'s properties correctly, in the rest of the paper we will use SS-BYZ-Q and $\mathcal{Q}$ interchangeably.

6 Constructing the Erratic-Pulser Algorithm

The ERRATIC-PULSER algorithm (Figure 2) is written in an event-driven fashion; that is, it is continuously executed in the background and no explicit initialization is needed. The algorithm requires invoking two $\mathcal{Q}$ instances per node ($\mathcal{Q}_{start}$ and $\mathcal{Q}_{end}$). In addition, each node has three timers TIMER$_{start}$, TIMER$_{end}$ and TIMER$_{main}$ with elapsed time of CYCLE$_{start}$, CYCLE$_{end}$ and CYCLE$_{main}$, respectively. When TIMER$_{start}$ or TIMER$_{end}$ elapse, an instance of SS-BYZ-Q is invoked ($\mathcal{Q}_{start}$ for TIMER$_{start}$ and $\mathcal{Q}_{end}$ for TIMER$_{end}$). TIMER$_{main}$ is used to determine the value of $WantToPulse$, which is used as the input value for $\mathcal{Q}_{start}$ and $\mathcal{Q}_{end}$. If TIMER$_{main}$ is elapsed, then $WantToPulse := 1$, and once TIMER$_{main}$ is reset, $WantToPulse := 0$ until it elapses again.

The intuition behind the algorithm is that $WantToPulse$ determines when p is willing to invoke a pulse. Once all correct nodes have $WantToPulse = 1$, the next time a SS-BYZ-Q instance is invoked, all of them will invoke pulses.

Remark: Notice that there is a difference between TIMER$_{start}$, TIMER$_{end}$ and TIMER$_{main}$. TIMER$_{start}$, TIMER$_{end}$ are timers that when they elapse, an event occurs, and the algorithm performs some action. These timers are always set, that is, once they elapse, they are reset immediately. TIMER$_{main}$, on the other

```
Algorithm Erratic-Pulser                                   /* executed at node p */
                                     /* the Qs are executed in the background */
                               /* the input value v_q for each Q instance, is the
                               value of WantToPulse at the time q joins Q */
 1. when TIMER_start elapses
        reset TIMER_start with CYCLE_large;
        reset TIMER_end;
        invoke $\mathcal{Q}_{start}(p)$;
 2. when TIMER_end elapses
        reset TIMER_end with CYCLE_large;
        invoke $\mathcal{Q}_{end}(p)$;
 3. $WantToPulse := 1$ if TIMER_main has elapsed, and $WantToPulse := 0$, otherwise;
 4. on returning from either $\mathcal{Q}_{start}(q)$ or $\mathcal{Q}_{end}(q)$ for some $q$ with value $V = 1$
    (a) invoke a pulse;
    (b) reset TIMER_main;
    (c) reset TIMER_start;

cleanup:
   if a TIMER is set with invalid value (below 0 or above its maximal value),
   reset it; for TIMER_main, 0 is a valid value;
```

Fig. 2. An algorithm achieving basic synchronized PULSEing

hand, will remain in its elapsed state until it is reset. That is, Line 3 is not executed only when TIMER_{main} elapses, but rather it is executed continuously. In a sense, when q wants to read its $WantToPulse$ variable value, it checks whether TIMER_{main} has elapsed; if so then it considers $WantToPulse = 1$, otherwise it reads $WantToPulse = 0$.

The following are the values of the constants used in ERRATIC-PULSER.

$$\text{CYCLE}_{start} := \text{CYCLE}_{main} := Cycle - \Delta_{max} - \Delta_{min};$$
$$\text{CYCLE}_{end} = \Delta_{min} - 10 \cdot d;$$
$$\text{CYCLE}_{large} := 2 \cdot (\Delta_{max} + \text{CYCLE}_{start} + \text{CYCLE}_{end}).$$

Note: CYCLE_{main} needs to be larger than $\Delta_{max} + 9 \cdot d$ time units, hence, $Cycle$ must be larger than $2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$ time units.

7 Erratic-Pulser's Correctness Proofs

Definition 7. *A correct node p* ***pulses-in-unison****, there is a pulsing point t, such that p invokes a pulse near t each time that p invokes a pulse. The system* ***pulses-in-unison****, if for every correct node p , p* ***pulses-in-unison***

Remark 3. The definition of "near t" implies that if p **pulses-in-unison** then each time p invokes a pulse there is a time interval $[t_1, t_2]$ such that $|t_2 - t_1| \leq 3 \cdot d$ and each correct node (including p) invokes a pulse within this interval. This also implies that if there exists a correct node p that **pulses-in-unison** then the system **pulses-in-unison**.

Lemma 4. *Once the system has been coherent for $\Delta_{\mathcal{Q}}$ time, the system pulses-in-unison.*

Proof. According to Lemma 2 and Lemma 3 (in Section 5), once the system has been coherent for $\Delta_{\mathcal{Q}}$time units, all copies of SS-BYZ-Q behave according to the requirements of $\mathcal{Q}$. This means that all correct nodes see the same output values. Since a correct node invokes a pulse only in accordance with the output of a $\mathcal{Q}$, if some correct node invokes a pulse, then within $3 \cdot d$ time units from its pulse, all correct nodes will also invoke pulses. This means that every correct node pulses-in-unison, which means that the system pulses-in-unison. □

The following lemma proves that a correct node will eventually invoke a pulse. The previous lemma claims that after some time, **if** a correct node invokes a pulse, then all the correct nodes invoke pulses.

Lemma 5. *Eventually some correct node will invoke a pulse. This happens no later than $\Delta_{\mathcal{Q}} + \Delta_{max} + \text{CYCLE}_{large} + \text{CYCLE}_{main} + 3 \cdot d$ time units after the point at which the system becomes coherent.*

Proof. Consider the system $\Delta_{\mathcal{Q}}$ after it becomes coherent: If a correct node invokes a pulse, the lemma holds. Otherwise, after CYCLE_{main} time units, all correct nodes will have $WantToPulse$ as 1. Eventually, after no more than CYCLE_{large},

TIMER_{start} at some correct p will expire, which will initiate $\mathcal{Q}_{start}(p)$, that terminates no more than $\Delta_{max} + 3 \cdot d$ time units afterwards (by *strong termination*), and will have the output value $V = 1$ (since all correct nodes had the input value of $v = 1$). By line 4, of the Erratic-Pulser, p will invoke a pulse. □

Lemma 6. *Once the system pulses-in-unison, let t_1 be a time unit at which a correct node p invokes a pulse. Let t_2 be the last time at which p invokes a pulse in the interval $[t_1, t_1 + \Delta_{max} + 3 \cdot d]$. p does not invoke a pulse in the interval $[t_2, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$. p invokes a pulse at some time t_3, where $t_3 \leq t_2 + \text{CYCLE}_{main} + 6 \cdot d + \Delta_{max}$.*

Proof. According to the lemma's assumption the system pulses-in-unison. Hence, when p invokes a pulse at t_1 all correct nodes invoke pulses before time $t_1 + 3 \cdot d$. Define $[t_s, t_e]$ to be the time interval in which all correct nodes have invoked a pulse, such that $t_1 \in [t_s, t_e]$ and $t_e - t_s \leq 3 \cdot d$. All correct nodes execute lines 4.a, 4.b and 4.c during the interval $[t_s, t_e]$. Therefore, the correct nodes' timers $\text{TIMER}_{main}, \text{TIMER}_{start}$ are reset. Hence, after t_e all correct nodes' values of $WantToPulse$ are 0, and hence any correct node that joins any $\mathcal{Q}$ instance after t_e has an input value $v_q = 0$. This holds until TIMER_{main} elapses at some correct node, that is until $t_s + \text{CYCLE}_{main}$. In other words, no correct node joins any $\mathcal{Q}$ instance in the interval $[t_e, t_s + \text{CYCLE}_{main}]$ with input value of 1.

By definition, t_2 is the last time that p invoked a pulse in the interval $[t_1, t_1 + \Delta_{max} + 3 \cdot d]$. Hence, after $t_2 + 3 \cdot d$ all correct nodes have invoked pulses, and hence have $WantToPulse$ as 0 for at least $\text{CYCLE}_{main} - 3 \cdot d$ time units. Therefore, in the interval $[t_2 + 3 \cdot d, t_2 + \text{CYCLE}_{main} - 3 \cdot d]$ no correct node joins any instance of$\mathcal{Q}$ with an input value of 1. Since $\text{CYCLE}_{main} \geq \Delta_{max} + 9 \cdot d$, it holds that $t_2 + 3 \cdot d \in [t_e, t_s + \text{CYCLE}_{main}]$, hence during the time interval $[t_e, t_2 + \text{CYCLE}_{main} - 3 \cdot d]$ no correct node joins any $\mathcal{Q}$ instance with an input value of 1. Hence, in the time interval $[t_e + \Delta_{max}, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$ no correct node invokes a pulse. Since $t_1 + 3 \cdot d + \Delta_{max} \geq t_e + \Delta_{max}$ and since t_2 is the last time p invoked a pulse before $t_1 + 3 \cdot d + \Delta_{max}$, it holds that p did not invoke a pulse in the time-interval $[t_2, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$.

Lastly, after $t_2 + 3 \cdot d$ time units have elapsed, all correct nodes have reset TIMER_{main} and TIMER_{start}. Since $\text{CYCLE}_{main} = \text{CYCLE}_{start}$, we have that when TIMER_{start} elapses at some correct node, then $WantToPulse = 1$ at that correct node. By time $t_2 + 3 \cdot d + \text{CYCLE}_{main}$ all correct nodes have set their value of $WantToPulse$ to 1. Consider the last correct node to do so, it starts executing $\mathcal{Q}$, as instructed by Line 1 (the elapsing of TIMER_{start}), and since the input values of all correct nodes are 1, it terminates with an output value of 1. This happens no later that $t_2 + 6 \cdot d + \text{CYCLE}_{main} + \Delta_{max}$. That is, p invokes a pulse no later than $t_3 = t_2 + 6 \cdot d + \text{CYCLE}_{main} + \Delta_{max}$. □

Note that the above lemma shows that a correct node p invokes a pulse in some pattern. That is, after each pulse there is a period of uncertainty, and afterwards there is a long period of no PULSEing. Then p invokes a pulse again, and so on. Note that in the above lemma, the TIMER_{end} was never used; it will be used in the following lemma, which claims the "uncertainty" period is of a constant length, and at the end of it a pulse is invoked. This lemma will give us the required properties, since with it the PULSEing pattern of a correct node p will be constant, and since the system pulses-in-pattern, the entire PULSEing pattern of the system will be determined.

Lemma 7. *Consider t_1, t_2 to be as defined in Lemma 6. Once the system pulses-in-unison, the value of t_2 is in the interval $[t_1 + \Delta_{min} - 13 \cdot d, t_1 + \Delta_{max} + 3 \cdot d]$.*

Proof. According to Lemma 6, after the last pulse there is a silent period during which TIMER$_{main}$ and TIMER$_{start}$ tick away. Once they elapse (they both elapse together), the following happens. First, $WantToPulse$ is set to 1 (until the next pulse). Second, TIMER$_{end}$ is reset; and third, a $\mathcal{Q}$ instance is initiated.

Since the system pulses-in-unison, after the last pulse (at time t_2) all correct nodes reset TIMER$_{start}$. This means that TIMER$_{start}$ elapses at all correct nodes within a $3 \cdot d$ interval, which implies that TIMER$_{end}$ elapses at all correct nodes within a $3 \cdot d$ interval. Consider the last node q to have had TIMER$_{start}$ elapse (at time t'). No correct node has had TIMER$_{start}$ elapse before time $t' - 3 \cdot d$, hence at time $t' - 3 \cdot d + \Delta_{min}$ all correct nodes still have $WantToPulse = 1$ (no $\mathcal{Q}$ instance managed to finish yet). Therefore, when q's TIMER$_{end}$ elapses at time $t' - 10 \cdot d + \Delta_{min}$ (since CYCLE$_{end} = \Delta_{min} - 10 \cdot d$), all correct nodes are guaranteed to join q's $\mathcal{Q}(q)$ instance with input value of 1, and hence in time interval $[t' + \Delta_{min} - 10 \cdot d + \Delta_{min}, t' + \Delta_{min} - 7 \cdot d + \Delta_{max}]$ q invokes a pulse.

t'_1, t'_2 represent the same meaning as t_1, t_2, just for the "PULSEing cycle" that starts after t_2. Consider t'_1 to be the first time value at which a correct q' invokes a pulse after t_2 (note that according to Lemma 6, $t'_1 \in [t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}, t_3]$). For q' to invoke a pulse, at least one correct node should have $WantToPulse = 1$ in the interval $[t'_1 - \Delta_{max}, t'_1 - \Delta_{min}]$. Since t'_1 is the first time some node invokes a pulse, and since $t' - 3 \cdot d$ is the first time some correct node has $WantToPulse = 1$ in the current "PULSEing cycle", we have that $t' \in [t'_1 - \Delta_{max}, t'_1 - \Delta_{min} + 3 \cdot d]$. Therefore, q invokes a pulse due to TIMER$_{end}$'s elapsing is in the interval $[t'_1 - \Delta_{max} + \Delta_{min} - 10 \cdot d + \Delta_{min}, t'_1 - 4 \cdot d + \Delta_{max}]$. Since $\Delta_{max} = \Delta_{min} + 3 \cdot d$, we have that the above time interval is $[t'_1 - 13 \cdot d + \Delta_{min}, t'_1 - 4 \cdot d + \Delta_{max}]$. This implies that $t'_2 \geq t'_1 - 13 \cdot d + \Delta_{min}$. Which means that starting from the first pulse, the next "PULSEing cycle" will have that $t_2 \geq t_1 - 13 \cdot d + \Delta_{min}$. □

The above lemmata show that if the system has been coherent for $\Delta_{\text{Erratic-Pulser}} := \Delta_{\mathcal{Q}} + 3 \cdot Cycle$, then all correct nodes invoke pulses together, and they have a distinctive PULSEing pattern: say a node invokes a pulse at some time t_1; during the interval $[t_1, t_1 + \Delta_{min} - 13 \cdot d]$ there could be some additional pulses, then during the interval $[t_1 + \Delta_{min} - 13 \cdot d, t_1 + \Delta_{max} + 3 \cdot d]$ at least one pulse is invoked, and then there is an interval of at least CYCLE$_{main} + \Delta_{min} - 3 \cdot d$ time units during which no pulse is invoked, and finally, within the next $12 \cdot d$ there will be new pulses and a new PULSEing "cycle" will start.

Note that the length of this "cycle" is bounded from below by $\Delta_{max} +$ CYCLE$_{main} + \Delta_{min}$, and bounded from above by $\Delta_{max} +$ CYCLE$_{main} + \Delta_{min} + 12 \cdot d$. In addition, notice that each such "cycle" starts with a "possibly noisy period" of length $\Delta_{max} + 3 \cdot d$, and ends with a "quiet period" of CYCLE$_{main} + \Delta_{min} - 3 \cdot d$

time. Since $\text{CYCLE}_{main} \geq \Delta_{max} + 9 \cdot d$, we have that the quiet period is at least Δ_{min} longer than the first period. This remark is important for the next section.

8 Creating the Balanced-Pulser

The above ERRATIC-PULSER synchronizes the correct nodes into some repetitive PULSEing pattern. However, to solve the PULSE-synchronization problem, an additional algorithm is required. We now present the BALANCED-PULSER algorithm, which starting from an arbitrary state, shortly after the system is coherent, produces pulses approximately once in a *Cycle*, despite the permanent presence of *Byzantine* nodes.

```
Algorithm **Balanced-Pulser**                          /* executed at node p */

1. execute an instance $\mathcal{A}$ of ERRATIC-PULSER in the background;
2. when $\mathcal{A}$ produces a pulse
       if $\mathcal{A}$ has not produced a pulse for at least $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ time,
   invoke a pulse.
```

Fig. 3. An algorithm solving the PULSE-synchronization problem

Theorem 1. *Algorithm* BALANCED-PULSER *solves the* PULSE-*synchronization problem in a self-stabilizing and* Byzantine *tolerant manner.*

Proof. Once the system is coherent for Δ_Q time, by Lemma 4 the system pulses-in-unison. Hence, each time a correct node sees $\mathcal{A}$ PULSEing, within $3 \cdot d$ time units all other correct nodes see the same. In addition, by Lemma 6 and Lemma 7, the pulses that $\mathcal{A}$ produces have a distinct pattern. That is, a pulse, then a period of length $\Delta_{max} + 3 \cdot d$ with possible pulses and a period of length $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ with no pulses. Then, within $12 \cdot d$, another pulse.

If a correct node hasn't heard $\mathcal{A}$ producing a pulse for $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ time, it must mean that $\mathcal{A}$ has undergone the "quiet period", since the "possible noisy period" is short. Hence, the next PULSE produced must be the beginning of a new "cycle". Therefore, all correct nodes invoke pulses together in BALANCED-PULSER. In addition, all correct nodes invoke pulses only at the beginning of a "cycle", and they invoke pulses $3 \cdot d$ apart of each other. Since all correct nodes invoke pulses only at the beginning of a "cycle", we need only to argue about the length of the "cycle".

According to the lemmata in the previous section, the difference between the "long-cycle" and the "short-cycle" is at most $12 \cdot d$ time units. Setting $\text{CYCLE}_{min} := \Delta_{max} + \text{CYCLE}_{main} + \Delta_{min}$, and $\text{CYCLE}_{max} := \text{CYCLE}_{min} + 12 \cdot d$, we have that the system is in a synchronized_pulsing_state. That is, starting from any state, the system reaches a synchronized_pulsing_state; this proves *convergence*. In addition, according to the previous section, the PULSEing pattern remains as long is the system is coherent, thus *closure* also holds. □

The convergence time of BALANCED-PULSER is the same as the convergence of ERRATIC-PULSER + *Cycle*; that is, $\Delta_{\mathcal{Q}} + 4 \cdot Cycle$ time units.

9 Discussion

Time complexity: Once the system has become coherent, the BALANCED-PULSER algorithm converges in $O(f) + O(Cycle)$ time.

Message complexity: The BALANCED-PULSER algorithm executes $2 \cdot (n - f)$ SS-BYZ-Q instances each *Cycle*. Since SS-BYZ-Q has $O(f \cdot n^2)$ message complexity, then the message complexity becomes $O(f \cdot n^3)$ per CYCLE.

Executing fewer ss-Byz-Q: The main feature of ERRATIC-PULSER is that "eventually there will be a correct node that executes SS-BYZ-Q". As presented, ERRATIC-PULSER has each correct node execute SS-BYZ-Q once its timers elapse. The algorithm can be adapted such that only $f + 1$ of the nodes (predetermined and considered as part of the program, not memory) can invoke $\mathcal{Q}$. Since there will always be a correct node that invokes $\mathcal{Q}$, the correctness of the algorithm holds. This reduces the message complexity to $O(f^2 \cdot n^2)$.

Clock synchronization: The Digital clock synchronization problem consists of having all correct nodes agree on an integer value that progresses linearly with time. To build a digital clock synchronization algorithm using a PULSEing algorithm, all that is needed is to execute an agreement on the next clock's value each time a pulse is invoked. Setting the cycle of the PULSE to be long enough for the agreement algorithm to terminate, ensures that all correct nodes will agree on the clock value, and advance it appropriately. Note that the convergence time of such an algorithm is the convergence time of the underlying PULSE algorithm, plus an additional $cycle_{max}$time units. See [4] for a more detailed discussion.

Arbitrary *Cycle* values: According to the constraints of the BALANCED-PULSER algorithm, *Cycle* must be larger than $2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$ time units. For the purpose of clock synchronization it is enough to have *Cycle* in the order of Δ; for example, $Cycle = 5 \cdot \Delta$ would suffice for a linear convergence of the digital clock synchronization algorithm.

However, if one wishes to use PULSEing for other reasons, it is desired to be able to PULSE in any *Cycle*. To PULSE every $Cycle' < 2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$, set *Cycle* to be some multiplication of $Cycle'$ such that it falls within the constraints. Now, each time that BALANCED-PULSER produces a pulse, reset a timer of $Cycle'$ long, and when it elapses, invoke a pulse and reset the timer again. The PULSEing pattern will be a pulse by BALANCED-PULSER every *Cycle* and $Cycle/Cycle'$ pulses in between. This scheme is similar to what is done in [18]. The tricky part is to notice that if a pulse is invoked less than $Cycle'$ time before a PULSE by BALANCED-PULSER then the timer for the "small" pulses is reset, and hence a pulse is invoked again only in $Cycle'$ time units. Note that the difference between $cycle_{max}$ and $cycle_{min}$ is still $12 \cdot d$, hence there is no meaning to having $Cycle' \leq 12 \cdot d$. That is, $Cycle'$ should always be larger than $12 \cdot d$ time units.

References

1. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
2. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
3. Daliot, A., Dolev, D.: Self-stabilization of byzantine protocols. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, Springer, Heidelberg (2005)
4. Daliot, A., Dolev, D., Parnas, H.: Linear time byzantine self-stabilizing clock synchronization. In: Papatriantafilou, M., Hunel, P. (eds.) OPODIS 2003. LNCS, vol. 3144, Springer, Heidelberg (2004), A corrected version appears in http://arxiv.org/abs/cs.DC/0608096
5. Sakurai, Y., Ooshita, F., Masuzawa, T.: A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 283–298. Springer, Heidelberg (2005)
6. Nesterenko, M., Arora, A.: Local tolerance to unbounded byzantine faults. In: IEEE SRDS 2002, pp. 22–31 (2002), citeseer.ist.psu.edu/nesterenko02local.html
7. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: 22nd Int. Conference on Distributed Computing Systems (2002)
8. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. Journal of the ACM 51(5), 780–799 (2004)
9. Hoch, E.N., Dolev, D., Daliot, A.: Self-stabilizing byzantine digital clock synchronization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, Springer, Heidelberg (2006)
10. Daliot, A., Dolev, D.: Self-stabilizing byzantine agreement. In: PODC 2006. Proc. of the Twenty-fifth ACM Symposium on Principles of Distributed Computing, Denver, Colorado (July 2006)
11. Liskov, B.: Practical use of synchronized clocks in distributed systems. In: Proceedings of 10th ACM Symposium on the Principles of Distributed Computing, ACM Press, New York (1991)
12. Arora, A., Dolev, S., Gouda, M.G.: Maintaining digital clocks in step. Parallel Processing Letters 1, 11–18 (1991)
13. Dolev, S.: Possible and impossible self-stabilizing digital clock synchronization in general graphs. Journal of Real-Time Systems 12(1), 95–107 (1997)
14. Dolev, S., Welch, J.L.: Wait-free clock synchronization. Algorithmica 18(4), 486–511 (1997)
15. Papatriantafilou, M., Tsigas, P.: On self-stabilizing wait-free clock synchronization. Parallel Processing Letters 7(3), 321–328 (1997)
16. Herman, T.: Phase clocks for transient fault repair. IEEE Transactions on Parallel and Distributed Systems 11(10), 1048–1057 (2000)
17. 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)
18. Dolev, D., Hoch, E.N.: On self-stabilizing synchronous actions despite byzantine attacks. In: DISC2007. LNCS, vol. 4731, pp. 193–207. Springer, Heidelberg (2007)
19. Fischer, M.J., Lynch, N.A., Merritt, M.: Easy impossibility proofs for distributed consensus problems. Distributed Computing 1, 26–39 (1986)
20. Daliot, A., Dolev, D.: Self-stabilizing byzantine pulse synchronization. Technical report, Cornell ArXiv, (August 2005), http://arxiv.org/abs/cs.DC/0608092
21. Freiling, F.C., Ghosh, S.: Code stabilization. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, Springer, Heidelberg (2005)

A The Use of ss-Byz-Agree

The mode of operation of the ss-Byz-Agree, a self-stabilizing Byzantine agreement protocol presented in [10] is as follows: A node that wishes to initiate agreement on a value does so by disseminating an initialization message to all nodes that will bring them to (explicitly) invoke the ss-Byz-Agree protocol. Nodes that did not invoke the protocol may join in and execute the protocol in case enough messages from other nodes are received during the protocol. The protocol requires correct initiating nodes not to disseminate initialization messages too often. In the context of the current paper, an (*Initiator*, p, *) message serves as the initialization message.

When the protocol terminates, the ss-Byz-Agree protocol returns (in each correct node q) a triplet (p, m, τ_q^p), where m is the agreed value that p has sent. The value τ_q^p is an estimate, on the receiving node q's local clock, as to when node p has sent its value m. We also denote it as the "recording time" of (p, m). Thus, a node q's decision value is $\langle p, m, \tau_q^p \rangle$ if the nodes agreed on (p, m). If the sending node p is faulty then some correct nodes may agree on $(p, \perp)$, where $\perp$ denotes a non-value, and others may not invoke the protocol at all. The function $rt(\tau_q)$ represents the time at which the local clock of q reads τ_q.

The ss-Byz-Agree protocol satisfies the following typical Byzantine agreement properties:

Agreement: If the protocol returns a value ($\neq\perp$) at a correct nodes, it returns the same value at all correct nodes;
Validity: If all correct nodes are triggered to invoke the protocol ss-Byz-Agree by a value sent by a correct node p, then all correct nodes return that value;
Termination: The protocol terminates within a finite time;

The proof uses the following properties of the ss-Byz-Agree protocol ([10]):

Timeliness-Agreement Properties:

1. (agreement) For every two correct nodes q and q' that decide $\langle p, m, \tau_q^p \rangle$ and $\langle p, m, \tau_{q'}^p \rangle$ at local times τ_q and $\tau_{q'}$ respectively: $|rt(\tau_q) - rt(\tau_{q'})| \leq 3d$.
2. (validity) If all correct nodes invoked the protocol in the interval $[t_0, t_0 + d]$, as a result of some initialization message containing m sent by a correct node p that spaced the sending by at least $6d$ from the completion of the last agreement on its message, then for every correct node q, the decision time τ_q, satisfies $t_0 - d \leq rt(\tau_q) \leq t_0 + 3d$.
3. (termination) The protocol terminates within Δ time units following its explicit invocation, and within $\Delta + 7d$ time units, in case it was not explicitly invoked[7].
4. (separation) Let q be any correct node that decided on any two agreements regarding p at local times τ_q and $\bar{\tau}_q$, then $t_2 + 5d < \bar{t}_1$ and $rt(\tau_q) + 5d < \bar{t}_1 < rt(\bar{\tau}_q)$, where t_2 is the latest time at which a correct node invoked ss-Byz-Agree in the earlier agreement, and $\bar{t}_1$ is the earliest time that ss-Byz-Agree was invoked by a correct node in the later agreement.

[7] $\Delta := 7(2f + 3)d$.

Magnifying Computing Gaps[⋆] Establishing Encrypted Communication over Unidirectional Channels (Extended Abstract)

Shlomi Dolev[1], Ephraim Korach[2], and Galit Uzan[1]

[1] Department of Computer Science,
Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
{dolev,poucema}@cs.bgu.ac.il
[2] Department of Industrail Enginering,
Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
korach@bgu.ac.il

Abstract. Consider two, not necessarily identical, powerful computers or computer-grids connected by a unidirectional communication link that should transfer a long stream of information in the presence of a listening adversary that is slightly weaker. We present schemes that enhance the computation strength gap between the powerful computers and the adversary. In other words, the gap between the amount of information decrypted by the adversary and the information decrypted by the receiver grows with time.

We also suggest schemes based on the shortest vector problem in which only the receivers are computationally powerful. The scheme is self-stabilizing in the sense that it can establish a security level without relying on (previously distributed private keys that are part of) the state. The iterative nested approach suggested, can be used for enhancing the security of the classical protocol of Ralph Merkle [19]. Several applications for sensor networks and for secure communication with survivors are suggested.

Keywords: unidirectional encryption, combinatorial optimization problems.

1 Introduction

Truly unidirected encryption. Modern cryptography is based on complexity theory ensuring that the resources required to reveal the secret, i.e., computing time and/or space, are too big. One such very useful example is public key cryptography. However, public key schemes require two way communication — the sender should know in advance the public key of the receiver. One would like

[⋆] Partially supported by the Lynne and William Frankel enter for computer science. The first author is also supported by the Rita Altura trust chair in computer science.

to encrypt a message without having any information concerning the recipients. In other words, we would like to have an encryption scheme that can be used in a unidirected communication link.

We start with an intuitive example. Imagine a teacher in front of a class that would like to convey a (long) message only to the smartest students in class without identifying them. Also consider the case where the teacher may allow the content of each portion of the message to be revealed some period of time after message transmission. The teacher uses a (preferably randomly chosen and hard) puzzle, so that the smartest students can solve it faster than the rest. The teacher uses the solution of the puzzle to encrypt the first portion of a message he/she would like to convey. The group of slower students may collaborate to solve a puzzle, thus the method is useful as long as such a collaboration still requires a longer time for obtaining a solution in relation to any of the smartest students. To continue conveying the next portions, the teacher may repeat the procedure used for the first portion. However, the slower group members may distribute the different puzzles to be solved independently by the slower members. Still the teacher would like to avoid the possibility of revealing the next message portion as fast as the first was revealed. To do so, the teacher also encrypts the description of the next puzzles using the solutions of the previously sent puzzles. Thus, over time, the number of puzzles solved by the smartest students, but not yet solved by the other students, grows.

Applications of the methods may include cases in which a satellite would like to broadcast a file with instructions to one or many on-ground units. Another scenario is when a sensor that (repeatedly) communicates a secure message (in the style of S.O.S. with more details concerning location, status. etc.) that should be understood only by a computationally strong entity (satellite). The schemes proposed here are designed for the case in which there is communication in one direction, from a sender to a receiver. The adversary can listen to the transmitted information along the transmission path, but cannot change it. Self-stabilization [5,6] has influenced security and cryptography in designing proactive security schemes [7,8] that start in a consistent state, where private keys are part of the states. In this work we demonstrate that, under some assumptions, security may be established from an arbitrary state. A somewhat similar approach has been suggested in [19], and is essentially in the heart of public key cryptography that assumes two-way communication. This work's main focus is establishing security for directed communication channels assuming the adversary is computationally restricted.

Must rely on computation gap. Clearly, a listening adversary can reveal the clear-text if it has the same capabilities as the recipients since we assume that the encryption protocol is known to all listeners. To overcome the above obvious limitation, we propose to equip each receiver with a powerful computer or a computer-grid that is able to compute, maybe only a specific problem, rapidly. Note that the computation gap abstracts particular limitations either in memory (as assumed in [11,20,23]) or in processing or in communication. Any combination of such resources/capabilities that enforces the computation gap fits our schemes. One may consider a computer with a computing primitive that is capable of solving a version of the multicriteria traveling salesman problem for big enough input, see e.g., [15,18,22]. In this case, a listening adversary that is not equipped with these or equivalent powerful computers/computer-grids, may need much more time to decrypt the secret.

Choosing a computation task. As an example, we suggest a scheme based on the lattice problems defined in [1,3,12], for which a random instance is proved to be as hard as a worst case instance of a certain computational problem on lattices (which is believed to be hard on the worst-case). The choice of the lattice problem is due to lack of a prove that existing methods that use modulo operations and primes are hard in average though in practice there are parameters that are believed to produce such instances. The parameters of the lattice problems may be tuned to fit the desired hardness of finding a solution (by using exhaustive search). We note that even though the lattice-based problems may be hard to be solved on average, there are easy to solve instances. The way we magnify the computation gaps ensures that these easy to solve instances will be averaged with other instances, and therefore will not give the adversary a real benefit.

The approach in a nutshell. The idea is to use the solution of the first problem instance to mask the description of the second problem instance, and in general to use the previous solutions of the problem instances to mask the description of the current instance and the current secret.

The rest of the paper is organized as follows. In the next section we present the settings for the competition between the secure protocol participants, namely the sender and the receiver, and the adversary. Section 3 discusses the complexity of the average instance and presents a solution based on a lattice problem. The case in which only the receiver is superior to the adversary is addressed in Section 4. Concluding remarks appear in Section 5.

2 The Settings

We assume that we have a *fast*, possibly optical, device, or a computing system, that can perform a single combinatoric task $\mathcal{T}$ within x time units. We compete with an adversary that has a constant number k of slow computers that each can perform $\mathcal{T}$ in at least $y > x$ time units on the average. The computation of $\mathcal{T}$ is either by a single slow computer or by several computers, including the overhead of distributing the input and collecting output. Assume that we repeatedly compute tasks $\mathcal{T}$. One can try to compute each instance of the problem by a different set of computers, and by doing so to eliminate the waiting time for the result of the predecessor instances of the problem. Roughly speaking, if each iteration of the problem does not depend on the previous results, and the number of slow computers (k) is large enough, we may perform the entire computation in a total time that is only $y - x$ longer than the time it takes for the fast computer.

We may take it to the extreme and compute the output for all the possible inputs having a computer for each possible input as a preprocessing stage. The above "solution" may require an unreasonable amount of computing and/or amount of storage. In the sequel we design schemes in which the result of previous iterations is an input for the current iteration. Thus, the number of task instances that one needs to compute in order to continue the computation in a non serial manner, is very large.

Consider a sequence of encrypted secrets $s_1, s_2, \cdots, s_l$, for which the sequence of the respective (decrypted) clear-texts is $cts_1, cts_2, \cdots, cts_l$. Assume that s_1, $s_2, \cdots, s_l$ are sequentially sent from the sender to the receiver.

Definition 1. *An infinite iterative sequential instance computation is a computation in which in every iteration a secret is sent and revealed (using the entire information gained from the previous iterations). We define a time unit to be the maximal time required for revealing cts_i, the clear-text of s_i, given the results of the processes that revealed the clear-texts of $s_1, s_2, \cdots, s_{i-1}$.*

Definition 2. *We define the fan-out to be the number of problem instances that should be computed if computations for revealing cts_{i+1} starts, by additional computers, before the current iteration that reveals cts_i is completed. Where one of these computations is identical to, or as fast as, the computation that is performed when the process of revealing $cts_1, cts_2, \cdots, cts_i$ is completed.*

A large fan-out implies a requirement for more computing power and memory. Interestingly, the *fan-out* notion is meaningful when there is no useful computation for revealing cts_{i+1}, before knowing cts_i. For example, consider the suggestion to use the scheme presented in [21] to encrypt a long file by using a function over all previous secrets, which are file portions, and the current secret as the new secret. In other words, this is a double encryption scheme where first, a function over the previous secrets and the new secret, $f(s_1, s_2, \cdots, s_{i-1}, s_i)$, is used to encrypt the new secret, s_i, and then the scheme presented in [21] is used to encrypt the result of f. Thus, a non-trivial choice of f will force the adversary to decrypt the new secret, which is a file portion, only after all previous secrets were decrypted. In other words, one needs to reveal all previous secrets in order to reveal the current secret. For example, $f(s_1, s_2, \cdots, s_{i-1}, s_i)$ can be a bitwise *xor* of s_i and s_{i-1}. Note that if f is defined over all previous secrets then the fan-out does not become larger, since the only missing result when we start computing s_i is s_{i-1}.

At first glance the above scheme seems to have a large fan-out, the number of possible file portion contents. Only in this case we may reveal s_{i+1} almost together with s_i using the preprocessing for each possible value of s_i. However, there is still a useful *pipelined computation* for revealing s_{i+1} that we may start before revealing the clear-text of s_i. Decrypting s_{i+1} can be started in a pipelined fashion by revealing the result of $f(s_1, s_2, \cdots, s_i, s_{i+1})$ in parallel with the computation that reveals s_i. Once $f(s_1, s_2, \cdots, s_{i+1})$ is revealed, the inexpensive $xor(cts_i, f(s_1, s_2, \cdots, s_{i+1}))$ may rapidly reveal cts_{i+1}.

Another important aspect of the problem is the number k of computers (that may compute $\mathcal{T}$ in y time units) that the adversary has for encryption. If k is greater than or equal to the fan-out, then the adversary may always reveal the secret almost together with the receiver. Thus, we assume that k is much smaller

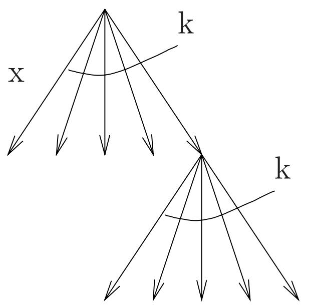


Fig. 1. k fan-out of x time unit computations

than the fan-out. In other words, we prefer schemes that have greater fan-out, which in turn can cope with larger k.

3 Worst Case Average Case Equivalent Lattice Problem

In this section we present a scheme based on the lattice shortest vector problem [1,3,12]. A lattice is a set of points in space such that every point is a combination $\Sigma_{i=1}^{l} a_i v_i$, where a_i are integers and v_i, $1 \le i \le l$, are l independent vectors; each v_i is of dimension (at least) l. Finding the point of the lattice, that is not the origin but is closest to the origin, is called the (SVP) shortest vector problem (which is proved to be NP-hard for polynomial random reduction [2]).

An approximation problem is defined such that a solution for a randomly chosen instance of the approximation problem implies a solution for the worst case instance of three famous worst-case problems related to the shortest vector problem of a lattice. This reduction and proof were introduced in [1]. The approximation problem suggested in [1] is now defined (in fact we follow the description of Ajtai's random lattice problem in [12]).

For a given integer n we choose c_1 and c_2 and compute $m = c_1 n \log n$ and $q = n^{c_2}$. c_1 and c_2 are chosen such that m and q are integers and (1) $c_2 \ge 7$ and, (2) $n \log q < m \le \frac{q}{2n^4}$ and (3) $m < n^2$.[1]

The input of the problem is a set of m vectors $\lambda = (v_1, v_2, \ldots, v_m)$ of length n and an integer q. The $m-1$ vectors $v_1, v_2 \ldots, v_{m-1}$ are chosen randomly from the set of all vectors $(x_1, x_2, \ldots, x_n)$, where $0 \le x_i \le q-1$. Then $m-1$ values $\delta_1, \delta_2, \ldots, \delta_{m-1}$ are chosen randomly in $\{0,1\}$ and v_m is computed to be $v_m = -\sum_{i=1}^{m-1} \delta_i \cdot v_i \ mod \ q$.

A set $\Lambda(\lambda, q)$ is defined to be the set of all vectors $h = (h_1, h_2, \ldots, h_m)$ for which $\sum h_i \cdot v_i \equiv 0 \ mod \ q$. The length of a vector h is defined as the usual Euclidean norm $\|h\| = (h_1^2 + h_2^2 + \ldots + h_m^2)^{1/2}$. Given λ, q as an input, the problem is to find a nonzero vector h with length of at most n, $\|h\| \le n$. Note that by the construction of v_m $\Lambda(\lambda, q)$ includes a vector $h = (\delta_1, \delta_2, \ldots, \delta_{m-1}, 1)$

[1] Note that given (1) and (3) above, it is obvious that $m \le \frac{q}{2n^4}$ for $n > 2$.

of length at most $(1^2 + 1^2 + \ldots + 1^2)^{1/2} = m^{1/2}$, which (by requirement 3 above) is no greater than $(n^2)^{1/2} = n$.

Here we define a harder problem requiring that: (r1) the solution h is in the form of $h_i \in \{0, 1\}^m$. This requirement further restricts the set of possible solutions but includes at least the constructed solution. (r2) the solution h is the *shortest* nonzero vector among all solutions that satisfy the requirements above; furthermore, if there are several such shortest nonzero vectors then the solution is the vector h that is the smallest among them in a lexicographic order. Again, we further restrict the set of possible solutions for the original approximation problem, having a solution to the restricted version implies a solution to the original approximation problem, and therefore the new defined problem is at least as hard as the original approximation problem. We use the term *shortest 01λq-vector* for the problem we have just defined. Note that the shortest 01λq-vector has the flavor of the subset sum problem (modulo q).

Program for Sender

```
1: Initialization:
2:     choose $n \geq 4, c_2 > 7, c_1$
3:     such that $n \log n^{c_2} < c_1 n \log n \leq n^2$
4:     $m := c_1 n \log n$
5:     $q := n^{c_2}$
6:     Send $(m, q)$
7:     for $i := 1$ to $m - 1$
8:         $v_i := random(x_1, \ldots, x_n)\ mod\ q$
9:     $\delta^{m-1} := random(\delta_1, \ldots, \delta_{m-1}) \mid \delta_i \in \{1, 0\}$
10:    $v_m := -\sum(\delta_i \cdot v_i)\ mod\ q$
11:    for $i := 1$ to $m$ Send$(v_i)$

12: Repeat
13:    $\lambda := (v_1, \ldots, v_m)$
14:    OTP:=FindShortest$(\Lambda(\lambda, q))$
15:    OTPS:=OTPS $\circ$ OTP
16:    CipherText:=OTP$\oplus$ Secret
17:    Send (CipherText)
18:    for $i := 1$ to $m - 1$
19:        $v_i := random(x_1, \ldots, x_n)\ mod\ q$
20:    $\delta^{m-1} := random(\delta_1, \ldots, \delta_{m-1}) \mid \delta_i \in \{1, 0\}$
21:    $v_m := -\sum(\delta_i \cdot v_i)\ mod\ q$
22:    Send $((v_1, v_2, \ldots, v_m)\oplus$ psuffix(OTPS) )
23: Until file transfered
```

Program for Receiver

```
1: Initialization:
2:     Recv $(m, q)$
3:     Recv $(v_1, \ldots, v_m)$

4: Repeat
5:     $\lambda := (v_1, \ldots, v_m)$
6:     OTP:=FindShortest$(\Lambda(\lambda, q))$
7:     OTPS:=OTPS $\circ$ OTP
8:     Recv(CipherText)
9:     Secret:=CipherText $\oplus$ OTP
10:    Recv $((v_1, v_2, \ldots, v_m)\oplus$ psuffix(OTPS))
11: Until file transfered
```

Fig. 2. Shortest 01λq-Vector

The code of the algorithm that uses the shortest 01λq-vector problem for magnifying the computing gap appears in Figure 2. In lines 2 and 3 of the code, the sender chooses $n \geq 4$, $c_2 \geq 7$, and c_1 such that (m and q defined in lines 4 and 5 are integers and) $n \log n^{c_2} < c_1 n \log n \leq \frac{n^{c_2}}{2n^4}$. Then in lines 4, 5, and 6 the sender computes $m = c_1 n \log n$ and $q = n^{c_2}$ and sends m and q to the receiver. In lines 7 and 8 the sender chooses $m - 1$ random vectors $v_1, \ldots, v_{m-1}$ from the set of all vectors $(x_1, ...x_n)$ with $0 \leq x_i \leq q - 1$. In line 9 the sender chooses $m-1$ random values $\delta_1, \delta_2 \ldots, \delta_{m-1}$ such that $\delta_i \in \{1,0\}$. Then in line 10 the sender computes the vector $v_m = -\sum(\delta_i \cdot v_i)\ mod\ q$. The construction of v_m and the inclusion of v_m in the set of the vectors $v_1, v_2, \cdots, v_m$, ensures the existence of a solution to the shortest $01\lambda q$-vector problem. Namely, there is at least one vector $h = (\delta_1, \delta_2 \ldots, \delta_{m-1}, 1)$ such that $((\sum_{i=1}^{m}(\delta_i \cdot v_i)) + v_m)\ mod\ q = (\sum_{i=1}^{m-1}(\delta_i \cdot v_i) - \sum_{i=1}^{m-1}(\delta_i \cdot v_i))\ mod\ q \equiv 0$ and the length of h is $\|h\| \leq \sqrt{m} \leq n$. In line 11 the sender sends to the receiver the vectors $v_1, v_2, \ldots, v_m$.

The sender repeatedly executes lines 13 to 21 until the file to be transferred is encoded. In line 13 the sender defines λ as the set of vectors $(v_1, v_2, \ldots, v_m)$. In line 14 the sender finds the shortest $01\lambda q$-vector. Finding the shortest $01\lambda q$-vector may be performed by exhaustive search over the 2^m possible vectors $h = \delta^m = \{0,1\}^m$. For each such vector we check whether $\sum_{i=1}^{m}(\delta_i \cdot v_i)\ mod\ q \equiv 0$ (its length must be not greater than n); if so we include h in the set $\mathcal{S}$ of possible solutions and compute the length of h. Then we choose a vector from $\mathcal{S}$ with the shortest length among the vectors in $\mathcal{S}$. In case there is a set of two or more vectors $\mathcal{T} \subseteq \mathcal{S}$ with the shortest length then we choose the first vector in $\mathcal{T}$ according to a lexicographic order. Note that the construction of v_m ensures that $\mathcal{S}$ is not empty; still, the result of the above computation may be a vector h which is not the $\delta_1, \delta_2, \ldots, \delta_{m-1}, 1$ vector computed in lines 9 and 10. We use the first $m-1$ coordinates $h_1, h_2, \ldots, h_{m-1}$ of the shortest vector as a one time pad (OTP) for our encryption scheme.

Assume the user would like to encrypt a secret that may be in the form of a long file, we next describe the way consecutive portions of the secret file are encrypted and sent to the receiver. In line 16 the sender computes the Cipher-Text xoring (bitwise) the OTP and (a portion of) the Secret (that is of $m-1$ bits length). Then, in line 17, the sender sends the encrypted secret to the receiver. Note that in every iteration the sender first sends λ and then the secret encrypted with the newly obtained OTP. Similarly, the receiver first receives λ and then decrypts the secret with the obtained OTP. Lines 18 and 19 start the process for the next $m-1$ bits of the secret file, choosing $m-1$ random vectors $(v_1, v_2, \ldots, v_{m-1})$ *modulo* q (as done in line 8), and new $m-1$ random values $\delta_1, \delta_2, \ldots, \delta_{m-1}$ (as done in line 9). In line 21 the sender computes the value of $v_m = -\sum(\delta_i \cdot v_i) mod\ q$. Then the sender encrypts the new problem instance with the OTP portions that were computed so far. The sender sends to the receiver the vectors $(v_1, v_2, \ldots, v_m)\oplus$ psuffix(OTPS). The number of bits required to describe the matrix is $c_2 m \cdot n \cdot \log n$. We use a sequence of the OTP portions to encrypt the new instance of the problem defined by such a matrix. To do so we propose to use the psuffix(OTPS) function that xores the OTPS portions that were computed most recently. Let OTP$_{k+1}, \ldots$ OTP$_{k+l}$ be the last l OTPS portions used in our algorithm, where $l \cdot (m-1) \geq c_2 m \cdot n \cdot \log n > (l-1) \cdot (m-1)$. Let $mask_{k+i} = \oplus_{j\ \mid\ (j \leq k+l)\ \wedge\ (j\ mod\ (l+1) = i)}$ OTP$_j$, psuffix(OTPS)=pseudorandom(OTP$_{k+l}$) $\oplus mask_{k+1} \circ mask_{k+2} \circ \cdots mask_{k+l}$. Note that in the beginning we may have less than l OTP portions in OTPS; in this temporary period, we will use a pseudo random function with a seed obtained from all the OTP portions revealed so far (here we defined the seed as

OTP_{k+l}, but other choices like a seed defined by $\oplus_{i=1}^{k+l}$ OTP_i may fit as well). We note that in order to enlarge the fan-out of the problem one would like to further restrict the choice of (n, c_1 and c_2) q to ensure that $q = 2^i$ for some integer i.

In line 2 the receiver receives the values m and q. In line 3 the receiver receives the first $v_1, v_2, \ldots, v_m$. Then the receiver repeatedly executes lines 5 to 10 until the receiver receives the entire secret file. In line 5 we use λ to denote the set of vectors $(v_1, v_2, \ldots, v_m)$ that define the next problem instance. In line 6 the sender finds the shortest $01\lambda q$-vector. Then in line 8 the receiver receives the CipherText. The secret is decoded by xoring the computed shortest $01\lambda q$-vector as the one time pad (OTP) for the received CipherText. At last the receiver receives the new vectors $(v_1, v_2, \ldots, v_m)\oplus$ psuffix(OTPS). Thus, the results of previous iterations allow the receiver to compute the problem instance $v_1, v_2, \ldots, v_m$ for the next iteration.

First we address the "hardness" of a bit in psuffix(OTPS) using the results in [16,17,24]. By our random choice, every bit has the same probability to be hard and some bit(s) is(are) hard (since the solution is hard on average). Since we use xor of solutions to define masks, the hardness of bits in psuffix(OTPS) is averaged. When the pseudo random sequence is obtained from the seed $\oplus_{i=1}^{k+l}$ the above claim addresses also the "hardness" of bits of the seed. The security of our scheme is based on the security of [3], noting that an adversary that can compromise our scheme is able to compromise the cryptosystem of [3]. We conclude with the following lemma.

Lemma 1. *In every instance in which the receiver decrypted $c_2 \cdot (m/(m-1)) \cdot n \cdot \log n$ portions of the secret more than the adversary decrypted it, it holds that the fan-out is at least $2^{c_2 \cdot (m-1) \cdot n \cdot \log n}$.*

Proof. By our choice of $q = 2^i$, every input prefix of length $c_2 \cdot (m-1) \cdot n \cdot \log n$ defines a valid input prefix of $v_1, v_1, \cdots, v_{m-1}$ for the problem. If the receiver decrypted $c_2 \cdot (m/(m-1)) \cdot n \cdot log_2 n$ portions of length $m-1$ of the secret more than the adversary decrypted the fan-out is at least $2^{(m-1) \cdot c_2 \cdot n \cdot \log n}$.

4 One-Sided Computation

In this section we will consider the case in which only one of the participants (the receiver) has to use extensive computing power. In other words, the gap in the computation power used is only the gap between the receiver and the listening adversary, while the sender does not have to (or is not able to) perform a computation expensive process. The idea is to use the fact that the probability for having a large number of solutions to the short $01\lambda q$-vector problem is very small and of the possibility of notifying the receiver with the OTP path. The *short $01\lambda q$-vectors* problem is defined similarly to the definition of the shortest $01\lambda q$-vector of Section 3; however, it is harder in the sense that it has to return the set that consists of all the solutions $h = (\delta_1, \delta_2 \ldots, \delta_{m-1}, 1) \in \{0,1\}^{m-1}1$, such that $\sum_{i=1}^{m}(\delta_i \cdot v_i)) + v_m)$ mod $q \equiv 0$.

In this scheme the sender does not have to compute the solution of the short 01λq-vector. The sender builds the random instances $(v_1, v_2, \ldots, v_m)$ as in the previous section. Then the sender sends these instances masked by the psuffix function defined over the l last sequences of the OTP, where the OTP is the constructed solutions of the short 01λq-vector: $(\delta_1, \ldots, \delta_{m-1})$. The sender uses $\delta_1, \cdots, \delta_{m-1}$ as the OTP (the OTP may not include δ_m since the value of δ_m is always 1). Once every k iterations the sender will make one or more *synchronization phases* with the receiver by sending the $m-1$ $\delta_1, \delta_2, \ldots, \delta_{m-1}$ bits that were randomly chosen (in line 19 of Figure 2) together with the obtained $((v_1, v_2, \ldots, v_m)\oplus$ psuffix(OTPS)) computed in (line 22 of Figure 2) during this iteration in every synchronization phase. In other words, the sender sends $((v_1, v_2, \ldots, v_m), (\delta_1, \delta_2, \ldots, \delta_m)\oplus$ psuffix(OTPS)). The synchronization phase does not produce a new element for computing psuffix(OTPS) or used for encryption.

On the other side, the receiver may find more than one possible solution (including the one constructed by the sender) and may not determine the constructed solution among them. So, how will the sender know which is the constructed solution? The receiver will use a *computation tree* that represents the set of all the possible solutions and the problem they define. When the receiver receives the next instances masked by the constructed solution, the receiver will try to solve the new instance using each of the previous possible solutions. Each such possible solution will result in an instance of the short 01λq-vector problem. Then in a synchronization phase the receiver will have a very high probability to truncate all the possible computations but one. We next prove that the set of possible solutions never blows up when the period k for synchronization and the number of synchronization phases l are chosen appropriately. The proof of the next Theorem is omitted from this extended abstract (See [10] for more details).

Theorem 1. *There exist q, n, m, k, and l, for which the expected number of iterations to reveal the i iteration secret is less than the $i+k$ iteration.*

The reasoning used in the proof above is depicted in Figure 3. The paths rooted at the left node represent computations that start due to the ith constructed problem. The number of possible solutions including the $i+1$st constructed

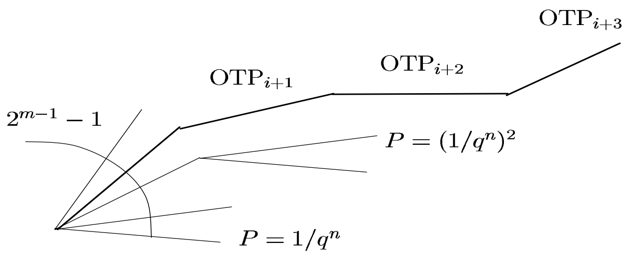


Fig. 3. The expected number of iterations required to reveal OTP_i

solution is at most $2^{m-1} - 1$ (all zero solution is not allowed). The bold path follows the constructed problems sequence; the other paths represent possible non constructed computation sequences.

5 Concluding Remarks

We define and present a framework for unidirectional encryption schemes. Imagine a person that survived a malicous attack, assume the survivor is surrounded by adversarial agents that may listen to his/her transmissions. The survivor being associated with a computationally superior organization may start randomly choosing matrices to convey his/her messages. Another scenario is a sensor in the field that should transmit a message to a satellite. Each sensor can be captured before hand, thereby nullifying the benefit of setting a private shared key of all sensors with the satellite before the distribution of the sensors. Our scheme can be used to communicate with the computationally strong satellite without issuing private keys a priori and without the need for maintaining private keys at the satellite.

In the sequel we present several examples that may well use our schemes. We view the mathematical aspect of taking a computing gap and bootstrapping it to an even wider gap by an algorithm as an interesting problem by itself that is not necessarily tied to an encryption task. There are several schemes that can be used by our magnifying computing gaps approach, we now list a few:

Private key establishment using unidirected link. The scheme presented in [11] (that uses shared secrets, and therefore does not fit our unidirected communication requirements) can be based on a secret that is sent as a suffix of a long file.

Anonymity of the receiver. Anonymity (e.g., [4,14]) is another issue that our scheme may support since the sender does not have to know the receivers.

Encrypted end-to-end and broadcast for unidirected network. Message delivery using unidirected communication links, where the adversary can listen to all the transmitted information is one of the main applications of our scheme. The sender can send a message to be forwarded to a remote receiver such that the message is transmitted over each hop in the path using our scheme for unidirectional delivery. An intermediate node may act only as a relay or decrypt and then encrypt the message again. Broadcast schemes may use the latter option resulting in decryption of the message by all nodes with superior computing capabilities over the listening adversary (whether it listens to the transmission over the links or in the form of a computation weak node). There are several potential users for the above schemes, for example satellite television broadcasters that would not like to update the satellite with every new subscriber.

Undirected encryption schemes:

Combating spam. We can use the one-sided computation scheme of Section 4 for combating junk mail [13]. For that, the email-sender will act as the receiver of the scheme and the email-receiver will act as the sender of the scheme; in this way the email-receiver will not have to compute the solution for the problem. In more detail, the email-sender sends a request for delivery, receives problems to solve (up to the first synchronization phase), then solves them and sends the email with the solution; the email-receiver will check the solutions and only when it is correct will it process the message (rare retransmissions will be needed when the set of solutions for the short 01λq-vector problem includes more than one solution after the synchronization phase).

Resource allocation. We can also use the above technique for controlling access to a shared resource. In order to access a resource one will need to solve a problem supplied by the resource administrator. Thus, the frequency of requests received from a particular client (in a way similar to the way we handled junk mail) will be controlled by the administrator (maybe also by tuning the complexity of the problem to solve) and clients may be effectively denied access when their computation power is beyond a certain threshold.

Private key strong-weak. Similar scheme for establishing a private key In an undirected network, when one of the two participants has weak computing power is possibly letting this participant play the sender in our one-sided computation scheme. For example, when a person equipped with a weak computational device would like to transmit, say, his/her current location, to a computationally strong base station without letting the adversary reveal the secret. One can use the random instances of the short 01λq-vector for implementing [19] such that the puzzles are hard on the average and the solution is known to the sender. Moreover, the encoded puzzles approach can enhance the security obtained by the scheme in [19]. Using the secret established by the protocol in [19], it is possible to encode a new set of n puzzles and establish a new key using this set. Such an iterative protocol will result in a higher security level. Roughly speaking, the chances that the secret is revealed at the same time it is created, are quadratically smaller than the chances that the secret is revealed in the original scheme. Note that the communication overhead of the additional iteration is only a constant factor of the original communication. Obviously, this approach can be extended beyond two iterations to gain even higher (in a factor of n and the number of iterations) security levels while communicating only $O(n)$ puzzles.

Overall comment. Note that it is possible to use a similar scheme where the sender and the receiver start with a key in common and thus have a (knowledge) gap that they further magnify to gain proactive security, e.g., [7,8]. We believe that the new defined shortest/short 01λq-vector problems are of independent interest.

Acknowledgments. It is a pleasure to thank Amos Beimel, Dan Boneh, Cynthia Dwork, Michael Fischer, Yinnon Haviv, Amir Herzberg, and the anonymous referees for useful comments.

References

1. Ajtai, M.: Generating Hard Instances of Lattice Problems. In: Proc. of STOC, pp. 99–108 (1996)
2. Ajtai, M.: The Shortest Vector Problem in L_2 is NP-hard for Randomized Reductions. In: Proc. of the 30th ACM STOC, ACM Press, New York (1998)
3. Ajtai, M., Dwork, C.: Public-Key Cryposystem with Worst-Case/Average-Case Equivalence, Electronic Colloquium on Computational Complexity, Report TR96-065 (1996)
4. Beimel, A., Dolev, S.: Busses for Anonymous Message Delivery. Journal of Cryptology 16(1), 25–39 (2003)
5. Dijkstra, E.W: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
6. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
7. Dolev, S., Kopetsky, M.: Secure Communication for RFIDs, Proactive Information Security within Computational Security. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, Springer, Heidelberg (2006)
8. Herzberg, A., Jarecki, S., Krawczyk, H., Yung, M.: Proactive Secret Sharing Or: How to Cope With Perpertual Leakage. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 339–352. Springer, Heidelberg (1995)
9. Cormen, T., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms, p. 118. MIT Press, Cambridge (1990)
10. Dolev, S., Korach, E., Uzan, G.: A Method for Encryption and Decryption of Messages. PCT Patent Application WO 2006/001006 (January 5, 2006)
11. Ding, Y.Z., Rabin, M.O.: Hyper-Encryption and Everlasting Security. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 1–26. Springer, Heidelberg (2002)
12. Dwork, C.: Positive Applications of Lattice to Cryptography. In: Privara, I., Ružička, P. (eds.) MFCS 1997. LNCS, vol. 1295, pp. 44–51. Springer, Heidelberg (1997)
13. Dwork, C., Naor, M.: Pricing via processing or combating junk mail. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (1993)
14. Dolev, S., Ostrovsky, R.: Xor-Trees for efficient anonymous multicast and reception. ACM TISSES 3(2), 63–84 (2000)
15. Feitelson, D.G.: Optical Computing: A Survey for Computer Scientists. MIT Press, Cambridge (1988)
16. Goldreich, O., Levin, L.: A hard-core predicate for all one-way functions. In: Proc. ACM Symp. on Theory of Computing, pp. 25–32 (1989)
17. Levin, L.A.: One-Way Functions and Pseudorandom Generators. Combinatorica 7(4), 357–363 (1987)
18. Lenstra, A.K., Shamir, A.: Analysis and Optimization of the TWINKLE Factoring Device. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 35–52. Springer, Heidelberg (2000)
19. Merkle, R.C.: Secure Communications Over Insecure Channels. CACM 21(4), 294–299 (1978)
20. Maurer, U.M.: Conditionaly-Perfect Secrecy and a Provable-Secure Randomized Cipher. Journal of Cryptology 5(1), 53–66 (1992)

21. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and time-release Crypto. Technical Report, MIT/LCS/TR-684
22. Reif, J.H., Tyagi, A.: Efficient Algorithms for Optical Computing with the DFT Primitive. Journal of Applied Optics (1997)
23. Vadhan, S.: On Constructing Locally Computable Extractors and Cryptosystems in Bounded-Storage Model. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, Springer, Heidelberg (2003)
24. Yao, A.C.: Theory and Application of Trapdoor Functions. In: 23rd FOCS, pp. 80–91 (1982)

STABILIZING TRUST and REPUTATION for Self-Stabilizing Efficient Hosts in Spite of Byzantine Guests*

(Extended Abstract)

Shlomi Dolev and Reuven Yagel

Department of Computer Science,
Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
`dolev,yagel@cs.bgu.ac.il`

Abstract. This work presents a general and complete method to protect a system against possible malicious programs. We provide concepts for building a system that can automatically recover from an arbitrary state including even one in which a Byzantine execution of one or more programs repeatedly attempts to corrupt the system state. Preservation of a guest execution is guaranteed as long as the guest respects a predefined contract, while efficiency is improved by using stabilizing reputation. We augment a provable self-stabilizing host operating system implementation with a contract-enforcement framework example.

Keywords: self-stabilization, security, host systems, Byzantine programs, trust and reputation.

1 Introduction

"Guests, like fish, begin to smell after three days" (Benjamin Franklin). A typical computer system today is composed of several self-contained components which in many cases should be isolated from one another, while sharing some of the system's resources. Some examples are processes in operating systems, Java applets executing in browsers, and several guest operating systems above virtual machine monitors (VMM). Apart from performance challenges, those settings pose security considerations. The *host* should protect not only its various *guests* from other possibly Byzantine guests [16,18,36], e.g. viruses, but also must protect it's own integrity in order to allow correct and continuous operation of the system [43]. Many infrastructures today are constructed with self-healing properties, or even built to be self-stabilizing. A system is *self-stabilizing* [13,14] if it can be started in any possible state, and subsequently it converges to a desired behavior. A *state* of a system is an assignment of arbitrary values to the system's variables.

* Partially supported by the Lynne and William Frankel Center for Computer Sciences and the Rita Altura trust chair in Computer Sciences.

Recovery with no utility. The fact that the system regains consistency automatically, does not guarantee that a Byzantine guest will not repeatedly drive the system to an inconsistent state from which the recovery process should be restarted. In this work we expand earlier self-stabilizing efforts for guaranteeing that eventually, some of the host's critical code will be executed. This ensures that eventually the host has the opportunity to execute a monitor which can enforce it's correctness in spite of the possibly existing Byzantine guests. In particular the host forces the Byzantine code not to influence the other programs' state. Finally, non-Byzantine programs will be able to get executed by the operating system, and provide their services.

Soft errors and eventual Byzantine programs. Even if we run a closed system in which all applications are examined in advance (and during runtime), still problems like soft-errors [38] or bugs that are revealed in rare cases (due to rare i/o sequence of the environment that was not tested/considered), might lead to a situation in which a program enters an unplanned state. The execution that starts from such an unplanned state may cause corruption to other programs or to the host system itself. This emphasizes the importance of the self-stabilization property that recovers in the presence of (temporarily or constantly) Byzantine guests. Otherwise, a single temporal violation may stop the host or the guests from functioning as required.

Host-guest enforced contract. The host guarantees preservation of the guest execution as long as the guest respects the predefined rules of a contract. The host cannot thoroughly check the guest for possible Byzantine behaviors (this is equivalent to checking whether the guest halts or not). Therefor the host will force a contract, that is sufficient for achieving useful processing for itself and the guests. The rules enforced by the host can be restrictive, e.g., never write to code segments, and allocate resources only through leases.

Stabilizing trust and reputation. Upon detecting a Byzantine behavior of a guest during run time (namely, sanity checks detect a contract violation) we can not prevent the guest from being executed, since the Byzantine behavior might be caused by a transient fault. Does this mean that we must execute all guests, including the Byzantine ones, with the same amount of resources? Furthermore, when we accumulate behavior history to conclude that a guest is Byzantine, the accumulated data maybe corrupted due to a single transient fault, thus we can not totally count on the data used to accumulate the history. Instead we continuously refresh our impression on the behavior of a guest while continuing executing all guests with different amount of resources. Details of violations are continuously gathered, and the impression depends more on recent behavior history. Such a trust and reputation function rates the guests with a suspicious level, and determines the amount of resources a guest will be granted. In this calculation, recent events are given higher weight, as past event are slowly forgotten. This approach copes with corruptions in the reputation data itself, since wrong reputation fades over time.

Table 1. Byzantine Threats

Mechanism	**Examples**	**Byzantine Threats**
Privileges, Address Space separation(MMU)	Commodity OSes	(i/o) Resources tampering, Separation algo. corruption
Type checking	JVM	Resource sharing Self modifying code
Emulation and Dynamic translator	Bochs, Qemu	Compiled code corruption
Hypervisor	Xen, VmWare	Rootkits, Privileged guest corruption

Byzantine guest examples. We review here some systems with their protection mechanisms and possible ways for Byzantine guests to attack. Commodity operating systems use standard protection mechanisms [49] such as several privilege levels and address space separation enforced by hardware, e.g., an MMU. A Byzantine guest can be executed with high privilege (by an unaware user), and corrupt the system's state. Additionally the lack of hardware i/o addresses separation in today's common processor architectures enables even kernel data corruption by, say, a faulty device driver. Managed environments like Java or the .NET CLR [22] use various methods of type checking, resource access control and also sandboxing [52]. These mechanisms rely on the correctness of the runtime loaders and interpreters and are also sensitive to self modifying code (see e.g., [12,23]).

Recently, there is a growing interest in virtualization techniques through virtual machine monitors. VMMs may form full emulators like Bochs [8] and Qemu [3] which interpret (almost) all of the guest's instructions (and thus can even ensure correct memory addressing). Other VMMs, like Xen [4], let the guest perform directly most of the processor instructions, especially the non-privileged ones (in [1,23,44] there is a classification of the various VMMs types). Many VMMs rely on one of the guests for performing complex operations such as i/o, and thus are vulnerable to Byzantine behavior of this special guest. Some studies, e.g., [44], show other problems in implementing virtualization above the x86 architecture [31], including some privileged instructions which are not trappable in user mode, and direct physical memory access through DMA. Recently, vendors augmented this architecture with partial corrections [40], but still not completely [24].

"Guests" that become super-hosts. Virtualized rootkits [34,43] were recently discussed. They load the original operating system as a virtual machine, thereby enabling the rootkit to even intercept all hardware calls that are made by the guest OS. They demonstrate the relative simplicity of a Byzantine program to take full control of the host. Table 1 summarizes some different mechanisms and their weaknesses.

Related research towards robust hosts. Various protection mechanisms were mentioned in the previous section. Some which emphasize separation and protection are detailed in the following. In [6], a Java virtual machine is enhanced with operating system and garbage collection mechanisms (type safety and write barriers) in order to prevent, cases like, "a Java applet can generate excessive amounts of garbage and cause a Web browser to spend all of its time collecting it". Virtual machine emulators have become tools to analyze malicious code [23]. Lately, several studies detailed ways of preventing malicious code from recognizing that it is executing as a guest [23,24,34,40,43,44] (see also [27] for VMM usage for security, and [43] which argues against relying on a full operating system kernel as a protection and monitoring base). In addition, well known hardware manufacturers intend to introduce soon IO-MMUs with isolation capability [1,10]. Operating system based emulators or hypervisors such as UML [15] or KVM [33] are used also to analyze suspected programs. [41] uses virtualization techniques to contain errors, especially in drivers, in realtime. Methods that are based on secure boot (e.g., [2,45,51]) are important in order to make sure that the host gets the chance to load first, and prevent *rootkits* from fooling it by actually running the host as a guest [34]. In [42], cryptography techniques are used in order to ensure that only authorized code can be executed.

Sandboxing techniques were presented in [52] (see also [26]). Sandboxing techniques make sure that code branches are in segment (a distinct memory section which exclusively belongs to a process), and also rely on different segments for code and data. For every computed address they add validation code that traps the system, or just masks addresses to be in the segment (this is actually a sandbox). They count on dedicated registers which hold correct addresses. Overview of trust and reputation can be found, e.g., in [25,37]. In [7] a Bayesian based approach with exponential decay is proposed.

The need for address space separation, the use of capabilities, minimal trusted base and other protection mechanisms were introduced in well known works [9,11,35,39,46,49]. Singularity [29,28] achieves process isolation in software by relying on type safety, and also prevents dynamic code. Self-modifying code certification is presented in [12].

Generally, extensive theoretical research has been done towards self-stabilizing systems [13,14,48] and autonomic - computing/ disaster - recovery/ reliability - availability - serviceability [30,32,50]. However, none of the above suggest a design for a host system that can automatically recover from an arbitrary state, even in the presence of Byzantine guests that repeatedly try to corrupt the system state.

Our contribution. (a) Identifying the need of combined self-stabilization, and techniques for enforcing a contract over the operations of a guest. We show that only such a combination will allow (useful) recovery. (b) The introduction of stabilizing trust and reputation and the use of the level of trust as a criteria for granting resources while continuing to evaluate the trust of the guests. (c) Concepts and a proof for designing hosts and contracts. (d) A running example.

Paper organization. Next, in Section 2, we briefly review results from previous works on self-stabilizing operating systems, which form our basis for a protected host system. Section 3 details the system settings and requirements. This is followed by Section 4 which presents a general framework for protecting against Byzantine programs. Section 5 presents an example of a simple Byzantine guest followed by the way a provable host implementation copes with such a Byzantine guest. As a result of paper length limitation, proofs and some technical and implementation details are omitted from this extended abstract.

2 Self-stabilizing Operating Systems – Foundations Overview

In previous works [19,20,21], we presented new concepts and directions for building a self stabilizing operating system kernel. A self-stabilizing algorithm/system makes the obvious assumption that it is executed. This assumption is not simple to achieve since both the microprocessor and the operating system should be self-stabilizing, ensuring that eventually the (self-stabilizing) applications/ programs are executed. An elegant composition technique of self-stabilizing algorithms [14] is used to show that once the underling microprocessor stabilizes the self-stabilizing operating system (which can be started in any arbitrary state) stabilizes, then the self-stabilizing applications that implement the algorithms stabilize. This work considers the important layer of the operating system.

One approach in designing a self-stabilizing operating system is to consider an existing operating system (e.g., Microsoft Windows, Linux) as a black-box and add components to monitor its activity and take actions accordingly, such that automatic recovery is achieved. We called this approach the black-box based approach. The other extreme approach is to write a self-stabilizing operating system from scratch. We called this approach the tailored solution approach. We have presented several design solutions in the scale of the black-box to the tailored solutions. The first simplest technique for the automatic recovery of an operating system is based on repeatedly reinstalling the operating system and then re-executing. The second technique is to repeatedly reinstall only the executable portion, monitoring the state of the operating system and assigning a legitimate state whenever required. Alternatively, the operating system code can be "tailored" to be self-stabilizing. In this case the operating system takes care of its own consistency. This approach may obviously lead to more efficient self-stabilizing operating systems, since it allows the use of more involved techniques.

Tailored Approach. An operating system kernel usually contains basic mechanisms for managing hardware resources. The classical Von-Neumann machine includes a processor, a memory device and external I/O devices. The tailored operating system is built (like many other systems) as a kernel that manages these three main resources. The usual efficiency concerns which operating systems must address, are augmented with stabilization requirements.

• **Process Scheduling.** The system is composed of various processes which are executing each in turn. The process loading, executing and scheduling part of the operating system usually forms the lowest and the most basic level. Two main requirements of the scheduler are fairness and stabilization preservation. Fairness means that in every infinite execution every running process is guaranteed to get a chance to run. Stabilization preservation means ensuring that the scheduler preserves the self-stabilization property of a process in spite of the fact that other processes are executed as well (e.g., the scheduler ensures that one process will not corrupt the variables of another process).

• **Memory Management.** We deal with two important requirements to the tasks of memory management. The first requirement is the *eventual memory hierarchy consistency*. Memory hierarchies and caching are key ideas in memory management. The memory manger must provide eventual consistency of the various memory levels. The second requirement is the *stabilization preservation* requirement. It means that stabilization proof for a single process p is automatically carried to the case of multiprocessing in spite the fact that context switches occur and the fact that the memory is actually shared. Namely, the actions of other processes will not damage the stabilization property of the process p.

• **I/O Device Drivers.** Device drivers are programs which are practically an essential part of any operating system. They serve as an adaptation layer by managing the various operation and communication details of I/O devices. They also serve as a translation layer providing consistent and more abstract interface for other programs and the hardware device resources (and sometimes they also add extra services not provided by the hardware devices). Device drivers are known to be a major cause of operating system failures [41].

In [21] we define two requirements which should be satisfied in order for the protocol between the operating system and an I/O device to be self-stabilizing. The first requirement (the ping-pong requirement) states that in an infinite system execution, in which there are infinitely many I/O requests, the OS driver and the device controller are infinitely often exchanging requests and replies. The second requirement is about progress and it states that eventually every I/O request is executed completely and correctly according to some protocol specification (e.g., the ATA protocol for storage devices). A device driver and device controller can be viewed as a master and a slave working together according to some protocol to achieve their mission. Thus, the device driver acting as a master can check that the slave is following, e.g. the ATA protocol, correctly.

The usage and usefulness of such a system in critical and remote systems cannot be over emphasized. For example entire years of work maybe lost when the operating system of an expensive complicated device (e.g., an autonomous spaceship) may reach an arbitrary state (say, due to soft errors) and be lost forever (say, on Mars). The controllers of a critical facility (e.g., a nuclear reactor or even a car) may experience an unexpected fault (e.g., an electrical spike) that will cause it to reach an unexpected state, from which the system will never recover, therein leading to harmful results. Our proofs and prototypes show that it is possible to design a self-stabilizing operating system kernel.

3 Settings and the Requirements

Definitions. We briefly define the system states and state transitions (see [19,20] for details concerning processor executions, interrupt, registers, read-only memories, a watchdog and additional settings). A *state* of the system is an assignment to its various memory components (including the program counter resister). A *clock tick* triggers the microprocessor to *execute a processor step* $ps_j = (s, i, s', o)$, where the inputs i and the current state of the processor s are used for defining the next processor state s', and the outputs o. The *inputs and outputs* of the processor are the values of its *i/o connectors* whenever a clock tick occurs. The processor uses the *i/o* connectors values for communicating with other devices, mainly with the memory, via its data lines. In fact, the processor can be viewed as a transition function defined by e.g., [31]. A *processor execution* $PE = ps_1, ps_2, \cdots$ is a sequence of processor steps such that for every two successive steps in PE, $ps_j = (s, i, s', o)$ and $ps_{j+1} = (\overline{s}, \overline{i}, \overline{s}', \overline{o})$ it holds that $\overline{s} = s'$.

Error model – arbitrary transient and Byzantine faults. The system state, including the program counter, system data structures and also the program code and data in RAM, may become arbitrarily corrupted, namely, assigned any possible value. This model is an extension of a previous one ([19]). The main feature of the extension is the removal of the assumption that all programs are self-stabilizing (or restartable [4]) so they might exhibit Byzantine behavior forever.
Requirements. We now define the requirements which should be satisfied for a host system to be self-stabilizing in spite of a Byzantine behavior.
(r1) Guest stabilization preservation. The fact that the host system may start in an arbitrary state, and execute code of Byzantine guests, will not falsify the stabilization property of each of the non-Byzantine guests in the system.
(r2) Efficiency guarantee. Non-Byzantine guests will eventually get the needed resources in order to supply their intended services.

Note that both (r1) and (r2) implicity require that a program that shares resources with others, will not block or will be blocked, outside of acceptable limits, due to this sharing (although in the worst case, due to the use of leases combined with a reputation system, resource will eventually be granted).

4 Concepts for Fighting the Byzantines

By combining techniques like secure booting, contract verification and enforcement together with self-stabilization we can protect a system against Byzantine guests in a provable way.

- **Secure booting** ensures that there is a minimal trusted computing base which runs programs and monitors.
- **Offline Byzantine behavior detectors** use code verification techniques, analyzing a program offline and looking for possible breaks of contracts.

– **Runtime anti-Byzantine enforcers** insert additional instructions in the executable for online sanity checks to enforce contract properties during a program execution.
– **Stabilizing trust and reputation** for determining the amount of resources a guest will be granted.
– **Self-stabilization** of these mechanisms and their composition [5,14] ensures that the system is eventually protected and functioning.

Secure booting is achieved through standard hardware based mechanisms (e.g., [2,45,51]). These are essential in order to guarantee that a Byzantine guest is not loaded first.

The system should be augmented with a detector framework which executes one or more upfront offline Byzantine detector plug-ins. A detector is built to enforce some aspect of a contract with a guest, and must be provable to perform its action completely and within acceptable time limits. These detectors scan the program code in advance for particular violations of the contract that are easy to check, and in case the scan reveals a Byzantine guest, this guest will not be loaded at all.

A program that passes the first check is augmented with sanity checks and access restrictions in sensitive code parts, where execution might do harm. The augmented code does not change the program semantics (up to stuttering) as long as the guest respect the contract. Upon detection of a violation in runtime, an enforcer can reload the program code and also update the trust and reputation level. An example for such an enforcer is one that enforces that segments used by the program are not changeable (meaning that self-modifying code is forbidden according to a contract). Runtime sanity checks, look for possible instruction sequences to make sure they do not violate the contract. Note that due to transient faults (that are rare by nature), a target address may change right after a sanity check, causing the system later to start a convergence stage as a self-stabilizing system should. In the case of a Byzantine program, the harm is prevented, although the detection and reloading will occur again and again (the trust and reputation record of a guest will limit the amount of processing used for this particular guest). In case the program is not Byzantine, the reload-of-code procedure will ensure correct behavior after which the trust and reputation will reach the maximal possible level.

Stabilizing trust and reputation can be achieved by using methods which favor recent events over past events. One example is [7] which combines a Bayesian approach with exponential decay. In such ways, trusted guests get more resources overtime, while suspected guests are not totally blocked and get chance to "shun evil and do good". Such approaches also cope with transient (fault) corruptions in the reputation data, since wrong reputation fades over time.

Theorem 1. *There exists a self-stabilizing host that can fulfill (r1) stabilization preservation and (r2) efficiency guarantees for guests.*

Sketch of proof: We list the mechanisms we use and the properties we establish by them. (a) The host is built above a self-stabilizing hardware ([17]) which guarantees eventually correct operation of the hardware from any state. (b) A self-stabilizing host operating system [19] which is guaranteed to periodically run some boot-code loaded in a secure way [2,45,51], without being subverted ([43]) (c) This trusted operating system guarantees eventual execution of all runnable processes including the contract offline detectors. (d) Code is being refreshed ([19,20]) periodically, so Byzantine or wrong behavior caused by transient faults to code segments are eventually fixed. (e) Contract properties are asserted by online enforcers. (f) Self-stabilizing programs might be supplied with a list of "initial" safe states. In such a case when recognizing Byzantine behavior, apart from preventing this behavior and refreshing the code, the closest state (using Hamming distance or some other metric) can be applied to the program. (g) All resource allocations are granted using leases with a self-stabilizing manager, as demonstrated in a previous work on dynamic memory [20], ensuring that resource allocations are eventually fair. The contract detectors and enforcers check also for behavior which violates the leasing rules. Resource are leased to a guest according to its trust and reputation level. (h) System calls and traps are also leased (again, according to the trust and reputation level), so a Byzantine guest is limited in the number of times it can cause long delay due to system calls. (i) Non-Byzantine programs are stabilizing in spite of faults and Byzantine behavior. (j) The interaction between those programs and other programs or devices is stabilizing too ([21]). (k) The stabilization process of one program only affects the state of this program and does not affect other programs. Thus, stabilization preservation and efficiency guarantee is achieved for guests. □

The implementations presented next, add sanity checks to branches and memory accesses, ensure correct use of leased resources, and enforce allowed patterns of out of memory accesses.

5 Host Implementation Example

In previous works we demonstrated the construction of a self-stabilizing operating system (SOS) [19,20,21,47]. Guest separation was achieved by using the segmentation mechanism of the Pentium processor [31], without MMU hardware protection. Additionally, we assumed that the code of the programs is hardwired and correct, thus a program does not contain instructions which affect the state of the other programs (including the system). When we introduce programs with arbitrary code, other programs, even the host/operating system itself, may be corrupted. In the current work we have implemented a prototype of a simple host that satisfies requirements (r1) and (r2) above the mentioned system.

To demonstrate the possible corruption of the system designed, we show an example of a threat in a program that accesses the operating system's segment and changes the scheduler state. The scheduler state is changed so that this program will be scheduled (again

```
mov ax, 0x8010
mov ds, ax
mov word [0x292], 3
```

Fig. 1. Byzantine Code

and again) instead of other guests. Figure 1 shows an example of such a 16-bit x86 assembly code. Lines 1-2 change the data segment pointer to the system's segment. Then, line 3 changes the process pointer contents to a value which will cause re-scheduling of this program.

One can argue that address-space separation, like found in commodity operating system kernels, can prevent this behavior. But if a Byzantine program manages to operate in a privileged mode, even once due to a transient fault, the separation algorithm itself might be subverted, followed by the above malicious behavior. Next we will describe our settings in order to show a provable solution.

To demonstrate these ideas we show: (a) an example containing added code that enforces memory access within a program's data segments (sandboxing). (b) Accessing shared resources through leases. (c) A prototype of a detector that performs offline verification that out of segment accesses are according to a list of known patterns allowed by a contract. (d) An example of stabilizing trust and reputation evaluation according to online sanity checks.

The suggested solution uses an architecture in which some code is read-only (Harvard model). A non-maskable interrupt (NMI) is generated by a simple self-stabilizing watchdog. Thus, the hardware triggers a periodic execution of the host monitoring (detectors) code. This architecture also guarantees that the monitoring code gets enough time to complete. A detector searches the code of every program to make sure it does not contain code that changes segments outside the scope of the program. Computed addresses are enforced to be within limits, by inserting sanity checks. The correct use of leased resources is also enforced during runtime. Additionally, from time to time the host refreshes the code of all guests (including sanity checks insertions), say from a CD-ROM, to allow self-stabilization of programs following a code refresh.

(a) Figure 2 line 1 demonstrates calculation of a segment selector value, as opposed to Figure 1 in which the address is fixed. Then, lines 2-5 are a sanity check added by the runtime anti-Byzantine enforcer. First the calculated address is validated to be in range (in this example it must have some fixed value), in case of a violation detection (line 3) the Increase-Bad-Reputation procedure is called to record the violation (see (d) below). Then in line 5, the correct address is enforced. Alternatively, a monitor could start actions of reloading code and data in case of detecting such a wrong access.

```
1   mov ax, <<computed address>>
// added sanity check
2   xor ax, SEGMENT_MASK
3   jz AfterSanityCheck
4   call Increase-Bad-Reputation
5   mov ax, FIXED_SEGMENT
AfterSanityCheck:
...
```

Fig. 2. Memory Access Enforcer

(b) Figure 3 presents the way a program uses a shared resource, in this case the dynamic memory heap. The contract is that all accesses to segments in this memory area must happen only through the segment selector register **fs**. Additionally, in order for this access to be leased, a program is not allowed to load a value in this register, but instead asks the system for a leased allocation (line 1). After this allocation request, and before every dynamic memory access, the program must check that the lease is still in effect, by checking the value in **fs** (lines 2-3). In case the allocation failed or expired, the value will be 0. Detectors and enforcers check that the use of shared resources is done according to this contract [20] and penalize the program in case of irregular use detection.

```
1   call MM_Alloc
After_MM_Alloc:
2   cmp fs, 0
3   jz TryLater
...
```

Fig. 3. Shared Resource Access

(c) The Pentium's operation code for moving a value into one of the 16-bit segment selector registers, is **8e**. Thus, the offline detector searches for commands starting with this code (assuming for simplicity that this is the only possible way). Note that the Pentium has variable length operations so in order to detect beginnings of operations we need to use disassembly techniques. Additionally, the mentioned operation code is sometimes used in legitimate ways, e.g. for accessing dynamic data segments. A possible solution is allowing known fixed patterns of access to other segments. An example pattern appears in Figure 4, where the program is accessing the video segment, which is needed for screen output. The **es** segment register is loaded in line 2 by the allowed value that is computed in line 1. In this case the **8e** op-code is preceded by the sequence **b8 00 b8** which is considered valid. Figure 5 presents the algorithm of the segment access detector. This detector is executed before loading the guest program. It scans a program's code segment for the **8e** code. When found, it verifies that it is preceded by one of the allowed patterns, otherwise the program is considered as one that does not respect the contract and therefore an upfront Byzantine program.

(d) Upon finding an online contract violation through performing the enforced sanity checks, the violation is recorded in the reputation history record (Figure 6). This record maybe kept as part of a *process entry* in the system's process table. Every predefine period of time (or steps) the system updates this record, as seen in in the Decay-Reputation procedure in Figure 6. The entries in the record are shifted in a way that the oldest entry is removed, thus implementing the needed decay and stabilization. This updated record is used for evaluating

```
1   mov ax, VIDEO_SEGMENT
2   mov es, ax
```

Fig. 4. An Allowed Pattern

```
BYZANTINE-DETECTOR($process\_entry, legal\_patterns$)
1  for each $instruction\_code(ic)$ in $process\_entry.code\_segment$
2  do if $ic$ starts_with $''8e''$
3        then for each $pattern$ in $leagal\_patterns$
4              do if $pattern$ precedes $ic$
5                    then continue_main_loop
6              $process\_entry.byzantine \leftarrow true$
7              return
```

Fig. 5. Out of Segment Access Detector

```
INCREASE-BAD-REPUTATION($process\_entry$)
1  $process\_entry.reputation[0]$ &= $BAD\_REPUTATION\_BIT$
2  $process\_entry.reputation[0] \ll 1$  ▷ Shift left.
3  return

DECAY-REPUTATION($process\_entry$)
1  for $i$ in $(MAX\_HISTORY - 1)$ .. 1
2  do $process\_entry.reputation[i] \leftarrow process\_entry.reputation[i-1]$
3  return $process\_entry.reputation$
```

Fig. 6. Update Trust and Reputation – Increase and Decay

the trust and reputation level of the relevant guest and granting resources in accordance.

Performance issues. The timing of the execution of code refreshing (and offline detectors) can be tuned according to the expected rate of soft-error corruptions to code. This processes do not have a great impact on the program execution performance, since the frequency of the checks may be balanced against the desired recovery speed.

One could suggest a performance gain by having an auxiliary processor (one core of a multi-core) for performing a repeated contract verification on the loaded code. However, a Byzantine guest might fool the auxiliary processor, say, by changing the sanity checks to be correct whenever the auxiliary processor is checking them. Still we can use such a processor for most of the cases to speed the indication on soft errors and to trigger code refreshing.

6 Concluding Remarks

In this work we presented an approach to use self-stabilizing reputation in order to gain efficient performance. We believe that self-stabilizing host systems that use stabilizing reputation are a key technology which can cope with Byzantine behavior in critical computing system. Source code examples can be found in [47].

References

1. Adams, K., Agesen, O.: A Comparison of Software and Hardware Techniques for x86 Virtualization. In: ASPLOS. Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems, CA (2006)
2. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: Proceedings of 1997 IEEE Symposium on Computer Security and Privacy, IEEE Computer Society Press, Los Alamitos (1997)
3. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: Proc. of USENIX Annual Technical Conference. FREENIX Track (2005)
4. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA (2003)
5. Brukman, O., Dolev, S., Haviv, Y., Yagel, R.: Self-Stabilization as a Foundation for Autonomic Computing. In: FOFDC. Proceedings of the Second International Conference on Availability, Reliability and Security, Workshop on Foundations of Fault-tolerant Distributed Computing, Vienna, Austria (April 2007)
6. Back, G., Hsieh, W.H., Lepreau, J.: Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In: OSDI. Proc. 4th Symposium on Operating Systems Design and Implementation, San Diego, CA (2000)
7. Buchegger, S., Le Boudec, J.-Y.: A Robust Reputation System for Mobile Ad-hoc Networks. Technical Report IC/2003/50, EPFL-IC-LCA (2003)
8. Bochs IA-32 Emulator Project. http://bochs.sourceforge.net/
9. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuchynski, M., Becker, D., Eggers, S., Chambers, C.: Extensibility, Safety, and Performance in the SPIN Operating System. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, Colorado, December (1995)
10. Ben-Yehuda, M., Xenidis, J., Mostrows, M., Rister, K., Bruemmer, A., Van Doorn, L.: The Price of Safety: Evaluating IOMMU Performance. In: OLS. The 2007 Ottawa Linux Symposium (2007)
11. Chase, J.S., Levy, H.M., Feeley, M.J., Lazowska, E.D.: Sharing and Protection in a Single-Address-Space Operating System. ACM Transactions on Computer Systems 12(4) (November 1994)
12. Cai, H., Shao, Z., Vaynberg, A.: Certified Self-Modifying Code. In: Proceedings of PLDI 2007, CA (2007)
13. Dijkstra, E.W.: Self-Stabilizing Systems in Spite of Distributed Control. Communications of the ACM 17(11), 643–644 (1974)
14. Dolev, S.: Self-Stabilization. The MIT Press, Cambridge (2000)
15. Dike, J.: A User-mode Port of the Linux Kernel. In: 5th Annual Linux Showcase and Conference, Oakland, California (2001)
16. Daliot, A., Dolev, D.: Self-stabilizing Byzantine Agreement. In: PODC 2006. Proc. of Twenty-fifth ACM Symposium on Principles of Distributed Computing, Colorado (2006)
17. Dolev, S., Haviv, Y.: Stabilization Enabling Technology. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 1–15. Springer, Heidelberg (2006)
18. Dolev, S., Welch, J.: Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults. In: UNLV. Proc. of the 2nd Workshop on Self-Stabilizing Systems (1995). Journal of the ACM, Vol. 51, No. 5, pp. 780-799, September 2004.

19. Dolev, S., Yagel, R.: Toward Self-Stabilizing Operating Systems. In: SAACS04,DEXA. Proceedings of the 15th International Conference on Database and Expert Systems Applications, 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems, Zaragoza, Spain, pp. 684–688 (August 2004)
20. Dolev, S., Yagel, R.: Memory Management for Self-Stabilizing Operating Systems. In: Proceedings of the 7th Symposium on Self Stabilizing Systems, Barcelona, Spain (2005). also in *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 2006.
21. Dolev, S., Yagel, R.: Self-Stabilizing Device Drivers. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 276–289. Springer, Heidelberg (2006)
22. ECMA International. ECMA-335 Common Language Infrastructure (CLI), 4th Edition, Technical Report (2006)
23. Ferrie, P.: Attacks on Virtual Machine Emulators. Symantec Advanced Threat Research, http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf
24. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility Is Not Transparency: VMM Detection Myths and Realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems, San Diego, CA (2007)
25. Guha, R., Kumar, R., Raghavani, P., Tomkins, A.: Propagation of trust and distrust. In: WWW. Proceedings of the 13th International World Wide Web conference (2004)
26. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems (1997)
27. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: Proceedings of SOSP 2003 (2003)
28. Hunt, G., Larus, J.: Singularity: Rethinking the Software Stack. Operating Systems Review 41(2) (April 2007)
29. Hunt, G., Aiken, M., Fhndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D., Wobber, T.: Sealing OS Processes to Improve Dependability and Safety. In: Proceedings of EuroSys2007, Lisbon, Portugal (March 2007)
30. Intel Corporation. Reliability, Availability, and Serviceability for the Always-on Enterprise, The Enhanced RAS Capabilities of Intel Processor-based Server Platforms Simplify 24 x7 Business Solutions, Technology@Intel Magazine (August 2005), http://www.intel.com/technology/magazine/Computing/Intel_RAS_WP_0805.pdf
31. Intel Corporation. The IA-32 Intel Architecture Software Developer's Manual (2006), http://developer.intel.com/products/processor/manuals/index.htm
32. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer, 41–50 (January 2003), See also http://www.research.ibm.com/autonomic
33. *KVM: Kernel-based Virtual Machine for Linux*, http://kvm.qumranet.com/
34. King, S.T., Chen, P.M., Wang, Y., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: IEEE Symposium on Security and Privacy (May 2006)
35. Lampson, B.W.: Protection. In: Proceedings of the 5th Princeton Symposium on Information Sciences and Systems, Princeton University (March 1971). Reprinted in ACM Operating Systems Review (January 1974)

36. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Trans. on Programming Languages and Systems 4(3), 382–401 (1982)
37. Mui, L.: Computational Models of Trust and Reputation: Agents, Evolutionary Games, and Social Networks. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (2002)
38. Mastipuram, R., Wee, E.C.: Soft errors' impact on system reliability. Voice of Electronics Engineer (2004), http://www.edn.com/article/CA454636.html
39. Neumann, P.G.: Computer-Related Risks. Addison-Wesley, Reading (1995)
40. Neiger, G., Santony, A., Leung, F., Rogers, D., Uhlig, R.: Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Intel Technology Journal 10(3) (August 2006)
41. Swift, M., Bershad, B.N., Levy, H.M.: Improving the reliability of commodity operating systems. In: SOSP 2003. Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY (October 2003). See also: M. Swift. Improving the Reliability of Commodity Operating Systems, Ph.D. Dissertation, University of Washington (2005)
42. Sharma, A., Welch, S.: Preserving the integrity of enterprise platforms via an Assured eXecution Environment (AxE). In: OSDI. A poster at the 7th Symposium on Operating Systems Design and Implementation (2006)
43. Rutkowska, J.: "Subvirting Vista Kernel For Fun and Profit — Part II Blue Pill", see also (2006), http://www.whiteacid.org/misc/bh2006/070_Rutkowska.pdf, http://www.whiteacid.org/papers/redpill.html
44. Robin, J., Irvine, C.: Analysis of the Intel Pentiums Ability to Support a Secure Virtual Machine Monitor. In: Usenix annual technical conference (2000)
45. Ray, E., Schultz, E.E.: An early look at Windows Vista security. Computer Fraud & Security 2007(1) (2007)
46. Schroeder, M.D.: Cooperation of Mutually Suspicious Subsystems in a Computer Utility. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA (September 1972)
47. SOS download page. http://www.cs.bgu.ac.il/~yagel/sos, 2007
48. http://www.selfstabilization.org
49. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1268–1308 (1975)
50. Sun Microsystems, Inc. 'Predictive Self-Healing in the Solaris^{TM} 10 Operating System", White paper (September 2004), http://www.sun.com/software/solaris/ds/self_healing.pdf
51. Tygar, J.D., Yee, B.: Dyad: A system for using physically secure coprocessors. In: Proceedings of IP Workshop (1994)
52. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient Software-based fault isolation. In: Proceedings of the Sym. On Operating System Principles (1993)

r-Semi-Groups: A Generic Approach for Designing Stabilizing Silent Tasks

Bertrand Ducourthial

Université de Technologie de Compiègne, Lab. Heudiasyc UMR CNRS 6599, France
bertrand.ducourthial@hds.utc.fr

Abstract. In [13,14,7], the modeling of silent tasks by means of so-called *r-operators* has been studied, and interesting relations have been shown between algebraic properties of a given operator and stabilizing properties of the related distributed algorithms. Modeling algorithms with algebraic operators allows to determine generic results for a wide set of distributed algorithms. Moreover, by simply checking some local algebraic properties, some global properties can be deduced. Stabilizing properties of shortest path calculus, depth-first-search tree construction, best reliable transmitters, best capacity paths, ordered ancestors list... have hence been established by simply reusing generic proofs, either in the read-write shared register models [13,14] or in the unreliable message passing models [7]. However, while this approach is promising, it may be penalized by the difficulty in designing new r-operators.

In this paper, we present the fundation of the r-operators by introducing a generalization of the idempotent semi-groups, called *r-semi-group*. We establish the requirements on the operators to be used in distributed computation and we show that the r-semi-groups fulfill them. We investigate the connections between semi-groups and r-semi-groups, in order to ease the design of r-operators. We then show how to build new r-operators, to solve new algorithmic problems.

With these new results, the r-semi-groups appear to be a powerful tool to design stabilizing silent tasks.

1 Introduction

1.1 Aim

Modeling silent tasks. Distributed algorithms resolve either static tasks (such as distance computation) or dynamic tasks (such as token circulation). The aim of a static task is to compute a global result, which means that after a running time, processors always produce the same output (such as the distance from a source). The specification of a static task is given by the final processor's outputs. To solve such a task, each processor uses a local algorithm, which builds some new outputs with the data owned by the processor itself and the data it has received from some other nodes.

These outputs can be seen as the result of an algebraic computation on the set to which belong all the data in the distributed system. The local algorithm is then assimilated to an algebraic operator, that operates on this set. For instance, for the distance computation problem, the local algorithm can be assimilated to the operator $\min(x, y+1)$ defined on the set $\mathbb{N}$ of the integers, where x denotes the private data of a node (either 0 for a source or any value larger than the diameter for the other nodes) and y denotes a data received from an ancestor [13].

Several operators could be used depending on the nodes, meaning that the modeling is not restricted to uniform distributed algorithms. Moreover, by adding some control information with the outputs, a node may take into account only a part of the received outputs, meaning that the modeling is not restricted to algorithms that always send the same output to all their neighbors (this is similar to point-to-point communications in a wireless network based on the receiver address field). Finally this modeling applies for the asynchronous message passing model where the outputs are sent asynchronously, and stored locally by the nodes before being used in their next local computation. Some generic results can then be derived to stronger models (shared memory, composite atomicity...).

Motivation. Modeling a distributed algorithm by means of algebraic computations allows to study the termination of a static task with classical methods, such as for instance fixed points [14]. Moreover the algebraic properties ensuring the termination can simply be checked on the local algorithm (*i.e.*, operator) instead of the whole system. And the results generally apply on a large range of applications by simply varying the operator, leading to generic proofs.

The modeling is also useful to study self-stabilizing algorithms. A distributed algorithm that solves a static task is self-stabilizing if it can reach a legitimate configuration while some transient failures occur [8]. A legitimate configuration is a configuration of the system (state of the processor's memories and communication channels) in which the processor's outputs fulfill the specification of the distributed algorithm. A self-stabilizing algorithm reaches such a configuration in finite time after the transient failures disappear. Self-stabilizing proofs are often long and difficult [16]. A generic modeling is then interesting to study a class of applications instead of a particular one: proofs and properties are shared by several distributed algorithms or applications.

1.2 r-Operators

In [10,13,14,7], the modeling of silent tasks by means of so-called *r-operators* has been studied, and interesting relations have been shown between algebraic properties of a given operator and stabilizing properties of the related distributed algorithms (Figure 1). The following array summarizes these results.

A large set of r-operators have been designed to solve different silent distributed tasks. On numerical sets, the minc operator defined by $\text{minc}(x, y) = \min(x, y+1)$ solves the distance computation and related problems. By using other r-mappings (such as $x \mapsto x + \text{weight}$), the multiple source shortest path problem is solved similarly [13]. The r-operator $\max(x, y \times \pi)$ solves the best

associative and comm. operator	silent task if there is no circuit	
associative, commutative and idempotent operator	silent task not self-stabilizing	[20] [13]
idempotent r-operator	silent task	[10,13]
strictly idempotent r-operator with total order	self-stabilizing read-write demon shared memory (registers) unreliable messages passing	 [13] [7]
strictly idempotent r-operator with partial order	self-stabilizing, shared memory and fully distributed demon	[14]

Fig. 1. Relations between algebraic properties of the operators and stabilizing properties of the related distributed algorithms

reliable paths related problems (π characterizes the link, $0 < \pi < 1$). The r-operator $\max(x, \min(x, \kappa))$ solves the best capacity paths related problems (κ is the capacity of the link). All these operators lead to self-stabilizing algorithms in unreliable messages passing systems [7]. Note that the operator max (special case of r-operator with the r-mapping identity) solves a silent task on any network, but is not self-stabilizing [13], except on networks without any circuit. In the same way, the r-operator $\max(x, y+1)$ leads to a self-stabilizing algorithm only if the network has no circuit (topological sort). The algorithm is not silent if there is a circuit.

With the (non numerical) set of identity's lists, ordered with the lexicographical order, and with the r-mapping that adds to the end of a list the identity of the node, one can build an r-operator that solves the depth-first-search tree [13]. With the set of lists of identity's sets, and with a mapping that shifts a list to the right, one can build an r-operator that gives for each node the ordered list of ancestors [14]. This is useful to discover the k-neighborhood (neighborhood until distance k), or the distance from the farthest node for instance (assuming there is a path between each pair of nodes in the network). To the contrary of the previous operators, this last one defines a partial order relation, and the self-stabilizing property of the resulting distributed algorithm is established for the fully distributed demon (instead of the read-write demon).

1.3 Contribution

The generic operators-based approach requires to model the algorithms with r-operators. A construction of the r-operators is then useful in order to take benefit of the already established general results. This paper introduces the r-semi-groups and explains how to build new r-operators from a practical point of view.

First, we establish the requirements on operators used in distributed computations. Next, we introduce the r-semi-groups and detail their construction. This leads to a more general definition of the r-operators than in previous work, because it allows to consider finite set in self-stabilizing proofs (which is interesting for implementation purpose). We prove fundamental properties of the r-semi-groups (requirements fulfilled, order relation) and we investigate the connections between r-semi-groups and semi-groups in order to ease the design of new r-operators. Finally, by using these results, we show how new r-operators can be designed for new distributed algorithms. This shows the interest of the r-operators to design or study stabilizing static tasks.

Several works deal with generic approaches to prove self-stabilizing properties of distributed algorithms [3,1,21,17,19]. However none of them relies on an algebraic modeling of the computations, though we think that this method is powerful, as the examples of already developed r-operators show.

Related works concern path algebra [2,15], while the path algebra structures require two laws (as in so-called *max-plus* algebra [4]). To the contrary of the r-operators, such algebra do not generalize the idempotent semi-groups but are some particular cases of semi-ring structures (*dioïd*); they have less applications for distributed algorithms. Other works deal with some relations between algebraic structures and computations on networks (see for instance [5]), but they do not define operators for useful distributed applications as the r-operators do.

Outlines. In the next section, a generic computing model is presented and some requirements on the operators are established. In Section 3, we present useful properties related to the modeling and to the requirements. Then, the *r-semi-groups* are introduced in Section 4, as well as their main properties. In particular, we show that they fulfill the requirements. In Section 5, the connections with the idempotent Abelian semi-groups are given. These results are used in Section 6 to build new r-operators. We explain step-by-step with an example how to design new r-operators for solving or studying new distributed silent tasks. Concluding remarks end the paper in Section 7.

2 Modeling Distributed Computations with Algebraic Operators

In this section, we explain how distributed computations can be modeled by means of algebraic operators, and we exhibit some requirements on the operators.

2.1 Generic Model

The network is modeled by a directed graph $G(V, E)$, composed of nodes (vertices $v \in V$) and unidirectional communication links (edges $(u, v) \in E$), such that each node owns a *private data*, some *inputs*, an *output* and an *operator* (Figure 2). The topology of the network is unknown.

The private data, inputs and outputs represent some kind of memory cells. The private data is fixed and cannot be corrupted (*e.g.*, it is stored in a read-only memory). The inputs and outputs are updated from time to time and could be corrupted. Reads and writes are atomic on these variables, meaning that a variable cannot be read (resp. write) if it is currently writing (resp. reading). Each edge (u, v) connects the output of u to the input of v, and is used to update the input of v by the output of u.

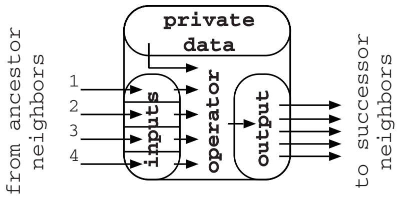


Fig. 2. A generic node

A *local computation* is performed by a node v, and consists in updating its output by applying its operator (denoted by $\diamond$) on its private data and its inputs as follows (Figure 2):

$$\text{output} \leftarrow \text{private data} \diamond \text{first input} \diamond \cdots \diamond \text{last input} \tag{1}$$

A *global computation* is composed of local computations (that update some outputs) and communications (that update some inputs with some ancestor's outputs). In a message-passing environment, a local computation is performed upon arrival of a new value and the output is sent to the descendant in the network when it has changed. However, to prevent any memory or message corruption, the local computations and the sending can be performed regularly. To model the non-determinism of the distributed system, it is assumed that the node activity is managed by a global scheduler, considered as an adversary (demon) [9,18,22].

2.2 Requirements

When the directed graph G has no circuit, the outputs of the sources (nodes without ancestors) are always equal to their private data. The output of a source's successor stabilizes on a value equal to the result of an expression composed with the private data and the operators of the ancestor sources and of the node itself. All the node outputs, from the sources to the sinks (nodes without successors), stabilize similarly. However the result of each node does depend on the underlying network (which is responsible of the form of the output expressions) while several different networks can be modeled by the same graph (see Figure 3). We then state the following requirement:

Requirement 1. *The operators should not be aware of the network wiring, in order to obtain a result that depends on the graph topology (and not on the local neighborhood numbering).*

Moreover when the directed graph G has some circuit, the outputs could never stabilize. This is the case on the graph represented in Figure 3 when using the addition on the (non nul) integers $\mathbb{N}$. We then state a second requirement:

Requirement 2. *The operators should not be aware of the circuits.*

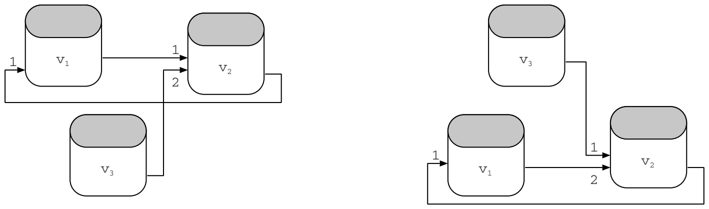


Fig. 3. A single graph but two networks (different wiring). Operators should not be aware of the wiring nor the loops.

2.3 Operators

In [20] it is shown that when instantiated by a so-called infimum (that is the law of an idempotent Abelian semi-group, such as the minimum on the integers), the global computation terminates. Note that these operators are associative, commutative and idempotent, and fulfil the above requirements related to the wiring and the circuits of the networks. However the applications of idempotent Abelian semi-groups are limited because a distributed computation with one of these operators gives the same result on all the nodes belonging to the same strongly connected component. Moreover, these operators do not support any transient failures [13]: the computation of the smallest value in the network with the minimum operator for instance can not recover if a value smaller than all private data appears in a node due to a transient failure.

In fact, the properties defining the idempotent commutative semi-groups are not necessary for computing on networks. In particular, some non associative nor commutative operators still fulfill the above requirements related to wiring and circuits in the network. In [10], the idempotent r-operators have been introduced. Since then, several applications have been developed both for parallel (see [12] and the references herein) and distributed computations [13,14,7]. In [10], it has been shown that the r-operators lead to the termination of static tasks, extending the result known for the infimum [20].

In [13], it has been shown that, to the contrary of the infimum, the strictly idempotent r-operators support transient failure. This has been proved in a shared memory model, with the weakest synchronization hypothesis (read-write demon), and assuming that the order relation induced by the operator is total. In [7], this result has been extended to the unreliable message passing networks; fair message loss, finite message duplication and arbitrary message reordering are supported.

In [14], a proof has been given for operators that only define a partial order relation, which enlarged the set of operators, and then the set of applications. In this case, a few more constraints is necessary regarding the nodes synchronization (fully distributed demon). The proof has been established using max-plus algebra [4] and fixed-point related techniques. The r-operators permitted to model the distributed computations by means of asynchronous matrix iterations.

3 Prerequisites

In this section, we introduce three prerequisites related to the initialization (identity element), the cancellation and the wiring awareness. We assume that all the data belong to the set $\mathbb{S}$, and that $(\mathbb{S}, \diamond)$ denotes a set $\mathbb{S}$ endowed by a law $\diamond$ (also called operator). In other words, $(\mathbb{S}, \diamond)$ is a so-called *magma* [6] Moreover in all this paper, we omit the left bracketing: $x \diamond y \diamond z = (x \diamond y) \diamond z$.

3.1 Right Identity Element

Since a local computation can be performed at any time, without any consideration on the communications, a node could perform a local computation before having received any value from some of its ancestors. We then admit the following hypothesis:

Hypothesis 3. *The operator* $\diamond$ *admits a* right identity element*.*

Supposing that all the inputs of a node v are initialized by a right identity element $e_\diamond$ of the local operator $\diamond$ used by the node, the output obtained by a local computation is equal to the private data x: $x = x \diamond e_\diamond \diamond \cdots \diamond e_\diamond$. Hence, this is equivalent to initialize the inputs by a right identity element of the local operator, and each output by the private data of the node. Note that self-stabilizing computations do not require initializations.

Weak left cancellation. Computations based on idempotent operators (such as the minimum on the integers) cannot be simplified with inverse elements (as we do with the addition) because there is no inverse element. An operator $\diamond$ is left cancellative on $\mathbb{S}$ if $\forall x, y, z \in \mathbb{S}, x \diamond y = x \diamond z \Rightarrow y = z$. We introduce a new weaker property that allows some simplifications.

Let y, z be two elements of $\mathbb{S}$ endowed by $\diamond$. We have $(y = z) \Rightarrow (\forall x \in \mathbb{S}, x \diamond y = x \diamond z)$ but the reciprocal could be false (consider the silly operator $\oslash_1$ defined by $x \oslash_1 y = x$ for any $x, y \in \mathbb{S}$).

Definition 1. *A magma* $(\mathbb{S}, \diamond)$ *is* weak left cancellative *iff:*

$$\forall y, z \in \mathbb{S}, \quad (\forall x \in \mathbb{S}, x \diamond y = x \diamond z) \Leftrightarrow (y = z).$$

Most usual operators are weak left cancellative on their definition set. This is in fact a really reasonable hypothesis that could be interpreted as follows: "if no element of $\mathbb{S}$ disagrees with the fact that y could be equal to z, then it is established that $y = z$".

To point out the difference between the left cancellative and the weak left cancellative property, let consider the set of integers $\mathbb{N}$ endowed by the min operator. If $(\mathbb{N}, \min)$ is left cancellative, then: $\forall x, y, z \in \mathbb{N}, (\min(x, y) = \min(x, z)) \Rightarrow (y = z)$. But this is false with $x = 2$, $y = 3$ and $z = 4$. Now, if $(\mathbb{N}, \min)$ is weak left cancellative, then: $\forall y, z \in \mathbb{N}, (\forall x \in \mathbb{N}, \min(x, y) = \min(x, z)) \Rightarrow (y = z)$. This is always true. Indeed, if $\min(x, y) = \min(x, z)$ is true for any $x \in \mathbb{N}$, by considering $x = y$ and $x = z$ we obtain $\min(y, y) = \min(y, z) = \min(z, z)$, and then $y = z$.

This example shows that an operator can be both weak left cancellative and idempotent, while this is not the case with the left cancellative property. Hence, while idempotent semi-groups are not regular, some simplifications in equations will be possible, on the basis of the weak left cancellation. We will see that this is very useful. The following proposition states that any usual algebraic structure is weak left cancellative (such as semi-groups and groups):

Proposition 1. *If the magma* $(\mathbb{S}, \diamond)$ *admits a left identity element, then it is weak left cancellative.*

Finally, Hypothesis 3 states that the operator $\diamond$ admits a right identity element. By weak left cancellative, it is unique:

Proposition 2. *If the magma* $(\mathbb{S}, \diamond)$ *is weak left cancellative, then it admits at most one right identity elements.*

Proof. Suppose that $e^1_\diamond$ and $e^2_\diamond$ are two right identity elements of $(\mathbb{S}, \diamond)$. Then they verify: $\forall x \in \mathbb{S}, x \diamond e^1_\diamond = x = x \diamond e^2_\diamond$. By weak left cancellation, we have $e^1_\diamond = e^2_\diamond$.

3.2 Rank 2 Commutativity

In order to ensure that the distributed computation is not aware of the wiring of the network, we introduce the following property. When it is fulfilled by an operator, it indicates that the operator is not aware of the order of the inputs received by its ancestors (note that, by convention, the first input of a local computation is always the private data of the processor, see Equation 1). This means that the same result will be obtained on the two representations of the same graph in Figure 3 for instance.

Definition 2. *The magma* $(\mathbb{S}, \diamond)$ *is rank 2 commutative on* $\mathbb{S}$ *if:*

$$\forall x, y, z \in \mathbb{S}, \quad x \diamond y \diamond z = x \diamond z \diamond y.$$

For instance, the $\oslash_2$ operator defined on the integers by $x \oslash_2 y = x + 2y$ is not commutative but it is rank 2 commutative: $x \oslash_2 y \oslash_2 z = x + 2y + 2z = x + 2z + 2y = x \oslash_2 z \oslash_2 y$.

In a weak left cancellative associative magma, if the operator is rank 2 commutative, then it is commutative. An associative and commutative operator is rank 2 commutative. Hence, the rank 2 commutativity property is only useful when the operator is not associative.

3.3 Rank 2 Idempotency

As shown by Equation 1, the local computations produce expressions built with the operators, the private data, and the received inputs, that can themselves be considered as expressions. In order to ensure the termination of the distributed computations even in presence of circuits (as in Figure 3), such expressions should be finite. We then introduce the following property, that allows to reduce the expressions, providing there is no brackets. We will show later how brackets can be "eliminated" from expressions even with non associative operators (note that other forms of idempotency have been introduced in [11]).

Definition 3. *A magma* $(\mathbb{S}, \diamond)$ *is* rank 2 idempotent *if it satisfies:*

$$\forall x, y \in \mathbb{S}, \quad x \diamond y \diamond y = x \diamond y.$$

For instance, the operator $\oslash_3$ defined on the integers by $x \oslash_3 y = x$ if $x < y$ and $x \oslash_3 y = 0$ if $x \geq y$ is not idempotent but is rank 2 idempotent [11].

4 r-Semi-groups

In this section, we define the r-semi-groups, which generalize the idempotent Abelian semi-groups [6,20,4,11].

4.1 Definition

We begin by three definitions, that generalizes the classical properties of associativity, commutativity and idempotency[1].

Definition 4. *Let* $r : \mathbb{S} \to \mathbb{S}$ *be an application. The magma* $(\mathbb{S}, \diamond)$ *is* r-associative *if it satisfies:* $\quad \forall x, y, z \in \mathbb{S}, \quad x \diamond (y \diamond z) = x \diamond y \diamond r(z).$

Definition 5. *Let* $r : \mathbb{S} \to \mathbb{S}$ *be an application. The magma* $(\mathbb{S}, \diamond)$ *is* r-commutative *if it satisfies:* $\quad \forall x, y \in \mathbb{S}, \quad r(x) \diamond y = r(y) \diamond x.$

Definition 6. *Let* $r : \mathbb{S} \to \mathbb{S}$ *be an application. The magma* $(\mathbb{S}, \diamond)$ *is* r-idempotent *if it satisfies:* $\quad \forall x \in \mathbb{S}, \quad r(x) \diamond x = r(x).$

For instance, the operator $\oslash_2$ (defined on $\mathbb{N}$ by $x \oslash_2 y = x + 2y$) is not associative nor commutative. But it is r-associative and r-commutative, with the mapping $r : x \mapsto 2x$. And the operator $\oslash_3$ (defined above) is not idempotent but is r-idempotent with the mapping $r : x \mapsto x - 1$. Of course, when the application r is the identity mapping $x \mapsto x$, these three properties are equivalent to the classical ones. We now define the r-semi-group using the previous definitions.

Definition 7 (r-semi-group). *Let* $(\mathbb{S}, \triangleleft)$ *be a weak left cancellative magma admitting the right identity element* $e_\triangleleft$*, and let* $r : \mathbb{S} \to \mathbb{S}$ *be an endomorphism. Then* $(\mathbb{S}, \triangleleft)$ *is an* r-semi-group *if it is r-associative, r-commutative, r-idempotent with the application* r.

An *endomorphism* r of $(\mathbb{S}, \diamond)$ is an application from $\mathbb{S}$ to $\mathbb{S}$ satisfying $r(x \triangleleft y) = r(x) \triangleleft r(y)$ for any $x, y \in \mathbb{S}$. The law of an r-semi-group is called *r-operator*, and the mapping r is called *r-mapping*. Note that in previous works [10], the r-operators relied on a bijective mapping instead of an endomorphism. This limited the self-stabilization proofs to infinite sets because the r-mapping needs to be expansive with respect to the order relation (see Proposition 6) [13,14]. Defining the r-operators with an endomorphism leads to the same properties (and then

[1] In the following, the prefix "r-" refers to the fact that an application is used in the definitions. However, while it is simpler to denote the application by r, it could be denoted differently without changing the definition's names.

the same proofs) while allowing to consider finite sets, which is important for implementation purpose (see [11] for more details). For instance, the operator $\oslash_4$ defined on $\mathbb{N}$ by $x \oslash_4 y = \min(x, 2y)$ is an r-operator on $\mathbb{N}$ with the mapping $x \mapsto 2x$, which is an endomorphism of $(\mathbb{N}, \oslash_4)$ but not a surjective mapping on $\mathbb{N}$ because odd integers are not reached.

4.2 Properties

By Proposition 2, the right identity element $e_\triangleleft$ is unique. In the same way, the r-mapping r of an r-semi-group is unique:

Proposition 3. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group. Then the following holds: (i)* r *is unique and satisfies* $\forall x \in \mathbb{S}$, $r(x) = e_\triangleleft \triangleleft x$*, (ii)* r *is injective, (iii)* $\forall x \in \mathbb{S}$, $r(x) = e_\triangleleft \Leftrightarrow x = e_\triangleleft$.

Proof. (i) For any $x, y \in \mathbb{S}$, $x \triangleleft r(y) = x \triangleleft e_\triangleleft \triangleleft r(y)$. Then, by r-associativity, we have $x \triangleleft r(y) = x \triangleleft (e_\triangleleft \triangleleft y)$. By weak left cancellation, we then obtain $r(y) = e_\triangleleft \triangleleft y$ and r is unique by Proposition 2. (ii) For any $a, b, x, y \in \mathbb{S}$, if $r(x) = r(y)$, we have $a \triangleleft b \triangleleft r(x) = a \triangleleft b \triangleleft r(y)$. Then, by r-associativity, we have $a \triangleleft (b \triangleleft x) = a \triangleleft (b \triangleleft y)$. Then, by weak left cancellation, we have $b \triangleleft x = b \triangleleft y$ and finally $x = y$. Thus r is injective. (iii) Since $r(e_\triangleleft) = e_\triangleleft$ and r is injective, if $r(x) = e_\triangleleft$, then $x = e_\triangleleft$.

This proof illustrates the interest of the weak left cancellation. We now prove that the r-semi-groups are rank 2 commutative and rank 2 idempotent, which are two important properties for computing in networks, as seen in the previous section.

Proposition 4. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group. Then* $\triangleleft$ *is rank 2 commutative on* $\mathbb{S}$.

Proof. Let $x, y, z \in \mathbb{S}$. We have $r(x \triangleleft y \triangleleft z) = r(x) \triangleleft r(y) \triangleleft r(z) = r(x) \triangleleft (r(y) \triangleleft z)$ (endomorphism and r-associativity). By r-commutativity, we have $r(x) \triangleleft (r(y) \triangleleft z) = r(x) \triangleleft (r(z) \triangleleft y)$. Then we have: $r(x) \triangleleft (r(z) \triangleleft y) = r(x) \triangleleft r(z) \triangleleft r(y) = r(x \triangleleft z \triangleleft y)$ (r-associativity and endomorphism). Hence, for any $x, y, z \in \mathbb{S}$, $r(x \triangleleft y \triangleleft z) = r(x \triangleleft z \triangleleft y)$. Thus, by Proposition 3, $x \triangleleft y \triangleleft z = x \triangleleft z \triangleleft y$ (injection).

Proposition 5. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group. Then* $\triangleleft$ *is rank 2 idempotent on* $\mathbb{S}$.

Proof. For any $x, y \in \mathbb{S}$, we have: $r(x \triangleleft y \triangleleft y) = r(x) \triangleleft r(y) \triangleleft r(y)$ (endomorphism). Then, by r-associativity, we obtain: $r(x \triangleleft y \triangleleft y) = r(x) \triangleleft (r(y) \triangleleft y)$. Then $r(x \triangleleft y \triangleleft y) = r(x) \triangleleft r(y)$ by r-idempotency, and finally $r(x \triangleleft y \triangleleft y) = r(x \triangleleft y)$ (endomorphism). Thus, by Proposition 3, $x \triangleleft y \triangleleft y = x \triangleleft y$ (injection).

An idempotent Abelian semi-group $(\mathbb{S}, \oplus)$ admits the order relation $\preceq_\oplus$ defined by $x \preceq_\oplus$ iff $x \oplus y = x$. The following proposition generalizes this result to r-semi-groups.

Proposition 6. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group. Then the following binary relation* $\preccurlyeq_{\triangleleft}$ *is an order relation:* $\forall x, y \in \mathbb{S}, \quad x \preccurlyeq_{\triangleleft} y \quad \Leftrightarrow \quad e_{\triangleleft} \triangleleft x \triangleleft y = e_{\triangleleft} \triangleleft x.$

Proof. (i) By rank 2 idempotency (Proposition 5), we have: $\forall x \in \mathbb{S}, e_{\triangleleft} \triangleleft x \triangleleft x = e_{\triangleleft} \triangleleft x$ and $\preccurlyeq_{\triangleleft}$ is reflexive. (ii) Suppose that $x \preccurlyeq_{\triangleleft} y$ and $y \preccurlyeq_{\triangleleft} x$ hold. Then $e_{\triangleleft} \triangleleft x \triangleleft y = e_{\triangleleft} \triangleleft x$ and $e_{\triangleleft} \triangleleft y \triangleleft x = e_{\triangleleft} \triangleleft y$. By rank 2 commutativity (Proposition 4), we have $e_{\triangleleft} y \triangleleft x = e_{\triangleleft} \triangleleft x \triangleleft y$ and then $e_{\triangleleft} \triangleleft x = e_{\triangleleft} \triangleleft y$. By Proposition 3, this gives $r(x) = r(y)$ and then $x = y$. Hence $\preccurlyeq$ is antisymmetric. (iii) Suppose that $x \preccurlyeq_{\triangleleft} y$ and $y \preccurlyeq_{\triangleleft} z$. Then we have $e_{\triangleleft} \triangleleft x = e_{\triangleleft} \triangleleft x \triangleleft y$ and $e_{\triangleleft} \triangleleft y \triangleleft z = e_{\triangleleft} \triangleleft y$. By rank 2 commutativity (Proposition 4), we have: $e_{\triangleleft} \triangleleft x = e_{\triangleleft} \triangleleft y \triangleleft x$. By substitution, we obtain $e_{\triangleleft} \triangleleft x = e_{\triangleleft} \triangleleft y \triangleleft z \triangleleft x$. By rank 2 commutativity, we have $e_{\triangleleft} \triangleleft x = e_{\triangleleft} \triangleleft x \triangleleft y \triangleleft z$. By substitution, we obtain $e_{\triangleleft} \triangleleft x = e_{\triangleleft} \triangleleft x \triangleleft z$. Hence $x \preccurlyeq_{\triangleleft} z$ and $\preccurlyeq_{\triangleleft}$ is transitive. Since $\preccurlyeq_{\triangleleft}$ is reflexive, antisymmetric and transitive, it is an order relation.

In an idempotent r-semi-group $(\mathbb{S}, \triangleleft)$, the r-mapping r is expansive ($\forall x \in \mathbb{S}, x \preccurlyeq_{\triangleleft} r(x)$) and reciprocally. Indeed, $e_{\triangleleft} \triangleleft x \triangleleft r(x) = e_{\triangleleft} \triangleleft (x \triangleleft x) = e_{\triangleleft} \triangleleft x$. Thus, when $x \preccurlyeq_{\triangleleft} r(x)$ and $x \neq r(x)$, the r-semi-group is called *strictly idempotent*. This property is useful for self-stabilizing proofs.

5 Connections with Idempotent Semi-groups

In this section, we precise the link between idempotent semi-groups and r-semi-groups. This allows to design new r-operators (see Section 6). Note that in previous works, the r-operators were defined with a surjective mapping, then bijective by Proposition 3. In this case, if $(\mathbb{S}, \triangleleft)$ is an r-semi-group, then $(\mathbb{S}, \oplus)$ is an idempotent Abelian semi-group with $\oplus$ defined by $x \oplus y = r(x) \oplus y$. The Corollary 1 established here is more general. Some propositions are needed before.

Proposition 7. *An r-semi-group* $(\mathbb{S}, \triangleleft)$ *is an idempotent commutative semi-group if and only if the r-mapping is the identity mapping* $r : x \mapsto x$.

Proof. The only points to check concern the weak left cancellation and the identity element. Since a semi-group owns an identity element both on the left and on the right, it is weak left cancellative by Proposition 1. Conversely, an r-semi-group with the r-mapping $r : x \mapsto x$ verifies: $\forall x \in \mathbb{S}$, $r(x) = x = e_{\triangleleft} x$ (Proposition 3). This means that $e_{\triangleleft}$ is an identity element (both on the left and on the right).

Note that Proposition 3 is still true when r is the identity mapping. Moreover, an idempotent commutative semi-group is rank 2 commutative and rank 2 idempotent. And its order relation $\preceq_{\oplus}$ appears as a particular case of the order relation $\preccurlyeq_{\triangleleft}$ of an r-semi-group. Indeed we have $x \preccurlyeq_{\triangleleft} y \Leftrightarrow r(x) \triangleleft y = r(x)$ and with $r : x \mapsto x$, we find $x \preccurlyeq_{\oplus} y \Leftrightarrow x \triangleleft y = x$.

We now show that any r-semi-group induces an idempotent Abelian semi-group. Let define the following property:

$$\forall x, y \in \mathbb{S}, \qquad r(x) \triangleleft y \in r(\mathbb{S}) \tag{2}$$

Proposition 8. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group satisfying Property* 2*. Then* $(r^2(\mathbb{S}), \oplus)$ *is a magma with* $\oplus$ *defined by* $r^2(x) \oplus r^2(y) = r^2(x) \triangleleft r(y)$*.*

Proof. For any $x, y \in \mathbb{S}$, we have: $r^2(x) \oplus r^2(y) = r^2(x) \triangleleft r(y)$ by definition of $\oplus$. Then, by homomorphism, we have: $r^2(x) \oplus r^2(y) = r(r(x) \triangleleft y)$. By Property 2, there exists $z \in \mathbb{S}$ such that $r(x) \triangleleft y = r(z)$. Then $r^2(x) \oplus r^2(y) = r^2(z) \in r^2(\mathbb{S})$ and $\oplus$ is an internal composition law on $r^2(\mathbb{S})$.

In the following, $\oplus$ denotes the internal law defined on $r^2(\mathbb{S})$ by: $r^2(x) \oplus r^2(y) = r^2(x) \triangleleft r(y)$.

Proposition 9. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group satisfying Property* 2*. Then* $\oplus$ *is associative on* $r^2(\mathbb{S})$*.*

Proof. For any $x, y, z \in \mathbb{S}$, we have by definition of $\oplus$ and homomorphism property of r on $(\mathbb{S}, \triangleleft)$: $r^2(x) \oplus (r^2(y) \oplus r^2(z)) = r^2(x) \oplus (r^2(y) \triangleleft r(z)) = r^2(x) \oplus r(r(y) \triangleleft z) = r^2(x) \triangleleft (r(y) \triangleleft z)$. Then, by r-associativity of $\triangleleft$, we have: $r^2(x) \triangleleft (r(y) \triangleleft z) = r^2(x) \triangleleft r(y) \triangleleft r(z)$. Since $r^2(x) \oplus r^2(y) \oplus r^2(z) = r^2(x) \triangleleft r(y) \triangleleft\ r(z)$, $\oplus$ is associative on $r^2(\mathbb{S})$.

Proposition 10. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group satisfying Property* 2*. Then* $\oplus$ *is commutative on* $r^2(\mathbb{S})$*.*

Proof. For any $x, y \in \mathbb{S}$, we have by definition of $\oplus$ and homomorphism property of r on $(\mathbb{S}, \triangleleft)$: $r^2(x) \oplus r^2(y) = r^2(x) \triangleleft r(y) = r(r(x) \triangleleft y)$. Then, by r-commutativity, we have: $r(r(x) \triangleleft y) = r(r(y) \triangleleft x)$. Then we have: $r(r(y) \triangleleft x) = r^2(y) \triangleleft r(x) = r^2(y) \oplus r^2(x)$ (definition of $\oplus$ and homomorphism).

Proposition 11. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group satisfying Property* 2*. Then* $\oplus$ *is idempotent on* $r^2(\mathbb{S})$*.*

Proof. For any $x \in \mathbb{S}$, we have by definition of $\oplus$ and homomorphism property of r on $(\mathbb{S}, \triangleleft)$: $r^2(x) \oplus r^2(x) = r^2(x) \triangleleft r(x) = r(r(x) \triangleleft x)$. Then, by r-idempotency, we have: $r(r(x) \triangleleft x) = r(r(x))$. Thus $r^2(x) \oplus r^2(x) = r^2(x)$ and $\oplus$ is idempotent on $r^2(\mathbb{S})$.

We then have the following corollary.

Corollary 1. *Let* $(\mathbb{S}, \triangleleft)$ *be an r-semi-group satisfying* $\forall x, y \in \mathbb{S}, r(x) \triangleleft y \in r(\mathbb{S})$*. Then* $(r^2(\mathbb{S}), \oplus)$ *is an Abelian idempotent semi-group, where* $\oplus$ *is defined by* $r^2(x) \oplus r^2(y) = r^2(x) \triangleleft r(y)$*.*

The reader will find in [11] a figure that summarizes the construction of the r-semi-groups, as well as the connections with idempotent Abelian semi-groups.

6 Designing New r-Operators

In this section, we explain how to build new r-operators to solve new algorithmic problems, thanks to the previous results. The idea is to start from an idempotent Abelian semi-group, and then to derivate an r-semi-group with an endomorphism.

1) Suppose that we seek an r-operator for solving a distributed silent task, without considering transient fault for the moment. In order to build the legitimate output of a node, a local computation can be considered as discarding some inputs while keeping some others. For the distance computation by example, if a node receives the values 3 from a first neighbor and 5 from a second, it should keep 3 and discards 5. From this choice, we may deduce an order relation $\preccurlyeq$ on the set $\mathbb{S}$ to which belong all the values such that the discarded values are larger than the others (the natural order on the integers for the distance computation). From this order, we may deduce an operator $\oplus$ such that $(\mathbb{S}, \oplus)$ is an idempotent commutative semi-group. For the distance, we find $(\mathbb{N}, \min)$ for instance.

2) If the output of a node is larger than the smallest of its input, then we may deduce an application r that increases its input (*i.e.*, $x \preccurlyeq r(x)$) such that the output of the node is equal to the smallest of its inputs increased by r. In other word, the output would be equal to $x_0 \oplus r(x_1) \oplus \cdots \oplus r(x_p)$ where x_0 is the private data of the node and $x_1 \dots x_p$ are its inputs. For instance, for the distance computation, if a node receives both 3 and 5, its output (representing its distance from a source in the network) will be equal to 4. We then define $r : \mathbb{N} \to \mathbb{N}$ by $r(x) = x + 1$.

3) On an other hand, if the output of a node is equal to the smallest of its inputs, this may indicate that the silent task is solved with an operator issued from an idempotent commutative semi-group (*i.e.*, r is the identity mapping $x \mapsto x$). Note that if the output is strictly smaller than its inputs for every nodes, then the task may not be silent when there are some circuits in the network: the output will be smaller and smaller.

4) From the results of Section 5, we can then define an r-operator using $\oplus$ and r using the relation $r^2(x) \triangleleft r(y) = r^2(x) \oplus r^2(x)$. The characteristic of both $\preccurlyeq$ and r allow to determine the properties regarding termination and self-stabilization of the related distributed algorithm. For the distance computation, the order $\leq$ is total on $\mathbb{N}$ and $x \mapsto x + 1$ is strictly expansive. Hence the r-operator minc – defined by $\text{minc}(x, y) = \min(x, y + 1)$ – leads to a self-stabilizing distributed algorithm even in unreliable messages passing networks.

This simple and generic method is convenient to prove the termination or the self-stabilizing property of a given static task. Stabilizing properties of a distributed algorithm appear as a result from an equilibrium between the data decreasing (semi-group, min for the distance) and increasing (endomorphism, $x \mapsto x + 1$ for the distance), which leads to the convergence. For more complex problems, a preliminary work may be necessary to define the set $\mathbb{S}$ (see examples in Section 1.2).

Assuming a synchronous demon, the time complexity of the generic distributed algorithm relying on r-operators presented in [13,14,7] is in $O(D + |\mathbb{S}|)$ where D denotes the diameter of the network and $|\mathbb{S}|$ denotes the cardinal of the definition set of the operator.

7 Conclusion

A distributed algorithm resolving a static task stabilizes when it reaches a terminal configuration. It is self-stabilizing if it can reach such a legitimate terminal configuration while some transient failures may occur in the network. Proving the self-stabilization property of such an algorithm leads generally to long and complex proofs. Several works deal with the design of a generic approach instead.

We developed an algebraic approach and we pointed out the connection between the algebraic structure to which belongs the local operators and the stabilization of the distributed algorithms. Hence, when some proofs have been established for a given algebraic structure, it is sufficient to check whether the local algorithm is or not an operator of such a structure instead of writing some potentially long proofs from scratch. An important point is that the behavior of the whole distributed system is characterized by some local properties. Several r-operators have been designed for solving different static tasks such as distance related problems (weighted distance, single or multiple source shortest path...), depth-first-search tree computation, ordered list of ancestors, topological sort, diameter computation... These operators lead to silent or self-stabilizing algorithms in different distributed systems, including the unreliable messages passing networks, depending on their algebraic properties. However, while this approach is promising, it may be penalized by the difficulty in designing new r-operators.

This paper focused on the construction of the r-operators in the aim of facilitating the development of new ones. We introduced the r-semi-group, a generalization of the idempotent semi-group, and presented their construction and properties. By these properties, we explained how to derivate new r-operators for solving new static tasks. Moreover the r-semi-group construction we introduced allows to extend the previous definition of the r-operators to finite sets, which is more convenient for an implementation purpose.

The r-semi-groups have many applications both in parallel [12] and distributed computations. They appear as a powerful tool for studying silent or self-stabilizing static tasks. We guess that new static tasks will be solved and studied this way, especially in wireless ad hoc networks, which are unreliable messages passing directed networks, and where transformers cannot apply.

Open problems concern the relationship between the partial versus total order relation and the kind of admissible demon, as well as the completeness of the r-operators.

Acknowledgment. The author wishes to thank Sébastien Tixeuil for interesting remarks.

References

1. Afek, Y., Bremler, A.: Self-stabilizing unidirectional network algorithms by power supply. Chicago Journal of Theoretical Computer Science 4(3), 1–48 (1998)
2. Aho, A., Hopcropft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. In: Series in Computer Science and Information Processing, Addison-Wesley Publishing, Reading (1974)

3. Arora, A., Attie, P., Evangelist, M., Gouda, M.: Convergence of iteration systems. Distributed Computing 7, 43–53 (1993)
4. Baccelli, F., Cohen, G., Olsder, G., Quadrat, J.-P.: Synchronization and Linearity, an algebra for discrete event systems. In: Series in Probability and Mathematical Statistics, Wiley, Chichester (1992)
5. Bilardi, G., Preparata, F.: Characterization of associative operations with prefix circuits of constant depth and linear size. SIAM Journal of Computing 19(2), 246–255 (1990)
6. Bourbaki, N.: *Algèbre*. In: Éléments de Mathématiques, 2nd edn. Hermann, Paris.(Fascicule IV, Livre II) (1964)
7. Delaët, S., Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators revisited. Journal of Aerospace Computing, Information, and Computation, A previous version appeared in SSS'2005, Barcelona (2006)
8. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
9. Dolev, S., Israeli, A., Moran, S.: Self stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing 7, 3–16 (1993)
10. Ducourthial, B.: New operators for computing with *associative nets*. In: Proc. of SIROCCO 1998, Amalfi, Italy, pp. 55–65. Carleton Scientific (1998)
11. Ducourthial, B.: *r*-semi-groups. Technical report, Heudiasyc UMR CNRS 6599, UTC, 2006. http://www.hds.utc.fr/~ducourth/bib/rap-rsemigroups-BDucourthial.pdf
12. Ducourthial, B., Sicard, N., Mérigot, A.: Efficient neighborhood-based computations on regions using scans. In: Proceeding of IEEE ICIP 2005, IEEE Computer Society Press, Los Alamitos (2005)
13. Ducourthial, B., Tixeuil, S.: Self-stabilization with r-operators. Distrib. Comp. 14(3), 147–162 (2001)
14. Ducourthial, B., Tixeuil, S.: Self-stabilization with path algebra. Theoretical Computer Science (Special issue on max-plus algebra)1(293), 219–236 (2003)
15. Gondran, M., Minoux, M.: Graphes et Algorithmes. Eyrolles, Paris (1979)
16. Gouda, M.G.: The triumph and tribulation of system stabilization. In: Helary, J.-M., Raynal, M. (eds.) WDAG 1995. LNCS, vol. 972, pp. 1–18. Springer, Heidelberg (1995)
17. Gouda, M.G., Schneider, M.: Maximizable routing metrics. IEEE/ACM Transaction on Networking 11(4), 663–675 (2003)
18. Israeli, A., Jalfon, M.: Uniform self-stabilizing ring orientation. Inform. and Comput. 104, 175–196 (1993)
19. Oehlerking, J., Dhama, A., Theel, O.: Towards automatic convergence verification of self-stabilizing algorithms. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, Springer, Heidelberg (2005)
20. Tel, G.: Topics in Distributed Algorithms. In: Cambridge International Series on Parallel Computation, vol. 1, Cambridge University Press, Cambridge (1991)
21. Theel, O.: Exploitation of Ljapunov theory for verifying self-stabilizing algorithms. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, Springer, Heidelberg (2000)
22. Tsai, M.S., Huang, S.T.: A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. Parallel Proc. Letters 4(1-2), 65–72 (1994)

Global Predicate Detection in Distributed Systems with Small Faults

Felix C. Freiling[1] and Arshad Jhumka[2]

[1] Department of Computer Science, University of Mannheim, Germany
[2] Department of Computer Science, University of Warwick, UK

Abstract. We study the problem of global predicate detection in presence of permanent and transient failures. We term the transient failures as small faults. We show that it is impossible to detect predicates in an asynchronous distributed system prone to small faults even if nodes are equipped with a powerful device known as *failure detector sequencer* (denoted by Σ). To redress this impossibility, we introduce a theoretical device, known as a *small fault sequencer* (denoted by Σ_{SF}), and show that Σ_{SF} is necessary and sufficient for predicate detection. Unfortunately, we also show that Σ_{SF} cannot be implemented even in a synchronous distributed system. Fortunately, however, we show that predicate detection *can* be achieved with high probability in synchronous systems.

1 Introduction

The problem of detecting whether a predicate holds on the global state of the system (global predicate detection) lies at the heart of several important problems in distributed computing, e.g., testing and distributed debugging. For fault-free systems, predicate detection has already been extensively studied and is rather well-understood [11,2]. Predicate detection has also been studied in faulty systems, especially under the crash fault model [12,13,14]. In such cases, the predicates encompass predicates about the operational state of processes. In this stream of work, predicate detection has profited from the large body of work in the area of failure detection pioneered by Chandra and Toueg [5]. Gärtner and Pleisch [13] showed that it is impossible to detect general predicates even if a perfect failure detector is used, explaining why Garg and Mitchell [12] needed to restrict the class of predicates which they could detect. Gärtner and Pleisch [14] later defined an extension of a failure detector, called *failure detection sequencer*, denoted as Σ, which is necessary and sufficient to implement predicate detection in the presence of crashes. A failure detection sequencer yields perfect information both about the operational state of a process and about the state of the communication channel from some process to another. Gärtner and Pleisch [14] showed that Σ can be implemented in perfectly synchronous systems. For a complete survey, please refer to Gärtner and Pleisch [10].

To the best of our knowledge, no work has yet addressed the problem of predicate detection in presence of transient failures, which we address in this paper. We unify this work with previous work on predicate detection in presence of permanent failures [14]. There exists work that has addressed the problem of developing protocols that are resilient to both permanent and transient faults. In such instances, the studies have focused on the interplay between fault tolerance, which focuses on tolerating permanent failures of a *subset* of processes, and self-stabilization, which focuses on tolerating transient failures on *all* processes [1,4,15,7]. In other related work, Beauquier et al. [3] studied the concept of *detecting* transient failures in a self-stabilizing way. As a matter of contrast, in this paper, we consider the class of *small faults*, i.e., transient failures that only occur on application processes, and not on monitor processes, hence we do not address the fault class and the problem of self-stabilization.

Nesterenko and Arora [19] studied the problem of *dining philosophers* in presence of a new fault class, namely *malicious crash*, which is a special form of crash and transient failures where transient failures eventually lead to the crash of a process. So apart from focussing on a different problem (predicate detection), we also assume a different fault model than Nesterenko and Arora [19].

Contributions. In this paper, we study the problem of predicate detection in systems prone to crash and small faults. Investigation under such fault assumptions is desirable since such models offer a higher assumption coverage.Our contributions are the following:

1. We show that it is impossible to solve the problem of global predicate detection in asynchronous systems prone to small faults, even in the presence of Σ (failure detector sequencer, see Sect. 4).
2. We introduce a novel and powerful device called a *small fault sequencer*, denoted by Σ_{SF}. We present its interface, and we show that Σ_{SF} is necessary and sufficient for global predicate detection in Sect. 5.
3. We show that Σ_{SF} cannot be implemented even in a synchronous system.
4. Because Σ_{SF} cannot be implemented in a synchronous system, we show that, through careful analysis, an almost correct Σ_{SF} can still detect small faults with high probability. For this, we analyze the conditions under which Σ_{SF} can miss small faults, and perform a probabilistic analysis of their occurrences in Sect. 6.

Before presenting these results, we formalize the models used in the paper in Sect. 2 and define the predicate detection problem in Sect. 3. For lack of space, we relegate proofs to a technical report [9].

2 Model

2.1 Asynchronous Distributed System

An *asynchronous distributed system* consists of a set of *application processes* $\Pi = \{p_1, \ldots, p_n\}$ which communicate using message passing over bidirectional channels. Each process p_i has a local state s_i, which is determined by the values of its local variables. The state space of processes can be infinite but we assume that the set of all states of every process is countable. Furthermore, every process has a well-defined initial state. The *local algorithm* A_i of process p_i describes *state transitions* of s_i, denoted as *events*, which are formalized in a *next state relation* δ_i. We assume that for any state s, the set of possible next states $\delta(s)$ is finite. We distinguish

- *internal* events which just affect the local state of p_i,
- *send* events describing the sending of a message m to some process, and
- *receive* events representing the reception of a message m from some process.

The communication channels are assumed to be reliable, i.e., every message is sent to a designated recipient, only the recipient may receive the message, and the recipient is notified of the identity of the sender. No message is received twice. Furthermore, messages are delivered in FIFO order, meaning that if p first sends message m before message m' to q then m is received before m' at q.

We define the *causal order* as the smallest relation $\rightarrow$ on the events of an execution, which satisfies:

- If e and f are different events on the same process and e happens before f, then $e \rightarrow f$.
- If e is a *send* event and f is the corresponding *receive* event, then $e \rightarrow f$.
- $\rightarrow$ is transitive, i.e. for any events e, f, g such that $e \neq f$, if $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

The global state $S = (s_1, \ldots, s_n)$ is composed of the local states of the system's processes. Consequently, the *global intitial state* is the global state consisting of all initial states of the processes. A *distributed algorithm* $A = (A_1, \ldots, A_n)$ is the collection of the local algorithms. Hence the events of the local algorithms yield the events of the distributed algorithm: Internal events modify the state of the corresponding process, send and receive events additionally modify the set of messages in transit. A particular execution of a distributed algorithm is called a *distributed computation*. We depict distributed computations using space/time diagrams. As a programming notation we adapt the event-based programming technique used by Guerraoui and Rodrigues [16].

2.2 Failures

A *fault model* defines the way in which certain components of a system might be affected by faults. We consider two different types of faulty behavior in this paper: crash faults and small faults. We model both types of faults as special *fault events*.

A process suffers a *crash fault* if it simply stops to execute events. We say that *a process crashed in state s* if s is the state in which the crash event occurs.

A process suffers a *small fault* if it experiences a spontaneous state change which is neither a send, receive nor internal event. A small fault can change the values stored in variables in an arbitrary fashion. Instances of small faults are bit flips, stuck-ats etc. A state transition causing a planned program event is by definition no small fault. In this paper, we focus on small faults occurring in application processes, and only in those parts of the application processes which do not belong to the predicate detection algorithm or subsystem.

Note that fault events are also part of the causal order defined above. For example, if at some process a small fault event e_1 happens and afterwards a small fault event e_2 then $e_1 \to e_2$. Note that a causality point of view fault events are internal events of a process.

2.3 Observations

Since every local state s_i corresponds to a prefix of events of a local computation of p_i, a global state can be equivalently expressed as the union of all these events for every process. A global state S is *consistent* if the corresponding event set is left-closed with respect to $\to$. Intuitively, this means that all receive events have a corresponding send event in S.

It is well known that the set of all consistent global states of a distributed computation form a lattice [18]. An *observation* is a sequence of global states $S_1, S_2, S_3, \ldots$ such that S_1 is the global initial state and for every $i > 1$, S_i results from executing a single event on some process in global state S_{i-1}. In other words, an observation is a path through the lattice starting at the empty global state (bottom).

As an example consider Fig. 1. The starting global state is the point with coordinate $(0, 0)$. Coordinate $(0, 1)$ is not a consistent global state since event e_2 cannot occur before event e_1. Hence, any coordinates on the p_2 axis are not consistent global states. One possible observation by M is

$$(0, 0), (1, 0), (1, 1), (1, 2), (2, 2).$$

Another observation is

$$(0, 0), (1, 0), (2, 0), (2, 1), (2, 2).$$

In fact, any path through the lattice, starting from $(0, 0)$, with non-decreasing p_1 and p_2 coordinates is a possible observation.

2.4 Observation System

To perform predicate detection, we need a system which observes the application processes. For this purpose, we add a set of *monitor* processes to the system. The set of monitor processes is denoted by Φ. We augment the local algorithms of application processes in such a way that when an event e occurs at a process p_k (denoted by e_k), a *control message* is sent to all the monitor processes. Control messages do not interfere with the application messages sent by the application processes in their algorithms. Throughout a computation, monitor processes continuously collect control messages and construct a set of observations which are subsequently used for predicate detection. In practice, the same hardware running an application process also runs a monitor process.

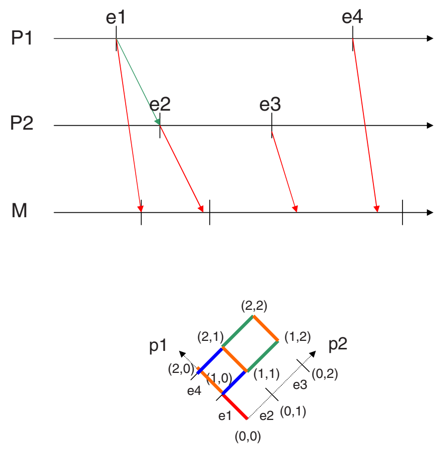


Fig. 1. Example of a lattice of consistent global states and observation

2.5 Failure Detector Sequencers Σ

We use the failure detection sequencer definition of Freiling, Henkel and Widder [8]: The sequencer Σ consists of a set of modules, one for each monitor process. The sequencer triggers events of the form $\langle$crash, p, $H\rangle$ where H is a sequence of (application of fault) events ending in a state s. In this case we say that "process p is suspected to have crashed in state s". The sequencer guarantees the following properties:

– (Integrity) Every process is suspected at most once for every crash.
– (Accuracy) If a process is suspected to have crashed in state s, it did crash in state s.
– (Completeness) If a process crashes in some state s, then it will eventually be suspected to have crashed in s.

Σ is similar to the concept of a failure detector but it is strictly stronger than any failure detector class as defined by Chandra and Toueg [5]. Gärtner and Pleisch [14] show that Σ can be implemented in synchronous systems.

3 Predicate Detection Problem

The predicates we wish to detect are *global predicates*, meaning that they are defined on global states. Using the lattice of consistent global states and given some global predicate φ, there can be different notions of what it means that "φ holds" in a computation (see for example *Possibly* and *Definitely* [2] or others [17]). Here we abstract from the particular modality. Instead we postulate a boolean function *modality* which tests the particular modality on a lattice of consistent global states.

Given a global predicate φ over the state of a distributed system, the predicate detection problem [6] consists of finding an algorithm that satisfies the following two requirements:

– (Safety) The algorithm does not raise the exception unless φ holds in the computation, and
– (Liveness) If φ holds, the algorithm will eventually raise the exception.

The assumption behind this definition is that the monitored computation never terminates. This is sometimes called *online predicate detection*, on which we focus in this paper.

4 Insufficiency of Σ

In a system prone to small faults, the occurrence of such a fault can cause a global predicate to hold. Of bigger importance is the fact that the predicate can hold between fault occurrences, making it necessary to detect the occurrence of every fault during system execution. To achieve this, the system needs to be equipped with a device that returns the state of the program. One such device is the failure detector sequencer Σ. This device returns the final state of a process prior to its crash. Unfortunately, even this very strong abstraction does not help to solve predicate detection under crash and small faults.

Proposition 1 (Impossibility of Predicate Detection)

1. *Given an asynchronous system in which* only *small faults may occur it is impossible to solve predicate detection for a general predicate φ.*
2. *Given an asynchronous system in which crash* and *small faults may occur it is impossible to solve predicate detection for a general predicate φ even if Σ is available.*

So even with as powerful a device as a failure detector sequencer, it is impossible to detect a general predicate in a system prone to small and crash faults. The problem is in fact that, when there is no crash, the sequencer does not offer any helpful information. To address this impossibility result, in the next section we identify a more powerful device that helps us solve the problem.

5 Solving Predicate Detection with Small Faults

5.1 Defining Σ_{SF}

To be able to detect the small faults, a more powerful device needs to be identified that captures the various state changes when faults occur. The device, called a *small fault Σ* (denoted Σ_{SF}), continuously monitors both the operational and functional state changes of the various processes, and has the following interface operations, i.e., this device yields two types of events:

1. $\langle$crash, p, $H\rangle$: process p crashed in the state following event sequence H.
2. $\langle$sf, p, e, $H\rangle$: p experienced a small fault event e following event sequence H.

Definition 1 (Σ_{SF}). *The* small fault failure detection sequencer *denoted Σ_{SF} is a device satisfying the following properties:*

– *Σ_{SF} does not issue the event ⟨crash, p, H⟩ unless p executed event sequence H and crashed in the state following H.*
– *If p crashes in a state following event sequence H then eventually Σ_{SF} issues the event ⟨crash, p, H⟩.*
– *Σ_{SF} does not issue the event ⟨sf,p, e, H⟩ unless p executed event sequence H and the small fault event e occurred following H at p.*
– *If small fault e happens at process p following event sequence H then eventually Σ_{SF} issues the event ⟨sf, p, e, H⟩.*

The idea behind Σ_{SF} is that the device retains at least the same capability with respect to crash faults as Σ, and can also detect occurrences of small faults.

5.2 Equivalence with Predicate Detection

We now show that Σ_{SF} is in fact necessary and sufficient to solve predicate detection in the presence of crash and small faults.

```
1  Upon ⟨Init⟩ do
2      $L = \{\}$ /* set of global states with ordering information */
3  Upon ⟨rcDeliver, p, e⟩ do
4      $L := L \cup e$
5      $L := \textit{join-closure}(L)$
6      if $\textit{modality}(L, \varphi)$ then
7          trigger ⟨detected, $\varphi$⟩
8  Upon ⟨crash, p, H⟩ do
9      $L := L \cup H$
10     $L := L \cup \textit{crash}(p, H)$ /* construct crash event after $H$ */
11     $L := \textit{join-closure}(L)$
12     if $\textit{modality}(L, \varphi)$ then
13         trigger ⟨detected, $\varphi$⟩
14 Upon ⟨sf, p, e, H⟩ do
15     $L := L \cup H \cup e$
16     $L := \textit{join-closure}(L)$
17     if $\textit{modality}(L, \varphi)$ then
18         trigger ⟨detected, $\varphi$⟩
```

Fig. 2. Algorithm for detecting global predicate φ using Σ_{SF}

Sufficiency. Fig. 2 depicts a predicate detection algorithm that uses Σ_{SF}. The central variable is the lattice data structure L which is a set of global states together with an ordering relation. To build the lattice, we assume that all events are tagged with corresponding information: For application events, this information is derived from the vector timestamp used for causal delivery (rcDeliver, line 3). For crash events and small fault events it can be derived from the event sequence H attached to every event from Σ_{SF}. We assume that a boolean function *modality* exists to check if a given predicate holds on the lattice for the chosen modality.

All event handlers of the algorithm are structurally similar: First some events are "added" to L. This means that they extend the axis of the process to which they are associated. Now L may not be a lattice anymore so we have to compute the closure of L. Note that we only have to compute the *join* closure since events are only appended to "the top" of the lattice (for the definition of the operations join and meet in the lattice of consistent global states see Mattern [18]). Subsequently L is checked using $modality(L, \varphi)$. If the function returns *true* the detection of φ is triggered.

The monitor has to process three kinds of events: (1) control message deliveries, (2) crash events issued by Σ_{SF}, and (3) small fault events issued by Σ_{SF}. In case (1), event e is added to L. In case (2), an artificial event is generated representing the crash of process p after executing event sequence H (denoted by $crash(p, H)$). In case (3), the information in H is needed to reconstruct the ordering information of the small fault event e before it is added to L. Note that in cases (2) and (3) H may contain application events that are not yet part of L. Events in cases (2) and (3) are always "parallel" (independent, internal) events of a particular process.

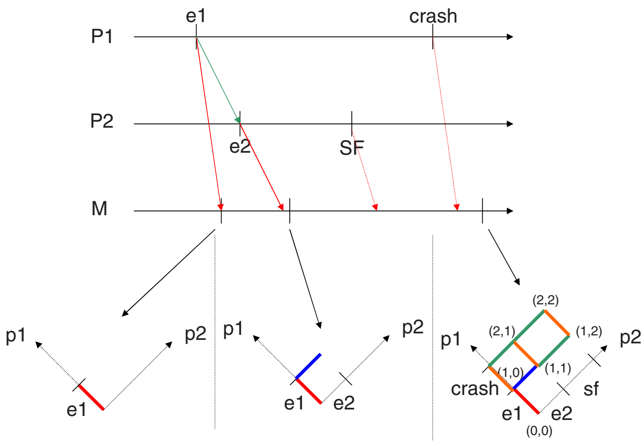


Fig. 3. Example execution of predicate detection algorithm using Σ_{SF}. The figure shows the lattice of consistent global states constructed at a monitor at three different points in time. The rightmost lattice depicts also the internal coordinates of the states in the lattice.

Fig. 3 depicts an example execution of the algorithm with two processes p_1 and p_2. Application events are delivered to monitor m using causal broadcast. Other events (crash of p_1 and small fault at p_2) are delivered locally to m by Σ_{SF}. If events are causally dependent, causal broadcast allows to eliminate inconsistent global states (e.g., state with coordinates $(0, 1)$). The central idea of the proof is to prove that the lattice L is always a prefix of or equal to the "real" lattice of the computation.

Lemma 1. *Let R be the lattice of consistent global states of a computation C observed using the algorithm in Fig. 2. The algorithm guarantees two things:*

1. *$L \subseteq R$.*
2. *Any event in C is eventually represented in L.*

Lemma 2. *Global predicate detection is possible given Σ_{SF}.*

Necessity. We now investigate the question whether Σ_{SF} is necessary to solve predicate detection. To prove this, we construct an algorithm that emulates Σ_{SF} using only the properties of some assumed predicate detection algorithm A. Note that the existence of small faults makes the problem substantially different from the problem tackled by Gärtner and Pleisch [14] who used an incremental number of instances of A to detect crashes of processes in certain states. The difficulty here is that small faults can be arbitrary state transitions so that (1) we need to anticipate certain state transitions to distinguish good ones (program events) and bad ones (small faults), and (2) we need to keep the number of parallel instances of A finite although the number of possible states is infinite.

Like Gärtner and Pleisch [14] we assume that A can be "forked" into parallel instances, each checking a different global predicate. Each new instance checks the complete history of the computation no matter when it is started. Also, we assume states of the computation to be unique (distinguishable).

We tackle the above two problems as follows: We assume that the next state relation δ of the algorithm driving the observed computation is known. Given the current global state s, δ identifies a finite set $S = \{s_1, s_2, \ldots\}$ of possible next states. For each such state we define a predicate φ_{s,s_i} and fork a new instance of A for that predicate. We use the detections of these predicates to keep track of the current state of the computation. They do not trigger any actions of the emulated Σ_{SF}.

To detect crashes, for every process p_i we define a predicate φ_{s,p_i} stating that "p_i crashed in state state s" and fork a new instance of A. Once A detects such a predicate, we trigger a crash event at the emulated interface of Σ_{SF}. To detect small faults in state s we could potentially fork an instance of A for *every other* possible state transition (the complement of $\delta(s)$). However, having infinitely many states this set is infinite and we can not have an infinite number of parallel instances of A (this is problem (2) described above). We tackle this problem in the following way: In any state s we first define the "complement predicate"

$$\phi_s = \neg\left(\bigvee_{s_i \in S} \varphi_{s,s_i} \vee \bigvee_{p_i \in \Pi} \varphi_{s,p_i}\right)$$

and fork an instance of A detecting ϕ_s. Once this predicate is detected, we need to find out which element of ϕ_s caused A to trigger. Since the set of all global

```
Upon ⟨Init⟩ do
    $H[1, \ldots, n] = \{\}$ /* local event sequences of processes */
    $s = I$ /* global initial state */
    trigger ⟨main loop⟩
Upon ⟨main loop⟩ do
    $S := \delta(s)$ /* compute set of next possible program states */
    forall $t \in S$ do
        $\varphi_{s,t} :=$ "system is in state $t$"
        $fork(\varphi_{s,t})$
    forall $p \in \Pi$ do
        $\varphi_{s,p} :=$ "$p$ crashes in state $s$"
        $fork(\varphi_{s,p})$
    $\phi_s := \neg(\bigvee_{t \in S} \varphi_{s,t} \vee \bigvee_{p \in \Pi} \varphi_{s,p})$
    $fork(\phi_s)$ /* fork detection of complement predicate */
Upon ⟨$\varphi_{s,t}$ detected⟩ do
    $s := t$
    update $H[1, \ldots, n]$
    trigger ⟨main loop⟩
Upon ⟨$\varphi_{s,p}$ detected⟩ do
    trigger ⟨crash, $p$, $H$⟩
    update $H[1, \ldots, n]$
Upon ⟨$\phi_s$ detected⟩ do
    /* now loop through $\phi_s$ to find state $t$ following $s$ resulting from small
       fault */
    trigger ⟨loop, $first(\phi_s)$, $\phi_s \setminus first(\phi_s)$⟩
Upon ⟨loop, $s'_s$, $\phi'_s$⟩ do
    $fork(s'_s)$
    trigger ⟨loop, $next(\phi'_s)$, $\phi'_s \setminus next(\phi'_s)$⟩
Upon ⟨$t_s$ detected⟩ do
    /* found small fault $(s, t_s)$ */
    trigger ⟨sf, $p$, $(s, t_s)$, $H$⟩
    update $H[1, \ldots, n]$
```

Fig. 4. Emulating Σ_{SF} using an abstract algorithm A for predicate detection. The next state relation of the algorithm driving the computation is given as δ. The operations *first* and *next* are iterators on a set and yield the first and subsequently next element of the given set.

states is countable, also the subset defined by ϕ_s is countable. Therefore we can enumerate in an incremental fashion all states $s' \in \phi_s$ and fork a corresponding instance of A. Eventually the new state of the computation will be chosen and subsequently detected by A. Since the state transition from s to s' was not defined by δ, it must be the result of a small fault and so we trigger a corresponding event at the interface of Σ_{SF}.

Lemma 3. *Σ_{SF} is necessary for global predicate detection.*

Theorem 1 (Equivalence of predicate detection and Σ_{SF}). *Given an asynchronous system in which crash faults and small faults may occur. Global predicate detection is possible if and only if the system is augmented with Σ_{SF}.*

5.3 Implementing Σ_{SF}

Using Σ_{SF} it is possible to solve predicate detection. So how can we implement Σ_{SF}? Since Σ_{SF} is strictly stronger than Σ and Σ is equivalent to a synchronous system if crash faults can happen, Σ_{SF} at least also needs the strength of a synchronous system to be implemented.

```
at every application process $p$:
  with every step from local state $s$ to $s'$
    causally broadcast $\langle (s, s'), p \rangle$ to all monitors
at every monitor process $m$:
  **Upon** $\langle$*Init*$\rangle$ **do**
      $laststate[1, \dots, n] := I$
      $t[1, \dots, n]$ /* vector of timers */
  **Upon** $\langle$*causal deliver*, $(s, s')$, $p\rangle$ **do**
      **if** $laststate[p] \neq s$ **then**
          **trigger** $\langle$sf, $p$, $(laststate[p], s)$, $H\ \rangle$
      $laststate[p] := s'$
      reset timer $t[p]$
  **Upon** $\langle$*expiry of timer* $t[p]\rangle$ **do**
      **trigger** $\langle$crash, $p$, $H\rangle$
```

Fig. 5. A naive and incorrect (almost correct) algorithm to implement Σ_{SF} in a synchronous system

Fig. 5 shows a simple instructive algorithm which *almost* solves the problem. The algorithm works as follows: Every process p_i sends its latest transition, rather than its latest state, to the monitors. On the other hand, monitors need to keep track of state changes of various processes to track small faults, and they need to keep a timer for each process to determine when they have crashed. When a timer for a process expires, it returns a *crash* event. Further, when it receives the latest transition from a process, it compares the initial state of the transition with the last known state for that process. A discrepancy indicates occurrence of small faults.

Unfortunately, the above algorithm is not correct if more than one small fault occurs within two synchronous steps. In fact this is also the reason that in general Σ_{SF} is not implementable.

Theorem 2 (Impossibility of implementing Σ_{SF} in synchronous systems). *It is impossible to implement Σ_{SF} in a synchronous distributed system.*

A corollary of Theorems 1 and 2 is that the predicate detection problem cannot be solved in synchronous systems. In the next section we investigate weakening the detection problem to tackle this unfortunate result.

6 Probabilistic Predicate Detection

Because of the impossibility of implementing Σ_{SF}, we require processes to send their states to monitors at regular intervals. To increase the chance of detecting every fault, the state sampling periods at the processes should be set sufficiently small so that all faults can be detected. Because fault occurrences are random, even arbitrarily small sampling period may miss faults.

Hence, we introduce the problem of *probabilistic predicate detection*. This problem is a weaker version of the more general problem in that it only requires the predicate detection specification to be satisfied with high probability. We now define the probabilistic predicate detection problem: given a global predicate φ, the probabilistic predicate detection problem consists of finding an algorithm that satisfies the following two requirements:

- The algorithm does not raise the exception unless φ holds with probability $1 - \Pi_s$
- If φ holds, the algorithm will eventually raise the exception with probability $1 - \Pi_l$.

Here, Π_s (resp. Π_l) is the probability of violating the safety (resp. liveness) specification. We will now provide a probabilistic analysis of the problem. The requirements for probabilistic predicate detection can be violated in two ways: (i) safety, and (ii) liveness. Safety specification violation occurs when processes miss the occurrences of faults between two successive steps, while violation of liveness occurs when faults “revert” themselves within a step, i.e., a sequence of faults occurs where the first and final states are equal. This has the property of “masking” the occurrences of faults.

There are two possible ways of analyzing the problem. The first approach is based on the sampling and failure rates, where the analysis is on the probability of either the safety or liveness specification to be violated. Based on this analysis, the sampling rate can be adjusted such that the probability of violation is reduced to below a certain threshold. The second approach is a randomized approach, whereby, at each time instant, each process p_i sends a control message to monitors with probability ρ_i. This approach allows determining a suitable ρ_i such that the probability of violation is reduced to a level less than a predetermined threshold level. However, we argue that the randomized approach is not very good, since it is better to send a “no transition” (looping transition) than not sending anything. This implies that the monitors are able to track state changes at all times. Hence, randomization will not help in implementing Σ_{SF}. Thus, in this paper, we will adopt the first approach.

6.1 Violation of Safety

Safety property is violated when more than one fault occurs between two steps. Since a control message is sent at every step, occurrence of more than one fault implies that some faults will not be notified, thus violating safety. We now analyze the probability of this occurring.

Assume that σ faults occur in δ steps. We also assume that faults occurrences are independent. This means that $\frac{\sigma}{\delta}$ faults occur within one step, i.e., between two consecutive steps.

The probability of violating safety can now be computed as the probability that the number of faults f within one step will exceed one:

$$\begin{aligned} \Pi_s &= \text{Prob}[f > 1] \\ &= 1 - (\text{Prob}[f = 0] + \text{Prob}[f = 1]) \end{aligned}$$

Assuming that fault occurrences f follow a Poisson distribution (since occurrences are independent) we have:

$$\begin{aligned} &1 - (\text{Prob}[f = 0] + \text{Prob}[f = 1]) \\ &= 1 - \exp^{-(\frac{\sigma}{\delta})}(\frac{\sigma}{\delta}^0 + \frac{\sigma}{\delta}^1) \\ &= 1 - \exp^{-(\frac{\sigma}{\delta})}(1 + \frac{\sigma}{\delta}) \end{aligned}$$

Thus, when the rate at which fault occurs is high, i.e., more than one fault occurs between two steps, the probability of safety violation is high, whereas if the rate of fault occurrences is low, the probability of safety violation is very low. One can therefore preset the value of δ such that $\frac{\sigma}{\delta}$ is very low.

6.2 Violation of Liveness

The liveness property of predicate detection is violated when the following happens: a sequence of faults occurs between two consecutive steps such that the final state after the sequence is the same as the start state before the sequence of faults. In other words, the state sequence $s_k \cdot s_{k+1} \cdot \ldots \cdot s_{k+n} \cdot s_l$ is such that every transition between s_k and s_l is a fault transition, and $s_k = s_l$, and s_k, and s_l occur at the "borders" of a given step.

We make the following assumptions:

1. We consider a sequence of faults occurring between 2 steps, with state sequences (of length n) $x_0 \cdot x_1 \ldots x_{n-2} \cdot x_0$, such that the first and final states are equal.
2. Let the size of the state space of the program be N, where $N \gg n$.

When a fault occurs, two consecutive states cannot be equal. In any such state sequences, there are $n - 1$ faults. Since consecutive states cannot be equal, we need to compare the third state onwards with the first state. Thus, there are $n-2$ state comparisons for state equality. Also, the probability of two states being equal is $\frac{1}{N}$ (probability of states being different is $\frac{N-1}{N}$).

The probability of the first state being equal to the final state is equal to

$$(\tfrac{N-2}{N-1})^{(n-3)} \cdot (\tfrac{1}{N-1}) \Rightarrow (1 - \tfrac{1}{N-1})^{(n-3)} \cdot (\tfrac{1}{N-1})$$

Assume that the probability of a fault occurring to be p_f. The probability of a sequence of $n-1$ faults to occur is p_f^{n-1}. Hence, the probability of liveness violation is given by $\Pi_l = \frac{p_f^{n-1}}{N-1} \cdot (1 - \frac{1}{N-1})^{(n-3)}$.

References

1. Anagnostou, E., Hadzilacos, V.: Tolerating transient and permanent failures. In: Schiper, A. (ed.) WDAG 1993. LNCS, vol. 725, pp. 174–188. Springer, Heidelberg (1993)
2. Babaoğlu, Ö., Marzullo, K.: Consistent global states of distributed systems: Fundamental concepts and mechanisms. In: Mullender, S. (ed.) Distributed Systems, 2nd edn. ch. 4, pp. 55–96. Addison-Wesley, Reading (1993)
3. Beauquier, J., Delaët, S., Dolev, S., Tixeuil, S.: Transient fault detectors. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 62–74. Springer, Heidelberg (1998)
4. Beauquier, J., Kekkonen-Moneta, S.: On FTSS-solvable distributed problems. In: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, 21–24 August,1997 Santa Barbara, California, p. 290 (1997)
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
6. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
7. Delaët, S., Tixueil, S.: Tolerating transient and intermittent failures. Journal of Parallel and Distributed Computing 62, 961–981 (2002)
8. Freiling, F.C., Henkel, S., Widder, J.: Network synchronization in the crash-recovery model. Technical Report TR-2006-009, Department for Mathematics and Computer Science, University of Mannheim(May 2006)
9. Freiling, F.C., Jhumka, A.: Global predicate detection in distributed systems with small faults. Technical Report TR-2007-008, Department for Mathematics and Computer Science, University of Mannheim(August 2007)
10. Freiling, F.C., Pleisch, S.: Predicate detection in asynchronous systems with crash failures. In: Zomaya, A.Y., Diab, H.B. (eds.) Dependable Computing Systems: Paradigms, Performance Issues and Applications, ch. 7, pp. 171–212. Wiley, Chichester (2005)
11. Garg, V.: Elements of Distributed Computing. Wiley, Chichester (2002)
12. Garg, V.K., Mitchell, J.R.: Distributed predicate detection in a faulty environment. In: ICDCS 1998. Proceedings of the 18th IEEE International Conference on Distributed Computing Systems, IEEE Computer Society Press, Los Alamitos (1998)
13. Gärtner, F.C., Pleisch, S.: ImPossibilities of predicate detection in crash-affected systems. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 98–113. Springer, Heidelberg (2001)

14. Gärtner, F.C., Pleisch, S.: Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 280–294. Springer, Heidelberg (2002)
15. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance. In: PODC 1993. Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing, pp. 195–206. ACM Press (1993)
16. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer, Heidelberg (2006)
17. Kshemkalyani, A.D.: A fine-grained modality classification for global predicates. IEEE Trans. Parallel Distrib. Syst. 14(8), 807–816 (2003)
18. Mattern, F.: Virtual time and global states of distributed systems. In: Cosnard, M., et al. (eds.) Proceedings of the International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, pp. 215–226. Elsevier Science Publishers, Amsterdam (1989). Reprinted on pages 123–133 in [20]
19. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: Proc. ICDCS, pp. 191–198 (2002)
20. Yang, Z., Marsland, T.A. (eds.): Global States and Time in Distributed Systems. IEEE Computer Society Press, Los Alamitos (1994)

The Truth System: Can a System of Lying Processes Stabilize?

Mohamed G. Gouda and Yan Li

Department of Computer Sciences
The University of Texas at Austin
1 University Station (C0500)
Austin, Texas 78712-0233
{gouda,yanli}@cs.utexas.edu

Abstract. We introduce a new abstract system, called the truth system. In the truth system, a process deduces a true value, with high probability, from an incoming stream of both true and false values, where the probability that a value in the incoming stream is true is at least 0.6. At each instant, the receiving process maintains at most one candidate of the true value, and eventually the process reaches the conclusion that its candidate value equals, with high probability, the true value. In this paper, we present three versions of the truth system, discuss their properties, and show how to choose their parameters so that their probability of error is small, i.e. about 10^{-6}. The third version, called the stable system, is the most valuable. We employ the stable system as a building block in a stabilizing unidirectional token ring of n processes. When n is small, i.e. about 100 or less, the stable system can be considered error-free and we argue that the resulting token ring is stabilizing with high probability. We simulate the token ring, when n is at most 100, and observe that the ring always stabilizes even though each process lies about its state 40% of the time.

Keywords: Distributed Systems, Network Protocols, Self-Stabilization.

1 Introduction

Faults, that are often assumed to plague the communications between different processes in a distributed system, can be distinguished into natural faults and malicious faults [7]. On one hand, *natural communication faults* are assumed to occur independently of the underlying computation of the distributed system, and are usually assumed to be random. On the other hand, *malicious communication faults* are assumed to occur in the worst possible times for the underlying computation of the distributed system, and are usually assumed to be deliberate and based on complete knowledge (by the adversary) of the underlying computation.

Distributed systems that tolerate natural communication faults are elegant, inexpensive, and practical to implement and use. However, such systems cannot tolerate malicious communication faults if they happen to occur. By contrast, distributed systems that tolerate malicious communication faults are complex, expensive, and sometimes impossible to design.

Well-known examples of natural communication faults and how to tolerate them are as follows.

- *Loss*: Some sent values from one process to another are lost. These faults can be tolerated by making the sending process send the same value repeatedly until the sending process receives an "acknowledgement" from the receiving process [9].
- *Delay*: Some sent values from one process to another are delayed for an unbounded time period before these values are received by their intended receivers. Sometimes, these faults cannot be tolerated as discussed in [6]. This realization has led researchers to adopt weaker model of faults, for example imperfect fault detectors, that can be tolerated [3] and [16].
- *Corruption*: Some sent values are corrupted randomly after they are sent by one process and before they are received by another process. These faults can be detected by adding a checksum to each sent value. In this case, any random corruption of a sent value and its checksum can be detected, with high probability, by the receiving process [14].
- *Topology Change*: The topology of the distributed system changes over time, for example due to the mobility of the processes within the system. Methods for tolerating these "faults" are discussed in [15].
- *Anonymity*: Each sent value does not include the identity of the sending process and is received by an arbitrary process in the distributed system. Methods for tolerating these "faults" are discussed in [1].
- *Modification*: A sent value from one process to another is *modified* before it is received as follows. The value is replaced by any wrong (possibly malicious) value in such a way that the receiving process cannot tell, by examining the received value, that the received value is in fact a false value different from the true value that was sent. For example, if a checksum is attached to the sent value, then a modification fault causes both the value and its checksum to be replaced as follows. The true value is replaced by any false (possibly malicious) value, and the checksum is replaced by the checksum of the false value and so the receiving process cannot tell that the true value and its checksum are replaced by a false value and its checksum. Methods for tolerating modification faults in the context of distributed voting are presented in [13] and [11].

Well-known examples of malicious communication faults and how to tolerate them are discussed in [10] and [12].

In this paper, we present a new method, which we call the truth system, for tolerating modification faults. The truth system is different from the distributed voting systems, discussed in [13] and [11], in several ways. Most notably, the two systems have different objectives. The objective of the truth system is to deduce, with a predefined high probability, the true value from an *unbounded* stream of both true and false values. In particular, the parameters of the truth system can be set such that the probability of the deduced value being wrong is very small, say 10^{-6}. By contrast, the objective of a distributed voting system is to deduce the value that has the highest probability of being true from a *bounded* stream of both true and false values, but the probability of the deduced value being wrong can be as high as 0.5.

To save space, we omit the proofs of all theorems from this paper. An interested reader can get these proofs from the full version of the paper [8].

2 Three Versions of the Truth System

The goal of this paper is to introduce a new abstract system, called the truth system, discuss its properties, and show that this system can tolerate, with high probability, modification faults. In the truth system, a process deduces a true value from a mixed stream of both true and false values. The probability, that a value in the mixed stream is true (or false), is at least 0.6 (or at most 0.4, respectively). The process correctly deduces the true value with a high probability, that is around ($1\text{-}10^{-6}$).

It is important to explain why we chose the probability, of a value in the mixed stream being true, to be at least 0.6. First, if we chose this probability to be at most 0.5, then no system can deduce the true value with any probability greater than 0. Second, if we chose this probability to be higher than 0.5 but less than 0.6, then as shown at the end of Section 3 the truth system may need up to 1700 values in the input stream to deduce the true value. This is 10 times the number of values needed in the input stream, to deduce the true value, when the probability of a value in the mixed stream being true, is at least 0.6.

The truth system consists of two processes: source and monitor. Process source has an integer state, and process monitor attempts to correctly deduce the integer state of process source.

Periodically, process source sends an integer s to process monitor. With a probability of at least 0.6, the sent s is the true value of the state of process source, and with a probability of less than 0.4, the sent s is an arbitrary integer. Process monitor receives the s integers, one by one, and maintains at most one candidate for the state of process source. Eventually process monitor reaches the conclusion that its maintained candidate for the state of process source equals, with high probability, the true state of process source.

The state of process source can change over time, but we assume that this change occurs at a slow rate. This is because if the state of source is changed at a fast rate, the monitor process may never be able to catch up and deduce the state of source. As shown at the end of Section 3, to ensure that the change rate of the state of process source is slow, we assume that the state of source is not changed until process source has sent out this state 170 or more times.

Our presentation of the truth system in this paper consists of three steps. In each step, we present a version of the truth system and discuss its properties. We then point out some problem with this version and so clear the way for the next version that is to be presented in the next step, and so on. The version presented in the third (and last) step has no problems, as far as we can tell.

In the first step, we present a version of the truth system where the monitor process terminates as soon as it concludes that its maintained candidate for the state of process source equals, with high probability, the true state of process source. The problem of this truth system is that the monitor process deduces exactly one true state of process source, even if the state of process source is changed many times afterwards. We refer to this truth system as the *one-shot system*.

In the second step, we modify the one-shot system to make the monitor process continue to operate indefinitely, even after it concludes that its maintained candidate for the state of process source equals, with high probability, the true state of process source. We refer to this truth system as the *continuing system*. The problem with the continuing system is that the conclusion reached by process monitor (that its candidate for the state of process source equals, with high probability, the true state of source) is not stable, but it can fluctuate wildly over time, even when the true state of process source remains fixed for a long time period.

In the third step, we modify the continuing system to ensure that, when the true state of process source remains fixed for a long time period, the conclusion reached by process monitor (that its candidate for the state of process source equals, with high probability, the true state of source) remains stable over time. We refer to this truth system as the *stable system*.

3 The One-Shot System

In this section, we present our first version of the truth system, called the one-shot system. In the one-shot system, as soon as process monitor concludes, that its maintained candidate for the state of process source equals, with high probability, the true state of process source, process monitor terminates.

Process source in the one-shot system is specified in Protocol 1. This process has one action that it executes over and over, since the guard of the action is **true**. During each execution of the action, process source sends an integer s to process monitor. With probability p/100, the sent s is the value of input state, and with probability (100-p)/100, the sent s can be any integer, where p is an input of process source.

Input p can be regarded as the *probability of process source telling the truth*, and input *state* can be regarded as the *true state of process source*. From the received s integers, process monitor is expected to deduce the true value of input *state* in process source. It is straightforward to show that if the value of p is in the range 0..50, then process monitor can never deduce the true value of *state*. At the end of Section 3, we argue that if the value of p is in the range 51..59, then process monitor may need to receive up to 1700 s integers (instead of 170) in order to deduce the true value of *state* with high probability. That is why we specified the value of p to be in the range pmin..99, where *pmin* is a constant whose value is in the range 60..99.

Protocol 1. process source

```
const     pmin  : 60..99
input     p     : pmin..99
          state : integer  {state of source}
variable  r     : 0..99    {random number}
          s     : integer  {sent state}
begin
     true →
          r := random
          if r ≥ p then
             s := any          {assign malicious value}
          else
             s := state
          end
          send s to monitor
end
```

Both *p* and *state* are inputs to process source. Thus, their values can be changed over time by an outside agent. As shown at the end of Section 3, we assume that once the value of state is changed, the value of state remains fixed until process source executes its action 170 or more times.

Process monitor in the one-shot system is specified in Protocol 2. Process monitor has three variables: c, st and s. Variable c is a counter whose value is in the range $0..cmax$, where $cmax$ is a constant of process monitor. Variable st stores the latest candidate for the state of process source. Variable s stores the latest received integer from process source. The value of counter c indicates whether process monitor can conclude that the current value of st equals, with high probability, the value of state in process source. Process monitor reaches this conclusion when, and only when, the value of counter c is $cmax$.

Process monitor has only one action that is executed each time the process receives an integer s from process source. When an integer s is received, process monitor checks the value of its counter c. If $c = 0$, then variable st is assigned s and counter c is assigned 1. If $c > 0$ and st is different from the received s, then c is decreased by 1. If $c > 0$ and st equals the received s, then c is increased by 1 (provided that c does not exceed its maximum value $cmax$). Then process monitor compares the values of c with $cmax$. If $c = cmax$, then process monitor concludes that the current value of its variable st equals, with high probability, the value of *state* in process source.

To complete the specification of process monitor, we need now to compute the value of constant $cmax$ in monitor. The value of $cmax$ should be chosen such that the probability of error of the one-shot system is kept small, say around 10^{-6}.

The *probability of error*, denoted *p(error)*, of the one-shot system is the probability that starting from its initial global state where $c = 0$, the system reaches a global state where $c = cmax$ and $cs \neq state$.

Protocol 2. process monitor of the one-shot system

```
const     pmin :  60..99   {same as pmin in source}
          cmax :  integer
variable  c    :  0..cmax  {counter, init. 0}
          s    :  integer  {received state}
          cs   :  integer  {candidate state}
begin
    rcv s from source →
        if c = 0 then
            c := 1
            cs := s
        else if cs ≠ s then
            c := c - 1
        else
            c := min(c+1, cmax)
        end
        if c = cmax then
            {conclude: cs = state} terminate
        end
end
```

In Theorem 1 below, we give a formula that describes the relationship between *pmin*, *cmax*, and *p(error)* for the one-shot system. Our proof of this theorem is based on two simplifying assumptions. First, we assume that the state of process source does not change over time. This assumption is acceptable given our understanding that the change rate of the state of process source is slow anyway. Second, we assume that whenever process source sends an arbitrary integer *s* to process monitor, process source always sends the same integer that is different from the state of process source. This assumption represents the worst case scenario that assigns *p(error)* its highest value. We adopt these two assumptions in proving all the theorems in this paper. (Recall that the proofs of all the theorems are in [8].)

Theorem 1 (pmin, cmax, and p(error) for the one-shot system)

$$p(error) \;=\; \frac{(1-pmin)^{cmax}}{(1-pmin)^{cmax} \;+\; (pmin)^{cmax}} \;. \qquad \square$$

For many applications, it is reasonable to expect that *p(error)* should be around 10^{-6}. In this case, we can use the formula in Theorem 1 to produce the relationship between *pmin* and *cmax*, for the one-shot system, shown in Table 1.

An execution *step* of the one-shot system consists of two parts. First, process source executes its (sending) action, then process monitor executes its (receiving) action.

If *pmin* in this system is 0.6 and the value of variable *cs* in process monitor is the correct state of process source, then in each step of the system, counter *c* in process monitor is incremented by 1 with probability 0.6, and is decremented

Table 1.

pmin	*cmax*
0.6	34
0.7	16
0.8	10
0.9	6

by 1 with probability 0.4. In other words, each step of the system increments counter c by 0.2 on the average. Thus, the system needs to execute cmax/0.2 steps on the average before counter c reaches its maximum value *cmax* and the system terminates. Because *cmax* in this system is 34 from Table 1, the system needs to execute 170 steps before it terminates.

If we choose *pmin* in this system to be 0.51, and assume that *cmax* remains 34 (rather than being increased in value as it should), and follow the same analysis in the previous paragraph, we conclude that the system will execute on average (34/0.02)= 1700 steps before it terminates. In other words, choosing *pmin* to be in the range from 0.51 to 0.59 can lead to (sometimes substantial) increase in the number of steps to be executed. This should explain our choice of *pmin* to be at least 0.6.

The above analysis for computing the average number of steps that need to be executed by the one-shot system before it terminates is based on the assumption that the state of process source does not change during execution. This is an important assumption; for instance, if the state is changed at least once every 5 execution steps, the one-shot system may never terminate. This should explain our above requirement that the state of process source remains fixed for the duration of 170 steps.

4 The Continuing System

The problem of the one-shot system is that process monitor terminates as soon as it concludes that the value of its *cs* variable equals, with high probability, the value of input state in process source. Thus, monitor cannot observe any change in the state of process source. To remedy this problem, we modify process monitor such that the process continues to execute indefinitely. This modification is achieved by replacing statement **terminate** by a statement **skip** in the action of process monitor. We refer to the resulting system as the continuing system.

Because the continuing system is nonterminating, the initial state of the system, where $c = 0$, is irrelevant. Rather, we define the *probability of error p(error)* of the continuing system as the steady state probability that the system is in a global state where $c = cmax$ and $cs \neq state$. The following theorem gives a formula that describes the relationship between *pmin*, *cmax*, and *p(error)* for the continuing system.

Theorem 2 (pmin, cmax, and p(error) for the continuing system)

$$p(error) \;=\; \frac{(1-pmin)^{2\times cmax}}{(1-pmin)^{2\times cmax} \;+\; (pmin)^{2\times cmax}} \;. \qquad \Box$$

Assuming that *p(error)* is around 10^{-6}, we can use the formula in Theorem 2 to produce the relationship between *pmin* and *cmax*, for the continuing system, shown in Table 2. Notice that the *cmax* values in the one-shot system, shown in Table 1, are twice the *cmax* values in the continuing system, shown in Table 2.

The continuing system has an interesting problem. Even if the state of process source remains fixed for a long time period T, the value of counter c in process monitor can fluctuate during period T between $c < cmax$ (when process monitor cannot conclude that $cs = state$) and $c = cmax$ (when process monitor can conclude that $cs = state$). This observation suggests the following definition.

The *probability of no conclusion*, denoted *p(no-conclusion)*, of the continuous system is the steady state probability that the system is in a global state where $c < cmax$. The following theorem gives a formula for computing *p(no-conclusion)* as a function of *pmin* and *cmax*.

Theorem 3 (p(no-conclusion) for the continuing system)

$$p(no\text{-}conclusion) \;=\; \frac{\sum_{i=1}^{2\times cmax-1}(\frac{1-pmin}{pmin})^{i}}{\sum_{j=0}^{2\times cmax}(\frac{1-pmin}{pmin})^{j}} \;. \qquad \Box$$

Using the formula in Theorem 3 and the values of *pmin* and *cmax* in Table 2, we can compute the values of *p(no-conclusion)*, for the continuing system, as shown in Table 2.

From the first row in Table 2, if *pmin* and *cmax* for the continuing system are 0.6 and 17 respectively, then *p(no-conclusion)* for this system is 0.667. This means that even if the state of process source remains fixed for a long time period T, process monitor cannot conclude that $cs = state$ for 66.7% of the time during period T. Clearly, this is not acceptable and a modification of the continuing system is in order. In the next section, we describe how to modify the continuing system to remedy this problem. We refer to the modified system as the stable system.

Table 2.

pmin	*cmax*	*p(no-conclusion)*
0.6	17	0.667
0.7	8	0.429
0.8	5	0.250
0.9	3	0.111

5 The Stable System

The stable system is obtained from the continuing system, discussed in the previous section, by making the following two modifications to process monitor (in the continuing system). First, a new variable named *ss* is added to process monitor. Variable *ss* stores the latest stable estimate (by process monitor) of the state of process source. Second, the last **if**-statement in the action of process monitor is modified to become as shown in Figure 1. (The first **if**-statement in the action of process monitor remains unchanged.)

```
if c = cmax then
    {conclude: cs = state} ss := cs
end
```

Fig. 1.

The *probability of error p(error)* of the stable system is defined as the steady state probability that the system is in a global state where $ss \neq state$. The following theorem describes the relationship between *pmin*, *cmax*, and *p(error)* for the stable system.

Theorem 4 (pmin, cmax, and p(error) for the stable system). *Given that p(error) for the stable system is around* 10^{-6}*, we have the following relationship between pmin and cmax for the stable system.*

A *step* of the stable system consists of two parts. First, process source executes its action once. Second, process monitor executes its action once. The *convergence span* of the stable system is the average number of steps that needs to be executed by the stable system in order to change the global state of the system from one where $c = cmax$ and $ss \neq state$ to one where $c = cmax$ and $ss = state$. The following theorem gives an approximate formula for computing the convergence span of the stable system.

Theorem 5 (convergence span of the stable system).

$$convergence\ span \approx \frac{2 \times cmax}{2 \times pmin - 1}. \qquad \square$$

Table 3.

pmin	*cmax*
0.6	20
0.7	9
0.8	5
0.9	4

□

Table 4.

pmin	*cmax*	*convergence span*
0.6	20	200
0.7	9	45
0.8	5	17
0.9	4	10

Using the formula in this theorem and the values of *pmin* and *cmax* from Table 3, we compute the convergence span of the stable system as shown in Table 4.

6 A Stabilizing Token Ring

In this section, we discuss how to employ the stable system, presented in the previous section, as a building block in constructing a stabilizing unidirectional token ring of up to 100 processes, where each process can lie about its state at most 40% of the time. The use of the stable system as a building block in constructing a stabilizing token ring, (where each process can lie about its state at most 40% of the time) can be roughly viewed as an example of the cross-over composition proposed in [2].

We start our discussion by presenting a unidirectional token ring, in Protocol 3, where processes do not lie about their states. Note that this ring is similar to Dijkstra's token ring in [4] with two exceptions. First, the execution of this ring is synchronous, whereas the execution of Dijkstra's token ring is asynchronous. Second, this ring uses message passing primitives whereas Dijkstra's token ring uses shared memory primitives.

Protocol 3. process p[i : 0..n-1] in the original token ring

```
variable s  :  0..n-1 {sent/received state}
         ss :  0..n-1 {state}
begin
    true  →
        s := ss
        send s to p[i+1 mod n]

||  rcv s from p[i-1 mod n]  →
        if i > 0 then
           ss := s
        else if ss = s then
           ss := ss + 1 mod n
        end
end
```

In this ring, each process p[i] has two variables, *s* and *ss*, where variable *s* stores the latest state that p[i] has sent or received, and variable *ss* stores the state of p[i]. Each p[i] also has two actions: a sending action where p[i] sends its own state to p[i+1 mod n], and a receiving action where p[i] receives the state of p[i-1 mod n] then modifies its own state based on the received state.

A *global state* of this ring is defined by a value for each *ss* variable in the ring. (This means that the *s* variables are not considered part of the global state of the ring.)

A *transition* of this ring is a pair (S, S') of global states of the ring such that if the ring is in a global state S and a "step" is executed, then the ring becomes in a global state S'. Executing a step in the ring consists of two parts. First, each process in the ring executes its sending action, then each process in the ring executes its receiving action. Thus, each process in the ring ends up executing both its (sending and receiving) actions in a step.

A *computation* of this ring is an infinite sequence S.0, S.1, ... of global states of the ring such that each pair (S.i, S.(i+1)) of consecutive states in the sequence is a transition of the ring.

It is straightforward to show that each computation of this ring reaches a legitimate global state where the values of all the ss variables are equal after at most 2n transitions, and so this ring is stabilizing.

Clearly, stabilization of the ring in Protocol 3 depends heavily on the fact that the ring processes do not lie when they send their states to other processes. To allow the ring processes to lie about their states, 40% of the time, and still retain the stabilization of the ring, we employ the stable system, discussed in the previous section, in constructing the new ring. Specifically, each process p[i] in the ring is modified to act as a source when p[i] sends a state *s* to process p[i+1 mod n], and act as a monitor when p[i] receives a state *s* from process p[i-1 mod n]. The new ring is shown in Protocol 4.

A *global state* of the new ring is defined by a value for each *ss* variable in the ring. (This means that none of the other variables, namely *r*, *c*, *s*, and *cs*, is considered part of the global state of the new ring.)

A *transition* of the new ring is a pair (S, S') of the global states of the new ring such that if the ring is in a global state S and a "step" is executed, then the ring becomes in a global state S'. Executing a step in the new ring consists of two parts. First, each process in the ring executes its sending (or source) action. Second, each process in the ring executes its receiving (or monitor) action.

A *computation* of the new ring is an infinite sequence S.0, S.1, ... of global states of the ring such that each pair (S.i, S.(i+1)) of consecutive states in the sequence is a transition of the new ring.

The new ring can be viewed as consisting of n stable systems, and each ss variable in the ring can be viewed as belonging to the monitor of one of those stable systems. For the new ring to stabilize, if it is to stabilize, each *ss* variable needs to be assigned around 2×n new values. From Table 4 and given that pmin is 0.6, a stable system needs to execute on average 200 steps in order to assign a new value to its *ss* variable. Therefore, for the new ring to stabilize, if it is to

Protocol 4. process p[i : 0..n-1] in the new token ring

```
const      pmin :  60..99     {pmin = 60}
           cmax :  integer    {cmax = 20}
variable   r    :  0..99      {random number}
           c    :  0..cmax    {counter, init. 0}
           s    :  0..n-1     {sent/received state}
           cs   :  0..n-1     {candidate state}
           ss   :  0..n-1     {stable state}
begin
      true  →
            r := random
            if r ≥ pmin then
               s := any
            else
               s := ss
            end
            send s to p[i+1 mod n]

||    rcv s from p[i-1 mod n]  →
            if c = 0 then
               c := 1
               cs := s
            else if cs ≠ s then
               c := c - 1
            else
               c := min(c+1, cmax)
            end
            if c = cmax then
               if i > 0 then
                  ss := cs
               else if ss = cs then
                  ss := ss + 1 mod n
               end
            end
end
```

stabilize, each of the n stable systems in the ring needs to execute around 400×n steps. By choosing n to be relatively small, say 100, each stable system in the ring needs to execute a small number of steps, around 40000 steps in order for the ring to stabilize. Because the probability of error of a stable system is very small, around 10^{-6}, it is reasonable to assume that whenever any *ss* variable is assigned a value, in the first 40000 transitions of a computation, it is assigned a correct value. We refer to this assumption as the *no-use-lying* assumption.

Now consider a computation S.0, ..., S.40000, ... of the new ring. Under the no-use-lying assumption, whenever an *ss* variable is assigned a value in the first 40000 transitions of this computation, it is assigned a correct value. Therefore, the global state S.40000 in this computation is a legitimate state, with high probability. Therefore, the new ring is stabilizing, with high probability.

The probabilistic stabilization of the new ring depends heavily on the validity of the no-use-lying assumption. To check the validity of this assumption, we have run 100 simulations of the new ring. For each simulation, we chose n (the number of processes in the ring) to be 100, the initial global state of the ring to be random, and the wrong state that each process sends in place of its correct state to be 0. We observed that each simulation has stabilized to a legitimate global state, after no more than 32440 transitions. These simulation results justify our adoption of the no-use-lying assumption.

A probabilistic token ring is proposed in [5]. This ring is significantly different from our new token ring in Protocol 4 in the following sense. In the probabilistic ring in [5], when p[i] receives a (possibly wrong) value s from p[i-1 mod n], p[i] uses the received s to update its own state. By contrast, in our new ring in Protocol 4, when p[i] receives a (possibly wrong) value s from p[i-1 mod n], p[i] first uses its counter c to check whether s is correct with high probability, and only when p[i] is certain that s is correct with high probability, does p[i] use s to update its own state.

7 Concluding Remarks

The truth system is a building block that can be employed in a distributed system to ensure that the system performs its intended function, with high probability, even if up to 40% of the sent values by each process in the system are completely arbitrary. In this paper, we presented three versions of the truth system: the one-shot system, the continuing system, and the stable system. We also compared the properties of these three versions and concluded that the stable system is superior to the other two. Finally we showed how to employ the stable system in a unidirectional token ring so that the ring performs its intended function even if up to 40% of the values sent by each process in the ring are arbitrary.

This paper suggests a number of interesting problems that merit further research. First, are there interesting versions of the truth system other than those discussed in this paper? Second, are there algorithms that take a distributed system that performs a function f under the assumption of perfect communication and produce a distributed system that employs a version of the truth system as a building block and performs function f, with high probability, under the assumption that up to 40% of the values sent by each process in the system are arbitrary. Third, are there effective methods to compute the probability of error and the convergence span for a distributed system where a version of the truth system is employed as a building block?

Acknowledgments. The work of M. G. Gouda is supported in part by a grant 0520250 from the National Science Foundation. The authors are grateful to the referees who made useful suggestions to improve the presentation. We are especially grateful to the referee who remembered Gouda's early proposal to reject any self-stabilization paper that uses Dijkstra's token ring as an example, and who accepted our paper nonetheless. We appreciate your tolerance!

References

1. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: On the power of anonymous one-way communication. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 396–411. Springer, Heidelberg (2006)
2. Beauquier, J., Gradinariu, M., Johnen, C.: Cross-over composition - enforcement of fairness under unfair adversary. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 19–34. Springer, Heidelberg (2001)
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for asynchronous systems. Journal of the ACM 43(2), 225–267 (1996)
4. Dijkstra, E.W.: Self-stabilization systems in spite of distributed control. In: CACM, pp. 643–644 (November 1974)
5. 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)
6. 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 (1984)
7. Fischer, M.J., Merritt, M.: Appraising two decades of distributed computing theory research. Distributed Computing 16(2-3) (2003)
8. Gouda, M.G., Li,Y.: The truth system: Can a system of lying processes stabilize? UTCS Technical Report TR-07-42, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2007.
9. Herzberg, A., Kutten, S.: Early detection of message forwarding faults. SIAM Journal of Computing, August-October issue (2000)
10. Hoch, E.N., Dolev, D., Daliot, A.: Self-stabilizing byzantine digital clock synchronization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 350–362. Springer, Heidelberg (2006)
11. Kumar, A., Malik, K.: Voting mechanisms in distributed systems. IEEE Transactions on Reliability 40(5), 593–600 (1991)
12. Malekpour, M.R.: Byzanting-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)
13. Paquette, M., Pelc, A.: Optimal decision strategies in byzantine environments. Journal of Parallel and Distributed Computing 66(3), 419–427 (2006)
14. Peterson, W.W., Brown, D.T.: Cyclic codes for error detection. In: Proceedings of the IRE, vol. 49, pp. 228–235 (1961)
15. Walter, J.E., Welch, J.L., Vaidya, N.H.: A mutual exlcusion algorithm for ad hoc mobile networks. Wireless Networks (2001)
16. Yang, J., Gafni, E., Neiger, G.: Structured derivations of consensus algorithms for failure detectors. In: PODC, Puerto Vallarta, Mexico (1998)

Temporal Partition in Sensor Networks*

Ted Herman[1], Sriram Pemmaraju[1], Laurence Pilard[1,2], and Morten Mjelde[1,3]

[1] Department of Computer Science,
University of Iowa, Iowa City, IA 52242-1419, U.S.A.
{herman,sriram}@cs.uiowa.edu
[2] pilard.laurence@gmail.com
[3] University of Bergen, Norway
mortenm@ii.uib.no

Abstract. Sensor networks are composed of nodes embedded in physical environments where applications may be tasked to run for years without human maintenance and without continuous external power supply. Strategies for power conservation are thus important in sensor network protocols and system architecture. One such strategy is to arrange node sleeping schedules so that radios are powered off until communication is necessary. Nodes cannot receive messages during periods when the radio is turned off. In this setting, there can arise situations where groups of network nodes have somehow become *temporally partitioned*: due to having different sleeping schedules, groups of nodes could be unaware of each other. The paper presents several self-stabilizing protocols to solve the problem of temporal partition; starting from an arbitrary temporally partitioned state, these protocols lead the network to a state in which all nodes have a perfectly aligned sleep schedule. Our techniques include using randomly chosen relatively prime sleep periods and occasional, and possibly random, probing of extra time slots. Our protocols aim for fast convergence while imposing only a small energy consumption overhead.

1 Introduction

Efficient power utilization is widely studied in the Wireless Sensor Network (WSN) community. WSNs are composed of nodes embedded in physical environments that may need to operate for years using only battery power, so techniques reducing power consumption are important. Among the ideas for reducing power consumption, powering down the processor is the simplest to implement (modern processor architectures typically enable fast switching to low power idle states). Auxiliary devices in a sensor node, including flash memory and radio, can also be put into low power modes or turned off completely. To illustrate the utility of turning off the radio, consider the widely used CC2420 radio chip, which essentially has four different power modes: transmitting, receiving, idle (awake to receive), and off. For this chip, receiving a message, which takes about 1.472 milliseconds, consumes the same power as running in idle mode for about 65

* Research supported by NSF Grant 0519907.

milliseconds. Judiciously powering off the CC2420 (which consumes no power while off) rather than running in idle mode can thus significantly extend the lifetime of a WSN application dependent on batteries.

The WSN literature refers to techniques that power off the radio (and other devices) intermittently as duty cycling. A common strategy is to arrange periodic time intervals, called cycles, that begin with the radio on for some fixed period of activity, followed by turning the radio off for the remainder of the cycle. The fixed period with the radio on is called the *active period*, and the remainder of the cycle is the *sleeping period* (see Figure 1(a)). The *duty cycle* is defined as the ratio of the active period length to the length of the entire cycle (sum of active and sleeping periods). Typical WSN targets for duty cycles are 1% or below, which can extend to years the life of a sensor network running on batteries. Though duty cycling seems obviously a good idea to conserve power, there is some risk associated with duty cycle implementation, as the following paragraph explains.

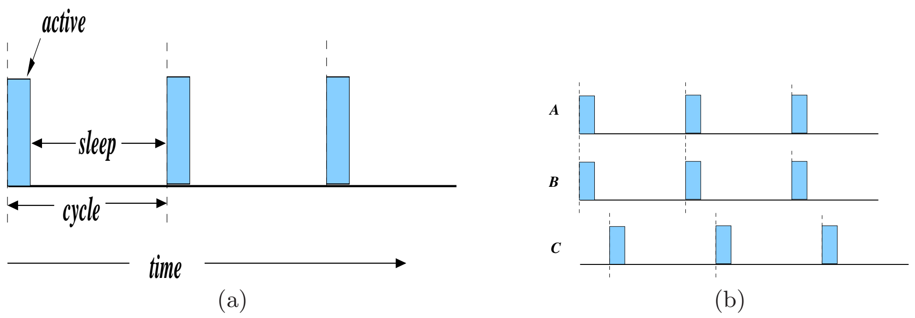


Fig. 1. (a) The typical cycle of an active period followed by a sleep period. (b) Nodes A and B are aligned, whereas node pairs (A, C) and (B, C) are displaced.

Selection of duty cycle parameters must be balanced against application and network protocol performance. The question of what are optimal parameters for duty cycling has been addressed in previous research (Section 3 covers some results). Some other issues concerning duty cycle implementation are robustness to deployment errors, adaptation dynamic changes to the network, and mobility of sensor nodes. The basic problem we examine in this paper is *temporal partition*. If different subnetworks of a WSN follow duty cycles that are "displaced" such that one subnetwork is sleeping while another is active, can the duty cycling protocol automatically bring the subnetworks together to a common duty cycle? At any time, a set R of nodes in the WSN are called *aligned* if their cycles have the same lengths (active and sleeping periods) and the time instants of when the next active period starts, for nodes in R, all lie in some time interval of length ϵ (a platform-dependent tolerance factor related to timekeeping abilities). If a pair (x, y) of nodes is not aligned, then we say x and y are *displaced*; more generally, if a set R is aligned and a set S is aligned, but if $R \cup S$ is not aligned, then

R and S are displaced (see Figure 1(b)). This terminology enables a succinct statement of the problem we consider, which is a problem of self-stabilization: starting from an initial state where each node has some arbitrary cycle, does the protocol eventually converge to state where the set of all nodes is aligned, and remains aligned thereafter? Implicit in this problem statement is the fact that a pair of displaced nodes may have no awareness of each other and cannot gain this awareness until they are aligned.

One could argue that, in practice, temporal partition rarely if ever occurs, because transient faults of state are very rare (we do not have any data on the occurrence of such faults) and networks are carefully deployed. The addition of new nodes to a network is simple to accommodate: after initialization, new nodes remain active until they overhear the current cycle and then join the network gracefully. Nevertheless, a protocol that can stabilize from temporal partition has the virtue of relaxing constraints on deployment, say, that nodes must be initialized in a controlled order, or disallowing that some group of nodes accidentally initialized together could later be brought to the network as an active group with their own established cycle.

2 Assumptions, Definitions, and Results

If two WSN nodes p and q are in bidirectional communication range, we say there is a link between p and q. Let G be the graph induced by the nodes and bidirectional links; we assume G is connected at any instant. Nodes have immutable, distinct identifiers: let $\mathsf{ID}(p)$ denote the identifier of node p. A subset S of nodes in G is *stably aligned* if all nodes in S are aligned and remain aligned forever in the absence of any transient faults in the network. The length of the active period is given; we assume that this is a fixed constant and sufficient for a node to perform local bidirectional communication, i.e., with all neighbors. This assumption is partly justified by the *local broadcast model* of communication that is typically used for WSNs; in this model a message transmitted by a node can be heard by *all* other nodes in its communication range. (Our protocols remain correct even if the active periods have larger length, e.g., proportional to the diameter of the network, and if these lengths vary from cycle to cycle). We are given an upper bound, denoted sleep_{upper}, on the length of the sleeping period. The motivation for sleep_{upper} is that the WSN application is typically required to sense and report at some given periodicity, and this requirement would not be satisfied if nodes sleep longer than sleep_{upper}. Given this upper bound, $1/(1+\mathsf{sleep}_{upper})$ is a lower bound on the duty cycle that protocols can achieve.

We now restate the *temporal partition* problem. The nodes of G are partitioned into subsets $N_1, N_2, \ldots$ such that nodes in each subset N_i are initially aligned, whereas nodes in any pair of distinct subsets (N_i, N_j), $i \neq j$, are displaced. Devise a self-stabilizing protocol such that, eventually, the set of all nodes in G is stably aligned. We use two measures for the performance of our protocols: (i) duty cycle and (ii) convergence time, which we define (as usual) for deterministic protocols to be the worst case time from initial state to a state in which the set of all nodes in G is stably aligned. For randomized protocols we compute the expected convergence time and when possible the convergence time guaranteed with high probability (i.e., $1 - 1/n$, where n is the number of nodes in the network).

We now state our assumptions related to other important issues such as communication, mobility, and clock synchrony. Nodes send and receive messages to communicate; transmission of a message is not acknowledged. For simplicity of exposition, we assume that messages are not lost. Our results extend to the case where messages can be lost, but if (p, q) remains a bidirectional link for some constant number of cycles, then at least one of p's messages is received by q and vice versa. Due to space restrictions, we assume that nodes are not mobile, though our protocols are correct even in the presence of limited node mobility.

Sensor nodes have discrete clocks that can provide relative timing functions and scheduled wakeup signals, which support cycle scheduling of active and sleeping periods. We assume that the WSN uses a (self-stabilizing) clock synchronization protocol, which exchanges messages during the active period of the cycle. Note that if a set of nodes has synchronized clocks, and if the protocol for scheduling active and sleeping periods is purely a function of the clock value, then stable alignment is simple: all nodes become active and sleep together because their clocks are the same. Suppose (p, q) is a bidirectional link, but p and q are not stably aligned. One mechanism to achieve alignment is called *discovery*, which would occur if p and q have active periods that overlap sufficiently long to exchange protocol messages: each node thereby "discovers" the other, and future cycle scheduling can be adjusted to enforce mutual alignment.

To simplify exposition, we normalize the active period length and take it to be 1 time unit. We assume the cycle period length is a multiple of the active period length, hence the cycle and sleeping period lengths are integral values. A time interval of length 1 is called a *slot*, and we assume that in any state of the WSN, nodes wake up at integral times, that is, on slot boundaries. This assumption simplifies our analysis, though in a real network, nodes could be displaced by some arbitrary real number. For two nodes to become aligned, the active periods need to overlap long enough for discovery to occur. One way to interpret our assumption about nodes waking on slot boundaries would be to extend, in implementation, the actual radio-on time for an active period so that sufficient overlap is ensured. Such implementation details are beyond the scope of this presentation.

Results. The paper presents two classes of self-stabilizing protocols for the temporal partition problem. The first class of protocols (Section 4.1) use a "no cost" approach, which permits nodes to *only* vary their sleep periods and does not allow them to remain active outside their given active periods. This class of protocols provide $O(diam(G) \cdot z^2)$ convergence time with $1/z$ duty cycle for any $z \leq \mathsf{sleep}_{upper}$. The key step in these protocols is a randomized choice of relatively prime sleep periods. The second class of protocols (Section 4.2) use *probing.* These protocols permit nodes to remain active for a small number of time slots outside their active period, for the purposes of probing other components. We consider deterministic as well as randomized probing techniques. We present a family of deterministic probing protocols that provide $O(diam(G) \cdot \frac{z^2}{c})$ convergence time with a duty cycle of $\frac{1+c}{z}$, for any integer c, $1 \leq c \leq z-1$. We show that these protocols are optimal by providing matching lower bounds. For dense, constant-diameter networks, randomized probing provides $O(z \cdot (\log z + \log \log n))$ convergence time with $2/z$ duty cycle, for any $z \leq \mathsf{sleep}_{upper}$. We provide a comparison of the results for these classes of protocols in Section 5.

3 Related Work

The majority of WSN research on radio power conservation is investigated at the MAC layer, which can optimize for different communication patterns (broadcast, convergecast, etc.) and take advantage of platform-specific abilities. The application layer, however, may have extra knowledge that enables greater optimization in power control [8]. For instance, at the application layer it could be known that the WSN should sleep for eight hours. The protocols in this paper are intended for middleware or for the application layer, which fits the paradigm of cycles with active and sleeping periods described in Section 2. At the MAC layer, protocols are not geared to cycles; instead, a node can initiate transmission at any time, which is generally suited to low-latency, event-driven applications.

MAC-layer protocols for power control are either *asynchronous* [11,6] or *synchronous* [1,4,2,3]. The asynchronous protocols arrange the timing of wakeup and sleep using patterns so that discovery can occur for each transmission operation [11]. For example, the B-MAC protocol [6] introduces Low-Power Listening (LPL), in which nodes sleep for some time, followed by brief sampling of the radio to detect upcoming transmissions. LPL requires that transmitters send a preamble to each message such that the length of the preamble is at least the length of the LPL sleeping time. In synchronous MAC protocols, nodes may build tables to represent the schedule of each neighbor, and arrange active periods to suit the targets of transmission. See [10] for a comparative analysis of power management protocols and some optimality results in this area. Typical duty cycles for asynchronous MAC protocols range from 1% to 30%. For synchronous MAC protocols, it is possible to obtain about 0.1% duty cycles [5]. To get duty cycles much below 0.1% seems to require knowledge of application behavior [9].

4 Protocols

We distinguish two approaches for protocol design: (i) the "no cost" approach varies only the length of the sleep period up to the sleep_{upper} bound; (ii) the "probing" approach adds active periods for the purpose of accelerating convergence.

Before we present our protocols in detail, we discuss the issue of how components merge with each other after discovery; this pertains to all our protocols. When the active period of a node a overlaps with the active period of a node b, both nodes acquire information about each other. Then using an unspecified "merge rule," either a decides to join b's component or vice versa. There can be a variety of deterministic or randomized merge rules. For example, a node with smaller clock value may join the component of node with larger clock value; alternately, IDs of component leaders may be used to make this decision. Ideally a merge rule should be simple and not impose any communication overhead. Besides the merge rule, there is also the issue of whether or not a node a takes its component along when it joins node b's component. If we assume yes, then node a needs to communicate its decision to the rest of the nodes in its component and this imposes a communication overhead. The advantage of this approach is that for any component, each of its nodes contributes to the discovery of other components, thereby increasing the possibility that this component will join others. Despite this advantage, our protocols assume that nodes individually make the decision to join other components, without taking the rest of their component along. This choice was made mainly to keep the communication overhead low and keep our analysis simple. Exploring the trade-offs between these two approaches is for the future.

4.1 No Cost Approaches

This section presents an efficient "no cost" protocol to solve the temporal partition problem. The basic structure of protocols being considered here is the following.

1. If v receives a message from a node u such that $clock(v) \neq clock(u)$ and if the "merge rule" is satisfied then v copies $clock(u)$ and other associated information.
2. Node v picks its sleep period $s_v \leq \mathsf{sleep}_{upper}$.

The above two steps are executed as part of node v's active period. After completion of its active period, v sleeps for s_v slots and then repeats the above protocol. Due to the assumption that the active period of a node occupies one time slot, the cycle length of node v satisfies $z_v = 1 + s_v$. Variants of the above protocol are obtained by varying how the sleep period s_v is chosen. For example, s_v may vary from node to node and from one cycle to the next, s_v may be chosen using a deterministic rule or a randomized rule, etc. In the rest of this subsection we assume that all nodes in a stably aligned component choose the same sleep period in each cycle, though this may vary from cycle to cycle. This can be achieved without any communication because these nodes share common information (a synchronized clock, the ID of the component leader, etc.) and the sleep period is a function of such information. This is true even if the choice of the sleep period is random; if all nodes in a stably aligned component use their clock value as a seed for the random number generator, they obtain identical sleep periods with no need for additional communication. The following lemma uses elementary number-theoretic arguments to show that the choice of s_v is critical.

Lemma 1. *Let z_a and z_b respectively be the cycle lengths of nodes a and b, in every cycle.*

(i) If z_a and z_b are not relatively prime then there exists an initial displacement of a and b such that the active periods never overlap.

(ii) If z_a and z_b are relatively prime then the active periods of a and b overlap within at most $s_a \cdot s_b$ time slots, regardless of the initial displacement.

Proof. (i) Let $f > 1$ be a common factor of z_a and z_b. For any pair of positive integers m_a and m_b, the quantity $m_a \cdot z_a - m_b \cdot z_b$ is a multiple of f and is therefore distinct from 1. If nodes a and b start with an initial displacement of 1, then no matter how many cycles a runs for and no matter how many cycles b runs for, the active periods of a and b never overlap.
(ii) Without loss of generality, suppose that $z_a > z_b$. Then for any possible displacement k, the equation $m_a \cdot z_a - m_b \cdot z_b = k$ has a positive integer solutions for m_a and m_b satisfying $m_a \leq s_b$ and $m_b \leq s_a$. Thus the active periods of a and b overlap in at most $s_a \cdot s_b$ time slots. □

The above lemma suggests the assignment of relatively prime cycle lengths to different components in order to guarantee convergence. For a pair of components, an easy choice would be integers z and $z+1$ satisfying $2 \leq z < z+1 \leq \mathsf{sleep}_{upper}$ because such a pair is guaranteed to be relatively prime. See Figure 2 for an il-

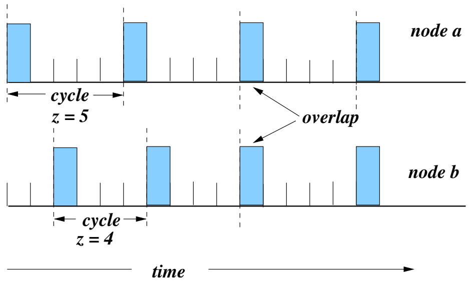


Fig. 2. Illustration for two nodes, a with cycle length 5 and b with cycle length 4. In this example, after the active periods of a and b overlap, node b adopts the cycle length of a and the two nodes will continue to be aligned forever.

lustration of how convergence takes place for $z = 4$. However components are unaware of each other's existence and are therefore unaware of the choices that other components make. Therefore deterministically guaranteeing that the two components make distinct choices is difficult and so we resort to randomization. Let z be a fixed integer satisfying $2 \leq z \leq \mathsf{sleep}_{upper}$. Each node v performs the following step in determining its sleep period.

> $\mathsf{NoCost1}(v)$: In each cycle, node v picks its sleep period s_v uniformly at random from $\{z-1, z\}$.

Thus the cycle length of each node v is either z or $z+1$. Note that the above protocol assumes that all nodes are aware of z. This is justified by assuming that all nodes are informed of sleep_{upper} and use a common, deterministic rule for picking an integer z, $2 \leq z \leq \mathsf{sleep}_{upper} - 1$ (e.g., pick the largest integer z: $2 \leq z \leq \mathsf{sleep}_{upper} - 1$). We now prove an upper bound on the number of time slots it takes for the active periods of a pair of nodes using the NoCost1 rule to overlap. Define the *displacement* of a pair of nodes to be the minimum distance, in time slots, between the active periods of the nodes.

Lemma 2. *In a network in which nodes use the NoCost1 rule for picking sleep periods, for any two nodes a and b, the active periods of a and b will overlap in expected $O(z^3)$ time.*

Proof. Let Z be the random variable denoting the displacement between nodes a and b. Suppose that at the beginning of a cycle $0 < Z < \lfloor z/2 \rfloor$. Then at the end of the cycle, Z increases by 1 with probability 1/4, decreases by 1 with probability 1/4, and retains the same value with probability 1/2. At the two extreme values $Z = 0$ and $Z = \lfloor z/2 \rfloor$, the random variable behaves as follows: (i) if $Z = 0$, then at the end of the cycle, Z increases by 1 with probability 1/2 and retains the same value with probability 1/2; (ii) if $Z = \lfloor z/2 \rfloor$, then at the end of the cycle Z decreases by 1 with probability 1/2 and retains the same value with probability 1/2. This behavior of Z as a function of the number of cycles is a 1-dimensional random walk with reflecting barriers and it is well known [7] that such a random walk will reach $Z = 0$ in expected $O(z^2)$ steps (cycles), which translates to $O(z^3)$ time slots, independent of what the initial value of Z is. □

The random walk described in the above proof is a somewhat inefficient way of removing the displacement between a pair of nodes, especially given the fact that if we could somehow force one component to pick z and the other to pick $z+1$ we could get convergence in $O(z)$ cycles ($= O(z^2)$ time slots) because in each cycle the displacement between the two components would *consistently* change by one slot. This observation motivates the following rule for picking a sleep period.

> NoCost2(v): Node v picks s_v uniformly at random from $\{z-1, z\}$ and retains the same sleep period for $2z$ cycles.

The use of NoCost2 shaves off a factor of "z" from the expected time to overlap and gets us to within a constant factor of the optimal deterministic selection.

Lemma 3. *In a network in which nodes use the NoCost2 rule for picking sleep periods, for any two nodes a and b, the active periods of a and b will overlap in expected $O(z^2)$ time.*

Proof. Let a *phase* denote $2z$ consecutive cycles in which a node has the same sleep period. Although a and b may not begin phases simultaneously, each phase of one node will be concurrent for at least z cycles with a phase of the other. If a and b pick distinct cycle lengths then they overlap within one phase; the probability of this event is 1/2. On the other hand, with probability 1/2, a and b might pick the same cycle length and continue to have the same displacement at the end of the phase. Therefore, the random variable P, denoting the number of phases before overlap, has the geometric distribution $\text{Prob}[P = k] = 1/2^k$ and therefore $\mathrm{E}[P] = O(1)$. This means that the active periods of a and b overlap in expected $O(1)$ phases, which translates to expected $O(z)$ cycles or expected $O(z^2)$ time slots. $\square$

The above lemma considers only two components. We now consider the general case where the network has arbitrarily many stably aligned components of arbitrary sizes. Each node executes NoCost2 and to be concrete we assume the merge rule: if $clock(a) < clock(b)$ then node a joins node b's component.

Theorem 1. *In expected $O(diam(G) \cdot z^2)$ time slots, the network will have exactly one stably aligned component. Here $diam(G)$ refers to the diameter of the network G.*

Proof. Let v be a node with largest clock value. From Lemma 3 we see that in expected $O(z^2)$ time slots, all neighbors of v would have inherited v's clock value. Note that the stably aligned component containing v will continue to have the largest clock value in the network. Continuing inductively, we see that in expected $O(t \cdot z^2)$ time slots, all nodes within t hops from v would have inherited v's clock, leading to the theorem. $\square$

Corollary 1. *There is a self-stabilizing protocol that solves the temporal partition problem without using any extra active time periods in $O(diam(G) \cdot z^2)$ time with a $1/z$ duty cycle.*

A cycle length of z and a duty cycle of $1/z$ prevents the nodes from using any extra active time periods, besides the given active periods they are required to have. In such a setting information takes $\Omega(z)$ time slots to traverse $O(1)$ hops in the network and therefore we get an $\Omega(diam(G) \cdot z)$ lower bound on the convergence time for any no cost protocol. The above result is thus a factor of "z" away from this lower bound.

4.2 Probing Approaches

While the no-cost approach shows convergence with no additional power consumption beyond what is needed for the prescribed active period, it is possible to speed up convergence by increasing the duty cycle by only a constant factor. This is done by permitting nodes to be active for additional time slots outside their prescribed active periods for the purposes of "probing" for other components. During the additional active period, each node will probe for nodes not belonging to its own stably aligned component. During its probing period a node will repeatedly send messages and also listen for messages from other nodes. Contrary to the no-cost approach, we make the sleep period length identical for all nodes in all cycles.

In this section we will consider two methods for probing: deterministic and randomized. In both approaches, during a component's prescribed active period, every node will select a probing period from within the sleeping period during the probing period it will turn its radio on. This probing period may be selected deterministically or randomly, as we shall see in the following.

Deterministic Probing. We start this section with a deterministic probing protocol. Let z be the cycle length and assume that time slots in a cycle are labeled $0, 1, \ldots, z-1$. For an integer parameter c, $1 \leq c \leq z-1$, we define the *probing period* to be a set of size c of consecutive slots in $[1, z-1]$. The basic structure of the protocol is the following: during its active period, a node chooses its next probing period to immediately follow its current probing period. For example, if the current probing period of a node is $[x, x+c-1]$, then its next probing period is $[x+c, x+2c-1]$. The endpoints of the probing period are computed modulo z so as to allow the next probing period to start at the beginning of the cycle, in the event that the previous probing period had reached the end.

Assuming that each component has an ID which is the greatest ID of any node in the component, we have the following merging rule. Suppose that during its active or probing period, the node $v \in A$ discovers a node $u \in B$, where A and B are distinct components. If the ID of B is higher than the ID of A, then v immediately joins B by adjusting its clock. Otherwise, v sends a message containing its clock and the ID of A to u; on receiving v's message u immediately joins A. Recall two assumptions we made in Section 2: (i) the length of the active period (i.e., 1 time unit) is sufficiently large to allow reliable bidirectional communication and (ii) the length of overlap between any two time periods is integral. These two assumptions along with the fact that v and u are neighbors implies that all of the communication described above completes before the end of the active or probing periods of v and u. It is easy to see that the duty cycle of the protocol is $(1+c)/z$, and that in $\lceil z/c \rceil$ cycles, all slots of a cycle are probed by each node. This leads to the following result.

Lemma 4. *In a network where nodes use the deterministic probing protocol with integral* c*,* $1 \leq c \leq z-1$*, any two adjacent nodes will discover each other in* $O(\frac{z^2}{c})$ *time.*

Using the above lemma and an argument essentially identical to that in the proof of Theorem 1, we get the following result.

Theorem 2. *The convergence time of the deterministic probing protocol is* $O(diam(G) \cdot \frac{z^2}{c})$*.*

We now show that the above protocol is optimal within a constant factor by proving a $\Omega(diam(G) \cdot \frac{z^2}{c})$ lower bound on the convergence time of any deterministic ID-based protocol (i.e., a protocol that uses the nodes IDs to govern the merging of two or more components) with a duty cycle of $\frac{1+c}{z}$. We say that discovery occurs between two connected components if the probing period of one overlaps with the active period of the other. We assume that discovery is a sufficient condition to ensure that at least one node belonging to one of the two components will merge with the other.

We define a *probing frame* as the union of all probing periods (over several cycles). For instance, if the probing period of a node is $\{1, 2, \ldots, z-1\}$ (the entire sleeping period) each cycle, then the probing frame of the node is the sleeping period itself. In the same way, if the probing period alternates between $\{1,2\}$ and $\{3,4\}$, the probing frame is $\{1,2,3,4\}$. Moreover, we assume that the probing frame begins just after the active period. Note that all nodes in one stably aligned component have the same probing frame. The next lemma considers the minimum length of the probing frame needed to guarantee that discovery occurs between two connected components.

Lemma 5. *Let A and B be two stably aligned components. Regardless of the initial temporal displacement, when using a deterministic probing protocol discovery is guaranteed between A and B if and only if the length of the probing frame is at least $\lfloor \frac{z}{2} \rfloor$ slots.*

Proof. In this proof we denote the length of the probing frame as p slots. Recall that the length of the sleeping period is $z-1$. We first show that if discovery is to be guaranteed, then $p \geq \lfloor \frac{z}{2} \rfloor$. Assuming that the active period of A does not overlap with the probing frame of B, then since B's probing period consists of the first part of its sleeping period it follows that the active period of A overlaps with B's remaining sleeping period. Thus in order to ensure that the probing frame of A overlaps with the active period of B we require that $z-1-p \leq p \Rightarrow \frac{z}{2} - \frac{1}{2} \leq p$.

Since the length of the probing frame must be an integer number of slots, we get that $p \geq \lfloor \frac{z}{2} \rfloor$.

It is easy to show that if $p \geq \lfloor \frac{z}{2} \rfloor$ then discovery is guaranteed, but due to page constraints we omit the details. □

Given that the length of one probing period is c slots, it follows from Lemma 5 that each probing frame consists of $\lfloor \frac{z}{2} \rfloor \cdot \frac{1}{c}$ probing periods. From this we see that $\Omega(\frac{z}{c})$ cycles are required to guarantee discovery between two components, and since the length of one cycle is z slots, we get the following result.

Lemma 6. *Given a protocol solving temporal partition using a deterministic probing approach. The minimum time complexity ensured by the protocol for discovery to occur between two stably aligned components is $\Omega(\frac{z^2}{c})$ slots where c is the length of the probing period.*

In the following, we will denote a *component graph* $G_c = (V_c, E_c)$ such that every stably aligned component in the network is a node in V_c and there exists an edge between two nodes A and B in G_c if and only if there exists an edge between at least one node in A and one node in B in the network.

Theorem 3. *There does not exist a deterministic ID-based protocol for solving the temporal partition problem that with a duty cycle of $\frac{1+c}{z}$ converges in less than $\Omega(\frac{z^2}{c} \cdot diam)$ slots where diam is the diameter of the network.*

Proof. Assume that such a protocol exists, we will show that this leads to a contradiction. We consider a component graph that is a chain consisting of the components $C_1, C_2, ..., C_{diam}$ where C_i is a neighbor of C_{i-1} and C_{i+1} for every $2 \leq i \leq diam - 1$. We assume that initially every component consists of a single node, and we assign an ID to each node such that $\mathsf{ID}(C_1) > \mathsf{ID}(C_2) > \ldots > \mathsf{ID}(C_{diam})$.

Lemma 6 states that $\Omega(\frac{z^2}{c})$ slots are required before a component is guaranteed to discover a neighboring component (if one exists). Observe that $N(C_i) = \{C_{i-1}, C_i, C_{i+1}\}$ for any $1 \leq i \leq diam$ (within the constraints that $i-1 \geq 1$ and $i+1 \leq diam$). Thus following $\Omega(\frac{z^2}{c})$ slots C_2 will merge with C_1, C_3 will merge with C_2 etc. Then, after the merging, we obtain a new component graph consisting of the components $C_1 \cup C_2, C_3, \ldots, C_{diam}$ connected in a chain such that $\mathsf{ID}(C_1 \cup C_2) > \mathsf{ID}(C_3) > \mathsf{ID}(C_4) > \ldots > \mathsf{ID}(C_{diam})$, since $\mathsf{ID}(C_1 \cup C_2) = \mathsf{ID}(C_1)$. In this new component graph, the diameter has been reduced by 1. Note that the new component graph retains the same properties as the previous one, and that every component C_3, C_4, ..., C_{diam} still only consists of a single node. This allows us to apply an inductive argument.

We see that one component is removed from the graph every $\Omega(\frac{z^2}{c})$ slots. Thus the convergence time is $\Omega(\frac{z^2}{c} \cdot diam)$, contradicting the initial assumption. Since each node is active during $1+c$ time slots each cycles, we see that the duty cycle is $\frac{1+c}{z}$. □

Randomized Probing. In this section we will use *randomized probing* to solve the temporal partition. The basic idea of randomized probing is that each node picks one slot at random, from its sleep period, and remains active during that slot. This approach works best when there is a large component and its nodes have picked different slots to probe, thus covering a large fraction of the sleep period. Randomization helps to achieve this and our analysis uses standard arguments similar to those in the "birthday paradox" or the "coupon collector" problem to assert that if the size of a component is a "$\log n$" factor times the length of a cycle, the component will cover the entire cycle with high probability.

To keep exposition simple, we assume that the underlying network is a clique; the technique works for more general constant-diameter, dense graphs. The basic structure of the protocol is the similar to deterministic probing; recall that there we assumed that all nodes have the same cycle length z in all cycles. The key differences between our randomized probing protocol and our deterministic probing protocol are enumerated below.

1. The "merge rule" is different. Node v merges with the component of node u provided u's component is larger in size. If the sizes of the two components are identical, then the IDs of leaders are used to break the tie. We will show later that using the sizes of components as part of the merge rule plays a critical role in ensuring fast convergence.
2. Let the time slots in a cycle be labeled $\{0, 1, \ldots, z-1\}$, with 0 denoting the active period. During the active period each node also picks a time slot t uniformly at random from $\{1, 2, \ldots, z-1\}$. After its active period, each node goes to sleep for the next $z-1$ time slots, with the exception of time slot t; this time slot is used for probing.

For the purposes of ensuring enough "coverage" of the sleep period, it is critical for this protocol that the random choices of t be independent even for the nodes in the same component.

We start the analysis of this protocol be assuming that there are k components, labeled $C_1, C_2, \ldots, C_k$ such that $|C_1| \geq |C_2| \geq \cdots \geq |C_k|$ and furthermore if $|C_{i+1}| = |C_i|$ then the ID of the component leader of C_{i+1} is greater than the ID of the leader of C_i. Note that this labeling is just for the purposes of the proof and is not computed by the algorithm. This ordering of components guarantees that if a node $v \in C_i$ leaves its component to join another, then that component is one of $C_1, C_2, \ldots, C_{i-1}$. Our analysis assumes that $z \leq n/8 \log n$ (used in the proof of Lemma 7). There are two cases depending on the size of C_1.

"Large" C_1. Suppose that $|C_1| \geq 2z \cdot \log n$. We show that in this case, with high probability, in one cycle, all nodes in $C_2 \cup C_3 \cup \cdots \cup C_k$ will join component C_1. Consider a node $v \in C_2 \cup C_3 \cup \cdots \cup C_k$. The probability that v's extra active period will not overlap with the active periods of any node in C_1 is

$$\left(1 - \frac{1}{z}\right)^{|C_1|} \leq \left(1 - \frac{1}{z}\right)^{2z \cdot \log n} \sim e^{-2 \log n} = \frac{1}{n^2}.$$

Since there are n nodes, by using the union bound we see that with probability at most $1/n$ there is a node $v \in C_2 \cup C_3 \cup \cdots \cup C_k$ whose extra active period does not overlap with the extra active period of any node in C_1. Therefore, with probability at least $1 - 1/n$, every node outside C_1 will join C_1 in one cycle (of length z time slots).

"Small" C_1. Here we suppose that $|C_1| < 2z \cdot \log n$. In this case we show that the number of components decreases by a constant fraction in each cycle. Consider a permutation π of all nodes in which C_1 comes first, followed by C_2, followed by C_3, and so on. The nodes in each C_i appear in some arbitrary order in π. Let v be the node with rank $\lfloor n/2 \rfloor$ in π and let C_j be the component that contains v. Call C_j the *middle component*, $C_1, C_2, \ldots, C_{j-1}$, the *big components*, and $C_{j+1}, C_{j+2}, \ldots, C_k$, the *small components*. We will show two properties.

Property 1. The number of small components is at least a constant fraction of the number of large components.

Property 2. In one cycle, all the small components will disappear, with high probability.

Together these properties lead to the claim that in the "small" C_1 case a constant fraction of the components disappear in each cycle. Property (2) follows from the same argument that was used to deal with the "large" C_1 case. Property (1) is proved in the following lemma.

Lemma 7. *The number of small components is at least a constant fraction of the number of large components.*

Proof. Since $|C_1| < 2z \log n$, every component has size at most $2z \log n$. Using the assumption that $z \leq n/8 \log n$ yields an upper bound of $n/4$ on the size of every component. In particular, the middle component C_1 has size at most $n/4$ and from this it follows that the union of the small components has size at least $n/4$. Let the number of small components be s. The average size of the small components is at least $n/4s$. Therefore every big component has size at least $n/4s$, implying that the number of big components is at most $2s$. $\square$

Note that if the number of components is $n/2z \log n$ or fewer, then there is at least one component of size at least $2z \log n$, putting us in the "large" C_1 case and guaranteeing convergence in one cycle (with high probability). So suppose that the number of components in more than $n/2z \log n$. Even if the network starts off with n components (i.e., every node is a components by itself), the progress we make in the "small" C_1 case implies that in $O(\log(z \log n))$ cycles, we will reach a state in which there at at most $n/2z \log n$ components. Given that each cycle has z time slots, we get the following theorem.

Theorem 4. *In expected $O((\log z + \log \log n) \cdot z)$ time slots, the network will have exactly one stably aligned component.*

5 Conclusions

The no-cost approach provides an $O(diam(G) \cdot z^2)$ convergence time with $1/z$ duty cycle for any $z \leq \mathsf{sleep}_{upper}$. The optimality of this result is currently unclear to us and it is possible that further randomization could improve the convergence time to $O(diam(G) \cdot z \log z)$. The deterministic probing approach yields a convergence time of $O(diam(G) \cdot z^2/c)$ for any c, $1 \leq c \leq z - 1$. This leads to a spectrum of convergence times and duty cycles, obtained by varying c relative to z such that the product of the convergence time and duty cycle equals $O(diam(G) \cdot z)$. For example, if c is picked close to $\sqrt{z}$, we get a convergence time of $O(z^{3/2})$ and a duty cycle that is approximately $1/\sqrt{z}$. While this flexibility might seem like an advantage that the deterministic probing approach has over the no-cost approach, it is worth pointing out that we can pick any $z \leq \mathsf{sleep}_{upper}$ for the no-cost approach and obtain a similar flexibility. In the light of this, it is not clear if the deterministic probing approach has any advantage over the no-cost approach, at least in the worst case. However, the randomized probing approach does yield asymptotically faster convergence (with the same duty cycle) relative to the no-cost approach under certain circumstances. If the network is assumed to be a clique, then the no-cost approach guarantees an $O(z^2)$ convergence time for a $1/z$ duty cycle. Comparing this with the $O((\log z + \log \log n) \cdot z)$ convergence time of the randomized probing approach, we note that whenever $\log \log n = o(z)$, we get an asymptotically faster convergence time using randomized probing. Informally speaking, unless z is very small, the randomized probing approach is much faster than the no cost approach for dense network. Given this positive news for the randomized probing approach, it may be worthwhile to expand this approach to more general classes of graphs. For example, we are currently analyzing the randomized probing approach for classes of graphs whose min-cut value is bounded from below.

References

1. Ye, W., Heidemann, J., Estrin, D.: An energy-efficient MAC protocol for wireless sensor networks. In: INFOCOMM 2002. Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communication Societies, IEEE Computer Society Press, Los Alamitos (2002)
2. van Dam, T., Langendoen, K.: An adaptive energy-efficient MAC protocol for wireless sensor networks. In: SenSys 2003. Proceedings of the First ACM Conference on Embedded Networked Sensor Systems, pp. 171–180. ACM Press, New York (2003)
3. Rajendran, V., Obraczka, K., Garcia, J.J., Aceves, L.: Energy-efficient collision-free medium access control for wireless sensor networks. In: SenSys 2003. Proceedings of the First ACM Conference on Embedded Networked Sensor Systems, pp. 181–193. ACM Press, New York (2003)
4. Ye, W., Heidemann, J., Estrin, D.: Medium access control with coordinated adaptive sleeping for wireless sensor networks. IEEE Transactions on Networking (2004)
5. Ye, W., Silva, F., Heidemann, J.: Ultra-low duty cycle MAC with scheduled channel polling. In: SenSys 2006. Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, pp. 321–334 (2006)
6. Polastre, J., Hill, J., Culler, D.: Versatile low power media access for wireless sensor networks. In: SenSys 2004. Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, pp. 95–107 (2004)
7. Motwani, R., Raghavan, P.: Randomized Algorithms. Combridge University Press, New York (1995)
8. Ramanathan, N., Yarvis, M., Chhabra, J., Kushainagar, N., Krishnamurthy, L., Estrin, D.: A stream-oriented power management protocol for low duty cycle sensor network applications. In: EMNETS 2005. Proceedings of the Second IEEE Workshop on Embedded Networked Sensors, IEEE Computer Society Press, Los Alamitos (2005)
9. Li, Y., Ye, W., Heidemann, J.: Energy efficient network reconfiguration for mostly-off sensor networks. In: SECON 2006. Proceedings of the Third IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks, pp. 527–535. IEEE Computer Society Press, Los Alamitos (2006)
10. Cao, H., Parker, K.W., Arora, A.: O-MAC: a receiver centric power management protocol. In: ICNP 2006. Proceedings of the 14th IEEE International Conference on Network Protocols, IEEE Computer Society Press, Los Alamitos (2006)
11. Zheng, R., Hou, J.C., Sha, L.: Asynchronous wakeup for ad hoc networks. In: MOBIHOC 2003. Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 35–45. ACM Press, New York (2003)

Secure and Self-stabilizing Clock Synchronization in Sensor Networks

Jaap-Henk Hoepman[1], Andreas Larsson[2], Elad M. Schiller[2], and Philippas Tsigas[2]

[1] TNO ICT, and Radboud University Nijmegen
jaap-henk.hoepman@tno.nl
[2] Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University
{larandr,elad,tsigas}@chalmers.se

Abstract. In sensor networks, correct clocks have arbitrary starting offsets and nondeterministic fluctuating skews. We consider an adversary that aims at tampering with the clock synchronization by intercepting messages, replaying intercepted messages (after the adversary's choice of delay), and capturing nodes (i.e., revealing their secret keys and impersonating them). We present the first self-stabilizing algorithm for secure clock synchronization in sensor networks that is resilient to such an adversary's attacks. Our algorithm tolerates random media noise, guarantees with high probability efficient communication overheads, and facilitates a variety of masking techniques against pulse-delay attacks in the presence of captured nodes.

Keywords: Secure and Resilient Computer Systems, Sensor-Network Systems, Clock-synchronization, Self-Stabilization.

1 Introduction

Accurate clock synchronization is imperative for many applications in sensor networks such as mobile object tracking, detection of duplicates, and TDMA radio scheduling. Broadly speaking, existing clock synchronization protocols are too expensive for sensor networks because of the nature of the hardware and the limited resources that sensor nodes have. The unattended environment, in which sensor nodes typically reside, necessitates secure solutions and autonomous system design criteria that are self-defensive against a malicious adversary.

To illustrate an example of clock synchronization importance, consider a mobile object tracking application which monitors objects that pass through the network area (see [2]). Nodes detect the passing objects, record the time of detection, and send the estimated trajectory. Inaccurate clock synchronization would result in an estimated trajectory that could differ significantly from the actual one.

We propose the first self-stabilizing algorithm for clock synchronization in sensor networks with security concerns. We consider an adversary that intercepts messages that it later replays. Our algorithm guarantees automatic recovery after the occurrence of arbitrary failures. Moreover, the algorithm tolerates message omission failures that might occur, say, due to the algorithm's message collisions or due to random media noise.

The propagation delay of messages in short distance wireless communications allows nodes to use broadcast transmissions to approximate pulses that mark the time of real physical events (i.e., beacon messages). In the *pulse-delay* attack, the adversary snoops messages, jams the synchronization pulses, and replays them at the adversary's choice of time (see [9,10,19]).

We are interested in fine-grained clock synchronization, where there are no cryptographic counter measures for such pulse-delay attacks, e.g., the *nonce* techniques strive to verify the freshness of a message by issuing pseudo-random numbers for ensuring that old communications could not be reused in replay attacks (see [18]). Unfortunately, the lack of fine-grained clock synchronization implies that the round-trip time of message exchange cannot be efficiently estimated.

The system strives to synchronize its clocks while forever monitoring the adversary. We assume that the adversary cannot break existing cryptographic primitives for sensor networks by eavesdropping (e.g., [18,22]). However, we assume that the adversary can *capture* nodes, reveal their entire state (including private variables), stop their execution, and impersonate them.

We assume that, at any time, the adversary has a distinct location in space and a bounded influence radius, uses omnidirectional broadcasts from that distinct location, and cannot intercept broadcasts for an arbitrarily long period. (Namely, we consider a model that is comparable to the one of Gilbert et al. [11], which considers the minimal requirements for message delivery under broadcast interception attacks.) We explain how, by following these realistic assumptions, we can sift out responses to delayed beacons.

A secure synchronization protocol should mask attacks by an adversary that aims to make the protocol give an erroneous output. Unfortunately, due to the unattended environment and the limited resources, it is unlikely that all the designer's assumptions hold forever, e.g., over time, the number of captured nodes becomes sufficiently large for the adversary to tamper with the clock.

We consider systems that have the capability of monitoring the adversary , and then stopping it by external intervention. In this case, the nodes start executing their program from an arbitrary state. From that point on, we require rapid system recovery. Self-stabilizing algorithms [3,4] cope with the occurrence of transient faults in an elegant way. Self-stabilizing systems can be started in *any* configuration, which might occur due to the occurrence of an arbitrary combination of failures. From that arbitrary starting point, the algorithm must ensure that it accomplishes its task if the system obeys the designer's assumptions for a sufficiently long period.

We focus on the fault-tolerance aspects of secure clock synchronization protocols in sensor networks. Our objective is to design a distributed algorithm for sampling n clocks in the presence of t incorrect nodes (i.e., faulty or captured). The clock sampling algorithm facilitates clock synchronization using a variety of existing masking techniques to overcome pulse-delay attacks in the presence of captured nodes, e.g., [9,10] uses Byzantine agreement (requires $3t + 1 \leq n$), and [19] considers the statistical outliers (requires $2t + \epsilon \leq n$, where $\epsilon \in O(1)$).

Our Contribution. We present the first design for secure and self-stabilizing clock synchronization in sensor networks that is resilient to an adversary that can capture nodes and launch pulse-delay attacks. Our design tolerates transient failures that may occur due to temporary violation of the designer assumption, e.g., the adversary captures more than t nodes and then stops. After the system resumes operation according to designer assumption, the algorithm secures with high probability clock precision that is $O((\log n)^3)$ times the optimum, where $\Omega(n^2)$ is the optimum and n is the number of sensor nodes. We assume that (before and after the system's recovery) there are message omission failures, say, due to random media noise or the algorithm's message collision. The correct node sends beacons and responds to the other nodes' beacons. We use a randomized strategy for beacon scheduling that guarantees collision avoidance with high probability.

Document structure. We start by describing the system settings (Section 2) and formally present the algorithm (Section 3). Then we review the literature and draw our conclusions (Section 4).

2 System Settings

We model the system as one that consists of a set of communicating entities, which we call processors (or nodes). We denote the set of processors by P, where $|P| \leq N$; N is an upper bound on the number of processors and is known by the processors themselves. In addition, we assume that every processor $p_i \in P$ has a unique identifier, i.

Time, Clocks, and Their Notation. We follow a model compatible with the one of Herman and Zhang [12]. We consider three notations of time: *real time* is the usual physical notion of continuous time, used for definition and analysis only; *native time* is obtained from a native clock, implemented by the operating system from hardware counters; *local time* builds on native time with an additive adjustment factor in an effort to approximate a cluster-wise clock.

We consider applications that require the clock interface to include the *read* operation, which returns a timestamp with T possible states. Let C_k^i and c_k^i denote the value $p_i \in P$ gets from the k^{th} *read* of the native or local clock, respectively. Moreover, let r_k^i denote the real-time instant associated with that k^{th} *read* operation.

Clock counters do not increment at ideal rates, because the hardware oscillators have manufacturing variations and the rates are affected by voltage and temperature. The clock synchronization algorithm adjusts the local clock in order to achieve synchronization, but never adjusts the native clock. We define the native clocks *offset* $\delta_{i,j}(k,q) = C_k^i - C_q^j$, where $\Delta_{i,j}(k,q) = r_k^i - r_q^j = 0$. We assume that, throughout system execution, the native clock offset is arbitrary. Moreover, the *skew* of p_i's clock $\rho_i = \lim_{\Delta_{i,i}(k,q)\to 0} \delta_{i,i}/\Delta_{i,i}(k,q)$ is in $[\rho_{\min}, \rho_{\max}]$, where $\rho_{\min}$ and $\rho_{\max}$ are known constants. Thus, the clock skew is the first derivative of the clock offset value with respect to real time. Because clock skew is generally not constant, higher order derivatives of the clock rate are nonzero. The relative clock skew is $\rho_{i,j} = \rho_i - \rho_j$. We assume that $1 - \kappa \leq \rho_i \leq 1 + \kappa$. The second derivative of the clocks' offset is called *drift*. We follow the approach of Herman and Zhang [12] and allow non-zero drift (as long as $\rho_i \in [\rho_{\min}, \rho_{\max}]$).

Communications. Wireless transmissions are subject to collision and noise. The processors communicate among themselves using a local broadcast primitive, *LBcast* and *LBrecv*, with a transmission radius of at most R_{lb}. We consider the potential of any pair of processors to communicate directly, or to interfere with each others communications.

We associate every processor, p_i, with a fixed and unknown location in space, L_i. We denote the potential set of processors that processor $p_i \in P$ can directly communicate with (with whose communications, processor p_i can interfere) by $G_i \subseteq \{p_j \in P | R_{lb} \geq |L_i - L_j|\}$ (respectively, $\overrightarrow{G_i} \subseteq \{p_j \in P | 2R_{lb} \geq |L_i - L_j|\}$). We assume that $n \geq |\overrightarrow{G_i}|$ is a known upper bound on the node's degree.

Communication Operations. We model the communication channel, $queue_{i,j}$, from processor p_i to processor $p_j \in G_i$ as a FIFO queuing list of the messages that p_i has sent to p_j and p_j is about to receive. When p_i broadcasts message m, the operation *LBcast* inserts a copy of m to every $queue_{i,j}$, such that $p_j \in G_i$. Every message $m \in queue_{i,j}$ is associated with a particular time at which m arrives at p_j. Once m arrives, p_j executes *LBrecv*. We require that the period between the time in which m enters the communication channel and the time in which m leaves it, is at most a constant, d. We assume that d is a known and efficient upper bound on the communication delay between two neighboring processors.

Accessing the Communication Media. We assume that processor p_i uses the following optimization, which is part of many existing implementations. Before accessing the communication media, p_i waits for a period d and broadcasts only if there was no message transmitted during that period. Thus, processor p_i does not intercept broadcasts that have started (and did not finish) before time $t - d$, where t is the time of the broadcast by p_i.

Security Primitives. The existing literature describes many elements of the secure implementation of the broadcast primitives *LBcast* and *LBrecv* using symmetric key encryption and message authentication (e.g., [18,22]). We assume that neighboring processors store predefined pairwise secret keys. In other words, $p_i, p_j \in P : p_j \in G_i$ store keys $s_{i,j} : s_{i,j} = s_{j,i}$. The adversary cannot efficiently guess $s_{i,j}$. Confidentiality and integrity are guaranteed by encrypting the messages and adding a message authentication code. We can guarantee messages' freshness by adding a message counter (coupled with the beacon's timestamp) to the message before applying these cryptographic operations, and by letting receivers reject old messages, say, from the clock's previous incarnation. Note that this requires maintaining, for each sender, the index of the last properly received message. As explained above, the freshness criterion is not suitable for fine-grained clock synchronization in the presence of pulse-delay attacks.

The Interleaving Model. Every processor p_i executes a program that is a sequence of *(atomic) steps*. For ease of description, we assume the interleaving model where steps are executed atomically, a single step at any given time. An input event, which can be either the receipt of a message or a timer going off, triggers each step of p_i. Only steps that start from a timer going off may include (at most once) an *LBcast* operation. We note that there could be steps that read the clock and decide not to broadcast.

Since no self-stabilizing algorithm terminates (see [4]), the program of a processor consists of a do-forever loop. An iteration is said to be complete if it starts in the loop's first line and ends at the last (regardless of whether it enters branches). A processor executes other parts of the program (and other programs) and activates the loop upon a time-out. We assume that every processor triggers the loop's time-out within every period of $u/2$, where $u > w + d$ is the *(operation time) slot*, where w is the time it takes to execute a complete iteration of the do-forever loop, including all messages received in that slot, assuming that there is a known upper bound on the number of those. Since processors execute programs other than the clock synchronization, the actual time in which the timer goes off is hard to predict. Therefore, for the sake of simplicity, we assume that the this time has a uniform distribution. We note that a simple random scheduler can be used for the case in which the this time can be characterized.

The *state* s_i of a processor p_i consists of the value of all the variables of the processor (including the set of all incoming communication channels, $\{queue_{j,i} | p_j \in G_i\}$). The execution of a step in the algorithm can change the state of a processor. The term *system configuration* is used for a tuple of the form $(s_1, s_2, \cdots, s_n)$, where each s_i is the state of processor p_i (including messages in transit for p_i). We define an *execution* $E = c[0], a[0], c[1], a[1], \ldots$ as an alternating sequence of system configurations $c[x]$ and steps $a[x]$, such that each configuration $c[x+1]$ (except the initial configuration $c[0]$) is obtained from the preceding configuration $c[x]$ by the execution of the step $a[x]$. We often associate the notation of a step with its executing processor p_i using a subscript, e.g., a_i.

Tracing Timestamps and Communications. The communication operations that we use, *LBcast* and *LBrecv*, have a time notation that we call *timestamp*. We assume that all timestamps have T possible states. We assume the existence of an efficient algorithm for timestamping the message in transfer (see [22]).

That is, the sent message includes the estimated value of the native clock at sending time. The timestamp of an *LBcast* operation is the native time at which message m is sent. When processor p_i executes the *LBrecv* operation, an event is triggered with the arguments j, t, and $\langle m\rangle$: p_j is the sending processor of message $\langle m\rangle$, which p_i receives when p_i's native clock is (approximately) t. We note that every step can be associated with at most one communication operation and therefore with one access to the native clock counter during or at the end of the operation. We denote by $C^i(a_i)$ the native clock value associated with the communication operation in step a_i, which processor p_i takes.

Adversarial Message Omission and Delay. We assume that at any time, the adversary and all processors have distinct (unknown) locations in space. We assume that there is a single adversary and that its radio transmitter sends omni-directional broadcasts (using antennas that radiate equally in space). Therefore, the adversary cannot arbitrarily control the distribution in space of the set of recipients for which the beacon's broadcast is delayed or omitted. We assume that it chooses a sphere that divides the set of processors in two: (1) The correct receivers are outside the sphere and receive all beacons on time, and (2) The late receivers are inside the sphere and receive either no beacon or beacons after a delay that is greater than a known constant.

Concurrent vs. Independent Broadcasts. We say that processor p_i performs an *independent broadcast* in a step $a_i \in E$ if there is no processor $p_j \in P$ that broadcasts in a step $a_j \in E$, such that either (1) a_j is performed after a_i and before step a_k^r that receives the message that was sent in a_i (where $p_k \in P$), or (2) a_i is performed after a_j and before step a_k^r that receives the message that was sent in a_j. We say that processor $p_i \in P$ performs a *concurrent broadcast* in a step a_i if a_i is dependent (i.e., "not independent"). Concurrent broadcasts can cause message collisions.

Fair Communications. The processors reside in the unattended environment and malicious adversarial activity is not the only reason why communication links may fail. Therefore, we consider message omission due to either random media noise or message collisions that the algorithm causes.

Gilbert et al. [11] consider the minimal requirements for message delivery under broadcast interception attacks and assume that the adversary intercepts no more than β broadcasts of the algorithm, where β is a known constant. We note that the result of Gilbert et al. is applicable in a model in which, in every period, the algorithm is able to broadcast at most α messages the adversary can intercept at most β. In other words, our assumption regarding the ratio of β/α is comparable to the model of Gilbert et al. [11]. The parameter $\xi \geq 1$ denotes the maximal number of repeated transmissions required for a single successful message transfer whenever there are no message collisions due to the algorithm's concurrent broadcasts. We assume that all processors know ξ.

We say that execution E has *fair communications*, if, whenever processor p_i independently broadcasts ξ successive messages in steps $a_i^{\xi} \in E$, every processor receives at least one of these messages. We note that fair communication does not imply reliable communication even for $\xi = 1$, because processors might broadcast concurrently when there is no agreed broadcast schedule or when the clock synchrony is not tight.

The Environment. The environment that restricts the adversary's ability to launch message interception attacks guarantees fair communication. The environment can execute the operation $\mathit{omission}_i(m)$ (which is associated with a particular message, m, sent by processor p_i) immediately after $\mathit{LBcast}_i(m)$. The environment selects a subset of p_i's neighbors ($R_i \subseteq G_i$) to remove any message m_i from their queues $queue_{i,j}$ (such that $p_j \in R_i$). We assume that the environment arbitrarily selects R_i when invoking *omission* due to algorithm message collision. The adversary, under the environment's supervision, selects messages to remove due to random media noise. The adversary launches message interception attacks by selecting R_i. The environment supervises so the adversary does not violate the fair communication requirements.

System Specifications ***Fair Executions.*** An execution E is *fair* if the communications are fair and every correct processor, p_i, executes steps in a timely manner (by letting the loop's timer go off in the manner that we explain above).

The Task. We define the system's task by a set of executions called *legal executions* (LE) in which the task's requirements hold. A configuration c is a *safe configuration* for an algorithm and the task of LE provided that any execution that starts in c is a legal execution (belongs to LE). An algorithm is *self-stabilizing* with relation to the task of LE if every infinite execution of the algorithm reaches a safe configuration with relation to the algorithm and the task.

Clock Synchronization Requirements. Roughly speaking, without any attacks or failures, the native clocks follow similar characteristics. Processors can synchronize their local clocks by revealing these characteristics. The task's output decodes the coefficient vector of a finite degree polynomial $P_{i,j}(t)$ that closely approximates the native clock value of processor p_j at time t, where t is a value of p_i's native clock. Römer et al. [16] explain how to calculate $\{P_{i,j}(t)\}_{j \neq i}$.

Elson et al. [7,6] explain how to calculate the global and the local clocks using $\{P_{i,j}(t)\}_{j \neq i}$. We note that the local c^i could be agreed in different manners, one of which is based on clustered networks. In each cluster, every processor considers a predefined set of processors, call the *cluster head*, for which it tries to estimate a common local time using a predefined deterministic function.

This paper presents an algorithm for sampling n neighbouring clocks. We measure the algorithm's performance by looking at the period, $\Gamma(n)$, it takes n processors to send at least one beacon that all processors respond to. In other words, we are interested in the minimal period in which all processors are able to complete roundtrip message exchange.

Let p_i, p_j, and p_k be three correct nodes such that p_i and p_j are of type (1) and p_k is of type (2). Suppose that p_j broadcasts a message that p_k receives (after a delay) and p_k then sends a response message that p_i receives (possibly $i = j$). We require that p_k detects that p_j has responded to a delayed message in the presence of at most t captured nodes.

3 Secure and Self-stabilizing Clock Synchronization

In order to explain better the scope of the algorithm, we present a generic organization of secure clock synchronization protocols. The objectives of the clock synchronization protocol are to: (1) periodically broadcast beacons, (2) respond to beacons, and (3) aggregate beacons with their responses in records and deliver them to the upper layer. Every node estimates the clock after sifting out responses to delayed beacons. Unlike objectives (1) to (3), the clock estimation task is not a hard realtime task. Therefore, the algorithm outputs records to the upper layer that synchronizes clocks after neutralizing the effect of pulse-delay attacks (see section 4 for more details). The algorithm focuses on the following two tasks:

• *Beacon Scheduling:* The nodes sample clock values by broadcasting beacons and waiting for their responses. The task is to guarantee round-trip message exchange.
• *Beacon and Response Aggregation:* Once a beacon completes the round-trip exchange, the nodes deliver to the upper layer the records of a beacon and its set of responses.

We present a design for an algorithm that samples clocks of neighboring processors by continuously sending beacons and response. Without synchronized clocks, the nodes cannot efficiently follow a predefined schedule. Moreover, assuring reliable communication becomes hard in the presence of random media noise and message collision. The celebrated Aloha protocol [1] (which does not consider nondeterministic fluctuating skews) inspires us to take a randomized strategy for scheduling broadcasts and overcome the above difficulties by showing that with high probability there are no concurrent broadcasts. Our scheduling strategy is simple; the processors choose a random time to broadcast from a predefined period D. We use time redundancy to overcome the clocks' asynchrony and the difficulty in measuring D. Moreover, we use a parameter, ℓ, used to trade off between minimal size of D and the probability of having a collision free schedule.

Beacon and Response Aggregation. The algorithm allows the use of clock synchronization techniques such as *round-trip synchronization* [9,10] and *reference broadcasting* [6]. For example, in the round trip synchronization technique, the sender p_j sends a timestamped message $\langle t_1 \rangle$ to receivers, p_k, which receive the message at time t_2. The receiver p_k responds with the message $\langle t_1, t_2, t_3 \rangle$, which p_k sends at time t_3 and p_j receives at time t_4. Thus, the output records are in the form of $\langle j, t_1, \{\langle k, \langle t_2, t_3, t_4 \rangle \rangle\} \rangle$, where $\{\langle k, \langle t_2, t_3, t_4 \rangle \rangle\}$ is the set of all received responses sent by nodes p_k.

We piggyback beacon and response messages. For the sake of presentation simplicity, let us start by assuming that all beacon schedules are in a (deterministic) Round Robin fashion. Given a particular node p_i and a particular beacon that p_i sends at time t_s^i, we define t_s^i's *round* as the set of responses, $\langle t_s^j, t_r^j \rangle$, that p_i sends to node p_j for p_j's previous beacon, t_s^j, where t_r^j

is the time in which p_i received p_i's beacon t_s^j. Node p_i piggybacks its beacon with the responses to nodes, p_j, and the beacon message, $\langle v_i \rangle$, is of the form: $\langle \langle t_s^1, t_r^1 \rangle, \ldots \langle t_s^{i-1}, t_r^{i-1} \rangle, t_s^i, \langle t_s^{i+1}, t_r^{i+1} \rangle, \ldots \langle t_s^n, t_r^n \rangle \rangle$.

Now, suppose that the schedules are not done in a Round Robin fashion. We denote p_j's sequence of up to $BLog$ most recently sent beacons with $[t_s^j(k)]_{0 \leq k < BLog}$, among which $t_s^j(k)$ is the k-*th* oldest and $BLog$ is a predefined constant. (We note that $BLog$ may depend on the safety parameter, ℓ, for assuring that all nodes successfully broadcast.) We assume that, in every schedule, p_i receives at least one beacon from p_j before broadcasting $BLog$ beacons. Therefore, p_i's beacon message, $\langle v_i \rangle$, can include a response to p_j's most recently received beacon, $t_s^j(k)$, where $0 \leq k < BLog$.

Since not every round includes a response to the last beacon that p_i sends, then p_i stores its last $BLog$ beacon messages a FIFO queue, $q_i[k] = [t_s^j]_{0 \leq k < BLog}$. Moreover, every beacon message includes all responses to the $BLog$ most recently received beacons from all nodes. Let $q_j = q[k]_{0 \leq k < BLog}$ be p_i's FIFO queue of the last $BLog$ records of the form $\langle t_s^j(k), t_r^j(k) \rangle$, among which $t_s^j(k)$ is p_i's k-*th* oldest beacon from p_j, $t_r^j(k)$ is the time at which it was received and $i \neq j$. The new form of the beacon message is: $\langle q_1, \ldots q_{i-1}, q_i, q_{i+1}, \ldots q_n \rangle$. In the round trip synchronization, the nodes take the role of a *synchronizer* that sends the beacon and waits for responses from the other nodes. The program of node p_i considers both cases in which p_i is, and is not, respectively the synchronizer.

The Algorithm's Pseudo-code. The pseudo-code, in Figure 1, includes two procedures: (1) a do-forever loop that schedules and broadcasts beacon messages (lines 53 to 63) and (2) an upon message arrival procedure (lines 66 to 68).

The Do-Forever Loop. Recall that by our system settings assumptions (Section 2), we assume that the do-forever loop's timer will go off within any period of $u/2$. Moreover, since the actual time cannot be predicted, we assume that the actual schedule has a uniform distribution over the period u. (A straightforward random scheduler can assist, if needed, to enforce the last assumption.) The do-forever loop periodically tests whether the "timer" has expired (in lines 53 to 58). In case the beacon's next schedule is in the "too far in the past" or "too far in the future", then processor p_i "forces" the "timer" to expire (line 55). The algorithm tests that all the stored beacon messages are ordered correctly and refer to the last $BLog$ beacons (line 56). In the case where the stored beacon messages are incorrect, then the algorithm flushes the queues (line 57).

When the time slot arrives, the processor outputs a synchronizer case record, a response to the beacons that processor p_i has sent $BLog$ rounds ago (line59). These data can be used for the round-trip synchronization and delay detection in the upper layer. Then, p_i enqueues the timestamp of the beacon it is about to send during this schedule (line 60). The next schedule for processor p_i is set (lines 61 and 62) just before it broadcasts the beacon message (line 63).

The Message Arrival. When a beacon message arrives (line 65), processor p_i outputs a record of the non-synchronizer case (line 68). This is not done before

```
   Constants:
2    $i =$ id of executing processor
     $n =$ total number of processors
4    $w =$ compensation time between lines 53 and 63
     $d =$ upper bound on message propagation delay
6    $u =$ size of a slot in time units $(u > d + w)$
     $BLog = \lceil(\xi + 2)(\rho_{\max} - \rho_{\min})\rceil$, backlog size
8    $\ell$ = the safety parameter
     $D = \ell\, n \log n$, the broadcast time slots
10   $T =$ number of possible states of a timestamp $(T \gg Du)$

12 Variables:
     native_clock : immutable storage of the native clock
14   $m[n]$ = all received messages and timesamps
       each entry is an array $v[n]$
16       each entry is a queue $q[BLog]$
           each entry is a pair $\langle s, r\rangle$
18   cslot : $[0, D\text{-}1]$ = current slot in use
     next : $[0, T\text{-}1]$ = schedule of next broadcast
20   $cT$ = last do-forever loop's timestamp

22 External functions:
     output$(R)$ : delivers record $R$ to the upper layer
24   choose$(S)$ : uniform selection of an item from the set $S$
     siz$(Q)$ : size of the queue
26   fst$(Q)$ : least recently enqueued element in $Q$, number 0
     lst$(Q)$ : most recently enqueued element in $Q$
28   flush$(Q)$ : empties the queue $Q$
     get$(t,Q)$ : list elements of field $t \in \{s,r\}$ in $Q$
30
   Macros and inlines:
32   border$(t)$ : $(D\text{-}cslot)u + t \mod T$
     schedule$(t)$ : $cslot \cdot u + t \mod T$
34   leq$(x, y)$ : $(\exists\, b : 0 \le b \le 2\, BLog\, D\, u\ \wedge$
       $y \mod T = x + b \mod T)$
36   enq$(q, m)$ : {while full$(q)$ do dequeue$(q)$; enqueue$(m)$ }
     cvec$(v,t)$ : siz$(v) = 0 \vee ($leq(fst$(v),t) \wedge$ leq(lst$(v),t) \wedge$
38     $\{\forall\, b_1 < b_2, \{b_1,b_2\} \subseteq [1,$siz$(v)]$ : leq$(v[b_1], v[b_2])\}$
     checki$(t)$ : cvec(get$(s, m[i].v[i].q), t)$
40   check$(t)$ : $\wedge\ \{\forall\, j \in P\text{-}\{i\}$ : cvec(get$(r, m[i].v[j].q), t)\}$
   (* Get response-record for $p_j$, for $p_i$ as the synchronizer *)
42   tsi$(s, j)$ : {if $\nexists\, b : s = m[j].v[i].q[b].s$ then return $\perp$
       else return
44       $\langle m[j].v[i].q[b].r$, lst$(m[i].v[j]).s$, lst$(m[i].v[j]).r\rangle$ }
     matches$(j)$ : $\{b :$ tsi$(m[i].v[i].q[b].s, j) \ne \perp\}$
46   sci$(j)$ : if matches$(j) = \emptyset$ then return $\perp$
       else return min(matches$(j)$)
48 (* Get response-record for $p_k$, for $p_j$ as the synchronizer *)
     ts$(s, j, k)$ : details appear in [14].
50   sc$(j, k)$ : details appear in [14].

52 Do forever, every $u/2$
     let $cT$ = read(native_clock) + $w$
54   if $\neg$ (leq$(next\text{-}2Du, cT) \wedge$ leq$(cT, next{+}u)$) then
       $next \leftarrow cT$
56   if $\neg$ (checki$(cT) \wedge$ check$(cT)$) then
       $\forall\, j,k \in P$ : flush$(m[j].v[k].q)$
58   if leq$(next, cT) \wedge$ leq$(cT, next + u)$ then
       output $\langle i, \{\ \langle$sci$(j), j,$ tsi(sci$(j), j)\rangle : j \in P\text{-}\{i\}\}\ \rangle$
60     enq$(m[i].v[i].q, \langle cT, \perp\rangle)$
       $(next, cslot) \leftarrow$ (border$(next)$, choose$([0, D\text{-}1])$)
62     $next \leftarrow$ schedule$(next)$
       LBcast$(m[i])$
64
   Upon LBrecv$(j, r, v)$          $(*\ i \ne j\ *)$
66   enq$(m[i].v[j].q, \langle$lst$(v[j].q).s, r\rangle)$
     $m[j] \leftarrow v$
68   output $\langle j, \{\langle$sc$(j,k), k,$ ts(sc$(j, k), j, k)\rangle : k \in P\text{-}\{i, j\}\}\rangle$
```

Fig. 1. Secure and self-stabilizing native clock sampling algorithm (code for $p_i \in P$)

processor p_i stores the arrival time of the message (line 66) and the message itself (line 67). These data can be used for the reference broadcast in the upper layer. Once p_i receives a beacon from node p_k, node p_i scans $m[\,]$ for responses that refer to p_k's previous beacons.

The Correctness. We divide the correctness proof of the algorithm presented in Figure 1 into two parts. The first part relates to the task of random broadcast scheduling and the second relates to the task of beacon and response aggregation. The second part's proofs simply verify that the pseudo-code aggregates the right responses with the right beacon. Due to space limits, some parts of the correctness proof of the random broadcast scheduling, and the correctness proof of the aggregation task appears in [14].

We analyze our random broadcasting strategy as a ball throwing game in a team of n players that throw balls into bins. The bins represent the timeslots. For the sake of simplicity, we consider every timestamp as a single information unit, which we call a ball. The players' coordination is poor and resembles the clocks' partial synchrony. We measure the team performance by looking at the number, $\Gamma(n)$, of bins it takes the team to get each at least n balls into bins. A detailed game description and the correctness proof of corollary 1 appears in [14].

Corollary 1. *$\Gamma(n) \in \Omega(n^2)$ and the random broadcasting strategy of the algorithm presented in Figure 1 secures with probability $1 - 2^{-\ell}$ that $\Gamma(n) \in O(n^2(\log n)^3)$.*

Let E be an execution and $a_i \in E$ an atomic step in which processor p_i broadcasts. Let $c \in E$, be the configuration that immediately follows a_i. We define the first *round* from a_i, $E_{a_i}(1)$ as a (finite) subsequence of E that starts in c and ends in the atomic step $a'_i \in E$, that is the first step after a_i in which p_i broadcasts. We define the second round $E_{a_i}(2) = E_{a'_i}(1)$. Similarly, the *x-th* round $E_{a_i}(x)$, $\forall x > 1, x \in \mathbb{N}$, is defined as $E_{a_i}(x) = E_{a'_i}(x-1)$. We say that processor p_j *skips* a round $E_{a_i}(1)$ if p_i does not receive a broadcast from p_j in $E_{a_i}(1)$. The *beacon broadcast period* (BBP) of processor p_i for a given broadcast in atomic step $a_i \in E$ is the real time length of the round $E_{a_i}(1)$.

Definition 1. *We define the set LE_{rbs} of legal executions with respect to the task of random broadcast scheduling, such that it includes every execution E in which: (1) The expected* beacon broadcast period *(BBP) of processor p_i is within $[Du/\rho_{\max}, Du/\rho_{\min}]$ and (2) The probability that no processor skips ξ consecutive rounds $E_{a_i}(x), \ldots E_{a_i}(x+\xi)$ is in $O(1-2^{-\ell})$, where $x \in \mathbb{N}$ and $a_i \in E$, is an atomic step in which p_i broadcasts).*

Let E be a fair execution of the algorithm presented in Figure 1 and $c \in E$ a configuration in which $\alpha_i = (\mathsf{leq}(next_i - 2Du, cT_i) \wedge \mathsf{leq}(cT_i, next_i)$ holds. We say that c is safe with respect to LE_{rbs}.

We show that cT_i follows the native clock.

Lemma 1. *Let E be a fair execution of the algorithm presented in Figure 1, and c a configuration that is at least u after the starting configuration. Then, it holds that $(\mathsf{leq}(C^i - u, cT_i - w) \wedge \mathsf{leq}(cT_i - w, C^i))$ in c.*

Proof. Since E is fair, the do-forever loop's timer goes off in every period of $u/2$. Hence, within a period of u, processor p_i performs a complete iteration of the do-forever loop in an atomic step a_i.

Suppose that c immediately follows a_i. According to line 53, the value of $cT_i - w$ is the value of C^i in c. Let $t = cT_i - w = C^i$. It is easy to see that $\mathsf{leq}(t-u, t) \wedge \mathsf{leq}(t, t)$ in c.

Let a_i^r be an atomic step that includes the execution of lines 66 to 68, follows c, and immediately precedes $c' \in E$. Let $t' = C^i$ in c'. Then, within a period of at most $u/2$, processor p_i executes step $a'_i \in E$, which includes a complete iteration of the do-forever loop. Since the period between a_i and a'_i is at most $u/2$, we have that $t' - t < u/2$. ∎

We show that starting from an arbitrary configuration a fair execution researches a safe configuration.

Lemma 2. *Let E be a fair execution of the algorithm presented in Figure 1. Then, within a period of u, a safe configuration is reached.*

Proof. Let p_i be a processor for which α_i does not hold in the starting configuration of E. We show that within the first complete iteration of lines 53 to 63, the predicate α_i holds. According to Lemma 1, all processors, p_i, complete at least one iteration of lines 53 to 63, within a period of u.

Let $a_i \in E$ be the first step in which processor p_i completes the first iteration. If α_i does not hold in the configuration that immediately precedes a_i, then the predicate in line 54 holds and processor p_i executes line 55.

Immediately after the execution of line 55, the predicate $\neg(\mathit{leq}(next_i - 2Du, cT_i) \wedge \mathit{leq}(cT_i, next_i))$ does not hold, because $\neg(\mathit{leq}(t-2Du, t) \wedge \mathit{leq}(t, t))$ is false for any t. Moreover, the predicate in line 58 holds, since $\mathit{leq}(t, t+u)$ holds for any t. Therefore, p_i executes lines 59 to 63.

Claim. Suppose that the predicate $\neg(\mathit{leq}(next_i - 2Du, cT_i) \wedge \mathit{leq}(cT_i, next_i))$ (line 54) does not hold and the predicate $\mathit{leq}(next_i, cT) \wedge \mathit{leq}(cT, next_i + u)$ (line 58) holds. If processor p_i executes lines 59 to 63, then α_i holds for the configuration that immediately follows.

Proof. Among the lines 59 to 63, only lines 61 to 62 can change the values of α_i. Let $t_1 = next_i$ immediately after line 58 and let $t_2 = next_i$ immediately after the execution of line 62. We denote by $A = t_2 - t_1$ the value that lines 61 to 62 adds to $next_i$, i.e., $A = (y + D - x)u$, where $0 \leq x, y \leq D - 1$. Note that x is the value of $cslot_i$ before line 61 and y is the value of $cslot_i$ after line 61.

Therefore, $A \in [u, (2D - 1)u]$. By the claim's assertion, we have that $\mathit{leq}(cT_i, t_1 + u)$ holds before line 61. Since $u \leq A$, it holds that $\mathit{leq}(cT_i, t_1 + A)$ and therefore $\mathit{leq}(cT_i, t_2)$ holds. Moreover, by the claim assertion we have that $\mathit{leq}(t_1, cT_i)$ holds. Since $A \leq (2D-1)u$, it holds that $A - 2Du \leq -u$. This implies that $\mathit{leq}(t_1 - 2Du + A, cT_i)$. Therefore $\mathit{leq}(t_2 - 2Du, cT_i)$ holds. ■

We show that a safe configuration follows the configuration of Definition 1.

Lemma 3. *Let E be a fair execution of the algorithm presented in Figure 1 that starts in a configuration c, in which α_i holds. Then, every configuration in E is safe with respect to LE_{rbs}.*

Proof. Let t_i be the value of p_i's native clock in configuration c and $a_i \in E$ is the first step of processor p_i. According to Lemma 1 and by the fairness of E, without loss of generality, we can assume that $C^i - t_i \bmod T \leq u/2$.

We show that α_i holds in configuration c' that immediately follows a_i. Lines 66 to 68 do not change the value of α_i. By the proof of Lemma 2, if a_i executes lines 59 to 63 within one complete iteration, then α_i holds in c'. Therefore, we look at step a_i that includes the execution of line 53 to 58, but does not include the execution of lines 59 to 63.

Let $t_1 = cT_i$ in c and $t_2 = cT_i$ in c'. We show that while a_i executes line 54, the predicate $\neg(\mathit{leq}(next_i - 2Du, cT_i) \wedge \mathit{leq}(cT_i, next_i))$ does not hold in a_i.

Let $A = next_i - Du$ and $B = next_i$ in c. The values of $next_i - Du$ and $B = next_i$ do not change in c'. Since α_i is true in c, it holds that $\mathit{leq}(A, t_1) \wedge \mathit{leq}(t_1, B)$. We claim that $\mathit{leq}(A, t_2) \wedge \mathit{leq}(t_2, B+u)$. Suppose, in a way of contradiction, that $\mathit{leq}(A, t_2) = \mathit{leq}(A, t_1 + u/2)$ does not hold. Then, $\mathit{leq}(next_i - Du, t_1 + u/2)$ does not hold in configuration c, which implies that $\mathit{leq}(t_1 - Du, t_1 + u/2)$ because $\mathit{leq}(t_1, next_i)$ hold in c. Hence, a contradiction.

Since $\mathit{leq}(t_1, B)$ in c, we have that $\mathit{leq}(t_2, B+u)$ while p_i execute line 54 in a_i. By the assumption that $t2 - t1 \bmod T < u$ we have that $\mathit{leq}(t_1 + u/2, B+u) \implies \mathit{leq}(t_2, B+u)$. ■

We show that every execution (for which the safe configuration requirements hold) is a legal execution with regard to the random broadcast scheduling task.

Lemma 4. *Let E be a fair execution of the algorithm presented in Figure 1, where all configurations in E are safe. Then, $E \in LE_{rbs}$.*

Proof. **(1)** Let $a_i \in E$ be a step in which processor p_i broadcasts and $a'_i \in E$ is the first step after a_i in which p_i broadcasts. Let $c_1 \in E$ immediately precede a_i and $c_2 \in E$ immediately follow a_i. Let $c_3 \in E$ immediately precede a'_i and $c_4 \in E$ immediately follow a'_i. Let $n_1 = next_i$ in c_1, $t_1 = cT_i$ in c_2, $n_2 = next_i$ in c_3, $t_2 = cT_i$ in c_4. The BBP can be expressed as B/ρ_i, where $B = t_2 - t_1 \bmod T$.

Processor p_i broadcasts in line 63 only when the predicate $\gamma_i = \mathit{leq}(next_i, cT_i) \wedge \mathit{leq}(cT_i, next_i + u)$ (line 58) holds. Claim 3 of Lemma 2 shows that in lines 61 to 62, $next_i$ is incremented (modulo T) by $A = (y + D - x)u$. Both integers x and y are chosen independently and from the same uniform distribution over $[0, D-1]$. Therefore, they have the same expected value. Therefore, the expected value of A is $\mathbf{E}((y + D - x)u) = (\mathbf{E}(y) + \mathbf{E}(D) - \mathbf{E}(x))u = (\mathbf{E}(D) + \mathbf{E}(y) - \mathbf{E}(y))u = \mathbf{E}(D)u = Du$.

Let $\hat{u}_1 = t_1 - n_1$ and $\hat{u}_2 = t_2 - n_2$. If we assume that $\hat{u}_1$ and $\hat{u}_2$ are independent and have the same distribution, the expected value of B is $\mathbf{E}((n_2 + \hat{u}_2) - (n_1 + \hat{u}_1)) = \mathbf{E}(n_2 - n_1) + \mathbf{E}(\hat{u}_2 - \hat{u}_1) = Du + \mathbf{E}(\hat{u}_2) - \mathbf{E}(\hat{u}_1) = Du$

Even if $\hat{u}_1$ and $\hat{u}_2$ are not independent and/or not from the same distribution, the expected value of B is Du as well, as the decrement of the BBP for a broadcast in a'_i within the period $[n_2, n_2 + u]$ implies a corresponding increment of the BBP for the broadcast in a_i. By the definition of $\rho_{\min}$ and $\rho_{\max}$ we have that $Du/\rho_{\max} \leq Du/\rho_i \leq Du/\rho_{\min}$ (since $\forall i : \rho_{\min} \leq \rho_i \leq \rho_{\max}$).

(2) Let a_i be a step in which in which processor p_i broadcasts, a'_i be the next step in which processor p_i broadcasts, and c be the configuration that immediately follows a_i.

Let r be the value of $next_i$ between lines 61 and 62 in a_i. The period of length Du that begins at r is divided in D slots of length u. A slot begins at time $r + xu$ and ends at time $r + (x+1)u$ for a unique integer $x \in [0, D-1]$. The slot in which a'_i broadcasts is *cslot* in c. By Corollary 1, the probability of no messages collides in the period r to $r + Du$ is in $O(1 - 2^{-\ell})$. ■

Performances. Several elements determine the precision of the clock synchronization. The clock sampling technique is one of them. Elson et al. [6] show that the reference broadcast technique can be more precise than the roundtrip synchronization technique. We allow the use of both techniques. Another important precision factor is the degree of the polynomial, $P_{i,j}(t)$, that approximates the native clock values of the neighboring processors p_i and p_j (see Römer et al. [16]). We consider any finite degree of the polynomial. Moreover, the clock synchronization precision improves, as neighboring processors are able to sample their clocks more frequently. However, due to the limited energy reserves in sensor networks, careful considerations are required.

The execution of a clock synchronization protocol can be classified between two extremes: *on-demand* and *continuous*. Nodes that wish to synchronize their clocks can invoke a distributed procedure for clock synchronization on-demand. The procedure terminates as soon as the nodes reach their target precision. An execution of a clock synchronization program is classified as continuous if no node ever stops invoking the clock synchronization procedure. Our generic design facilitates a trade-off between energy conservation (i.e., on-demand operation) and fine-grained clock synchronization (i.e., continuous operation). The trade-off allows budget policies to balance between application requirements and energy constraints.

Let us consider the continuous operation mode. The clock precision improves as the frequency of the beacons (and responses) that the correct processors are able send increases. Thus, the precision of $P_{i,j}(t)$ depends on $round(n)$, where $round(n)$ is the time it takes n processors to send n beacons and then to let n processors to respond to all n beacons. By Corollary 1, $round(n) \in \Omega(n^2)$ and $round(n) \in O(n^2(\log n)^3)$. Therefore, our design can secure clock precision that is $O((\log n)^3)$ times the optimum, with probability that is $1 - 2^{-\ell}$.

We note that the required storage is in $O(n^2 \log T)$. Moreover, existing sensor networks technology allows a message size of $14n+O(1)$ bytes. In [14], we explain how to further accommodate message size and to optimize performance.

4 Discussion

Sensor networks are particularly vulnerable to interference, whether as a result of hardware malfunction, environmental anomalies, or malicious intervention. When dealing with message collisions, message delays and noise, it is hard to separate malicious from non-malicious causes. For instance, it is hard to distinguish between a pulse delay attack from a combination of failures, e.g., a node that suffers from a hidden terminal failure, but receives an echo of a beacon. Recent studies consider more and more implementations that take security, failures and interference into account when protecting sensor networks (e.g., [11,5]). We note that many of the existing implementations assume the existence of a fined grained synchronized clock, which we implement.

Ganeriwal et al. [9,10] overcome the challenge of delayed beacons using the round-trip synchronization technique, and the Byzantine agreement protocol

[13]. Thus, Ganeriwal et al. requires $3t + 1 \leq n$. Song et al.'s [19] consider a different approach that uses the reference broadcasting synchronization technique. Existing statistics models refer to malicious time offsets as outliers. The statistical outlier approach is numerically stabile for $2t + \epsilon \leq n \leq 3t + 1$, where ϵ is a safety constant (see [19]). We note that both approaches are applicable to our work. However, based on our practical assumptions, we are able to avoid the Byzantine agreement overheads and follow the approach of Song et al. [19]. They assume the existence of a distributed algorithm for sending beacons and collecting their responses. This work presents the first design of that algorithm.

The generalized extreme studentized deviate (GESD) algorithm [17] can be used to detect outliers. We note that there exists self-stabilizing version of Song et al.'s [19] strategy. Let B be the set of "recently" delivered beacon records. By "recently", we mean that within a predefined period, $\varrho \in O(D)$, the node removes old records from B, where ϱ depends on ξ, i.e., the number of broadcasts it takes to assure message delivery. The algorithm tests set B for outliers.

Existing implementations of secure clock synchronization protocols [22,21,9,8,15,10,19] are not self-stabilizing. Thus, their specifications are not compatible with security requirements for autonomous systems. In autonomous systems, the self-stabilization design criteria are imperative for secure clock synchronization. For example, many existing implementations require initial clock synchronization prior to the first pulse-delay attack (during the protocol set up). This assumption implies that the system uses global restart for self-defense management, say, using an external intervention. We note that the adversary is capable of intercepting messages continually. Thus, the adversary can risk detection and intercept all pulses for a long period. Assume that the system detects the adversary's location and stops it. Nevertheless, the system cannot synchronize its clocks without a global restart.

Sun et al. [20] describe a cluster-wise synchronization algorithm that is based on synchronous rounds. The authors assume that a Byzantine agreement algorithm [13] synchronizes the clocks before the system executes the algorithm. Our algorithm is comparable with the requirements of autonomous systems and makes no assumptions on synchronous rounds or start.

Manzo et al. [15] describe several possible attacks on an (unsecured) clock synchronization algorithm and suggest counter measures. For single hop synchronization, the authors suggest using a randomly selected "core" of nodes to minimize the effect of captured nodes. The authors do not consider the cases in which the adversary captures nodes after the core selection. In this work, we make no assumption regarding the distribution of the captured nodes. Farrugia and Simon [8] consider a cross-network spanning tree in which the clock values propagate for global clock synchronization. However, no pulse-delay attacks are considered. Sun et al. [21] investigate how to use multiple clocks from external source nodes (e.g., base stations) to increase the resilience against an attack that compromises source nodes. In this work, there are no source nodes.

In [22], the authors explain how to implement a secure clock synchronization protocol. Although the protocol is not self-stabilizing, we believe that some of their security primitives could be used in a self-stabilizing manner when implementing our self-stabilizing algorithm.

Herman and Zhang [12] present a self-stabilizing clock synchronization algorithm for sensor networks. The authors present a model for proving the correctness of synchronization algorithms and show that the converge-to-max approach is stabilizing. However, the converge-to-max approach is prone to attacks with a single captured node that introduces the maximal clock value whenever the adversary decides to attack. Thus, the adversary can at once set the clock values “far into the future”, preventing the nodes from implementing a continuous time approximation function. This work is the first in the context of self-stabilization to provide security solutions for clock synchronization in sensor networks.

Conclusions. Designing secure and self-stabilizing infrastructure for sensor networks narrows the gap between traditional networks and sensor networks by simplifying the design of future systems. In this work, we consider realistic system settings and take a clean slate approach in designing a fundamental component; a clock synchronization protocol.

The designers of sensor networks often implement clock synchronization protocols that assume the system settings of traditional networks. However, sensor networks often require fine-grained clock synchronization for which the traditional protocols are inappropriate, e.g., the nonce techniques cannot resist pulse-delay attacks.

Alternatively, when the designers do not assume traditional system settings, they turn to reinforce the protocols with masking techniques. Thus, the designers assume that the adversary never violates the assumptions of the masking techniques, e.g., there are at most t captured nodes at all times, where $3t + 1 \leq n$. Since sensor networks reside in an unattended environment, the last assumption is unrealistic.

Our design promotes self-defense capabilities once the system returns to follow the original designer’s assumptions. Interestingly, the self-stabilization design criteria provide an elegant way for designing secure autonomous systems.

Acknowledgments. This work would not have been possible without the contribution of Marina Papatriantafilou in many helpful discussions, ideas, and analysis. Many thanks to Edna Oxman for improving the presentation.

References

1. Abramson, N., et al.: The Aloha System. Univ. of Hawaii (1972)
2. Demirbas, M., Arora, A., Nolte, T., Lynch, N.A.: A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 299–315. Springer, Heidelberg (2005)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
4. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)

5. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Gossiping in a Multi-Channel Radio Network, An Oblivious Approach to Coping with Malicious Interference. In: DISC 2007. LNCS, vol. 4731, pp. 130–145. Springer, Heidelberg (2007)
6. Elson, J., Girod, L., Estrin, D.: Fine-grained network time synchronization using reference broadcasts. Operating Systems Review (ACM SIGOPS) 36(SI), 147–163 (2002)
7. Elson, J., Karp, R.M., Papadimitriou, C.H., Shenker, S.: Global synchronization in sensornets. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 609–624. Springer, Heidelberg (2004)
8. Farrugia, E., Simon, R.: An efficient and secure protocol for sensor network time synchronization. J. Syst. Softw. 79(2), 147–162 (2006)
9. Ganeriwal, S., Capkun, S., Han, C.-C., Srivastava, M.B.: Secure time synchronization service for sensor networks. In: Ngu, A.H.H., Kitsuregawa, M., Neuhold, E.J., Chung, J.Y., Sheng, Q.Z. (eds.) WISE 2005. LNCS, vol. 3806, pp. 97–106. Springer, Heidelberg (2005)
10. Ganeriwal, S., Capkun, S., Srivastava, M.B.: Secure time synchronization in sensor networks. ACM Transactions on Information and Systems Security (March 2006)
11. Gilbert, S., Guerraoui, R., Newport, C.C.: Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 215–229. Springer, Heidelberg (2006)
12. Herman, T., Zhang, C.: Best paper: Stabilizing clock synchronization for wireless sensor networks. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 335–349. Springer, Heidelberg (2006)
13. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
14. Larsson, A., Schiller, E.M., Tsigas, P.: Secure and fault-tolerant clock synchronization in sensor networks. TR 2006:16, Computer Science and Engineering, Chalmers University of technology (September 2006)
15. Manzo, M., Roosta, T., Sastry, S.: Time synchronization attacks in sensor networks. In: SASN 2005. Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks, pp. 107–116. ACM Press, New York (2005)
16. Römer, K., Blum, P., Meier, L.: Time synchronization and calibration in wireless sensor networks. In: Stojmenovic, I. (ed.) Handbook of Sensor Networks: Algorithms and Architectures, pp. 199–237. John Wiley and Sons, Chichester (2005)
17. Rosner, B.: Percentage points for a generalized *esd* many-outlier procedure. Technometrics 25, 165–172 (1983)
18. Schneier, B.: Applied Cryptography, 2nd edn. John Wiley & Sons, Chichester (1996)
19. Song, H., Zhu, S., Cao, G.: Attack-resilient time synchronization for wireless sensor networks. Ad Hoc Networks 5(1), 112–125 (2007)
20. Sun, K., Ning, P., Wang, C.: Fault-tolerant cluster-wise clock synchronization for wireless sensor networks. IEEE Transactions on Dependable and Secure Computing 2(3), 177–189 (2005)
21. Sun, K., Ning, P., Wang, C.: Secure and resilient clock synchronization in wireless sensor networks. IEEE Journal on Selected Areas in Communications 24(2), 395–408 (2006)
22. Sun, K., Ning, P., Wang, C.: Tinysersync: secure and resilient time synchronization in wireless sensor networks. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) ACM Conference on Computer and Communications Security, pp. 264–277. ACM Press, New York (2006)

On the Probabilistic Omission Adversary

Taisuke Izumi and Koichi Wada

Nagoya Institute of Technology
Gokiso-cho, Showa-ku, Nagoya, Aichi 466-8555, Japan
{t-izumi, wada}@nitech.ac.jp

Abstract. This paper newly proposes a novel round-based synchronous system suffering crash and probabilistic omission failures. In this model, a novel class of adversaries, called *p-probabilistic omission adversary* (*p*-POA) is introduced. In addition to the ability of complete control of the crash-failure behavior, p-POA can select any subset of all transmitted messages as *omission candidates*. Then, each message in the omission candidates is lost with probability p. This paper investigates the feasiblity and complexity of the consensus problem under p-POA. We first show two impossibility results that (1) for any $p > 0$, there exists no uniform consensus algorithm tolerating more than or equal to $n/2$ crash failures, and that (2) for any $p > 0$, any uniform consensus algorithm cannot halt. We also show two consensus algorithms CPO and F-CPO. Both algorithms work under (1/2)-POA and respectively have distinct advantages. The algorithm CPO can tolerate at most $n/2 - 1$ crash failures and achieves $O(f)$ expected round complexity, where f is the actual number of crash failures. This implies that CPO has maximum crash-failure resiliency. While the second algorithm F-CPO assumes the maximum number of crash failures less than $n/3$, it achieves $f + O(1)$ round compexity in expectation. Since it is known that the lower bound for crash-tolerant consensus is $f + 1$, this result implies that only a constant number of extra rounds is nessesary to tolerate a drastic number of message omissions.

1 Introduction

Consensus problem is one of fundamental and important problems for designing fault-tolerant distributed systems. In the consensus problem, each process proposes a value, and has to agree on a common value that is proposed by a process unless it crashes. The consensus problem has many practical applications, e.g., atomic broadcast [3,10], shared object [1,11], weak atomic commitment [9] and so on. However, despite of such applications, it has no deterministic solution in asynchronous systems subject to only a single crash failure [7]. Thus, several consensus algorithms have been considered on systems with some additonal assumptions [5,6,16,3]. Especially, the *round-based synchrony* is one of the most commonly used assumptions for designing consensus algorithms[16,17]. Executions of the round-based synchronous systems proceed along consecutive communication-closed rounds, where "communication-closed" means that every message sent in a round is received in the same round.

In literatures about the synchronous consensus, the failure behavior other than crash is also considered. The *omission failure* is one of such failures, which allows a part of communication messages to be lost in transmission. Generally, for omission failures, two kinds of modelings have been considered. The most traditional one is the component failure model, which statically specifies the set of faulty components (i.e., faulty edges and/or faulty processes) and only messages related to faulty components can be omitted. Since the component failure model is a natural extension of crash-failure models, it is widely accepted in many studies [2,4,8,14,15,16]. However, in practical sense, it is not necessarily the best model: In real systems, the omission failures can occur on any component, but their occurrences are not permanent. Thus, the omission failures should be modeled as a *transient* and *ubiquitous* one, which is the contrary of the component failure model. The second model, called *dynamic failure model*, is one that absorbs these features [13,18]. While the component failure model restricts its failure behavior by specifying faulty components, the dynamic failure model bounds the behavior by the number of failures. More precisely, the dynamic failure model only assumes the existence of upper bound for the number of omission that can occur in one round. This model is more plausible than the component failure model in a certain type of real systems. Nevertheless, both the component and dynamic failure models leave the disadvantage that the failure influcence is too strong: Only a small fraction of failures makes a number of uniform tasks unsolvable. For example, in a complete network of n processes, if $2n-1$ omission failures are possible, the system can suffer the isolation of a non-crashed process, i.e., all messages from/to the process are lost by omission failures. Then, uniform solution of the consensus problem is clearly impossible. This disadvantage derives from the fact that the *adversary*, which is a daemon determining the failure behavior of system executions, can deterministically control the targets where failures appear.

In this paper, as the model to circumvent the above disadvantage, we newly introduce a round-based synchronous system suffering *crash* and *probabilistic omission failures*, and consider consensus algorithms on this system. Specifically, our model introduces the novel class of adversaries, called *p-probabilistic omission adversary* (p-POA). The p-POA is a generalization of the traditional crash-failure adversary, and thus it has all ability that the crash-failure adversary has. In addition, at each round, p-POA can select any subset of all transmitted messages (even the set consisting of all messages is possible) as *omission candidates*. Each message in the omission candidate is lost with probability p, where p is a system parameter of the adversary. Because of its probabilistic nature, the POA does not cause the isolation of a part of processes, and thus it can handle more drastic number of message omissions than existing omission failure models.

This paper investigates the feasibility of the consensus problem under the p-POA. Since communications liveness between any pair of processes is probabilistically guaranteed in our model, it seems not so surpising that the uniform consensus can be solved in our model (actually, we propose two uniform consensus algorithms under p-POA in this paper). However, it is quite nontrivial how probabilistic omissions affect the complexity of the consensus problem, which is the main interst of our study. In the followings, we summarize the contribution of this paper:

- We show two fundamental impossiblity result about the consensus problem under p-POA. The first result is that for any $p > 0$, there is no uniform consensus algorithm that can tolerate more than or equal to $n/2$ crash failures. The second one is that for any $p > 0$, there is no *halting* algorithm to solve the uniform consensus, i.e., in any consensus algorithm under p-POA, each process must continue to execute the algorithm even after it decides.
- We propose a consensus algorithm CPO working correctly under the (1/2)-POA. This algorithm assumes that the maximum number of failures t is less than $n/2$, which implies the maximum failure resiliency. The algorithm has $O(f)$ expected round complexity, where f is the actual number of crash failures. In the sense that the round complexity depends only the actual number of crash failures, this algorithm is *early-decideing*.
- We also propose another early-deciding consensus algorithm F-CPO working correctly under the (1/2)-POA. This algorithm requires the stronger assumption of $t < n/3$ than CPO, but has $f + O(1)$ round complexity in expectation. Interestingly, since it is known that the lower bound for crash-tolerant consensus is $f + 1$, this result implies that only a constant number of extra rounds is nessesary to tolerate a drastic number of omission (about a half of all messages).

The paper is organized as follows: In Section 2, we introduce the system model, and the definition of the consensus problem. Section 3 shows two impossibility results. Sections 4 and 5 provides the algoirhm CPO and F-CPO, respectively. Finally we conclude this paper in Section 6.

2 Preliminaries

2.1 Distributed Systems

We consider a distributed system with round-based synchrony. The distributed system consists of n processes $\mathcal{P} = \{p_0, p_1, p_2, \cdots, p_{n-1}\}$ that are completely connected by communication links, that is, any pair of processes can communicate with each other by directly exchanging messages. The system is round-based, that is, its execution is a sequence of synchronized *rounds* identified by $0, 1, 2,$ etc. Each round r consists of three steps:

Send step. Each process p_i sends messages.
Receive step. Each process p_i receives all the messages sent to p_i at the send step of the current round.
Local processing step. Each process p_i executes local computation.

2.2 Failure Models

An *adversary* is a daemon determining the behaivior of failures at each round. Notice that the adversary always makes a decision by worst possible cases. In this paper, to handle both crash and probabilistic omission, we introduce a novel class of adversaries, called *p-Probabilistic Omission Adversaries*(p-POA). Same as the traditional crash adversary, it can completely control the behavior of process crashes. If a process p_i is designated as crash during round r, it makes no operation subsequently. Then, the messages sent by p_i at round r may or may not be received, which is also controlled by the adversary. In addition to the ability of crash adversaries, p-POA can omit each in-trasmission message with probability p, where p is a system-specific parameter. More precisely, it determines *omission candidates* C_r at each round r, which is an arbitrary subset of all messages sent at round r. Every message $m \notin C_r$ is guaranteed to be transmitted correctly. On the other hand, each message $m \in C_r$ is lost with probability p. As a summary of the above model, the execution of one round r in our model can be described as follows:

1. The sending step is executed. Then, for each non-crashed process p_i, the adversary determines whether p_i crashes at the current round or not. In addition, if a process p_i is determined as crash, the adversary also determines its crash behavior (that is, determines which messages that will be sent by p_i at the current round are actually sent).
2. The adversary decides omission candidates C_r. Each message in C_r is lost with probability p.
3. The receiving step is executed. All remaining messages are correctly received.
4. The local processing step is executed.

It should be noted that the above failure model is a weaker model of the traditional crash-failure models: The p-probabilistic omission adversary can determinitically control the execution of the system so that it will behave as the traditional crash-fauilure systems. This implies that any complexity lower bounds for the crash-failure model also hold in our model.

There is an upper bound t on the number of processes that can crash. Throughout this paper, we assume $t < n/2$. This assumption is necessary to exist a consensus algorithm on our model (the necessary proof is given in Section 3). The actual number of crash processes is denoted by f $(\leq t)$. A process is said to be *correct* if it never crashes, and a round r is said to be *correct* when no process crashes during the round r.

2.3 Consensus Problem

In a consensus algorithm, each correct process initially proposes a value, and eventually chooses a decision value from the values proposed by processes so that all processes decide the same value. The *uniform consensus* is a stronger variant of the consensus. It disallows faulty processes to disagree on the decided value. The standard specification of uniform consensus is described as follows:

Termination. Every correct process eventually decides.
Uniform Agreement. No two processes decide different values.
Validity. If a process decides a value v, then, v is a value proposed by a process.

Since we consider the probabilistic omission, it is clear that there is no algorithm satisfying the above termination property in our model. For example, the execution where all messages are omitted is a possible execution in our model. In such execution, no algorithm can correctly reach decisions. Thus, we have to relax the specification of the consensus problem such that it allows a probabilistic guarantee of Termination property. In this paper, we define the consensus problem by the uniform agreement property, the validity property, and the following probabilistic termination property:

Probabilistic Termination. Every correct process decides with probability 1.

Notice that decision does not necessarily implies the *halt* of the algorithm. Even after a process decides, it may work for helping the decision of other processes.

3 Impossibility Results

In this section, we present two fundamental impossiblity results about the consensus problem on the system with the p-probabilistic omission adversary (for lack of space, we omit the proofs).

Theorem 1. *For any $p > 0$, there is no algorithm solving the consensus problem under the p-probabilistic omission adversary if $t \geq n/2$ holds.*

Theorem 2. *For any $p > 0$ and $t > 0$, there is no* halting *consensus algorithm under the p-probabilistic omission adversary.*

4 The $O(f)$-Round Algorithm for $t < n/2$

This section provides a consensus algorithm under the (1/2)-probabilistic omission adversary, which can tolerate less than $n/2$ crash processes. Because of Theorem 1, this algorithm achieves the maximum crash-failure resilience. The basic idea of our algorithm derives from Chandra and Toueg's algorithm using eventually strong failure detectors [3]. Before the presentation of our algorithm, we first discuss about the round complexity of a group communication primitive under the (1/2)-probabilistic omission adversary in the following subsection, which is used to construct our consensus algorithm.

4.1 The Round Complexity of All-to-All Broadcast

In our adversary model, there is no guarantee that a message sent by a correct process is necessarily received. Thus, if a process wants to broadcast an information d, one-time broadcasts bring only a slight possibility that all processes can corerctly receive d. Thus, this subsection investigates how many rounds are

```
Algorithm 2RB($d$): Code for $p_i$

1:  variable:
2:      $M_i$ : init $\perp$

3:  At round 1:
4:      if $p_i$ is the source process then
5:          Send$_i(d)$ to all processes
6:          $M_i \leftarrow d$
7:      endif
8:  At round 2:
9:      if $M_i \neq \perp$ then
10:         Send$_i(d)$ to all processes
11:     endif
12:     if a message $d$ is received from $p_j$ then
13:         $M_i \leftarrow d$
14:     endif
```

Fig. 1. Algorithm 2RB(d): 2-Round Broadcast Algorithm for Transmission Data d

sufficient to guarantee that all processes receive the information d with high probability.

First, we introduce an broadcast algorithm called *2-round Broadcast*(2RB). The pseudocode description of the 2RB algorithm for each process p_i is given in Figure 1. The variable d is the broadcast information, and the local variable M_i is the buffer of received messages. In the pseudocode, $M_i = d$ implies that the process p_i received the broadcast information d. The principle of 2RB is quite simple: The 2RB algorithm consists of 2 rounds. At the first round, the source process broadcasts the information d to all processes. At round 2, all processes that received d in previous rounds broadcast the information d.

The following lemma shows that the 2RB correctly broadcasts messages with high probability if no crash occurs during its execution.

Lemma 1. *Let p_i be the process that broadcasts an information using 2RB, and n' be the number of non-crashed processes at the beginnning of 2RB. If no process crashes during the execution of 2RB, all non-crashed processes receive the information d with probabilty $1 - O(n'/\alpha^{n'})$, where α is a constant more than one.*

Proof . *Let X_1 and X_2 be the random variables that represent the number of processes receiving d by the end of rounds one and two, respectively. Since each process receives d with the probability more than $1/2$ at round one, we have $E[X_1] \geq n'/2$. Using the Chernoff Bound, we can obtain the following inequality;*

$$\Pr\left[X_1 < n'/4\right] \leq e^{-\frac{n'}{2}\cdot\frac{1}{8}} \leq e^{-\frac{n'}{16}}. \tag{1}$$

This implies that at the end of round one, at least $n/4$ processes receive d with high probability. Next, under the assumption of $X_1 \geq n'/4$, we bound the probability of $X_2 = n'$.

$$\begin{aligned}\Pr[X_2 = n' | X_1 \geq n'/4] \\ &\geq \left(1 - \left(\frac{1}{2}\right)^{n'/4}\right)^{3n'/4} \\ &\geq 1 - \frac{3n'}{4}\left(\frac{1}{2}\right)^{n'/4} \\ &\geq 1 - \frac{3n'}{2^{n'/4+2}}. \end{aligned} \tag{2}$$

From the inequalities 1 and 2, we obtain $\Pr[X_2 = n'] \geq (1 - O(n'/\alpha^{n'}))$ *for some* $\alpha > 1$. □

Let us consider the case where all n' non-crashed processes concurrently broadcast informations using 2RB. Let $d_i (0 \leq i \leq n' - 1)$ be the information that is broadcast by p_i. Ths following lemma gives the bound for the probability that all processes receive all informations $d_0, d_1, \cdots d_{n'-1}$.

Lemma 2. *Let each process p_i $(0 \leq i \leq n' - 1)$ broadcasts an information d_i using* 2RB. *If no process crashes during the execution of* 2RB, *all non-crashed processes receive all informations $d_0, d_1, \cdots d_{n'-1}$ with probabilty $1 - O({n'}^2/\alpha^{n'})$, where α is a constant more than one.*

Proof . *Let E_i be the event that all processes receives d_i, and $\bar{E}_i$ be the complement of E_i. From Lemma 1,* $\Pr[\bar{E}_i] \leq O(n'/\alpha^{n'})$ *holds. Then, using the union bound, we obtain*

$$\begin{aligned}\Pr\left[\bigcup_{i=1}^{n'-1} \bar{E}_i\right] &\leq \sum_{i=1}^{n'-1} \Pr[\bar{E}_i] \\ &\leq n' \cdot O(n'/\alpha^{n'}) \leq O({n'}^2/\alpha^{n'}). \end{aligned} \tag{3}$$

Therefore, $\Pr[\cap_{i=1}^{n'-1} E_i] \geq 1 - O({n'}^2/\alpha^{n'})$ *holds. The lemma is proved.* □

4.2 Algorithm CPO

Algorithm Design. Using the 2RB primitive, we present an algorithm CPO that solves the consensus problem under the (1/2)-probabilistic omission adversary. Figure 2 is a pseudocode description of the algorithm. The algorithm CPO is designed using the *rotating coordinator* paradigm: An execution of CPO is divided into consecutive *cycles* that are identified by $0, 1, 2, \ldots$. A coordinator is assigned to each cycle. In the k-th cycle of the execution, process $p_{(k \bmod n-1)}$ works as the coordinator. In the algorithm CPO, each process maintains an candidate of decision values. In Figure 2, the candidate value of p_i is stored in variable e_i.

At initial configurations, the value of candidate variable e_i is p_i's proposal v_i. Each candidate variable e_i has a timestamp t_i, which stores the cycle number when the last update of e_i occurs. In each cycle, the coordinator tries to make all processes reach agreement by imposing its own candidate value to all other processes. One cycle consists of two phases, and one phase consists of two rounds. The first phase of each cycle is called *impose phase*, and the second one is called *commit-aggregation phase*. In impose phases, only the coordinator broadcasts e_i using 2RB to tell its candidate value to all other processes. If a process p_i receives the message with the candidate value e from the coordinator, it stores e to its candidate variable e_i, and also updates its timestamp t_i with the current cycle number. In commit-aggregation phases, each process p_i broadcasts the information (e_i, t_i) using 2RB. Commit-aggregation phases have two role: The first role is that each process checks whether more than $n/2$ processes have the same candidate value as one owned by itself or not. If a process p_i passes the check, it decides its candidate value. The second role is that the coordinator of the next cycle aggregates candidate values. If the process $p_{(r \bmod n-1)}$ aggregates more than $n/2$ candidate values at the commit-aggregation phase of cycle $r-1$, it updates its candidate value with one having the latest timestamp of all received candidates. At the next cycle, $p_{(r \bmod n-1)}$ can actually work as the coordinator only if it receives more than $n/2$ candidate values. Otherwise, it will be silent at the next cycle. In Figure 2, the variable act_i is the flag whether the coordinator p_i works or not.

Correctness. We prove the correctness of the algorithm CPO. We first introduce several notations used in the following proofs: Let $cod(c)$ be the identifier of the coordinator of cycle c (i.e., $cod(c) = (c \bmod n - 1)$. We define act_i^c, e_i^c and t_i^c as the values of act_i, e_i, and t_i at the beginning of a cycle c, respectively. We say that "*the coordinator of c is active*" if $p_{cod(c)}$ does not crash at the beginning of c and $act_{cod(c)}^c$ = TRUE holds. We say "*a cycle c is valid*" if no crash occurs during c and the coordinator of c is active. The cycle that is not valid is called *invalid*. For lack of space, several proofs are omitted (Lemma 4 and 6).

Lemma 3. *Letting p_i be the process that decides at cycle c. the decision value of p_i is $e_{cod(c)}^c$.*

Proof . *Let d be the decision of p_i. Since p_i decides at the cycle c, it receives the message (d, c) from the coordinator of cycle c. This implies $e_{cod(c)}^c$.* □

Lemma 4. *Let p_i be the process that decides a value d at cycle c. Then, for a process p_j and any cycle c' $(c' \geq c)$, if $t_j^{c'} \geq c$, then $e_j^{c'} = d$.*

Lemma 5 (Uniform Agreement). *No process decides differently.*

Proof . *Let p_i be the process that decides first (if two or more processes decide at a same round, arbitrary one of them is chosen), d be the decision value of p_i, and c be the cycle when p_i decides. Then, we prove that a process $p_j (\neq p_i)$ also decides d. From Lemma 4, any cycle $c' (\geq c)$ whose coordinator is active,*

```
Algorithm CPO: Code for $p_i$

1: variable:
2:     $e_i$ : init $v_i$ /* The $v_i$ is the $p_i$'s proposal */
3:     $t_i$ : init $-1$
4:     $act_i$ : init FALSE

5: for each cycle $c = 0, 1, 2, \cdots$ do :
6:     Impose Phase:
7:     if $(c \bmod n-1) = i$ and $act_i$ = TRUE then 2RB($e_i$) endif
8:     if a message is received from $p_{c \bmod n-1}$ then
9:         let $e$ be the message from the current coordinator
10:        $e_i \leftarrow e$; $t_i \leftarrow c$
11:    Commit-aggregation Phase:
12:    2RB($e_i, t_i$)
13:    if messages are received from more than $n/2$ processes then
14:        $act_i \leftarrow$ TRUE
15:        if $(c \bmod n-1) = i$ then
16:            let $(e', t')$ be the message with the latest timestamp
17:            $e_i \leftarrow e'$; $t_i \leftarrow c$
18:        endif
19:        if $\exists w : (w, c)$ is received more than $n/2$ times then
20:            $e_i \leftarrow w$; $t_i \leftarrow c$
21:            decide($e_i$)
22:        endif
23:    else $act_i \leftarrow$ FALSE endif
24: endfor
```

Fig. 2. Algorithm CPO: A consensus algorithm under (1/2)-POA

$e^{c'}_{cod(c')} = d$ holds. Since each process can decide at cycle x only if x's coordinator is active, at the cycle y when p_j decides, y's coordinator has the candidate value d at the beginning of y. From Lemma 3, this implies that p_j decides d. The lemma holds. □

Lemma 6 (Validity). *If a process decides a value d, then, d is a value proposed by a process.*

Lemma 7. *If cycles c and $c+1$ are valid, all processes decide by the end of $c+1$ with probability $1 - O(n^2/\alpha^n)$ (α is a constant).*

Proof . *Clearly, all processes decide at the end of $c+1$ if the following three events occur:*

- *The coordinator of $c+1$ becomes active, i.e., in the commit-aggregation phase of c, the coordinator of $c+1$ receives messages from more than $n/2$ processes.*
- *In the impose phase of $c+1$, all non-crashed processes receive the message from the coordinator.*

– *In the commit-aggregation phase of $c+1$, all non-crashed processes receive more than $n/2$ messages.*

Since c and $c+1$ are valid, from Lemma 2, each of the above three events occurs with a probability higher than $1-O(n^2/\alpha^n)$. Thus, all processes decide at the end of $c+1$ with proabability $(1-O(n^2/\alpha^n))^3 = 1-O(n^2/\alpha^n)$. □

Lemma 8 (Probabilistic Termination). *Every correct process decides with probability 1. In addition, the worst-case expected number of rounds until all correct processes decide is O(f).*

Proof . *We call two consecutive valid cycles a* block*. Two blocks are said to be* independent *if there is no cycle belonging to both blocks. Since at most f processes can crash, for any $k > 0$, the execution consisting of $4f+2k$ cycles includes k blocks independent with each other (notice that if a process p_j crashes during cycle c, it does not only makes the cycle c invalid, but also makes the cycle where p_j is coordinator invalid, i.e., one crash failure may make two cycles invalid). From Lemma 7, In every block, all non-crashed processes decide with probability $1-O(n^2/\alpha^n)$. Then, the probability that there exists the process that does not decide at the end of the cycle $4f+2k$ is $(O(n^2/\alpha^n))^k$, which converge to zero. Thus, every correct process decides with probability 1. In addition, The worst case expected number of cycles until all correct processes decide is $\sum_{k=1}^{\infty}(4f+2k)\cdot(1-O(n^2/\alpha^n))\cdot O(n^2/\alpha^n)^{k-1} = O(f)$. Since one cycle consists of four rounds, we proved the $O(f)$ worst-case expected round complexity.* □

5 The $(f+O(1))$-Round Algorithm for $t < n/3$

In this section, we show that a faster algorithm F-CPO, which achives $f+O(1)$ time complexity under the assumption of $t < n/3$.

Algorithm Design. We present an algorithm F-CPO in Figure 3. The algorithm F-CPO is based on the floodset algorithm, which is a traditonal crash-tolerant consensus algorithm [12]. Same as CPO, the algorithm F-CPO maintains candidate variables at each process, and its execution is divided into consecutive cycles that are identified by $1, 2, \ldots$. However, unlike CPO, one cycle of F-CPO consists of two rounds. In each cycle, all processes exchange their candidate values using 2RB primitive. At the end of each cycle, each process receiving sufficiently many number of messages updates its candidate variable by voting. More precicely, if a process p_i receives more than $2n/3$ candidate values, it updates its candidate variable with one that is received by p_i most frequently (if the frequencies of two or more values are same, the maximum one is choosen). In addition, if all received messages have a same value d, p_i decides d (but does not halt).

In the original floodset algorithm, one correct round (i.e., the round where no crash occurs) is sufficient to guarantee the termination because all non-crashed processes have a same candidate value after the correct round, and thus they decide at the following round. In contrast, since message omission is considered in our model, no-crash-failure round does not guarantee that all candidate values

```
Algorithm F-CPO: Code for $p_i$

1: variable:
2:     $e_i$ : init $v_i$ /* The $v_i$ is the $p_i$'s proposal */

3: for each cycle $c = 1, \cdots$ do :
4:     2RB($e_i$)
5:     let $E_i$ be the multiset of all received messages
6:     if $|E_i| > 2n/3$ then
7:         if $E_i$ includes only one value $v$ then
8:             decide($v$)
9:         else
10:            $e_i \leftarrow$ the most frequent value in $E_i$
11:        endif
12:endfor
```

Fig. 3. Algorithm F-CPO: A fast consensus algorithm on the system with (1/2)-POA for $t < n/3$

become a same one. However, in our model, one correct round (to be exact, one cycle including one correct round) triggers the agreement of all candidate values. Actually, in the algorithm F-CPO, it is guaranteed with high probability that all non-crashed processes have a same candidate if a cycle c includes one correct round and the number of crashes occuring the following cycle $c + 1$ is at most two. Notice that such pair of cycles necesarily exists in the execution of $f/2 + 1$ cycles, which is the key fact that our algorithm achives $f + O(1)$ expected round complexity (the detail is shown in the correctness proof).

Correctness. We prove the correctness of the algorithm F-CPO. Let $E(c)$ be the multiset consisting of candidate values of all non-crashed processes at the beginning of cycle c, and $\#_x(c)$ be the times that value x appears in $E(c)$. We define 1st(c) as the value d that maximize $\#_d(c)$ (if there are two or more values that maximize $\#_d(c)$, the largest candidate value is chosen as 1st(c)). We also define 2nd(c) as the value d that secondly maximizes $\#_d(c)$ after 1st(c) (Same as 1st(c), if two or more values secondly maximize, the largest one is chosen). A number of proofs are omitted for lack of space (Lemma 9, 10, and 11).

Lemma 9. *Assume that p_i decides a value d at cycle c. If a process p_j ($\neq p_i$) updates its candidate variable e_j at cycle $c'(\geq c)$, then its updated value is d.*

Lemma 10. *No process decides differently.*

Proof . *Using the lemma 9, we can prove this lemma by the same way as the lemma 5.* □

Lemma 11. *If a process decides a value d, then, d is a value proposed by a process.*

Lemma 12. *Let n' be the number of non-crashed processes at the beginning of c, and p_j be the process that broadcast an information d using* 2RB*. Then,*

1. *if p_j does not crash at round $2c-1$ (that is, the first round of the cycle c), and at most one process crashes at round $2c$ (the second round of c), the information d is received by all non-crashed processes with probability $1-O(n/\alpha^n)$, and*
2. *if p_j crashes at the first round of c and no process crashes at the second round of c, the information d is received by more than $n'/2+2$ or less than $n'/2-2$ processes with probability $1-O(n^{-1/2})$.*

Proof . *(***Proof of 1***) This lemma can be proved in the same way as Lemma 1. (***Proof of 2***) Let x be the number of processes that receive d at the beginning of round $2c$ (notice that the value x is not the random variable because the set of messages sent by p_j at round $2c-1$ is deterministically controlled by the adversary). Clearly, if $x=0$, no process receives messages d at the end of c. Thus, in the followings, we consider the case of $x>0$. We define P_1 as the set of processes that receive d with probability 1 (i.e., any process in P_1 receives the message with d that is not contained in the omission candidate C_{2c}). We also define n_1 as the cardinality of P_1. If $n_1 > n'/2+2$ holds, $n'/2+2$ processes necessarily receive d and thus the lemma holds. Thus, we assume $n_1 \leq n'/2+2$. Since there is no crash at round $2c$, any non-crashed process not in P_1 unreceives the information d with probability $(1/2)^x$ (denoted by q for short). Letting X as be the random vabriable representing the number of processes receiving d at the end of cycle c, we obtain*

$$\Pr[X=k] = \binom{n'-n_1}{n'-k} q^{n'-k}(1-q)^{k-n_1}. \tag{4}$$

Since this is the binomial distribution, it takes the maximum in the average case of $n'-k=(n'-n_1)q$. Thus,

$$\begin{aligned}\Pr[X=k] &\leq \binom{n'-n_1}{(n'-n_1)q} q^{(n'-n_1)q}(1-q)^{(n'-n_1)q} \\ &\leq O(n^{-1/2}), \end{aligned} \tag{5}$$

where we use a weaker variant of Stirling approximation $\binom{N}{qN} < q^{-qN}(1-q)^{-N(1-q)} \cdot O(N^{-1/2})$ and the fact of $(n'-n_1)=\Theta(n)$. From this inequality, using the union bound, we have $\Pr[(n'/2-2) \leq X \leq (n'/2+2)] \leq \sum_{z=-2}^{2} \Pr[X=(n'/2+z)] \leq 5 \cdot O(n^{-1/2})$. The lemma is proved. $\square$.

Lemma 13. *Let c be a cycle where at most one process crashes. Then, $|\#_{1st}(c+1)-\#_{2nd}(c+1)| > 2$ holds with probabilty $1-O(n^{-1/2})$.*

Proof . *If no crash occurs, the lemma clearly holds from Lemma 2 because all non-crashed processes receives a same set of messages with probabilty $1-O(n^2/\alpha^n)$. Thus, we only consider the case where one process crashes. Let $\hat{p}_0$,*

$\hat{p}_1, \cdots \hat{p}_{n'-1}$ be non-crashed processes at the end of c, and E_i be the event that all n' processes receives the information sent by $\hat{p}_i$. The process that crashes during c is denoted by p_x. We also intoduce the event E_X that the information broadcast by p_x is received more than $n'/2+2$ or less than $n'/2-2$ processes. For a event E, we denote its complement by $\bar{E}$. From Lemma 12, $\Pr[\bar{E_i}] \leq O(n/\alpha^n)$ and $\Pr[\bar{E_X}] \leq O(n^{-1/2})$ holds. Then, using the union bound, we obtain

$$\begin{aligned} \Pr\left[\left(\bigcup_{k=0}^{n'-1} \bar{E_i}\right) \cup \bar{E_X}\right] &\leq \sum_{k=0}^{n'-1} \Pr[\bar{E_i}] + \Pr[\bar{E_X}] \\ &\leq (n'-1)O(n/\alpha^n) + O(n^{-1/2}) \end{aligned} \tag{6}$$

Therefore, $\Pr[(\cap_{k=1}^{n-1} E_i) \cap E_X] \geq 1 - O(n^{-1/2})$ holds. This implies that at least $n'/2+2$ processes receive a same set of messages consisting of more than $2n/3$ informations (notice $n' > 2n/3$) with probability $1 - O(n^{-1/2})$. Then, more than $n/2+2$ processes update their candidate variables with a same value. It follows that $|\#_{1st}(c+1) - \#_{2nd}(c+1)| > 2$ holds with probability $1 - O(n^{-1/2})$. The lemma is proved. □

Lemma 14. *Let c be a cycle where at most two processes crash, and n' be the number of non-crashed processes at the beginning of c. Then, with probability $1 - O(n^2/\alpha^n)$, all processes that do not crash at the end of c receive at least $\max\{n'-2, 2n/3\}$ messages.*

Proof . *Using Lemma 12, we can prove this lemma in the same way as Lemma 2.* □

Lemma 15. *Assume $|\#_{1st}(c) - \#_{2nd}(c)| > 2$. If at most two processes crash during cycle c, all non-crashed processes have a same candidate value with probabilty $1 - O(n^2/\alpha^n)$.*

Proof . *Let n' be the number of processes that do not crash at the beginning of c. From Lemma 14, we can guarantee, with probability $1 - O(n^2/\alpha^n)$, that all non-crashed processes receive $\max\{n'-2, 2n/3\}$ messages. Since $|\#_{1st}(c) - \#_{2nd}(c)| > 2$ holds, the most frequent value of all received by each process is $\mathrm{1st}(c)$. This implies that all processes update its candidate variable with the value $\mathrm{1st}(c)$. The lemma is proved.* □

Lemma 16 (Probabilistic Termination). *Every correct process decides with probability 1. In addition, the worst-case expected number of rounds until all correct processes decide is $f + O(1)$.*

Proof . *We define a* block *as two consecutive cycles where the first cycle have at most one crash, and the second one has at most two crashes, respectively. Two blocks are said to be independent if there is no cycle belonging to both blocks. Let X_1 be the random variable representing the number of cycles until all processes have a same candidate value, and X_2 be the random variable representing the number of cycles taken by the decision of all processes from the cycle when all* *processe have a same candidate value. We first bound $E[X_1]$ under f_1 process crashes. For any $k > 0$, the execution consisting of $f_1/2 + 1 + 2k$ cycles includes $k + 1$ blocks independent with each other. From Lemmas 13 and 15, in every block, all non-crashed processes decides with probability $1 - O(1/\sqrt{n})$. Thus, we obtain $E[X_1] = \sum_{k=1}^{\infty}(f_1/2 + 1 + 2k) \cdot (1 - O(1/\sqrt{n})) \cdot O(1/\sqrt{n})^k = f_1/2 + O(1)$. Next, assuming f_2 crash failures are possible, we bound $E[X_2]$. From Lemma 14, if a cycle c has only two crash processes, all processes decides at the end of c with probability $1 - O(n^2/\alpha^n)$. Thus, we obtain $E[X_2] = \sum_{k=1}^{\infty}(f_2/2 + 1 + 3k) \cdot (1 - O(n^2/\alpha^n)) \cdot O(n^2/\alpha^n)^k = f_2/2 + O(1)$. These two bounds implies that the expected number of cycles until all correct processes decide is $E[X_1] + E[X_2] = (f_1 + f_2)/2 + O(1) = f/2 + O(1)$. Since one cycle consists of two rounds, we have proved the $f + O(1)$ worst-case expected round complexity.* □

6 Concluding Remarks

This paper has newly introduced a novel class of adversaries, called p-probabilistic omission adversary (p-POA). We also investigated the consensus algorithm working under p-POA, and its complexity. We proposed two impossibility results. The first one is that no algorithm can tolerate more than of equal to $n/2$ crash failures, for any $p > 0$. The second one is that for any $p > 0$, no algorithm can halt. We proposed two consensus algorithms CPO and F-CPO working under the (1/2)-probabilistic omission adversary. These algorithms have distinct advantages in the point of crash-failure resiliency and time complexity respectively.

References

1. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Transactions on Computer Systems 12(2), 91–122 (1994)
2. Basu, A., Charron-Bost, B., Toueg, S.: Simulating reliable links with unreliable links in the presence of process crashes. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 105–122. Springer, Heidelberg (1996)
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
4. Delporte-Gallet, C., Fauconnier, H., Freiling, F.C.: Revisiting failure detection and consensus in omission failure environments. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 394–408. Springer, Heidelberg (2005)
5. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM 34(1), 77–97 (1987)
6. Dolev, D., Reischuk, R., Strong, R.: Early stopping in byzantine agreement. Journal of ACM 37(4), 720–741 (1990)
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. Freiling, F.C., Herlihy, M., Penso, L.D.: Optimal randomized fair exchange with secret shared coins. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 61–72. Springer, Heidelberg (2006)

9. Guerraoui, R.: Revisiting the relationship between non-blocking atomic commitment and consensus. In: Helary, J.-M., Raynal, M. (eds.) WDAG 1995. LNCS, vol. 972, Springer, Heidelberg (1995)
10. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In: Mullender, S. (ed.) Distributed Systems, ch. 5, pp. 97–145. Addison-Wesley, Reading (1993)
11. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13, 124–149 (1991)
12. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
13. Moses, Y., Rajsbaum, S.: A layered analysis of consensus. SIAM Journal on Computing 31(4), 989–1021 (2002)
14. Neiger, G., Toueg, S.: Automatically increasing the fault-tolerance of distributed algorithms. Journal of Algorithms 11(3), 374–419 (1990)
15. Parvédy, P.R., Raynal, M.: Optimal early stopping uniform consensus in synchronous systems with process omission failures. In: SPAA. Proc. of the 16th annual ACM symposium on Parallelism in algorithms and architectures, pp. 302–310. ACM Press, New York (2004)
16. Perry, K.J., Toueg, S.: Distributed agreement in the presence of processor and communication faults. IEEE Transactions on Software Engineering SE-12(3), 477–482 (1986)
17. Raynal, M.: Consensus in synchronous systems: A concise guided tour. In: PRDC. Proc. of Pacific Rim International Symposium on Dependable Computing, pp. 221–228 (2002)
18. Santoro, N., Widmayer, P.: Majority and unanimity in synchronous networks with ubiquitous dynamic faults. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 262–276. Springer, Heidelberg (2005)

Upper Bounds for Stabilization in Acyclic Preference-Based Systems

Fabien Mathieu

Orange Labs, 38-40 rue du général Leclerc, 92794 Issy-les-Moulineaux, France
fabien.mathieu@orange-ftgroup.com
http://gyroweb.inria.fr/~fmathieu/

Abstract. Preference-based systems (p.b.s.) describe interactions between nodes of a system that can rank their neighbors. Previous work has shown that p.b.s. converge to a unique locally stable matching if an *acyclicity* property is verified. In the following we analyze acyclic p.b.s. with respect to the self-stabilization theory. We prove that the round complexity is bounded by $\frac{n}{2}$ for the adversarial daemon. The step complexity is equivalent to $\frac{n^2}{4}$ for the round robin daemon, and exponential for the general adversarial daemon.

Keywords: Preference-based systems, b-matching, acyclicity, round robin, adversarial and round robin daemons.

1 Introduction

A system is called *preference-based* if each of its nodes selfishly tries to activate its best edges according to some personal ranking. The description of the stable configurations — if any — of a p.b.s. is known as the stable b-matching problem. b-matching theory and its variants have applications in a variety of real-world situations, including dating agencies, college admissions, roommates attributions, assignment of graduating medical students to their first hospital appointment, or kidney exchanges programs [1,2,3,4,5].

Recently, Lebedev *et al.* showed that many preferences are acyclic. They also proved that an acyclic p.b.s. has a unique stable configuration, and always stabilizes[6,7]. This convergence result gave a theoretical proof for upload/download correlations in incentive-based networks like BitTorrent [8].

Our contribution is to analyze the convergence properties of acyclic p.b.s. by using the self-stabilization approach [9,10]. We use the convergence theorem proved in [6] to compute the step and round complexities for the round robin and the adversarial daemon.

The rest of this paper is organized as follows. In Section 2 we give a short introduction to the p.b.s. model and present the convergence theorem. In Section 3 we give the round complexity. The step complexity for the round robin and adversarial daemons are presented in Section 4.

2 Model

A preference-based system consists of a set V of n nodes, whose possible interactions are described by an acceptance graph G and a set of rankings $(<_i)_{i \in V}$.

The acceptance graph $G = (V, E)$ is an undirected graph. It describes compatibilities: a node i and a node j are capable of collaborating (we say that i is acceptable for j, and vice versa) if, and only if (iff) $\{i, j\} \in E$.

The ranking $<_i$ of a node i is a total order on the neighbors of i. If j and k are two distinct neighbors of i, then we say that i prefers j to k iff $j <_i k$.

Preference-based systems are also characterized by a quota vector b: a node i cannot sustain more that $b(i)$ simultaneous collaborations[1]. In this paper, we focus on the case $b = 1$, thus the state of collaborations at a given time is a matching of G, that we call configuration. $C(i)$ denotes the match of a node i, if any, in a configuration C.

The only action allowed in the model is the resolution of *blocking pairs*. A blocking pair is a pair of nodes that are not matched together, in which both could gain by collaborating (even if it means dropping the current match). A configuration with no blocking pair is *stable*.

We say that a node is *eligible* in a configuration if it belongs to at least one blocking pair. In this paper, we only consider *best-match* resolution: the action of an eligible node consists in choosing the node it prefers among the blocking pairs to which it belongs.

A *computation* is a maximal sequence of configurations such that for each configuration C_i, the next configuration C_{i+1} is obtained by the action of one eligible node in C_i. Maximality of a computation means that the computation is infinite or it terminates in a configuration where none of the actions are enabled, e.g. a stable configuration.

A *daemon* is an action scheduler that generates a computation. We consider here two kinds of daemons: the *round robin* daemon follows a round robin scheduling over V, and executes the action of i whenever i is selected by the scheduler and eligible; the adversarial daemon may select *any* eligible node at every step. The adversarial daemon encompasses the round robin.

2.1 Convergence Theorem for Acyclic p.b.s.

A preference cycle is a cycle of nodes $i_1, \ldots, i_k$, with $k \geq 3$, such that each node i_j prefers its successor i_{j+1} to its predecessor i_{j-1} ($i_{j+1} <_{i_j} i_{j-1}$). A p.b.s. without any preference cycle is called *acyclic*.

A particular kind of acyclic p.b.s. are the global-preferences systems, where the preferences come from an inherent total order. For any global-preferences system, there exists a labeling $i_1, \ldots, i_n$ of its nodes such that for any node i_j with distinct neighbors i_k and i_l, $i_k <_{i_j} i_l$ is equivalent to $k < l$. So, for sake of simplicity, the nodes of a global-preferences system are often labeled with the integers 1 to n.

[1] The b letter in *b-matching* stands for this quota vector.

In [6], Lebedev *et al.* proved that an acyclic p.b.s. has a unique stable configuration, and that all computations are finite. In other word, for any starting configuration, the adversarial daemon eventually converges to the stable configuration. Thus we can say that acyclic p.b.s. are self-stabilizing for their stable configuration.

Strong pairs. Strong pairs are a central concept in acyclic preference-based systems. A pair is strong in a configuration if:

- each node of the pair is ranked first by the other,
- or if there is one or more nodes that are ranked better, each of them is matched and forms a strong pair with its mate.

Strong pairs have many qualities (cf [6,7] for details):

- they are edges of the stable configuration,
- a strong blocking pair is stable until the pair is matched (the property is not affected by any daemon),
- the best-match resolution of any of the two nodes of a strong blocking pair matches these two nodes,
- once matched, a strong blocking pair is a stable matched pair,
- all non-stable configurations of a given acyclic p.b.s. admit at least one strong blocking pair.

Using strong blocking pairs, the question we will address in the following is the effective time complexity of the convergence under possibly adversarial scheduling regimes. We consider two measures for evaluating this complexity. The *round* complexity and the *step* complexity. A round is a sub-sequence of a computation in which every node that was eligible at the beginning of the round either is activated or ceases to be eligible during the round. The *step* complexity investigates the maximum length of a computation for all possible starting configurations.

3 Round complexity

The round complexity is simple to compute in the case of acyclic p.b.s., as shown by theorem 1:

Theorem 1. *Starting from any configuration, and under any daemon, an acyclic preference-based system stabilizes in $\lfloor \frac{n}{2} \rfloor$ rounds. Furthermore, there are instances where the round robin daemon requires $\lfloor \frac{n}{2} \rfloor$ rounds.*

Proof. The proof comes from the existence of strong blocking pairs in acyclic preference-based systems: after each round (until stabilization), we are sure that at least a a node of a strong blocking pair is selected, thus a stable edge is formed. As the stable configuration is a matching, it has at most $\lfloor \frac{n}{2} \rfloor$ edges, so the stabilization cannot last more than $\lfloor \frac{n}{2} \rfloor$ rounds. This bound is tight because it is reached for global preferences and complete acceptance graph, if we consider the round robin daemon with scheduling $n, (n-1), \ldots, 2, 1$, starting from the empty configuration $C_\emptyset$.

4 Step Complexity

In [6], the proof of the existence of a stable configuration relies on the following: in an acyclic p.b.s., all configurations of a given computation are distinct. Thus a first upper bound for the step complexity is the maximal number of matchings of a graph with n nodes. This number, also known as the number of involutions of a set of size n, has a factorial-like asymptotic behavior [11]. In the following we prove a tight quadratic bound for the round robin daemon and a tight exponential bound for the adversarial daemon.

4.1 Round Robin Daemon

The step complexity for the round robin daemon is given by theorem 2:

Theorem 2. *Starting from any configuration, the round robin daemon takes at most* $\sum_{k=0}^{\lfloor \frac{n}{2} \rfloor -1} (n-(2k+1))$ *steps to converge (hence the complexity is equivalent to* $\frac{n^2}{4}$*). This bound is tight.*

Proof. The reasoning is the same as for the round complexity. As long as the current configuration is not stable, there exists at least two nodes that belong to a strong blocking pair. Hence after at most $n-1$ steps, the round robin daemon is forced to match a strong blocking pair. The remaining non stable nodes are less than $n-2$, and if the configuration is not stable yet, at least two of them form a strong blocking pair, so after at most $n-3$ steps, a new stable edge is formed...

If we continue this process, we see that the number of steps cannot be more than

$$(n-1)+(n-3)+(n-5)... = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor -1} (n-(2k+1)) .$$

This is equal to $\frac{n^2}{4}+c$, where c is $-\frac{1}{4}$ when n is odd and 0 when n is even.

Like for the round complexity, one can see that this bound is reached for global preferences and complete acceptance graph, if we consider the round robin daemon with scheduling $n,(n-1),\ldots,2,1$, starting from $C_\emptyset$.

4.2 Adversarial Daemon

The step complexity is harder to compute for the adversarial daemon. First we will prove that the step complexity is not greater than $2^{n-1}-1$. Then we will introduce a daemon with complexity $\Omega(\mu^n)$, with $\mu \approx 1.6826$, thus proving that the complexity of the adversarial daemon stands somewhere between these two bounds.

2^n Upper Bound

Theorem 3. *The number of steps under any daemon is less than* $2^{n-1}-1$ *for an acyclic preference-based system for all possible initial configurations.*

Proof. We need to introduce $S(n,k)$, that denotes the maximal number of steps that can be made by using only a fixed subset of size k of the nodes of an acyclic system of length n (the system and its initial configuration are arbitrary). $S(n,n)$ is the maximal complexity, so we want to prove that $S(n,n) \leq 2^{n-1} - 1$.

Let C be an initial configuration of a system of size n. If C is not stable, C admits at least one strong blocking pair. Thus any computation can be split in three subsequences: before, during, and after the resolution of the strong blocking pair. The two nodes of the strong blocking pair cannot be selected before or after the pair is resolved, so we have:

$$S(n,n) \leq \underbrace{S(n,n-2)}_{\text{before resolution}} + \underbrace{1}_{\text{resolution}} + \underbrace{S(n,n-2)}_{\text{after resolution}}$$

Now, we need to express $S(n,k)$ for $1 \leq k < n$. First, we have $S(n,1) = 1$, because a single node i cannot be eligible after a selection: it is matched with the best node j it can (it cannot prefer the possible previous mate k of j, otherwise $\{i,j,k\}$ would be a preference cycle).

For $2 \leq k < n$, call A the set of nodes that can be selected, I the set of nodes that cannot be selected. One of the two following propositions is true:

– there exists a node in A that is never eligible as long as only nodes of A are selected,
– there exists a pair in $A \times A$ or in $A \times I$ that is strong with respect to the action of nodes from A.

In order to prove that, we consider the following path construction scheme: the successor, if any, of a node a in A is defined as the best choice for a that does not belong to a pair of I stable under the actions of A (stable nodes internal to I cannot interfere with other nodes, so one *forget* them). The successor of a non-stable node of I is its best choice among its neighbors from A plus its possible current neighbor. Note that a non stable node of I has always a successor.

Starting from an initial node a_0, one construct a path using this scheme. The successor of each node is the best node it can expect under the action of A. Because the system is acyclic and finite, the path eventually ends with a node without successor, or with two nodes j, k such that each one is the successor of the other.

If the path ends with a node without successor, this node belongs to A, and it cannot be eligible (its non-stable-neighbors list is empty). If it ends with two reciprocal successors i and j, then $\{i,j\}$ is strong under the action of A, and it belongs to $A \times A$ or $A \times I$ ($\{i,j\} \in I^2$ would imply that $\{i,j\}$ is a matched pair of I that is stable under the action of A).

We can now bound $S(n,k)$, using the same *before/during/after* argument than for $S(n,n)$:

$$S(n,k) \leq \begin{cases} S(n,k-1) & \text{if a node from } A \text{ is never eligible,} \\ 1 + 2S(n,k-2) & \text{if a pair from } A^2 \text{ is strong with respect to } A, \\ 1 + 2S(n,k-1) & \text{if a pair from } A \times I \text{ is strong with respect to } A. \end{cases}$$

In any case, we have $S(n,k) \leq 1 + 2S(n,k-1)$ for $2 \leq k < n$. An immediate recurrence gives $S(n,n-2) \leq 2^{n-2} - 1$, thus we have $S(n,n) \leq 2^{n-1} - 1$. This concludes the proof.

μ^n Lower Bound

Theorem 4. *There exists an acyclic preference-based system and a daemon such that the complexity is $\Omega(\mu^N)$, with*

$$\mu = \sqrt{\frac{\sqrt[3]{316 + 12\sqrt{249}}}{6} + \frac{20}{3\sqrt[3]{316 + 12\sqrt{249}}} + \frac{2}{3}} \approx 1.683$$

Proof. As the complexity is obviously increasing with n, we can restrain the proof to systems of even size. We consider a global-preferences system of size $N = 2n$ with complete acceptance graph.

The *worst eligible* daemon (w.e.d.) is defined, for global preference systems, as follows: as long as there exists an eligible node, the daemon select the worst-eligible node, i.e. the eligible node with the highest label.

Call T_C the number of steps taken by the w.e.d. to reach the stable configuration from an initial configuration C (the number of nodes and the knowledge graph are implicit). $T_\emptyset(n)$ stands for $T_{C_\emptyset(n)}$, where $C_\emptyset(n)$ is the empty configuration in the complete graph with $2n$ nodes. $T_{12}(n)$ stands for $T_{C_{12}(n)}$, where $C_{12}(n)$ is the stable configuration in the complete graph with $2n$ nodes, except the best pair, $\{1,2\}$, is not matched.

We will prove that $T_\emptyset(n) = \Theta(\lambda^n)$, with $\lambda = \mu^2$. This will guarantee that the complexity of the adversarial worst-eligible daemon is $\Omega(\mu^N)$ (other initial configurations may take more steps).

$T_\emptyset$ can be expressed as a recursive function of T_{12}. This shown by Equation (1).

$$T_\emptyset(n) = n^2 + \sum_{i=1}^{n-1} (n-i) T_{12}(i) \tag{1}$$

For proving (1), we need to understand how the w.e.d. performs from $C_\emptyset$. The basic idea is that at each step of the w.e.d., nodes can be split according to a pivot. Nodes above the pivot or matched with those form the upper part. Other nodes form the lower part. Upper nodes are never selected by the daemon, while lower nodes perform the w.e.d. in a recursive way, with little change on upper part configuration. The pivot increases one by one until the stable configuration is reached.

We start with $C_\emptyset$ where all nodes are single.

$$\underbrace{\{2n\}, \{2n-1\}, \ldots, \{n+1\}}_{\text{lower part}}, \underbrace{\{n\}, \ldots, \{2\}, \{1\}}_{\text{upper part}}$$

First, $2n$ goes with 1. $2n$ is not eligible, so $2n-1$ is selected and takes 1 from $2n$, that goes to 2. Then $2n-2$ matches with 1, forcing $2n-1$ to lower its matching, cascading to $2n$. After $\frac{n(n+1)}{2}$ steps, the w.e.d. reaches the configuration where each node of the lower part is matched with a node of the upper part as follows:

$$\{2n, \underbrace{n}_{\text{lowest eligible node}}\}, \{2n-1, n-1\}, \ldots, \{n+2, 2\}, \{n+1, 1\}$$

At this point, all nodes from what we have called the lower part are not eligible. So the first upper node, n, is selected and becomes the pivot, leading to the configuration

$$\underbrace{\{2n\}, \{n+1\}}_{\text{lower part}}, \underbrace{\{2n-1, n-1\}, \ldots, \{n+2, 2\}, \{n, 1\}}_{\text{upper part}}$$

The lower part ($\{2n\}, \{n+1\}$) performs $T_{\emptyset}(1) = T_{12}(1)$ (in other words, $2n$ matches with $n+1$), then $n+1$ swaps with $n+2$ as 2's mate. The lower part is now $\{2n\}, \{n+2\}$ and performs $T_{12}(1)$, then $n+2$ swaps with $n+3$ as 3's mate, and so on... Eventually, $2n-2$ swaps with $2n-1$ as $n-1$'s mate and $2n$ matches with $2n-1$, resulting in the following configuration :

$$\{2n, 2n-1\}, \{2n-2, \underbrace{n-1}_{\text{lowest eligible node}}\}, \{2n-3, n-2\}, \ldots, \{n+1, 2\}, \{n, 1\}$$

Now $n-1$ is the lowest eligible node (l.e.n.) and becomes the pivot.
More generally, the configuration before $2 \leq i \leq n$ becomes the pivot is

$$\{2n, 2n-1\}, \ldots, \{2i+2, 2i+1\}, \{2i, \underbrace{i}_{l.e.n.}\}, \ldots, \{i+2, 2\}, \{i+1, 1\}$$

After i is selected, we obtain

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+2, 2i+1\}, \{2i\}, \{i+1\}}_{\text{lower part}}, \underbrace{\{2i-1, i-1\} \ldots, \{i+2, 2\}, \{i, 1\}}_{\text{upper part}}$$

The upper part needs will mutate $i-1$ times (including i's selection) to become

$$\{2i-2, i-1\} \ldots, \{i+1, 2\}, \{i, 1\}$$

Each mutation is due to the selection of the highest node from the lower part. This is what we call a transitional step (selection of a lower part node to match with an upper part node). After a transitional step, the lower part is made of $2(n-i+1)$ nodes; More precisely $2(n-i)$ nodes perform a local $\{2k, 2k-1\}$ matching, while the two best nodes (of the lower part) are single. As only the best node of the lower part can interact with the upper part and as the w.e.d. never selects the best node, the lower part will perform $T_{12}(n-i+1)$ steps before the next transitional step. To summarize, the w.e.d. computation, starting from $C_{\emptyset}$ is made of:

- $\frac{n(n+1)}{2}$ steps to match each node of the upper half to a node of the lower half.
- for each pivot i $(n \le i \le 2)$, $i-1$ transitional steps, and $(i-1)T_{12}(n-i+1)$ low steps.

That leads to Equation (1) :

$$\begin{aligned}T_{\emptyset}(n) &= \tfrac{n(n+1)}{2} + \textstyle\sum_{i=2}^{n}(i-1)(1+T_{12}(n-i+1))\\ &= \tfrac{n(n+1)}{2} + \tfrac{n(n-1)}{2} + \textstyle\sum_{i=1}^{n-1}(n-i)T_{12}(i)\\ &= n^2 + \textstyle\sum_{i=1}^{n-1}(n-i)T_{12}(i)\end{aligned}$$

Now that we have an expression of $T_\emptyset$ that depends on T_{12}, we need to specify the behavior of T_{12}. This is given by Equation (2).

$$T_{12}(n) = \begin{cases} 0 & \text{if } n < 1 \\ 1 & \text{if } n = 1 \\ 4\left(n-1+\sum_{i=1}^{n-3} T_{12}(i)\right) + 3T_{12}(n-2) + T_{12}(n-1) & \text{if } n > 1 \end{cases} \qquad (2)$$

For $n \le 1$, (2) is obvious. For $n \ge 2$, we adapt (1)'s proof. The w.e.d. first matches all upper nodes, then combinatorics can be made using transitional and low steps derived from a pivot. The main difference is that due to pre-existing upper matchings, we need to distinguish odd and even pivots. Now, let us describe the different phases of the w.e.d. computation starting from C_{12} :

Initial phase. First $2n$ matches 1, $2n-1$ matches 1 and $2n$ matches 2. So after 3 steps, nodes form the configuration

$$\{2n-2, 2n-3\}, \ldots, \{4,3\}, \{2n,2\}, \{2n-1,1\}$$

Our first pivot will be $2n-2$ (even).

Even pivot. The configuration before $2i$ $(n-1 \le i \le 2)$[2] becomes the pivot is:

$$\{2n, 2n-1\}, \ldots, \{2i+4, 2i+3\}, \{2i, 2i-1\}, \ldots, \{4,3\}, \{2i+2, 2\}, \{2i+1, 1\}$$

After $2i$ is selected, we obtain

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+4, 2i+3\}, \{2i+1\}, \{2i-1\}}_{\text{lower part}}, \ldots, \{2i+2, 2\}, \{2i, 1\}$$

The lower part is made of $2(n-i)$ nodes that form a C_{12} configuration (the two best nodes, $2i+1$ and $2i-1$, are single). Low transitions occurs until the second best node $(2i+1)$ is selected and matches with 2. In other words, after $T_{12}(n-i) - T_{12}(n-i-1)$ transitions (we count the $T_{12}(n-i)$ transitions

[2] We treat the pivot 2 separately.

except we remove transitions that happens after node 2 is selected) we obtain the configuration

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+4\}, \{2i+2\}, \{2i+3, 2i-1\}}_{\text{lower part}}, \ldots, \{2i+1, 2\}, \{2i, 1\}$$

The lower part configuration is similar to $C_{12}(n-i)$ except for the four best nodes[3] : first and third nodes ($2i-1$ and $2i+3$) are together, while second and fourth nodes are single. The w.e.d. first performs the stable solution for the lower part minus $2i-1$ and $2i+3$ in $T_{12}(n-i-1)$ steps:

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+4, 2i+2\}, \{2i+3, 2i-1\}}_{\text{lower part}}, \ldots, \{2i+1, 2\}, \{2i, 1\}$$

Then $2i+2$ is selected and matches $2i-1$:

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+4\}, \{2i+3\}, \{2i+2, 2i-1\}}_{\text{lower part}}, \ldots, \{2i+1, 2\}, \{2i, 1\}$$

Lastly, after $T_{12}(n-i-1)$ steps, the lower part is stabilized and we have

$$\{2n, 2n-1\}, \ldots, \{2i+4, 2i+3\}, \{2i+2, \underbrace{2i-1}_{\text{l.e.n.}}\}, \ldots, \{2i+1, 2\}, \{2i, 1\} \quad (3)$$

$2i-1$ is now the lowest eligible node and is ready to become the pivot.

Odd pivot. (3) is the configuration before $2i-1$ ($n-1 \leq i \leq 2$) becomes the pivot. After that step, the configuration is

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+4, 2i+3\}, \{2i+2\}, \{2i\}}_{\text{lower part}}, \ldots, \{2i+1, 2\}, \{2i-1, 1\}$$

After $T_{12}(n-i)$ steps, the w.e.d. stabilizes the lower part:

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+4, 2i+3\}, \{2i+2, 2i\}}_{\text{lower part}}, \ldots, \{2i+1, 2\}, \{2i-1, 1\}$$

Then $2i$ swaps with $2i+1$ as 2's mate:

$$\underbrace{\{2n, 2n-1\}, \ldots, \{2i+4, 2i+3\}, \{2i+2\}, \{2i+1\}}_{\text{lower part}}, \ldots, \{2i, 2\}, \{2i-1, 1\},$$

[3] For $i = n-1$, there is only two nodes involved. However, the presented results stand if we set $T_{12}(n) = 0$ for any $n \leq 0$.

and after another $T_{12}(n-i)$ steps, the configuration is

$$\{2n, 2n{-}1\}, \ldots, \{2i{+}4, 2i{+}3\}, \{2i{+}2, 2i{+}1\}, \{\underbrace{2i-2}_{\text{l.e.n.}}, 2i{-}3\}, \ldots, \{2i, 2\}, \{2i{-}1, 1\}.$$

This is the point where $2i-2$ becomes the next pivot

2 as a pivot. Eventually, 2 is selected and matches with 1. After that, the w.e.d. performs $T_{12}(n-1)$ to produce the stable configuration.

We can now verify Equation (2):

$$\begin{aligned} T_{12}(n) = & \underbrace{3}_{\text{Initial phase}} + \underbrace{\sum_{i=2}^{n-1} (2 + T_{12}(n-i) + T_{12}(n-i-1))}_{\text{Even pivots}} \\ & + \underbrace{\sum_{i=2}^{n-1} (2 + 2T_{12}(n-i))}_{\text{Odd pivots}} + \underbrace{1 + T_{12}(n-1)}_{\text{2 as a pivot}} \\ = & \; 4 + T_{12}(n-1) + \textstyle\sum_{i=2}^{n-1} (4 + 3T_{12}(n-i)) + \sum_{i=3}^{n} (T_{12}(n-i)) \\ = & \; 4(n-1) + \textstyle\sum_{i=1}^{n-3} 4T_{12}(i) + 3T_{12}(n-2) + T_{12}(n-1) \end{aligned}$$

Equation and (2) allows us to affirm that T_{12} follows an asymptotic geometric progression. Moreover, the asymptotic common ratio can be explicitly found: it must be a positive solution of Equation (4).

$$x^2 = x^1 + 3 + \frac{4}{x-1} \tag{4}$$

This equation leads to Equation (5), a third degree equation whose only positive root is the asymptotic common ratio for T_{12}.

$$x^3 = 2x^2 + 2x + 1 \tag{5}$$

Analysis indicates that the positive root of Equation (5) is

$$\lambda = \frac{\sqrt[3]{316 + 12\sqrt{249}}}{6} + \frac{20}{3\sqrt[3]{316 + 12\sqrt{249}}} + \frac{2}{3} \approx 2.8312$$

Using Equation (1), we see that $T_\emptyset$ is also $\Theta(\lambda^n)$. That ends our proof.

5 Conclusion

We have shown that acyclic preference-based systems have a linear round complexity even for the adversarial daemon, and a quadratic step complexity for the round robin daemon. This means the the self-stabilization of such systems is good, as long as nodes cannot be eligible and not selected for an arbitrary long period of time. These bounds are tight for global p.b.s., but according to a previous work, the round complexity may be logarithmic for most acyclic p.b.s. that are not global [12]. On the other hand, the step complexity stands between $\Omega(\mu^n)$ and $2^{n-1} - 1$ for the adversarial daemon. This is a more precise result than the factorial-like upper bound that can be deduced from the convergence theorem. Note, that global p.b.s. with complete acceptance graph have been used whenever we needed to prove that a bound was reached. Thus the global p.b.s. with complete acceptance graph is a sort of extremum among the possible acyclic p.b.s., as empirically observed in [12].

References

1. Gale, D., Shapley, L.: College admissions and the stability of marriage. American Math. Monthly 69, 9–15 (1962)
2. Irving, R.W., Manlove, D., Scott, S.: The hospitals/residents problem with ties. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 259–271. Springer, Heidelberg (2000)
3. Irving, R.W., Manlove, D.F.: The stable roommates problem with ties. J. Algorithms 43(1), 85–105 (2002)
4. Roth, A.E.: The evolution of the labor market for medical interns and residents: A case study in game theory. Journal of Political Economy 92(6), 991–1016 (1984)
5. Roth, A.E., Sonmez, T., Utku Unver, M.: Pairwise kidney exchange. Journal of Economic Theory 125(2), 151–188 (2005)
6. Lebedev, D., Mathieu, F., Viennot, L., Gai, A.T., Reynier, J., Montgolfier, F.D.: On using matching theory to understand p2p network design. In: INOC (2007)
7. Gai, A.T., Lebedev, D., Mathieu, F., de Montgolfier, F., Reynier, J., Viennot, L.: Acyclic preference systems in p2p networks. In: Euro-Par (2007)
8. Gai, A.T., Mathieu, F., Reynier, J., De Montgolfier, F.: Stratification in P2P networks application to bittorrent. In: ICDCS (2007)
9. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
10. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
11. Chowla, S.: The asymptotic behavior of solutions of difference equations. In: Proceedings of the International Congress of Mathematicians, vol. I, 377, Amer. Math. Soc (1950)
12. Mathieu, F.: Self-stabilization in preference-based networks. In: P2P (2007)

A Self-stabilizing Weighted Matching Algorithm

Fredrik Manne and Morten Mjelde

University in Bergen, Norway
{fredrik.manne, mortenm}@ii.uib.no

Abstract. The problem of computing a matching in a graph involves creating pairs of neighboring nodes such that no node is paired more than once. Previous work on the matching problem has resulted in several self-stabilizing algorithms for finding a maximal matching in an unweighted graph. In this paper we present the first self-stabilizing algorithm for the weighted matching problem. We show that the algorithm computes a $\frac{1}{2}$-approximation to the optimal solution. The algorithm is simple and uses only a fixed number of variables per node. Stabilization is shown under various types of daemons.

Keywords: self-stabilizing algorithms, weighted matching.

1 Introduction

Given a graph with n nodes and m edges, a matching is a set of edges in a graph such that no node is incident to more than one selected edge. In a distributed setting a matching can model a situation where each node must choose exactly one of its neighbors for communication. The associated optimization problem then becomes to choose a matching of maximum cardinality.

The matching problem lends itself well to distributed solutions since progress towards a maximal solution can be made by selecting any edge in any order and adding it to the current matching just as long as the selected edge is not incident to an edge already included in the matching. It is well known that any maximal matching (i.e. where no more edges can be added) is also a $\frac{1}{2}$-approximation to the maximum matching.

Figure 1a shows an example of a non-maximal matching, while Figure 1b illustrates a matching that is maximal, but not maximum. A maximum matching is shown in Figure 1c. Finally, Figure 1d shows a set of edges that is not a matching, since they are incident on the same node.

Previous work on the matching problem has resulted in several self-stabilizing algorithms. Hsu and Huang [7] gave the first such algorithm and proved a bound of $O(n^3)$ on the number of moves assuming a sequential model under an adversarial daemon. This analysis was later improved to $O(n^2)$ by Tel [9] and finally to $O(m)$ by Hedetniemi et al. [6].

Gradinariu and Johnen [5] employed a method of randomization to assign an ID to each node that is unique within distance 2, and used this to run Hsu and

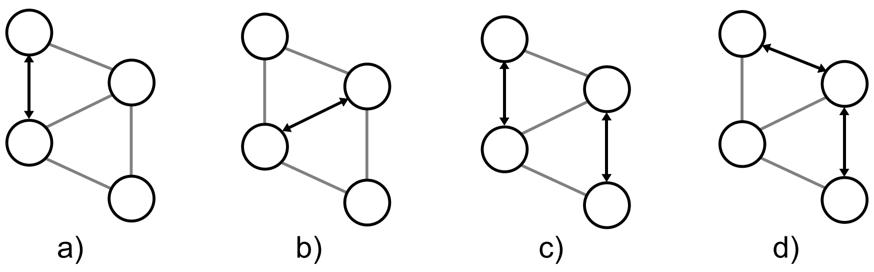


Fig. 1.

Huang's algorithm under an adversarial daemon. They show only finite stabilization time however. Using the same technique of randomized local symmetry breaking, Chattopadhyay et al. [1] later provided a maximal matching with $O(n)$ round complexity, but assuming the weaker fair distributed daemon.

In [4] Goddard et al. describe a synchronous version of Hsu and Huangs algorithm and show that it stabilizes in $O(n)$ time steps. Very recently, Manne et al. [8] presented an algorithm that stabilizes in $O(m)$ time steps in the more general distributed model. Goddard et al. [3] also gave a self-stabilizing algorithm for computing a maximal strong matching, however with exponential stabilization time.

In the current paper we present the first self-stabilizing algorithm for computing a *weighted* matching. As opposed to the unweighted case, we now assume an edge weighted graph and the objective is to compute a matching such that the sum of the weights of the edges in the matching is as large as possible. Again considering an example where every node in a network must choose exactly one neighbor to communicate with, the weighted matching problem can be used to model networks where not all lines of communication are equally desirable. For example, in a wireless system the weight assigned to a link might reflect the quality or bandwidth of the link. In this setting selecting a matching of maximum weight would ensure maximum information flow in the network.

We show that the presented algorithm computes a $\frac{1}{2}$-approximation to the optimal solution. As to speed of convergence, we show that the algorithm stabilizes in $O(n)$ rounds in the distributed model under a fair daemon. This implies that it also stabilizes in $O(n)$ time steps in the synchronous model which is the same as the algorithm by Goddard et al. [4] for the unweighted case. For an adversarial daemon we show that the algorithm stabilizes in a finite (exponential) number of steps both for the sequential and the distributed daemon.

It should be noted that computing a $\frac{1}{2}$-approximation for the weighted matching problem is an inherently more difficult problem than for the unweighted case. The reason for this is that in the weighted case any maximal solution can be arbitrarily bad compared to the optimal solution.

The rest of this paper is organized as follows. In Section 2 we give a short introduction to self-stabilizing algorithms and the computational environment we assume. In Section 3 we present our new algorithm while we show correctness and speed of stabilization in Section 4 before concluding in Section 5.

2 The Self-stabilizing Paradigm

A self-stabilizing algorithm is a distributed system where each node is an independent entity with knowledge only of itself and its neighbors. Unlike many other distributed systems, a self-stabilizing algorithm is not initialized. Instead the algorithm has to be able to reach a stable, or terminal, configuration regardless of its starting configuration. In this sense, a self-stabilizing algorithm is very resistant to transient faults and can also handle a dynamic environment where the structure of the underlying graph is changing.

A self-stabilizing algorithm is comprised of a set of rules, where each rule is made up of a predicate and a move. If a predicate evaluates to true for a node, the node is referred to as privileged and only then may it execute the corresponding move. An algorithm is stable if there are no privileged nodes in the graph.

The general distributed model allows a non-empty subset of the privileged nodes to perform one move each during a time step in the execution of the algorithm. The synchronous model is a sub-variant of this model, and requires that every privileged node executes a move in each step. Another sub-variant of the distributed model, the often-used sequential model, allows only a single node to make a move during each step.

If more than one node in the graph is privileged at that start of a particular time step there are several models that govern which nodes will perform a move. Common to all of these models is the notion of a daemon that makes the actual choice as to which subset of privileged nodes are selected for a move. We distinguish between a fair and an adversarial daemon. Under a fair daemon a privileged node will never have to wait an infinite number of time steps before it is permitted to make a move, while an adversarial daemon can select any privileged node for a move.

With the adversarial daemon, the moves complexity of an algorithm is measured in time steps for the distributed and synchronous model and in single moves for the sequential model. Under a fair daemon we measure moves complexity in rounds, where one round is a minimal sequence of time steps during which every node privileged at the start of the round has either made a move or become non-privileged. For further reading about self-stabilization algorithms see [2].

3 The Algorithm

In the following we present and motivate our self-stabilizing weighted matching algorithm. Note that at this stage we do not make any assumptions as to which model or which daemon the algorithm will execute under.

3.1 The Graph Model

Given an undirected weighted graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. We denote $w_{v,u} > 0$ as the weight of the edge $(v, u) \in E$. Furthermore, we assume that every node $v \in V$ has a unique, comparable, ID, denoted by ID_v. To ensure uniqueness of the edges we define a function $w(v, u) = (w_{v,u}, \max\{ID_v, ID_u\}, \min\{ID_v, ID_u\})$. This triplet, referred to as the *effective weight* of an edge, is used to define a total ordering of the edges. That is, the edges are first ranked by their weight and if two edges have the same weight they are ranked by the highest ID of the two nodes incident on that edge. If these are also equal the edges are ranked by the lowest ID of the two incident nodes. In this way, any node can compute the effective weight of all edges incident on it, and no two edges in the graph will ever be considered to be of equal weight (note that $w(v, null)$ is by definition 0). Thus the heaviest edge in any subset of E is the edge with greatest effective weight. For ease of presentation we will not distinguish between weight and effective weight in the rest of the presentation but merely assume that the weight of each edge is unique.

We also use the notation $N(v)$ for the neighborhood of v. That is, for a node $v \in V$, $u \in N(v) \Leftrightarrow (v, u) \in E$. We say that two edges are incident if they share at least one common end point. Note that an edge is incident to itself.

3.2 Variables

Every node $v \in V$ has two variables, m_v and h_v. The intent being that in a stable configuration m_v should point to the neighbor of v that v is matched with, while h_v is the weight of the edge (v, m_v). If the node v is not matched then m_v should be set to *null* and h_v to 0. During the execution of the algorithm a node v will use m_v and h_v to propose a matching with one of its neighbors by pointing to it. However, two neighbors v and w are only considered to be matched with each other if both $m_v = w$ and $m_w = v$. A matching M consists of all the matched edges in the graph, while the weight of M is the sum of the weights of the edges in M.

The algorithm also makes use of the set $N'(v)$ defined as follows: $N'(v)$ is a subset of $N(v)$ such that $u \in N'(v) \Leftrightarrow (u \in N(v) \wedge w(v, u) \geq h_u)$. That is, $N'(v)$ consists of all the neighbors of v that could achieve a match of equal or higher weight than their current one if they were to match with v. Note that $N'(v)$ is not a variable, but rather a set that v can compute during the execution of the algorithm.

3.3 The Algorithm

In this section we present our algorithm. It is quite simple, consisting of one function and one rule:

```
BestMatch(v)
    return $u : \max_{u \in N'(v) \cup \{null\}} w(v,u)$

SetMatch
    if $m_v \neq BestMatch(v) \bigvee h_v \neq w(v, m_v)$
    then
        $m_v = BestMatch(v)$
        $h_v = w(v, m_v)$
```

Algorithm 1.

The function **BestMatch(**v**)** returns the neighbor $u \in N'(u)$ such that $w(v,u)$ is maximal among all nodes in $N'(v)$, while the rule **SetMatch** sets m_v to point to the node returned by **BestMatch(**v**)** and also updates the value of h_v accordingly. Thus a node will always strive to match with the node in $N'(v)$ so that the resulting matched edge has maximal weight.

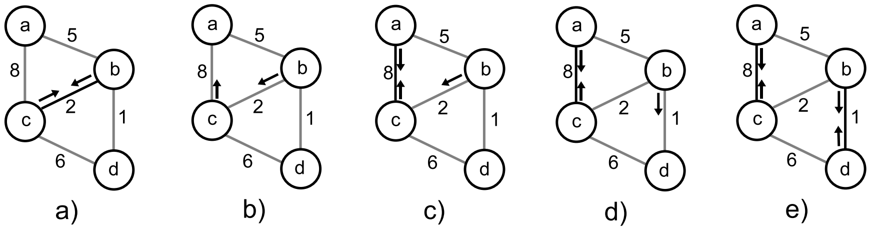


Fig. 2.

Figure 2 shows a possible execution of Algorithm 1. Starting from the configuration in Figure 2a, we observe that nodes b and c are matched. However, for both node b and c there exists an unmatched neighbor such that both the edge (a,b) and the edge (a,c) has a greater weight than (b,c). Assume now that c makes a move first and points to a (Figure 2b). Since (a,c) is the heaviest edge incident on a, the node a can now execute a move and point to c (Figure 2c) (note that the moves executed up till this point could have been done in any order). At this point b can no longer point to c, and since a is matched to c, it is left with d as its only unmatched neighbor. Thus b points to d (Figure 2d), and d, having now become privileged, points back (Figure 2e). Thus we have two pairs of matched nodes in the graph.

4 Proof of Correctness

In the following we will first show that when **Algorithm 1** has reached a stable configuration it also defines a matching that is a $\frac{1}{2}$-approximation to the maximum weight matching. We will then bound the number of steps the algorithm needs to stabilize both for the fair and for the adversarial distributed daemon. Note that the fair daemon is a subset of the adversarial one, thus any result for the latter also applies to the former.

4.1 Correct Stabilization

We now show that the algorithm, once stable, has found a $\frac{1}{2}$-approximation to the maximum weight matching problem. To do so, we first need the following observation which follows from the **BestMatch** function and from the predicate of **SetMatch**.

Observation 1. *In a stable configuration* $m_v \in N'(v) \cup \{null\}$ *and* $h_v = w(v, m_v)$ *for every node* $v \in V$*.*

The next step is to show that when stable, there is consensus in the graph as to which pairs of nodes are matched.

Lemma 1. *In a stable configuration* $m_v = u \Leftrightarrow m_u = v$ *for every edge* $(v, u) \in E$*.*

Proof. We note from Observation 1 that in a stable configuration $m_v \in \{N'(v), null\}$, $m_u \in \{N'(u), null\}$, $h_v = w(v, m_v)$, and $h_u = w(u, m_u)$. The rest of the proof is by contradiction.

We first show that $m_v = u \Rightarrow m_u = v$. Assume that $m_v = u$ while $m_u = y$ where $y \neq v$. Depending on the weights of (v, u) and (u, y) we have the following two possibilities: *i)* $w(v, u) > w(u, y)$, in which case u would be privileged, since v would be a better match for u than y. *ii)* $w(v, u) < w(u, y)$ in which case v is not a better match for u, thus $u \notin N'(v)$ and v is privileged. In either case, the algorithm is not stable. Using the same argument it also follows that $m_v = u \Leftarrow m_u = v$, thus proving the lemma. $\square$

In the following we refer to a *stable matching* as a set of edges M in a stable configuration such that for every edge $(x, y) \in E$, $(x, y) \in M \Leftrightarrow m_x = y$ and $m_y = x$. The next lemma shows that we cannot have a stable configuration where two adjacent nodes each have a matching of lower weight than that of the edge joining them.

Lemma 2. *In a stable configuration, for every edge* $(v, u) \in E$ *we have* $w(v, u) \leq \max(h_v, h_u)$*.*

Proof. From Observation 1 we know that in a stable configuration $h_x = w(x, m_x)$ for any node $x \in V$. The rest of the proof is by contradiction.

Assume that there exists an edge $(v, u) \in E$ in a stable configuration such that $w(v, u) > \max(h_v, h_u)$. Then since $w(v, u) > h_u$ we have $u \in N'(v)$. Also, since the current configuration is stable **BestMatch**(v) returns a node $x \in N'(v)$ such that $m_v = x$ and $w(v, x) > w(v, u)$. But since $h_v < w(v, u)$ it follows that $w(v, m_v) < w(v, u)$ and we must have $m_v \neq$ **BestMatch**(v) contradicting that the solution is stable. The same argument can also be used to show that u is privileged. $\square$

Corollary 1. *Let M be any stable matching given by Algorithm 1. Then every edge $(v, u) \in E$ is incident on at least one edge $(x, y) \in M$ such that $w(v, u) \leq w(x, y)$.*

Proof. Let (v, u) be any edge in E in a stable configuration. From Lemma 2 we know that $w(v, u) \leq \max(h_v, h_u)$. Assume (without loss of generality) that $h_u = \max(h_v, h_u)$. Then since $w(v, u) > 0$ we must also have $h_u > 0$ and it follows that u is matched in M. From Observation 1 we know that $h_u = w(u, m_u)$ implying that $w(v, u) \leq w(u, m_u)$ and the result follows. □

This enables us to show the main result of this section.

Theorem 1. *Any stable matching M given by Algorithm 1 is a $\frac{1}{2}$-approximation to the maximum weighted matching problem.*

Proof. Let M^* be a maximum weighted matching for G. From Corollary 1 we know that it is possible to associate every edge $(v, u) \in M^*$ with exactly one incident edge $(x, y) \in M$ such that $w(v, u) \leq w(x, y)$. Since at most two edges from M^* can be associated with each edge of M we have that $2w(M) \geq w(M^*)$ and the result follows. □

We note that it is fairly straight forward to show that the matching produced by Algorithm 1 is in fact exactly the same matching as the sequential greedy algorithm would give.

4.2 Convergence

We now show that Algorithm 1 stabilizes from any given starting configuration. Specifically, we will be looking at the rate of convergence first using the distributed adversarial model and then using the distributed fair model. The distributed model is the most general model, where a non-empty subset of the privileged nodes makes a move during each time step. Note that if at each time step only one node is allowed to make a move, this model is identical to the sequential model.

4.3 The Distributed Adversarial Model

We proceed to bound the number of time steps needed before Algorithm 1 stabilizes under an adversarial daemon. The proof is based on counting the number of moves needed before at least one node v stabilizes permanently. We then repeat the argument recursively for the remaining set of nodes $A = V - \{v\}$ obtaining our desired bound. First we show that if parts of the graph has stabilized permanently then there exists at least one node that can at most make two more moves.

Lemma 3. *Given a set $A \subseteq V$ where the nodes in A are the only nodes in the graph permitted to move. Then there exists at least one node in A that can make at most two moves.*

Proof. Let (v, u) be the heaviest edge in the set $\{(x, y) \, \forall \, x, y \in A : (x, y) \in E\} \cup \{(a, b) \, \forall \, a \in A, \; b \in V \backslash A : (a, b) \in E \wedge w(a, b) \geq h_b\}$. We assume without loss of generality that at least $v \in A$.

If $u \in V \backslash A$ then $w(v, u) \geq h_u$ and the only move v can make is one that sets $m_v = u$ (provided that this is not already the case). Since there does not exist any edge incident on v in A that is heavier than (v, u) and since u cannot make a move, it follows that v will not move again.

If $u \in A$ then there are two possibilities: *i)* $h_u \leq w(v, u)$ or *ii)* $h_u > w(v, u)$. In the first case it follows that the only move v can make is to match with u before becoming permanently stable (again assuming that only nodes in A may execute moves).

In the second case, u will need to make one move to correct its h-value (which is incorrect). During this time step v can also make a move. Following this move $h_u \leq w(v, u)$, and as in case *i)*, the only move v can make is to match to u, again becoming permanently stable. Thus v has executed at most two moves not counting any moves executed before u's first move. If v executes any move before u, we can simply switch the roles of v and u and repeat the argument.

In either of the above cases we see that there has to exist at least one node in A that can move at most twice. □

Based on Lemma 3 we can now give a recursive formula for the number of moves that Algorithm 1 can execute on the remaining nodes that have not yet stabilized permanently.

Lemma 4. *Let $A \subseteq V$ be a set where $|A| = k$ and let $t(k)$ be the maximum number of moves needed for A to stabilize given that only nodes in A are permitted to move. Then $t(k) \leq 3 \cdot t(k-1) + 2$.*

Proof. Recall from Lemma 3 that there exists at least one node $v \in A$ that can execute at most two moves. From the premise of the lemma we know that at most $t(k-1)$ moves can be made by the nodes in $A \backslash \{v\}$ before v makes its first move. Following this, another $t(k-1)$ moves can be made before v's second move. And finally at most $t(k-1)$ subsequent moves can be made as a result of v's second and final move. Thus at most $3 \cdot t(k-1) + 2$ moves can be made in a set of size k. □

Theorem 2. *Algorithm 1 stabilizes after $O(3^n)$ time steps under the distributed adversarial model.*

Proof. From Lemma 4 we know that the time needed for a subset of nodes of size k to become stable is $t(k) \leq 3 \cdot t(k-1) + 2$. Since $t(1) = 1$ it follows that the maximum number of moves needed to ensure stabilization is $t(n) \leq 2 \cdot 3^{n-1} - 1$. Thus the number of time steps used by the algorithm is $O(3^n)$. □

For the sequential adversarial model the bound from Theorem 2 can be improved to $O(2^n)$. This follows by noting that for any unstable configuration there exists at least one node that can make at most one more move before becoming permanently stable. We omit the details.

4.4 The Distributed Fair Model

We now look at the convergence rate of Algorithm 1 assuming a distributed model under a fair daemon, and prove that the algorithm stabilizes after at most $2 \cdot |M| + 1$ rounds where M is the final matching found by the algorithm. We remind the reader that in the distributed fair model, complexity is measured in rounds, where one round is a minimum period of time during which every node that was privileged at the start of the round has either made at least one move or at some point become non-privileged.

We first note that since the distributed fair model is a subset of the distributed adversarial model it follows from Theorem 2 that Algorithm 1 will eventually stabilize with a matching M that is a $\frac{1}{2}$-approximation to the maximum weighted matching problem. Thus it is meaningful to refer to the resulting matching M. We now proceed to bound the number of rounds before Algorithm 1 stabilizes.

Lemma 5. *After at most one round $m_v \in N(v) \cup \{null\}$ and $h_v = w(v, m_v)$ for every $v \in V$.*

Proof. Recall from the predicate of **SetMatch** that a node v is privileged if its m-value is incorrect or if $h_v \neq w(v, m_v)$. In either case **SetMatch** will set m_v to some node $u \in N'(v) \cup \{null\}$ and h_v to $w(v, m_v)$. Since $N'(v) \subseteq N(v)$ the result follows. □

Note that one cannot guarantee that $m_v \in N'(v) \cup \{null\}$ after the first round as $N'(v)$ might change after v has made its move.

Lemma 6. *After at most two rounds, the heaviest edge $(v, u) \in E$ is part of M. Furthermore, the algorithm will never cause (v, u) to leave M.*

Proof. From Lemma 5 we know that after the first round every node has a correct h-value and $m_v \in N(v) \cup \{null\}$. If (v, u) is the heaviest edge in G it follows that $h_v \leq w(v, u)$ and $h_u \leq w(v, u)$, implying that $u \in N'(v)$ and $v \in N'(u)$. Thus if $m_v \neq u$ then v is privileged and if $m_u \neq v$ then u is privileged. In either case, at most one more round is needed before v and u are matched. This will happen since neither of the two nodes has a neighbor that can give a better matching than the other node.

Since there does not exist any edge in the graph with weight greater than (v, u) neither v nor u will become privileged again, and thus the edge (v, u) will never leave the matching. □

We can now give the final bound on the number of rounds needed before Algorithm 1 stabilizes.

Theorem 3. *Algorithm 1 converges after at most $2 \cdot |M| + 1$ rounds.*

Proof. Let $e_1, e_2, \ldots, e_{|M|}$ be the edges in M sorted in descending order. We show by induction that $e_1, e_2, \ldots, e_i$ are all part of the matching after at most $2i$ rounds, and that they will not leave M in subsequent rounds.

The base case is covered by Lemma 6 and it follows that e_1 must be the heaviest edge in E.

For the induction step assume that the algorithm has run for at most $2 \cdot (i-1)$ rounds and that the edges $M_{i-1} = \{e_1, e_2, \ldots, e_{i-1}\}$ have been permanently added to M. It follows that we do not need to consider any edge incident on M_{i-1} for future inclusion in M.

Let (v, u) be the heaviest edge not incident on M_{i-1}. Then by the same argument as in the proof of Lemma 6 it follows that within the next two rounds (v, u) will be permanently added to M. Since no edge of weight greater than $w(v, u)$ will be added to M in subsequent time steps it follows that $e_i = (v, u)$.

From the above is follows that at most $2 \cdot |M|$ rounds are needed to find the matching. However, since some nodes may not be part of the matching, one more round may be needed for these nodes to right any incorrect variables they may have. Thus the algorithm requires at most $2 \cdot |M| + 1$ rounds. □

It should be noted that the size of a matching in a graph $G = (V, E)$ cannot exceed $|V|/2$. Since this would imply that every node is matched, the number of rounds needed for the algorithm to stabilize in this case is at most $2 \cdot |M| = |V|$, not $2 \cdot |M| + 1$.

As was noted in Section 2, both the synchronous and sequential fair models are sub variants of the distributed fair model. Thus the bound from Theorem 3 also holds for either of these models.

5 Conclusion

We have presented the first self-stabilizing algorithm for computing a $\frac{1}{2}$- approximation for the maximum weighted matching problem. In addition to being short and simple, the complexity of the algorithm is linear over the number of nodes in the graph when using a distributed fair daemon. Furthermore the algorithm requires only two variables per node.

It is worth noting that while we in Section 3.1 require that all IDs are unique, this is in fact not needed. The algorithm requires only that every ID is unique within distance 2. That is, no node can have two or more neighbors with the same ID. On the same note, we do not need to create a global ordering of the edges in the graph. While a global ordering was used to make the proofs more understandable, a local ordering is sufficient for the algorithm.

One common method for improving the approximation ratio of a matching is by the use of augmenting paths. An augmenting path is a path such that exactly every other edge in the path is part of the current matching. The length of an augmenting path is the number of unmatched edges in it. It is well known that if a matching does not contain an augmenting path of length i then the matching is a $\frac{i}{i+1}$-approximation. Thus it would be of interest to see if it is possible to design self-stabilizing algorithms that can detect and correct augmenting paths of length larger than $i = 1$ as is done in the current paper, while at the same time limiting the number of variables and stabilization time. One possible way of doing this could be to use the same kind of augmentations as is used in the sequential linear time algorithm by Vinkelmeier and Hougardy [10] to produce a solution with approximation ratio arbitrarily close to 2/3.

References

1. Chattopadhyay, S., Higham, L., Seyffarth, K.: Dynamic and self-stabilizing distributed matching. In: PODC 2002. Proceedings of the twenty-first annual symposium on Principles of distributed computing, pp. 290–297. ACM Press, New York (2002)
2. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
3. Goddard, W., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Self-stabilizing distributed algorithm for strong matching in a system graph. In: HiPC 2003. LNCS (LNAI), vol. 2913, pp. 66–73. Springer, Heidelberg (2003)
4. Goddard, W., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In: IPDPS, p. 162 (2003)
5. Gradinariu, M., Johnen, C.: Self-stabilizing neighborhood unique naming under unfair scheduler. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, Springer, Heidelberg (2001)
6. Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Maximal matching stabilizes in time o(m). Inf. Process. Lett. 80(5), 221–223 (2001)
7. Hsu, S.-C., Huang, S.-T.: A self-stabilizing algorithm for maximal matching. Inf. Process. Lett. 43(2), 77–81 (1992)
8. Manne, F., Mjelde, M., Pilard, L., Tixeuil, S.: A new self-stabilizing maximal matching algorithm. In: Sirocco 2007. Proceedings of the 14^{th} International Colloquium on Structural Information and Communication Complexity, pp. 96–108. Springer, Heidelberg (2007)
9. Tel, G.: Maximal matching stabilizes in quadratic time. Inf. Process. Lett. 49(6), 271–272 (1994)
10. Vinkemeier, D.E.D., Hougardy, S.: A linear-time approximation algorithm for weighted matchings in graphs. ACM Trans. Algorithms 1(1), 107–122 (2005)

Self-stabilization and Virtual Node Layer Emulations

Tina Nolte and Nancy Lynch*

MIT CSAIL, Cambridge, MA, USA

Abstract. We present formal definitions of stabilization for the Timed I/O Automata (TIOA) framework, and of emulation for the timed *Virtual Stationary Automata* programming abstraction layer, which consists of mobile clients, virtual timed machines called virtual stationary automata (VSAs), and a local broadcast service connecting VSAs and mobile clients. We then describe what it means for mobile nodes with access to location and clock information to emulate the VSA layer in a self-stabilizing manner. We use these definitions to prove basic results about executions of self-stabilizing algorithms run on self-stabilizing emulations of a VSA layer, and apply these results to a simple geographic routing algorithm running on the VSA layer.

Keywords: self-stabilization, virtual stationary automata, virtual node layer, geocast, abstraction layer emulation, mobile ad-hoc networking, TIOA.

1 Introduction

A system with no fixed infrastructure in which mobile clients may wander in the plane and assist each other in forwarding messages is called an ad-hoc network. The task of designing algorithms for constantly changing networks is difficult. Highly dynamic networks, however, are becoming increasingly prevalent, especially in the context of pervasive and ubiquitous computing, and it is therefore important to develop and use techniques that simplify this task.

In addition, nodes in these networks may suffer from crashes or corruption faults, which cause arbitrary changes to their program states. Self-stabilization [2,3] is the ability to recover from an arbitrarily corrupt state. This property is important in long-lived, chaotic systems where certain events can result in unpredictable faults. For example, transient interference may disrupt wireless communication, violating our assumptions about the broadcast medium.

In this paper, we first develop a basic formal theory of stabilization for the Timed I/O Automata (TIOA) framework [11], used to describe and analyse timed systems (Section 2). We then describe the abstract timed *Virtual Stationary Automata (VSA)* layer presented in [6], used to simplify algorithm design for

* Research supported by AFRL contract number F33615-010C-1850, DARPA/AFOSR MURI contract number F49620-02-1-0325, NSF ITR contract number CCR-0121277, and DARPA-NEST contract number F33615-01-C-1896.

mobile networks (Section 3). The VSA layer is a *virtual* infrastructure, consisting of mobile client automata, timing-aware and location-aware machines at fixed locations (VSAs), and a local broadcast service connecting VSAs and clients.

We introduce a formal theory of self-stabilizing emulation of a VSA layer in Section 4. This provides proof obligations required to conclude that an algorithm successfully emulates the VSA layer, allowing an application programmer to write programs for the VSA layer without worrying about its implementation.

We finally show that a self-stabilizing VSA layer emulation running on the physical layer and instantiated with a self-stabilizing VSA layer service implementation, is a stabilizing implementation of the service on the physical layer (Section 5). This separates the reasoning about stabilization properties of a VSA layer emulation algorithm from those of the VSA layer service being run. We apply these results to a simple self-stabilizing VSA layer algorithm that provides geographic routing, a version of which appeared in [7].

Virtual Stationary Automata programming layer. In prior work [6,5,4], we developed a notion of "virtual nodes" for mobile ad hoc networks. A virtual node is an abstract, relatively well-behaved active node implemented using less well-behaved real physical nodes. The GeoQuorums algorithm [5] proposes storing data at fixed locations; however it supports only atomic objects, rather than general automata. A more general mobile automaton is suggested in [4].

The static infrastructure we use in this paper includes fixed, timed virtual automata with an explicit notion of real time, called *Virtual Stationary Automata* (VSAs), distributed at known locations over the plane [6] and connected as in a wired network. Each VSA represents a predetermined geographic area and has broadcast capabilities similar to those of the physical nodes, allowing nearby VSAs and physical nodes to communicate. VSAs have access to *virtual* clocks, guaranteed to not drift too far from real time; many algorithms depend significantly on timing, and it is reasonable to assume that many mobile nodes have access to reasonably synchronized clocks. This layer provides mobile nodes with a fixed timed virtual infrastructure, reminiscent of more traditional and better understood wired networks, with which to coordinate their actions.

Our clock-equipped VSA layer is emulated by physical nodes. Each physical node is periodically told its region by a reliable GPS oracle. A VSA for a particular region is then emulated by a subset of the physical nodes in its region: the VSA state is maintained in the memory of the physical nodes emulating it, and the physical nodes perform VSA actions on behalf of the VSA. If no physical nodes are in the region, the VSA fails; if physical nodes later arrive, it restarts.

The implementation in [6] was designed to be self-stabilizing. This paper provides the necessary formal machinery to both formally express and prove that such an implementation is a VSA layer emulation and that it is self-stabilizing.

Geographic routing. We use a basic geographic routing service [7], based on greedy depth-first search, to demonstrate concepts we introduce in this paper. Geocast algorithms [14,1], GPSR [10], AFR [13], GOAFR+ [12], and polygonal broadcast [8] are other examples of greedy geographic routing algorithms, forwarding messages to the neighbor geographically closest to the destination.

2 Definitions

We start by defining the Timed I/O Automata modeling framework for timed systems, and then outline basic definitions and facts with respect to stabilization.

2.1 Timed I/O Automata (TIOA)

Here we define Timed I/O Automata (TIOA) terminology used in this paper. TIOAs are nondeterministic state machines whose state can change in two ways: instantaneously through a discrete transition, or according to a trajectory describing the evolution, possibly continuous, of variables over time. The TIOA framework can be used to carefully specify and analyse timed systems. (Additional details can be found in [11].)

A *valuation* for a set V of variables is a function mapping each variable $v \in V$ to a value in $type(v)$. The set of such valuations is $val(V)$.

A *trajectory*, τ, for V is a function mapping a left-closed interval of time starting at 0 to the set of valuations for V, such that for $v \in V$, τ restricted to v is in the dynamic type of v.

- τ is *closed* if $domain(\tau)$ is both left and right-closed.
- $\tau.fstate$ is the first valuation of τ, and, for τ closed, $\tau.lstate$ is the last.
- The limit time of τ, $\tau.ltime$, is the supremum of $domain(\tau)$.
- The concatenation, $\tau\tau'$, of trajectories τ and τ', τ closed, is the trajectory resulting from the pasting of τ', shifted by $\tau.ltime$, to the end of τ.

A *Timed I/O Automaton (TIOA)*, $\mathcal{A} = (X, Q, \Theta, I, O, H, \mathcal{D}, \mathcal{T})$, consists of:

– Set X of internal variables.
– Set $Q \subseteq val(X)$ of states.
– Set $\Theta \subseteq Q$ of start states, nonempty.
– Sets I of input actions, O of output actions, and H of internal actions, each disjoint. $A = I \cup O \cup H$ is all actions. $E = I \cup O$ is all external actions.
– Set $\mathcal{D} \subseteq Q \times A \times Q$ of discrete transitions.
  We say action a is enabled in state x if $(x, a, x') \in \mathcal{D}$, for some $x' \in X$. We require $\mathcal{A}$ be input-enabled (every input action is enabled at every state).
– Set $\mathcal{T} \subseteq$ trajectories of Q. We require:
  - For every state x, the point trajectory for x must be in $\mathcal{T}$,
  - For every $\tau \in \mathcal{T}$, every prefix and suffix of τ is in $\mathcal{T}$,
  - For every sequence of trajectories in $\mathcal{T}$, where for every τ_i but the last, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$, the concatenation of the trajectory sequence is also in $\mathcal{T}$, and
  - Time-passage enabling: for every state x, there exists a $\tau \in \mathcal{T}$ where $\tau.fstate = x$, and either $\tau.ltime = \infty$ or τ is closed and some $l \in H \cup O$ is enabled in $\tau.lstate$.

Two TIOAs $\mathcal{A}$ and $\mathcal{B}$ are *compatible* if they share no internal variables, and their internal actions are not actions of the other. Two compatible TIOAs $\mathcal{A}$ and $\mathcal{B}$ can be composed into a new TIOA $\mathcal{A}\|\mathcal{B}$, which has $\mathcal{A}$ and $\mathcal{B}$ as components where an action performed in one component that is an external action of the other component is also performed in the other component.

Given a set A of actions and a set V of variables, an (A, V)-*sequence* is an alternating sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \cdots$ where: (a) Each a_i is an action in A, (b) Each τ_i is a trajectory for V, (c) If α is finite, it ends with a trajectory, and (d) Each τ_i but the last is closed.

- α is *closed* if it is a finite sequence and its final trajectory is closed.
- The limit time of α, $\alpha.ltime$, is the sum of limit times of α's trajectories.
- The concatenation, $\alpha\alpha'$, of two (A, V)-sequences α and α', α closed, is α followed by α', where the last trajectory of α is concatenated to the first trajectory of α'.
- For sets of actions A and A', and sets of variables V and V', the (A', V')-*restriction* of an (A, V)-sequence α, written $\alpha\lceil(A', V')$, is the (A', V')-sequence that results from projecting the trajectories of α on variables in V', removing actions not in A', and concatenating all adjacent trajectories.

An *execution fragment* of a TIOA $\mathcal{A}$ is an (A, V)-sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \cdots$, where each τ_i is a trajectory in $\mathcal{T}$, and if τ_i is not the last trajectory of α, then $(\tau_i.lstate, a_{i+1}, \tau_{i+1}.fstate) \in \mathcal{D}$. The set of execution fragments of $\mathcal{A}$ starting from a state in some $S \subseteq Q$ is referred to as $frags^S_{\mathcal{A}}$.

An execution fragment of $\mathcal{A}$, α, is an *execution* of $\mathcal{A}$ if $\alpha.fstate$ is in Θ. The set of executions of $\mathcal{A}$ is referred to as $execs_{\mathcal{A}}$.

A state of $\mathcal{A}$ is *reachable* if it is the last state of some closed execution of $\mathcal{A}$. The set of reachable states of $\mathcal{A}$ is referred to as $reachable_{\mathcal{A}}$.

A *trace* (external behaviour) of an execution fragment α of $\mathcal{A}$, $trace(\alpha)$, is α restricted to external actions of $\mathcal{A}$ and trajectories over the empty set of variables. $traces_{\mathcal{A}}$ is the set of traces of executions of $\mathcal{A}$.

2.2 Stabilization

We define stabilization in terms of sets of (A, V)-sequences. This is general enough to talk about stabilization of traces and execution fragments of TIOAs, and about stabilization of transformed versions of these (A, V)-sequences.

Definition 1. *Let α and α' be (A, V)-sequences, and t be in $\mathbb{R}^{\geq 0}$. α' is a* t-suffix *of α if a closed (A, V)-sequence α'' exists where $\alpha''.ltime = t$ and $\alpha = \alpha''\alpha'$.*

Definition 2. *Let $\alpha = \alpha''\alpha'$ be an (A, V)-sequence and t be in $\mathbb{R}^{\geq 0}$. α' is a* state-matched t-suffix of *α if it is a t-suffix of α, and $\alpha'.fstate = \alpha''.lstate$.*

Lemma 1. *Let α be an (A, V)-sequence and t be in $\mathbb{R}^{\geq 0}$ where either $t < \alpha.ltime$, or $t = \alpha.ltime$ and α is closed. A state-matched t-suffix of α exists.*

For the following definitions, let B be a set of (A^B, V)-sequences, C be a set of (A^C, V)-sequences, and D be a set of (A^D, V)-sequences, where A^B, A^C, and A^D are sets of actions, and V is a set of variables.

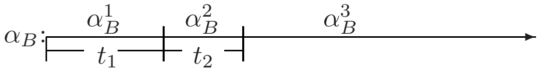
Definition 3. *Let t be a non-negative real. B* stabilizes in time *t* to *C if any state-matched t-suffix α of a sequence in B is a sequence in C.*

Since executions and traces of TIOAs are (A, V)-sequences, the above definition can be used to talk about executions or traces of one TIOA stabilizing to executions or traces of some other TIOA. The following lemma is a general result that can be used to show, for example, that if executions of one TIOA stabilize to those of another then its traces also stabilize to traces of the other.

Lemma 2. *Let A be a set of actions and V' a set of variables. If B stabilizes to C in time t, then $\{\alpha\lceil(A, V')|\alpha \in B\}$ stabilizes to $\{\alpha\lceil(A, V')|\alpha \in C\}$ in time t.*

Lemma 3 (Transitivity). *If B stabilizes to C in time t_1, and C stabilizes to D in time t_2, then B stabilizes to D in time $t_1 + t_2$.*

Proof sketch: Assume B stabilizes to C in time t_1, and C stabilizes to D in time t_2. Consider a sequence $\alpha_B = \alpha_B^1\alpha_B^2\alpha_B^3$ in B, where $\alpha_B.ltime \geq t_1 + t_2$, $\alpha_B^2\alpha_B^3$ is a state-matched t_1-suffix of α_B, and α_B^3 is a state-matched $t_1 + t_2$-suffix of α_B.



We will show that α_B^3 is in D:

Since B stabilizes to C in time t_1, $\alpha_B^2\alpha_B^3$ is in C. Also, since $\alpha_B^3.fstate = \alpha_B^2.lstate$, α_B^3 is a state-matched t_2-suffix of $\alpha_B^2\alpha_B^3$. Since C stabilizes to D in time t_2, and α_B^3 is a state-matched t_2-suffix of a sequence in C, α_B^3 is in D.

We conclude that B stabilizes to D in time $t_1 + t_2$. □

The following definitions capture the idea of a TIOA being self-stabilizing when composed with another TIOA, allowing us to write algorithms that can be started in an arbitrary state but take advantage of separate oracles, in order to eventually reach some legal state of the composed automaton. (The idea of a TIOA stabilizing given another can be used to arrive at layering results similar to those of fair composition, described in [3], showing that under certain conditions, if you have a self-stabilizing implementation A of a service that's used by a self-stabilizing implementation B of a higher level service, then B using A is still stabilizing.) We begin by defining a function that takes a TIOA and a state set L and returns the same TIOA with its start state set changed to L.

Definition 4. *Let $\mathcal{A}$ be any TIOA and L be any nonempty subset of $Q_{\mathcal{A}}$. Then $changeStart(\mathcal{A}, L)$ is defined to be $\mathcal{A}$ except with $\Theta_{changeStart(\mathcal{A},L)} = L$. We use notation $U(\mathcal{A})$ for $changeStart(\mathcal{A}, Q_{\mathcal{A}})$ (or $\mathcal{A}$ started in an arbitrary state), and $R(\mathcal{A})$ for $changeStart(\mathcal{A}, reachable_{\mathcal{A}})$ (or $\mathcal{A}$ started in a reachable state).*

Lemma 4. *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, $L \subseteq Q_{\mathcal{A}}, L' \subseteq Q_{\mathcal{O}}$, and $L'' \subseteq Q_{\mathcal{A}\|\mathcal{O}}$. Then:*

1. *$changeStart(\mathcal{A}, L)\|changeStart(\mathcal{O}, L') = changeStart(\mathcal{A}\|\mathcal{O}, L \times L')$.*
2. *$frags_{\mathcal{A}}^{L} = execs_{changeStart(\mathcal{A},L)}$.*
3. *$frags_{changeStart(\mathcal{A},L)\|changeStart(\mathcal{O},L')}^{L''} = frags_{\mathcal{A}\|\mathcal{O}}^{L''}$*
4. *For any $\alpha\alpha' \in traces_{U(\mathcal{A})\|R(\mathcal{O})}$, $\alpha' \in traces_{U(\mathcal{A})\|R(\mathcal{O})}$.*

Definition 5. *Let $\mathcal{A}$ be a TIOA, and $L \subseteq Q_{\mathcal{A}}$. L is a* legal set *for $\mathcal{A}$ if:*

1. *For every $(x, a, x') \in \mathcal{D}_{\mathcal{A}}$, if $x \in L$ then $x' \in L$.*
2. *For every closed $\tau \in \mathcal{T}_{\mathcal{A}}$, if $\tau.fstate \in L$ then $\tau.lstate \in L$.*

Definition 6. *Let $\mathcal{O}$ and $\mathcal{A}$ be compatible TIOAs, and L be a legal set for $\mathcal{A}\|\mathcal{O}$. $\mathcal{A}$* self-stabilizes in time t with respect to L and given $\mathcal{O}$ *if $execs_{U(\mathcal{A})\|\mathcal{O}}$ stabilizes in time t to $frags^{L}_{\mathcal{A}\|\mathcal{O}}$.*

Notice in the definition above that when $\mathcal{O} = R(\mathcal{O}')$ for some TIOA $\mathcal{O}'$, the TIOA $\mathcal{A}$ can recover from a corruption fault, where $\mathcal{A}$'s state can be changed arbitrarily: the resulting state s is in $Q_{\mathcal{A}} \times reachable_{\mathcal{O}'}$, meaning any execution fragment starting from s is in $execs_{U(\mathcal{A})\|\mathcal{O}}$.

3 Physical Layer and VSA Layer System Models

The physical layer consists of a bounded, tiled region of the plane, where mobile physical (real) nodes are deployed. These nodes are TIOAs susceptible to crash failures and restarts, and with access to a local clock. They also have access to a local broadcast service and a reliable *RW* (real world or GPS) automaton that models moves, failures, and restarts of the physical nodes and real-time. This layer can be used to emulate the VSA layer (we define emulation in Section 4).

The *Virtual Stationary Automata* abstraction layer [6] includes a modified version of *RW* called *RW'*, client nodes that correspond to physical nodes, virtual stationary automata (VSAs) the physical nodes emulate, and a local

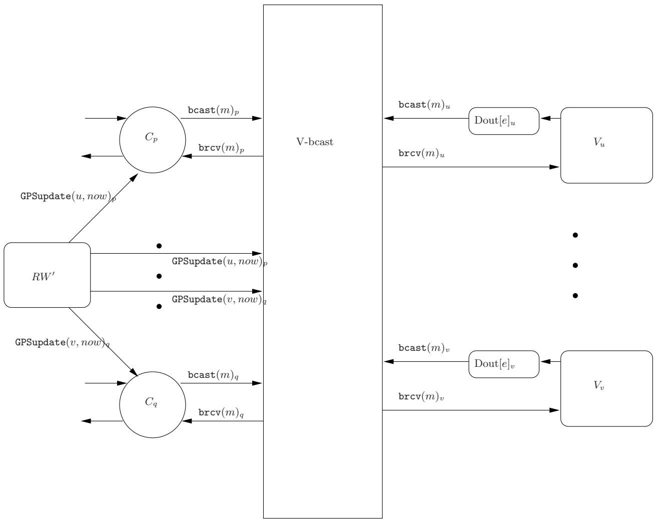


Fig. 1. Virtual Stationary Automata layer. VSAs and clients communicate locally using V-bcast. VSA outputs may be delayed in Dout.

broadcast service between them, V-bcast, similar to that of the physical layer (see Figure 1). Since each physical layer component has a corresponding virtual layer component, this section describes the more complicated VSA layer in depth and explains connections or differences from the physical layer as appropriate.

3.1 Network Tiling

The deployment space of the network is a fixed, closed, bounded portion of the two-dimensional plane called R. R is partitioned into known connected *regions*, with unique ids drawn from the set of region identifiers U. Distances between points in the same region are bounded by a constant r_{virt}. We also define a neighbor relation *nbrs* on ids from U: *nbrs* holds for any distinct region ids u and v where the distance between points in u and v is bounded by r_{virt}.

Connection to physical layer: The constant r_{virt} is the broadcast radius of the underlying physical nodes. The network tiling then ensures that any two physical nodes in the same or neighboring regions will be able to communicate.

3.2 Real World

Real world TIOA RW of the physical level models system time and mobile node region locations, failures, and restarts. It maintains a variable, *now*, that is considered the true system time, and two mappings, *locReg* and $fail$.

locReg, mapping the set of physical node ids, P, to U, indicates the region where a particular mobile node is located. RW outputs a $\mathsf{GPSupdate}(u, now)$ at a mobile node whenever the node changes regions, and every ϵ_{sample} time in addition, informing the node of the node's new region and the current time.

$fail$, mapping the physical node ids, P, to Booleans, indicates whether a physical node is failed. RW outputs fail_p at a node when it fails, setting $fail(p)$ to true, and outputs $\mathsf{restart}_p$ when the node restarts, setting $fail(p)$ to false. A fail only occurs at a non-failed node, and a restart only occurs at a failed node.

The real world TIOA RW' of the virtual level is an extension of RW that is also able to fail and restart regions in U (corresponding to failing and restarting VSAs). The mapping $fail$ is extended to also map region identifiers in U to a Boolean, and fail and restart actions similar to those for mobile nodes are added. In addition to the *locReg* and $fail$ variables of RW, RW' also maintains a *log* history variable, in which the execution of RW' up to *now* is stored.

RW and RW' outputs are also inputs to physical level broadcast and V-bcast services, respectively.

RW' is parameterized by failure and recovery conditions for regions, expressed as two precondition (allowed-to-happen) predicates, $failprec$ and *restartprec*, and two stopping condition (must-happen) predicates, $failstop$ and *restartstop*, each of which is parameterized by region id. These predicates are allowed to be over the variable *log* and the current time, *now*, and we require that for any region u, if $failstop[u]$ holds, then $restartprec[u]$ does not, and if $restartstop[u]$ holds, then $failprec[u]$ does not. Given these, the precondition of a fail_u, $u \in U$, action will be $\sim failed(u) \wedge failedprec[u]$. Similarly, the precondition of a $\mathsf{restart}_u$

action will be $failed(u) \wedge restartprec[u]$. The associated stopping conditions are $\sim failed(u) \wedge failstop[u]$, and $failed(u) \wedge restartstop[u]$.

Example predicates: One suitable $failprec$ for a region is that some failure or leave of a client occurred at the current time. The stopping condition can be that there are no clients in the region or none of the clients have been in the region for at least d time. (Region failures only occur in reaction to some mobile node fail or leave, and are guaranteed to happen if there are no clients populating their regions that have been around for some time.) For *restartprec*, we can require that there be some client in the region that's been in the region for at least $2d$ time. The stopping condition can be that the last client fail or leave was at least e time ago and there is some client that has been in the region at least $2d$ time. (Region restarts only occur if some mobile node has been around long enough, and are guaranteed to happen if none of the mobile nodes in the region have failed or left for some time. Constants e and d are explained in Section 3.5.)

3.3 Client Nodes

For each p in the set of physical node ids P, we assume a C_p from a set of TIOAs, $CProgram_p$. Each C_p has access to a local clock, *now*. Clients receive accurate information from the reliable GPS oracle, RW'. A $\mathsf{GPSupdate}(u, now)_p$ happens at C_p each time the client enters the system or changes region, indicating to the client the region u where it is currently located and the current system time. It also occurs every ϵ_{sample} time at each client. Clients accept this *now* real-time clock value as the value of their own local clock. For simplicity, this local variable progresses at the rate of real time. This implies that, outside of client failures and arbitrary initial states, the local value of *now* will equal real time.

C_p has access to V-bcast (see Section 3.5), allowing it to communicate with its own and neighboring regions' VSAs and clients with $\mathsf{bcast}(m)_p$ and $\mathsf{brcv}(m)_p$.

Clients can suffer crash failures. After a crash, a client performs no locally-controlled actions until restarted. If restarted, it starts from an initial state.

Additional arbitrary external interface and environment actions and local state used by algorithms running at the client are allowed. (Environment actions are external actions that are not actions of any other system component.)

Connection to physical layer: Each client node is hosted by its corresponding physical node. In addition, RW' inputs occur at a client node exactly when corresponding RW inputs occur at the physical node.

3.4 Virtual Stationary Automata (VSAs)

Here we describe VSAs; details on their implementation can be found in [6].

An abstract VSA is a clock-equipped virtual machine at a region in the network. We formally describe a timed machine for region u, V_u, as a TIOA from a set of TIOAs, $VProgram_u$. The state of V_u is referred to as *vstate* and is assumed to include a variable corresponding to real time, *vstate.now*. V_u's external interface is restricted to include only stopping failures, restarts, and the ability to broadcast and receive messages using V-bcast.

The VSA layer provides a *delay-augmented TIOA*, an augmentation of V_u with timing perturbations, represented with buffers $\mathrm{Dout}[e]_u$, composed with V_u's outputs, with the V_u outputs then hidden. The buffer delays messages by a nondeterministically-chosen time $[0, e]$. Programs must take into account e, as they would message delay. Also, a failure of region u also means a failure of $\mathrm{Dout}[e]_u$, clearing its buffer of messages.

Connection to physical layer: While an emulation of V_u would ideally be identical to a legitimate execution of V_u, an abstraction must reflect that, due to message delays or node failure, the emulation might be behind real time, appearing to be delayed in performing outputs by up to some time. This time is the e referred to with respect to Dout.

Since we emulate a VSA using physical nodes, its interface must be emulatable by them. This is why a VSA's external interface is restricted to include only the various failure and broadcast-related inputs and outputs. Also, its failures can be defined in terms of physical node fail status and movement, as described by the fail and restart predicates in Section 3.2.

3.5 Local Broadcast Service (V-bcast)

Communication is in the form of local broadcast service V-bcast, with message delay d. It allows communication between VSAs and clients in the same or neighboring regions. The service allows the broadcasting and receiving of message m at each port $i \in P \cup U$ through $\mathsf{bcast}(m)_i$ and $\mathsf{brcv}(m)_i$. It also receives GPSupdate, fail, and restart inputs from RW', informing the service of the location and failure status of nodes in the network.

V-bcast guarantees two properties: integrity and reliable local delivery. *Integrity* guarantees that for any $\mathsf{brcv}(m)_i$ that occurs, a $\mathsf{bcast}(m)_j, j \in P \cup U$ previously occurred. *Reliable local delivery* guarantees, roughly, that a transmission will be received by nearby ports: If port i, where i is a client or VSA port in any region u, transmits a message, then every port j, whether a client or VSA, in region u or neighboring regions during the entire time interval starting at transmission and ending d later receives the message by the end of the interval.

Connection to physical layer: V-bcast is implemented using the underlying physical nodes' broadcast capabilities. We assume that the physical layer broadcast satisfies, for physical nodes, integrity and reliable local delivery between physical nodes within distance r_{virt} of each other. The message delay d of V-bcast is the message delay of the underlying broadcast.

4 Self-stabilizing Emulations

In Section 3, we described the VSA layer and noted that it can be provided by a physical layer's broadcast and GPS-enabled physical nodes running an emulation algorithm. Here we formally define what it means for an algorithm to emulate an abstract VSA layer. We begin by providing definitions for a VSA layer algorithm and a VSA layer instantiation.

Definition 7. *A V-algorithm, alg, is a mapping from each mobile node id* $p \in P$ *to some TIOA* $C_p \in CProgram_p$*, and from each each* $u \in U$ *to some* $V_u \in VProgram_u$*. The set of all V-algorithms is referred to as* $VAlgs$*.*

Definition 8. *For each* $alg \in Valgs$*,* $VLayer[alg]$*, the* instantiation by *alg* of the abstract VSA layer*, is the abstract VSA layer where for each* $p \in P$*,* $C_p = alg(p)$*, and for each* $u \in U$*,* $V_u = alg(u)$*. More formally,* $VLayer[alg]$ *is the composition of V-bcast,* $alg(q)$ *for each* $q \in P \cup U$*, and* $Dout[e]_u$ *for each* $u \in U$*, where the bcast action between* V_u *and* $Dout[e]_u$ *is hidden.*

We are interested in the traces of a VSA layer, with VSA fails and restarts hidden (there is no natural analogue for such actions at the physical layer):

Definition 9. *Let* E *be the set of* $\mathsf{fail}(u)$ *and* $\mathsf{restart}(u)$ *actions for each* $u \in U$*, and let* $\mathcal{A}$ *be a TIOA. We refer to* $\mathcal{A}$*'s traces with* E *hidden,* $\{\beta\lceil(E_{\mathcal{A}} - E, \emptyset) | \beta \in traces_{\mathcal{A}}\}$*, as* $Htraces_{\mathcal{A}}$*.*

For any set S *of states of* $\mathcal{A}$*, we refer to* $\mathcal{A}$*'s traces with* E *hidden of execution fragments started in* S*,* $\{trace(\alpha)\lceil(E_{\mathcal{A}} - E, \emptyset) | \alpha \in frags^S_{\mathcal{A}}\}$*, as* $Htracefrags^S_{\mathcal{A}}$*.*

We then define an emulation of the abstract layer as a pair consisting of: (a) an emulation program, *amap*, and (b) a mapping, *tmap*, from traces of the emulation to traces of the abstract VSA layer without fails and restarts of regions. Like $VLayer$, *amap* is instantiated with programs from $Valgs$. However, unlike in our definition for $VLayer[alg]$, we do not restrict the instantiation of *amap* by *alg* to assign particular client and VSA programs to individual components; the mapping can be arbitrary. For example, for a particular *alg*, *amap*[*alg*] could be defined to be a physical layer in which each physical node's program is a composition of the client program in the VSA layer for that node, and an emulator portion where the physical node helps emulate its current region's VSA.

Definition 10. *An* emulation, $(amap, tmap)$, of the abstract VSA layer *has:*

- *Function* $amap : VAlgs \rightarrow \{\mathcal{T} | \mathcal{T} \text{ is a TIOA compatible with } RW\}$*.*
- *For each* $alg \in Valgs$*, function* $tmap[alg] : (E_{amap[alg]}, \emptyset)$*-sequences* $\rightarrow (E_{VLayer[alg]} - \{\mathsf{fail}(u), \mathsf{restart}(u) | u \in U\}, \emptyset)$*-sequences.*

We require that for any $alg \in Valgs$ *and trace fragment* β *of* $amap[alg] \| RW$*:*

1. *Let* B *be* $E_{RW} \cup \{environment\ actions\}$*.* $\beta\lceil(B, \emptyset) = tmap[alg](\beta)\lceil(B, \emptyset)$*.*
2. $\beta \in traces_{amap[alg] \| RW}$ *implies* $tmap[alg](\beta) \in Htraces_{VLayer[alg] \| RW'}$*.*

The following definition of a self-stabilizing emulation of a VSA layer says an emulation is self-stabilizing if any execution of $amap[alg] \| RW$ started in an arbitrary state of $amap[alg]$ and a reachable state of RW has a suffix in the set of execution fragments of $amap[alg] \| RW$ starting in a state in $L[alg]$. $L[alg]$ is a legal set for $amap[alg] \| RW$ with the added restriction that $tmap[alg]$ applied to any legal execution fragment is in $Htraces_{U(VLayer[alg]) \| R(RW')}$. This means once the emulation stabilizes, the mapped traces of the emulation look like those of the virtual layer with $VLayer[alg]$ started in an arbitrary state and RW'

started in some reachable state. This will allow us to guarantee that if the alg being emulated is such that $VLayer[alg]$ is self-stabilizing with respect to some legal set and given $R(RW')$, then the mapped traces of the emulation stabilize to traces of legal execution fragments of $VLayer[alg] \| RW'$ (Theorem 3).

Definition 11. *Let* $(amap, tmap)$ *be an emulation of the abstract VSA layer and* t *be in* $\mathbb{R}^{\geq 0}$. $(amap, tmap)$ *is a* self-stabilizing emulation of a VSA layer with stabilization time t *if for each* $alg \in VAlgs$*, there exists a legal set* $L[alg]$ *for* $amap[alg] \| RW$ *such that:*

1. $amap[alg]$ *is self-stabilizing with respect to* $L[alg]$ *and given* $R(RW)$ *in time* t*.*
2. *For each* $\alpha \in frags^{L[alg]}_{amap[alg] \| RW}$, $tmap[alg](trace(\alpha)) \in Htraces_{U(VLayer[alg]) \| R(RW')}$.

Let $null(t)$ *be the closed empty trajectory with* $ltime = t$*. We use* $Mtraces^{t,alg}_{amap,tmap}$ *to refer to:* $\{null(t)tmap[alg](\beta) |$ β *is a state-matched* t*-suffix of some element in* $traces_{U(amap[alg]) \| R(RW)}$.

We conclude that a transformed trace of a self-stabilizing emulation of the VSA layer started in an arbitrary state has a suffix in the traces of the VSA layer started in an arbitrary state:

Theorem 1. *Let* $(amap, tmap)$ *be a self-stabilizing emulation of the abstract VSA layer with stabilization time* t*, and let* alg *be any element of* $Valgs$*.*
Then $Mtraces^{t,alg}_{amap,tmap}$ *stabilizes to* $Htraces_{U(VLayer[alg]) \| R(RW')}$ *in time* t*.*

Proof sketch: Let β be a sequence in $Mtraces^{t,alg}_{amap,tmap}$, and β' be a state-matched t-suffix of β. We must show that $\beta' \in Htraces_{U(VLayer[alg]) \| R(RW')}$.

By definition of $traces$ and $Mtraces$, there exists some $\alpha\alpha' \in execs_{U(amap[alg]) \| R(RW)}$ such that α' is a state-matched t-suffix of $\alpha\alpha'$, and $\beta = null(t)tmap[alg](trace(\alpha'))$. By definition of self-stabilizing emulation, there exists some legal set $L[alg]$ for $amap[alg] \| RW$ such that properties 1 and 2 of the definition hold. Since α' is a state-matched t-suffix of a sequence in $execs_{U(amap[alg]) \| R(RW)}$ then property 1 implies $\alpha' \in frags^{L[alg]}_{amap[alg] \| R(RW)}$. By Lemma 4, $\alpha' \in frags^{L[alg]}_{amap[alg] \| RW}$. This implies by property 2 that $tmap[alg](trace(\alpha'))$ is in $Htraces_{U(VLayer[alg]) \| R(RW')}$.

Since β' is a state-matched t-suffix of $null(t)tmap[alg](trace(\alpha'))$, β' is a state-matched 0-suffix of $tmap[alg](trace(\alpha'))$. By Lemma 4, this implies $\beta' \in Htraces_{U(VLayer[alg]) \| R(RW')}$. □

Application to an Existing Emulation Algorithm

In prior work, we developed a self-stabilizing emulation algorithm for the VSA layer (details can be found in [6]). Physical nodes both implement their own corresponding client node and cooperate with other physical nodes to implement VSAs. For VSAs, at most one physical node in a VSA's tile is a leader (chosen by a stabilizing leader service), with primary responsibility for emulating the VSA and sole responsibility for performing VSA outputs. For fault-tolerance, other nodes receive VSA messages and maintain and update their own local versions of the VSA state, but do not perform broadcasts on behalf of the VSA.

Our implementation was made self-stabilizing with local correction and update and checksum messages. Update messages sent by a leader contain state information which overwrites VSA state information at other emulators, bringing emulators into agreement about VSA state. The leader also sends out checksum messages with an attached checksum. An emulator, when it receives the message, compares the attached checksum to the version it locally computed. If they differ, the emulator re-joins, ensuring its state is consistent with the leader's.

With our definitions, we can make *formal* correctness claims about [6]:

Theorem 2. *The VSA layer emulation algorithm of [6] is a self-stabilizing emulation of the VSA layer parameterized by the region failure and restart conditions described in Section 3.2.*

Proof sketch: We can show this by providing a mapping from states of the emulation algorithm to states of the abstract layer. For the emulation of an individual VSA, the mapping is from physical node and message channel states to a state of the abstract VSA. We then augment the emulation automaton to explicitly add fail and restart actions for regions, triggered based on conditions of the state of the emulation. A simple *tmap* function preserves interactions with *RW* and the environment, and renames certain physical node broadcasts and receives as V-bcast actions while hiding others. For example, a VSA receives a message sent to it the first time a client in the region receives it. We can then show that *tmap* applied to a trace of an execution of the emulation, combined with the added fail and restart actions for regions, has a suffix that is the trace of an execution of the abstract VSA layer started in an arbitrary initial state. □

5 Self-stabilizing Services on Emulated Layers

We can now combine a self-stabilizing emulation of the abstract VSA layer with a self-stabilizing algorithm run on the layer and conclude that the traces of the result stabilize to those of the algorithm running on the abstract VSA layer, with regions' fails and restarts hidden.

Theorem 3. *Let $(amap, tmap)$ be a self-stabilizing emulation of the abstract VSA layer, with stabilization time $t_1 \in \mathbb{R}^{\geq 0}$. For any $alg \in VAlgs, t_2 \in \mathbb{R}^{\geq 0}$, and legal set $vlegal[alg]$ for $VLayer[alg] \| RW'$, if $VLayer[alg]$ self-stabilizes with respect to $vlegal[alg]$ and given $R(RW')$ in time t_2, then $Mtraces^{t_1,alg}_{amap,tmap}$ stabilizes in time $t_1 + t_2$ to $Htracefrags^{vlegal[alg]}_{VLayer[alg] \| RW'}$.*

Proof sketch: Fix $alg \in VAlgs$ where $VLayer[alg]$ self-stabilizes with respect to $vlegal[alg]$ and given $R(RW')$ in time t_2. By Theorem 1, $Mtraces^{t_1,alg}_{amap,tmap}$ stabilizes to $Htraces_{U(VLayer[alg]) \| R(RW')}$ in time t_1. By definition of self-stabilization, since $VLayer[alg]$ self-stabilizes with respect to $vlegal[alg]$ and given $R(RW')$ in time t_2, $execs_{U(VLayer[alg]) \| R(RW')}$ stabilizes in time t_2 to

$frags^{vlegal[alg]}_{VLayer[alg]\|R(RW')}$, which by Lemma 4 is $frags^{vlegal[alg]}_{VLayer[alg]\|RW'}$. By Lemma 2, $Htraces_{U(VLayer[alg])\|R(RW')}$ stabilizes to $Htracefrags^{vlegal[alg]}_{VLayer[alg]\|RW'}$ in time t_2. With $Mtraces^{t_1,alg}_{amap,tmap}$ as B, $Htraces_{U(VLayer[alg])\|R(RW')}$ as C, and $Htracefrags^{vlegal[alg]}_{VLayer[alg]\|RW'}$ as D in Lemma 3, we conclude $Mtraces^{t_1,alg}_{amap,tmap}$ stabilizes in time $t_1 + t_2$ to $Htracefrags^{vlegal[alg]}_{VLayer[alg]\|RW'}$. □

Application to a Geocast Service

We now apply this theorem to a particular self-stabilizing service implemented using the VSA layer and conclude that the emulation of the VSA layer running this service eventually has traces that you could get in the abstract layer (minus regions' fail and restart events). We use a simple variant of a geocast service specification and implementation originally published in [7].

Specification. The geocast service is a timed channel automaton that allows a client C_p in region u to send a message m to region v via $\mathsf{geocast}(v, m)_p$, and to receive such a broadcast message via $\mathsf{geoRcv}(m)_p$, under certain conditions. For some constant ttl_{Geo}, say that a geocast by a client in region u to a region v at time t is *serviceable* if there exists at least one path of non-failed regions from u to v for the entire interval $[t, t + ttl_{Geo} + 2d + e]$. The geocast service's traces guarantee: (1) If a client geocasts a message at some time t and the geocast is serviceable, then all nonfailed clients in the destination region geoRcv the message by time $t + ttl_{Geo} + 2d + e$. (2) If a message is geoRcved by a client in region u, the message was geocast to region u within the last $ttl_{Geo} + 2d + e$ time.

Implementation. Geocast is implemented as a self-stabilizing $Valg$, alg_{Geo}, over the VSA layer. A client with a geocast input broadcasts the message to its local VSA, and the local VSA initiates VSA-to-VSA communication. VSA-to-VSA communication is based on a greedy depth-first search (DFS) procedure.

When a VSA receives a message for which it is not the destination, it greedily chooses a neighboring VSA using a function NxtNbr, mapping a set of region neighbors not yet tried, its own region, and the destination, to the next neighbor to forward the message to. (The selection is greedy in that the next neighbor chosen to receive the forwarded message is one on a shortest path to the destination VSA, after excluding paths that begin with neighbors associated with previous tries.) It then forwards the message in a forward message to that neighbor. If the VSA does not receive an indication through a found message that the message has been delivered to the destination within some bounded amount of time, it forwards the message to the next neighboring VSA returned by NxtNbr, etc.

Once the destination region is reached, the VSA at that region broadcasts the geocast message to its local clients, who then geoRcv it.

Self-stabilization is ensured by the use of a real-time timestamp to identify the version of the DFS for a forwarded message. Too old forwarded messages are eliminated from the system and newer forwarded messages do not impact the treatment of the older ones. Extending the results in [7], we can show that:

Lemma 5. *$VLayer[alg_{Geo}]$ self-stabilizes with respect to $reachable_{VLayer[alg_{Geo}]\|RW'}$ and given $R(RW')$ in time $e + 2d + ttl_{Geo}$.*

Theorem 4. *Let $(amap, tmap)$ be a self-stabilizing emulation of the abstract VSA layer with stabilization time t_{stab}. Then $Mtraces^{t_{stab},alg_{Geo}}_{amap,tmap}$ stabilizes to $Htracefrags^{reachable_{VLayer[alg_{Geo}]\|RW'}}_{VLayer[alg_{Geo}]\|RW'}$ in time $t_{stab} + ttl_{Geo} + 2d + e$.*

Proof sketch: By Lemma 5, $VLayer[alg_{Geo}]$ self-stabilizes with respect to $reachable_{VLayer[alg_{Geo}]\|RW'}$ and given $R(RW')$ in time $ttl_{Geo}+2d+e$. By Theorem 3, $Mtraces^{t_{stab},alg_{Geo}}_{amap,tmap}$ stabilizes to $Htracefrags^{reachable_{VLayer[alg_{Geo}]\|RW'}}_{VLayer[alg_{Geo}]\|RW'}$ in time $t_{stab} + ttl_{Geo} + 2d + e$. □

This means that if we run a self-stabilizing emulation of alg_{Geo}, the transformed trace of an execution of this emulation will eventually look like the suffix of a trace of the geocast specification.

By translating the geocast specification based on an abstract VSA layer's region failure and restart conditions, we can rephrase this result to be strictly in terms of the physical level. Consider the example region failure and restart conditions in Section 3.2. With those conditions, we can say that a region is *definitely non-failed* over some interval if some physical node at the start of the interval was non-failed and in the region for at least $2d$ time, and no failures or leaves of physical nodes occur during the interval or in the e time before the interval. This property is expressible with traces of physical node interactions with RW. The abstract geocast specification can then be transformed into a new, but weaker, physical level specification, $phys_{Geo}$, that replaces clients with physical nodes, replaces non-failed regions with definitely non-failed regions, and makes no mention of VSAs. We then get the corollary that traces of $U(amap[alg_{Geo}])\|R(RW)$ stabilize to traces of $phys_{Geo}$ in time $t_{stab} + ttl_{Geo} + 2d + e$.

6 Conclusions

We've presented a basic formal theory of self-stabilizing emulations for timed abstract virtual node layers. Abstract VSA layers can make the task of designing algorithms for mobile ad hoc networks considerably simpler than it would be in the absence of any infrastructure. Self-stabilizing algorithms were previously presented for emulation of VSA layers [6], and here we formalize the notion of emulation and self-stabilizing emulation. Such formalization provides a clear set of proof obligations required to conclude that an algorithm successfully provides an emulation of an abstract VSA layer, allowing an application programmer to program the VSA layer without worrying about how that layer is provided.

The formalization of self-stabilizing emulation also allows us to guarantee that if a self-stabilizing emulation of the abstract VSA layer is running a self-stabilizing VSA layer application, then the result is a system whose externally visible actions eventually look like those of a legal execution fragment of the application being run. This separates the reasoning about the stabilization properties of the emulation algorithm from those of the application being run.

These support application developers for unpredictable mobile networks by allowing them to safely and easily take advantage of timed virtual infrastructure to aid in problem solving.

References

1. Camp, T., Liu, Y.: An adaptive mesh-based protocol for geocast routing. Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad.-hoc Networking and Computing, 196–213 (2002)
2. Dijkstra, E.W.: Self stabilizing systems in spite of distributed control. Communications of the ACM, 643–644 (1974)
3. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
4. Dolev, S., Gilbert, S., Lynch, N., Schiller, E., Shvartsman, A., Welch, J.: Virtual Mobile Nodes for Mobile Ad Hoc Networks. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 230–244. Springer, Heidelberg (2004)
5. Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., Welch, J.: GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 306–320. Springer, Heidelberg (2003)
6. Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., Nolte, T.: Timed Virtual Stationary Automata for Mobile Networks. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, Springer, Heidelberg (2006)
7. Dolev, S., Lahiani, L., Lynch, N., Nolte, T.: Self-stabilizing Mobile Node Location Management and Message Routing. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, Springer, Heidelberg (2005)
8. Dolev, S., Herman, T., Lahiani, L.: Polygonal Broadcast, Secret Maturity and the Firing Sensors. In: FUN. Third International Conference on Fun with Algorithms, pp. 41–52 (May 2004). Also to appear in Ad Hoc Networks Journal, Elseiver.
9. Dolev, S., Israeli, A., Moran, S.: Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity. In: PODC 1990. Proceeding of the ACM Symposium on the Principles of Distributed Computing, pp. 103–117 (1990). Also in Distributed Computing 7(1), 3–16 (1993)
10. Karp, B., Kung, H.T.: GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In: Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, pp. 243–254. SCM Press (2000)
11. Kaynar, D., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Morgan and Claypool Publishers (2006)
12. Kuhn, F., Wattenhofer, R., Zhang, Y., Zollinger, A.: Geometric Ad-Hoc Routing: Of Theory and Practice. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 63–72 (2003)
13. Kuhn, F., Wattenhofer, R., Zollinger, A.: Asymptotically Optimal Geometric Mobile Ad-Hoc Routing. In: Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DialM), pp. 24–33. ACM Press, New York (2002)
14. Navas, J.C., Imielinski, T.: Geocast- geographic addressing and routing. In: Proceedings of the 3rd MobiCom, pp. 66–76 (1997)

Author Index