Alexander A. Shvartsman (Ed.)

LNCS 4305

Principles of Distributed Systems

10th International Conference, OPODIS 2006
Bordeaux, France, December 2006
Proceedings

Lecture Notes in Computer Science 4305

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

Alexander A. Shvartsman (Ed.)

Principles of Distributed Systems

10th International Conference, OPODIS 2006
Bordeaux, France, December 12-15, 2006
Proceedings

Volume Editor

Alexander A. Shvartsman
University of Connecticut
Computer Science and Engineering Department
371 Fairfield Rd., Storrs, CT 06269-2155, USA
E-mail: aas@cse.uconn.edu

Library of Congress Control Number: 2006937266

CR Subject Classification (1998): C.2.4, D.1.3, D.2.7, D.2.12, D.4.7, C.3

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

ISSN 0302-9743
ISBN-10 3-540-49990-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-49990-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11945529 06/3142 5 4 3 2 1 0

Preface

OPODIS, the International Conference On Principles Of Distributed Systems, is an annual forum for the exchange of state-of-the-art knowledge on principles of distributed computing and systems among researchers from around the world. The 10th anniversary edition of OPODIS was held during December 12–15, 2006, in Bordeaux, France.

This year over 230 papers were submitted, out of which 28 papers were accepted as regular papers and 3 as brief announcements. The decisions were made by the Program Committee during an electronic meeting held during the week of September 10th, 2006, following a review period. The Program Committee extends its thanks to all authors who submitted papers to OPODIS 2006. The chair thanks the members of the Program Committee and the External Reviewers for their hard work in reviewing and evaluating the submitted papers. We are convinced that a very good set of papers was selected for presentation at OPODIS 2006.

The symposium also featured keynote addresses by Amir Pnueli (Weizmann Institute, Israel), Butler Lampson (Microsoft, USA), Michel Raynal (IRISA, France), and Gerard Roucairol (Bull, France).

We believe that OPODIS has found its place among the conferences related to principles of distributed computing, networks, and systems. We hope that the 10th edition of OPODIS will contribute to the growth and the development of the conference and continue to increase its visibility.

October 2006

Alex Shvarsman
OPODIS 2006
Program Chair

Conference Organization

OPODIS, the International Conference On Principles Of Distributed Systems, is an open forum for the exchange of state-of-the-art knowledge on principles of distributed computing and systems among researchers from around the world. This conference was the 10th in a series of annual conferences.

General Chair

Ivan Lavallée	University of Paris 8, France

Program Chair

Alex Shvartsman	University of Connecticut, USA

Program Committee

James Anderson	University of North Carolina, USA
Anish Arora	Ohio State University, USA
Hagit Attiya	The Technion, Israel
Joffroy Beauquier	University of Paris 11, France
Riccardo Bettati	Texas A&M University, USA
Jiannong Cao	Polytechnic University, Hong Kong
Richard Castanet	ENSEIRB Bordeaux, France
Ajoy Datta	University of Nevada Las Vegas, USA
Michael Fischer	Yale University, USA
Paola Flocchini	Ottawa University, Canada
Chryssis Georgiou	University of Cyprus, Cyprus
Mohamed Gouda	University of Texas, USA
Teruo Higashino	Osaka University, Japan
Friedhelm Meyer auf der Heide	University of Paderborn, Germany
Mikhail Nesterenko	Kent State University, USA
Marina Papatriantafilou	Chalmers University, Sweden
Boaz Patt-Shamir	Tel Aviv University, Israel
Andrzej Pelc	University of Quebec, Canada
Giuseppe Prencipe	University of Pisa, Italy
Sergio Rajsbaum	UNAM, Mexico
Michel Raynal	IRISA, France
André Schiper	EPF Lausanne, Switzerland
Ulrich Schmid	Technical University of Vienna, Austria
Marc Shapiro	INRIA, France
Nir Shavit	Sun Microsystems, USA

David Simplot-Ryl	University of Lille, France
Paul Spirakis	University of Patras, Greece
Mark Tuttle	Intel, USA
Vincent Villain	University of Picardie, France
Roger Wattenhofer	ETH Zurich, Switzerland
Carlos Becker Westphall	University of St. Catarina, Brazil
Peter Widmayer	ETH Zurich, Switzerland

Organizing Committee

Thibault Bernard	University of Reims, France
Céline Butelle	University of Paris 8, France
Nicole Lavallée	
Devan Sohier	EPHE, France

Publicity Chair and Local Organization

Antoine Rollet	ENSEIRB Bordeaux, France

Steering Committee

Alain Bui	University of Reims, France
Marc Bui	EPHE, France
Hacène Fouchal	University of Antilles-Guyane, France
Roberto Gomez	ITESM-CEM, Mexico
Nicola Santoro	Carleton University, Canada
Philippas Tsigas	Chalmers University, Sweden

Invited Speakers

Amir Pnueli	Weizmann Institute, Israel
Butler Lampson	Microsoft, USA
Michel Raynal	IRISA, France
Gerard Roucairol	BULL, France

Reviewers

Marcos K. Aguilera	Sandip Bapat	Olaf Bonorden
Joufuk Ahmed	Robert Beers	Christian Boulinier
Keno Albrecht	Ismaël Berrada	Andre Brinkmann
James Aspnes	Carlo Bertolli	Marc Bui
Baruch Awerbuch	Ritwik Bhattacharya	Nicolas Burri

Jean-Michel Busca
Hui Cao
Aleksandr Charzewski
Bogdan Chlebus
Marek Chrobak
Mark Cieliebak
Antonio Cisternino
Thomas L. Clouser
Massimo Coppola
Alain Cournier
Xavier Devismes
Stéphane Devismes
Dave Dice
Stefan Dobrev
Mirosław Dynia
Sascha Effert
Niklas Elmqvist
Robert Elsaesser
Emre Ertin
Patrick Félix
Antonio Fernandez
Nathan Fisher
Roland Flury
Mark Foskey
Udo Fritzke Jr.
Leszek Gasieniec
Vincenzo Gervasi
Anders Gidenstam
Chris Gill
Roberto Gomez
Olga Goussevskaia
Rachid Guerraoui
Seif Haridi
Martin Hutle
David Hay
Felix Heine
Danny Hendler
Thomas Hérault
Eshcar Hillel
Phuong Ha Hoai
Colette Johnen
Ralf Klasing
Boris Koldehofe
Petr Kouznetsov
Dariusz Kowalski
Rastislav Kralovic
Evangelos Kranakis
Francine Krief
Danny Krizanc
Michael Kuhn
Ritesh Kumar
Andreas Larsson
Bill Leal
Pierre Lemarinier
Alessandro Leonardi
Thomas Locher
Euripides Markou
Ketan Mayer-Patel
Stéphane Messika
Yves Métivier
Mark Moir
Thomas Moscibroda
Vinayak Naik
Johan Nordlander
Regina O'Dell
Rotem Oshman
Yvonne Anne Oswald
Linda Pagli
René Peralta
Franck Petit
Olivier Pèrés
Laurence Pilard
Francesco Potorti
Tomasz Radzik
Sushant Rewaskar
Pascal von Rickenbach
Luis Rodrigues
Antoine Rollet
Stefan Ruehrup
Oliver Rutti
Lifeng Sang
Yoav Sasson
Elad Schiller
Stefan Schmid
John Smith
Nigamanth Sridhar
Mukundan Sridharan
Tami Tamir
Daniela Tulone
Pedro Trancoso
Nir Tzachar
Vasos Vassiliou
Pihui Wei
Demetris Zeinalipour
Hongwei Zhang
Yi Zhang
Michele Zorzi

Sponsoring Institutions

AUF, Agence Universitaire de la Francophonie, France
EPHE, L'Ecole Pratique des Hautes Etudes, France
LAISC, Laboratoire d'Informatique et des Systèmes Complexes, France
Laboratoire Cognition et Usage, Université Paris 8, France

Table of Contents

List of Breif Announcements

The following brief announcements were presented at OPODIS 2006. These papers appear in a technical report made available at the conference.

Conflict Managers for Self-Stabilization without Fairness Assumptions Maria Gradinariu and Sebastien Tixeuil

A Provably Correct Scalable Concurrent Skip List Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit

An Algorithm for Distributing and Retrieving Information in Sensor Networks Hugo Miranda, Simone Leggio, Luís Rodrigues, and Kimmo Raatikainen

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, p. 440, 2006.

Lazy and Speculative Execution in Computer Systems

Butler Lampson

Microsoft Research

Invited Talk Abstract

The distinction between lazy and eager (or strict) evaluation has been studied in programming languages since Algol 60s call by name, as a way to avoid unnecessary work and to deal gracefully with infinite structures such as streams. It is deeply integrated in some languages, notably Haskell, and can be simulated in many languages by wrapping a lazy expression in a lambda.

Less well studied is the role of laziness, and its opposite, speculation, in computer systems, both hardware and software. A wide range of techniques can be understood as applications of these two ideas. Laziness is the idea behind:

- Redo logging for maintaining persistent state and replicated state machines: the log represents the current state, but it is evaluated only after a failure or to bring a replica online.
- Copy-on-write schemes for maintaining multiple versions of a large, slowly changing state, usually in a database or file system.
- Write buffers and writeback caches in memory and file systems, which are lazy about updating the main store.
- Relaxed memory models and eventual consistency replication schemes (which require weakening the spec).
- Clipping regions and expose events in graphics and window systems.
- Carry-save adders, which defer propagating carries until a clean result is needed.
- "Infinity" and "Not a number" results of floating point operations.
- Futures (in programming) and out of order execution (in CPUs), which launch a computation but are lazy about consuming the result. Dataflow is a generalization.
- "Formatting operators" in text editors, which apply properties such as "italic" to large regions of text by attaching a sequence of functions that compute the properties; the functions are not evaluated until the text needs to be displayed.
- Stream processing in database queries, Unix pipes, etc., which conceptually applies operators to unbounded sequences of data, but rearranges the computation when possible to apply a sequence of operators to each data item in turn.

Speculation is the idea behind:

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 1–2, 2006.

Optimistic concurrency control in databases, and more recently in transactional memory. Prefetching in memory and file systems.

Branch prediction, and speculative execution in general in modern CPUs.

Data speculation, which works especially well when the data is cached but might be updated by a concurrent process. This is a form of optimistic concurrency control.

Exponential backoff schemes for scheduling a resource, most notably in LANs such as WiFi or classical Ethernet.

All forms of caching, which speculate that its worth filling up some memory with data in the hope that it will be used again.

In both cases it is usual to insist that the laziness or speculation is strictly a matter of scheduling that doesnt affect the result of a computation but only improves the performance. Sometimes, however, the spec is weakened, for example in eventual consistency.

I will discuss many of these examples in detail and examine what they have in common, how they differ, and what factors govern the effectiveness of laziness and speculation in computer systems.

In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-Free Set Agreement

Michel Raynal and Corentin Travers

IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France
{raynal, ctravers}@irisa.fr

Abstract. Asynchronous failure detector-based set agreement algorithms proposed so far assume that all the processes participate in the algorithm. This means that (at least) the processes that do not crash propose a value and consequently execute the algorithm. It follows that these algorithms can block forever (preventing the correct processes from terminating) when there are correct processes that do not participate in the algorithm. This paper investigates the wait-free set agreement problem, i.e., the case where the correct participating processes have to decide a value whatever the behavior of the other processes (i.e., the processes that crash and the processes that are correct but do not participate in the algorithm). The paper presents a wait-free set agreement algorithm. This algorithm is based on a leader failure detector class that takes into account the notion of participating processes. Interestingly, this algorithm enjoys a first class property, namely, design simplicity.

Keywords: Asynchronous algorithm, Asynchronous system, Atomic register, Consensus, Leader oracle, Participating process, Set agreement, Shared object, Wait-free algorithm.

1 Introduction

The consensus problem Consensus is a fundamental fault-tolerant distributed computing problem. As soon as processes cooperate, they have to agree in one way or another. This is exactly what the consensus problem captures: it allows a set of processes to agree on a critical data (called value, decision, state, etc.). Consensus can be informally defined as follows. Each process proposes a value, and a process that is not faulty has to decide a value (termination), such that there is a single decided value (agreement)[1], and that value is a proposed value (validity).

It is well-known that the consensus problem cannot be solved in asynchronous systems prone to even a single process crash, be these systems read/write shared memory systems [13], or message-passing systems [5]. So, one way to circumvent this impossibility is to enrich the asynchronous system with additional objects that are strong enough to allow solving consensus.

[1] We consider here the uniform version of the consensus problem. A faulty process that decides has to decide the same value as the non-faulty processes.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 3–19, 2006.

Shared memory systems equipped with objects defined with a sequential specification and more powerful than traditional atomic read/write registers have been investigated. This line of research has produced the notion of *consensus number* that can be associated with each object type defined by a sequential specification [9]. The consensus number of a type is the maximum number of processes for which objects of that type (together with atomic registers) allows solving consensus. For example, the objects provided with a Test&Set() operation have consensus number 2, while the objects provided with a Compare&Swap() operation have consensus number $+\infty$. The consensus number notion has allowed to establish a hierarchy among the objects (with a sequential specification) according to the synchronization power of the operations they provide to the processes [9].

Another research direction has been the investigation of objects that provide processes with information on failures, namely, the objects called *failure detectors* [2]. A failure detector class[2] is defined by abstract properties that state which information on failure is provided to the processes. According to the quality of that information, several classes can be defined. Differently from an atomic register or a Compare&Swap object, a failure detector has no sequential specification.

As far as one is interested in solving the consensus problem in an asynchronous system prone to process crashes, it has been shown that Ω is the weakest failure detector class that allows solving consensus in such a context [3]. "Weakest" means that any failure detector that allows solving consensus provides information on failures that allows building a failure detector of the class Ω.

A failure detector of the class Ω provides the processes with a primitive, denoted leader(), that returns a process identity each time it is called, and eventually always returns the same identity that is the id of a correct process, i.e., a process that does not crash when we consider crash failures. (Examples of message-passing Ω-based consensus algorithms can be found in [7,12,16]. These algorithms assume a majority of correct processes, which is a necessary requirement for Ω-based message-passing consensus algorithms.)

The set agreement problem. The k-set agreement problem [4] generalizes the consensus problem: it weakens the constraint on the number of decided values by permitting up to k different values to be decided (consensus is 1-set agreement). While k-set agreement can be easily solved in asynchronous systems where the number t of processes that crash is $< k$ (each of a set of k predetermined processes broadcasts its value, and a process decides the first value it receives), this problem has no solution when $k \geq t$ [1,11,19]. The failure detector approach to solve the k-set agreement problem in message-passing systems has been investigated in [10,14,15,20].

While (as indicated before) it has been established that Ω is the weakest failure detector class for solving consensus [3], let us remind that finding the weakest failure failure detector class for solving k-set agreement for $k > 1$ is still an open problem.

The main question. Failure detector-based consensus algorithms implicitly consider that all the processes participate in the consensus algorithm, namely, any process that

[2] We employ the words "failure detector class" instead of "failure detector type", as it is the word traditionally used in the literature devoted to failure detectors.

does not crash is implicitly assumed to propose a value and execute the algorithm. This is also an implicit assumption in the statement that Ω is the weakest failure detector to solve the consensus problem [3]. Basically, an Ω-based consensus algorithm uses the eventual leader to eventually impose the same value to all the processes. As the algorithm does not know when the leader is elected, its main work consists in guaranteeing that no two different values can be decided before the eventual leader is elected. The algorithm uses the eventual leader to *help* decide all the processes that do not crash.

The previous observation raises the following question: What does happen if the process that is the eventual leader does not participate in the consensus algorithm? It appears that the algorithm can then block forever, and consequently the termination property can no longer be guaranteed.

So, a fundamental question is the following: *What is the weakest failure detector to solve the consensus (or, more generally, the k-set agreement) problem when only a subset of the correct processes (not known a priori) propose a value and participate in the agreement problem?* This question can be reformulated as follows: What are the weakest failure detector classes to *wait-free* solve the consensus and the k-set agreement problems? Wait-free means here that a process that proposes a value and does not crash has to decide, whatever the behavior of the other processes (they can participate or not, and be correct or not). The previous observation on Ω shows that a failure detector of that class cannot be the weakest to wait-free solve the consensus problem.

Content of the paper. Answering the previous question requires to investigate new failure detector classes and show that one of them allows solving k-set agreement (sufficiency part) while being the weakest (necessity part). This paper addresses the sufficiency part. (On the necessity side, although we don't have yet formal results, we currently are inclined towards thinking that the failure detector class Ω^k_* -see below- is the weakest failure detector class for wait-free solving k-set agreement.)

More precisely, the paper presents a failure detector-based algorithm for shared memory systems that wait-free solves the k-set agreement problem whatever the number p of participating processes, and their behavior, in a set of n processes[3]. This algorithm assumes that, in addition to single-writer/multi-readers atomic registers, the shared memory provides the processes with a failure detector object of a class that we denote Ω^k_*. That class is an extension of the failure detector class introduced in [18], and the failure detector classes recently introduced in [6] and [17].

The failure detector class Ω^k introduced in [18] extends the classical Ω class [3] by allowing a set of up to k leaders to be returned by each invocation of the leader() primitive (Ω^1 boils down to Ω). The set of k leaders that is eventually returned forever includes at least one correct process. The aim of the class Ω^* introduced in [6] is to boost obstruction-free algorithms into non-blocking algorithms. That paper also shows

[3] Let us remind that all the algorithms based on an object O with consensus number n allows solving consensus whatever the number ($p \leq n$) and the behavior of the participating processes, i.e., they are wait-free consensus algorithms. (Such an object O has always a sequential specification.) In some sense, this paper looks for a failure detector class that, while being as weak as possible, is as strong as the object O, i.e., a class that allows designing wait-free failure detector-based set agreement algorithms. (Failure detectors cannot be defined from a sequential specification.)

that this failure detector class is the weakest for such a boosting. The failure detector class introduced in [17] extends Ω^k by explicitly referring to the notion of participating processes. It has been introduced to circumvent the $2p-1$ lower bound of the renaming problem [11] (where p is the number of participating processes). Using such a failure detector, the proposed renaming algorithm provides the processes with a renaming space whose size is reduced from $2p-1$ to $2p-\lceil \frac{p}{k} \rceil$ (where the value k comes from "k"-set agreement).

Roadmap. The paper is made up of 6 sections. Section 2 presents the computation model. Section 3 introduces the failure detector class Ω^k_*. Then, Section 4 presents the Ω^k_*-based k-set algorithm. This algorithm uses an underlying object denoted KA. So, Section 5 presents an algorithm constructing a KA object from atomic read/write registers. Finally, Section 6 concludes the paper.

2 Asynchronous System Model

2.1 Process and Communication Model

Process model. The system consists of n sequential processes that we denote $p_1, \ldots, p_n$. A process can crash. Given an execution, a process that crashes is said to be *faulty*, otherwise it is *correct* in that execution. Each process progresses at its own speed, which means that the system is asynchronous. In the following, $Correct$ denoted the set of processes that are correct in the run that is considered.

Coordination model. The processes cooperate and communicate through two types of reliable objects: two arrays of single-writer/multi-reader atomic registers and a shared object that we call KA (as shown in Section 5, such an object can be built from single-writer/multi-reader atomic registers). The processes are also provided with a failure detector object of the class Ω^k_* (see below).

Identifiers with upper case letters are used to denote shared objects. Lower case letters are used to denote local variables; in that case the process index appears as a subscript. As an example, $part_i[j]$ denotes the jth entry of a local array of the process p_i, while $PART[j]$ denotes the jth entry of the shared array $PART$.

2.2 The KA Object

The KA object is a variant of a round-based object introduced in [8] to capture the safety properties of Paxos-like consensus algorithms [8,12]. This object provides the processes with an operation denoted $\mathsf{alpha\_propose}_k()$. That operation has two input parameters: the value (v_i) proposed by the invoking process p_i, and a round number (r_i). The round numbers play the role of a logical time and allows identifying the invocations. The KA object assumes that no two processes use the same round numbers, and successive invocations by the same process use increasing round numbers. Given a KA object, the invocations $\mathsf{alpha\_propose}_k()$ satisfy the following properties ($\perp$ is a default value that cannot be proposed by a process):

- Termination (wait-free): an invocation of $\mathsf{alpha\_propose}_k()$ by a correct process always terminates (whatever the behavior of the other processes).
- Validity: the value returned by any invocation $\mathsf{alpha\_propose}_k()$ is a proposed value or $\perp$.
- Agreement: At most k different non-$\perp$ values can be returned by the whole set of $\mathsf{alpha\_propose}_k()$ invocations.
- Convergence: If there is a time after which the operation $\mathsf{alpha\_propose}_k()$ is invoked infinitely often, and these invocations are issued by an (unknown but fixed) set of at most k processes, there is then a time after which none of these invocations returns $\perp$.

A KA object can store up to k non-$\perp$ different values. A process invokes it with a value to store and obtains a value in return. If it is permanently accessed concurrently by more than k processes, the KA object might store anything. If there is a period during which it is accessed concurrently by at most $k' \leq k$ processes, it stores forever the corresponding k' proposed values.

3 The Failure Detector Class Ω^k_*

3.1 Definition

A failure detector of the class Ω^k_* provides the processes with an operation denoted $\mathsf{leader}()$. (As indicated in the introduction, this definition is inspired by the leader failure detector classes introduced in [6,17,18].) When a process p_i invokes that operation, it provides it with an input parameter, namely a set X of processes, and obtains a set of process identities as a result[4].

The semantics of Ω^k_* is based on a notion of time, whose domain is the set of integers. It is important to notice that this notion of time is not accessible to the processes. An invocation of $\mathsf{leader}(X)$ by a process p_i is *meaningful* if $i \in X$. If $i \notin X$, it is *meaningless*. The primitive $\mathsf{leader}()$ is defined by the following properties:

- Termination (wait-free): Any invocation of $\mathsf{leader}()$ by a correct process always terminates (whatever the behavior of the other processes).
- Triviality: A meaningless invocation returns any set of processes.
- Eventual multi-leadership for each input set X: For any $X \subseteq \Pi$, such that $X \cap \mathit{Correct} \neq \emptyset$, there is a time τ_X such that, $\forall \tau \geq \tau_X$, all the meaningful $\mathsf{leader}(X)$ invocations (that terminate) return the same set L_X and this set is such that:
  - $|L_X| \leq k$.
  - $L_X \cap X \cap \mathit{Correct} \neq \emptyset$.

The intuition that underlies this definition is the following. The set X passed as input parameter by the invoking process p_i is the set of all the processes that p_i considers as

[4] The definition of Ω^k_* is not expressed in the framework introduced by Chandra and Toueg to define failure detector classes [2]. More precisely, in their framework, the failure detector operation that a process can issue has no input parameter. It would be possible to express Ω^k_* in their framework. We don't do it in order to make the presentation simpler.

being currently *participating* in the computation. (This also motivates the notion of meaningful and meaningless invocations: an invoking process is trivially participating).

Given a set X of participating processes that invoke leader(X), the eventual multi-leadership property states that there is a time after which these processes obtain the same set L_X of at most k leaders, and at least one of them is a correct process of X. Let us observe that the (at most $k-1$) other processes of L_X can be any subset of processes (correct or not, participating or not).

It is important to notice that the time τ_X from which this property occurs is not known by the processes. Moreover, before that time, there is an anarchy period during which each process, as far as its leader(X) invocations are concerned, can obtain different sets of any number of leaders. Let us also observe that if a process p_i issues two meaningful invocations leader$(X1)$ and leader$(X2)$ with $X1 \neq X2$, there is no relation linking L_{X1} and L_{X2}, whatever the values of $X1$ and $X2$ (e.g., the fact that $X1 \subset X2$ imposes no particular constraint on L_{X1} and L_{X2}).

Let us consider an execution in which all the invocations leader(X) are such that $X = \Pi$ (the whole set of processes are always considered as participating). In that case, Ω^k_* boils down to the failure detector class denoted Ω^k introduced in [18]. If additionally, $k = 1$, we obtain the classical leader failure detector Ω introduced in [3].

When $X \subseteq \Pi$ and $k = 1$, Ω^k_* boils down to the failure detector class introduced in [6]. It is shown in [6] that Ω is weaker than Ω^1_* that in turn is weaker than $\diamond\mathcal{P}$ (the class of eventually perfect failure detectors: after some finite but unknown time, an eventually perfect failure detector suspects all the crashed processes and only them [2]).

3.2 The Family $\{\Omega^k_*\}_{1\leq k\leq n}$

It follows from the definition of Ω^k_*, that the failure detector class family $\{\Omega^k_*\}_{1\leq k\leq n}$ is such that $\Omega^k_* \subset \Omega^{k+1}_*$.

Moreover, as just indicated, when all the leader(X) invocations are such that $X = \Pi$, Ω^k_* boils down Ω^k (as defined in [18]), from which it follows that we have $\Omega^k \subseteq \Omega^k_*$. More generally, the failure detector classes Ω^k and Ω^k_* are related as indicated in Figure 1 where $A \rightarrow B$ means that any failure detector of the class A can be used to build a failure detector of the class B, while $A \cdots > B$ means that it is not possible to build a failure detector of the class B from a failure detector of the class A.

- The fact that $\Omega^k \subseteq \Omega^k_*$ (top-down plain arrows) follows from the definitions of the failure detector classes.
- The fact that Ω^n and Ω^n_* are the same class (top-down and bottom-up arrows at the right) follows directly from their definitions.

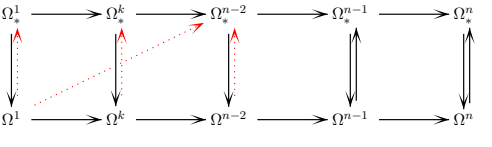


Fig. 1. Wait-free (ir)reducibility results between the families $(\Omega^x_*)_{1\leq x\leq n}$ and $(\Omega^y)_{1\leq y\leq n}$

- The fact that $\forall k : 1 \leq k < n-1, \forall k' : 1 \leq k' \leq n$, it is not possible to build a failure detector of the class Ω^k_* from a failure detector of the class $\Omega^{k'}$ (dotted arrows) is established in Theorem 1.
- The fact that it is possible to construct a failure detector of the class Ω^{n-1}_* from any failure detector of the class Ω^{n-1} is established in Theorem 2.

Theorem 1. *$\forall k : 1 \leq k < n-1, \forall k' : 1 \leq k' \leq n$, it is not possible to build a failure detector of the class Ω^k_* from a failure detector of the class $\Omega^{k'}$.*

Proof. To prove the theorem, it suffices to show that it is not possible to build a failure detector of the class Ω^{n-2}_* (the weakest class in the family $(\Omega^k_*)_{1\leq k\leq n-2}$) from a failure detector of the class Ω^1 (the strongest class in the family $(\Omega^k)_{1\leq k\leq n}$). The proof is by contradiction. Let us assume that there is an algorithm $\mathcal{A}$ that builds a failure detector of the class Ω^{n-2}_* from a failure detector of the class Ω^1 ([5]). We construct an infinite run in which at least one of the failure detectors Ω^1 or Ω^{n-2}_* fails to meet its specification. The construction uses the following claim. Let LEADER() denote the leader primitive of Ω^1. Recall that $\mathsf{leader}(X)$ is the leader primitive of Ω^{n-2}_*.

Claim C. Let R be an arbitrary run in which each process is correct. Moreover, in run R, each process p_i periodically invokes $\mathsf{leader}(X)$ for each X such that $i \in X \wedge |X| = n-1$. Let τ be a time at which the leadership properties of both Ω^1 and Ω^{n-2}_* are satisfied (i.e., all the invocations of LEADER() return the same process id, and, for any set X, all the invocations $\mathsf{leader}(X)$ return the same set). We claim that there is a run R_1 of the algorithm $\mathcal{A}$ such that

- R_1 is indistinguishable from R up to time τ.
- In R_1, there is a process p_x and a time $\tau_1 > \tau$ such that (1) the outputs of Ω^1 at τ and τ_1 are different, or (2) the outputs of Ω^{n-2}_* (for some set X) at τ and τ_1 are different.
- No process crashes in R_1.

Proof of the claim. By the claim assumption, the leadership properties of both Ω^1 and Ω^{n-2}_* are satisfied at time τ in R. In particular, we have:

1. Ω^1 outputs at each process the same leader identity ℓ.
2. Let $X = \Pi - \{\ell\}$. At time τ, $\exists L$ such that $|L| \leq n-2$, and, for each process $p_i \in X$, $\mathsf{leader}(X) = L$. Let $L' = L \cap X$. We have $|L'| \leq n-2$. As $|X| = n-1$, there is a process p_x such that $x \in X - L'$. Moreover, $x \neq \ell$ (because $X = \Pi - \{\ell\}$).

We show that, from the previous observations, we can build a run R_1, identical to R up to time τ, such that there exists a time $\tau' > \tau$ at which we have:

- Ω^1 outputs at some process a leader $\ell_1 \neq \ell$ or,
- At process p_x, $\mathsf{leader}(X) \neq L$.

[5] Let us recall that the output of Ω^1 at a given process p_i is *local*. This means that for the output of Ω^1 at p_i be known by the thread implementing the algorithm $\mathcal{A}$ at p_j, it is necessary that that output be written in the shared memory.

Let us consider the run R' defined as follows. R' is the same as R up to time τ. At time $\tau+1$, every process in L' crashes. Moreover, from τ, all the invocations LEADER() of Ω^1 output ℓ in R'. Due to the eventual multi-leadership property of Ω^{n-2}_*, there is a time $\tau_1 > \tau+1$ such that the invocations of $\mathsf{leader}(X)$ at process p_x return $L_1 \neq L$. This is because the set that is returned (namely, L_1) has to be such that $L_1 \cap X \cap Correct \neq \emptyset$, and, as the processes of L' have crashed, we have $L \cap X \cap Correct = \emptyset$ (recall that $L' = L \cap X$).

Let us now consider the run R_1 identical to R up to time $\tau + 1$. During the interval $[\tau + 1, \tau_1]$ the processes in L' do not take any step as far the algorithm $\mathcal{A}$ is concerned. The other processes behave exactly as in R'. If Ω^1 outputs $\ell' \neq \ell$ at some process p_y, the claim follows. Otherwise, let us observe that, for any process p_y (such that $y \notin L'$), R_1 cannot be distinguished from R'. In particular, the algorithm $\mathcal{A}$ outputs, at process p_x, $L_1 \neq L$ at time τ_1 when p_x issues $\mathsf{leader}(X)$. *End of the proof of the claim.*

Let us consider an arbitrary run R_0 in which each process is correct. There is a time τ_1 (1) at which the leadership properties of both failure detectors Ω^1 and Ω^{n-2}_* are satisfied, and (2) each process has taken at least one step. By Claim *C*, we can build a run R_1 identical to run R up to time τ_1 and such that the output of Ω^1 or Ω^{n-2}_* (for some input parameter X) at some process has changed at time $\tau'_1 > \tau_1$.

In run R_1, we can find a time $\tau_2 > \tau'_1$ such that each process has taken at least one step between τ'_1 and τ_2 and the leadership properties of Ω_1 and Ω^{n-2}_* are satisfied at time τ_2. By applying Claim *C*, we can build a run R_2 identical to R_1 up to time τ_2, etc. By iterating this process, we obtain an infinite run R and an infinite sequence of increasing times $(\tau_1, \tau'_1, \tau_2, \tau'_2, \ldots)$ such that $\forall i > 0, \exists p_{x(i)}$ such that, at $p_{x(i)}$ the output of Ω^1 or Ω^{n-2}_* (for some parameter X) is not the same at time τ_i and τ'_i. Due to the eventual leadership property of Ω, there is a time after which the output of Ω^1 does not change at each process. Consequently, it follows that in run R algorithm $\mathcal{A}$ fails to implement Ω^{n-2}_*. $\square_{Theorem\ 1}$

Theorem 2. *Given any failure detector of the class Ω^{n-1}, it is possible to build a failure detector of the class Ω^{n-1}_*.*

Proof. The proof is constructive. Let us consider any failure detector of the class Ω^{n-1}, and let LEADER() be its leader primitive. Let us consider the operation $\mathsf{leader}(X)$ defined as follows:

operation $\mathsf{leader}(X)$: **if** $X = \Pi$ **then** $\mathsf{return}\ \big(\text{LEADER}()\big)$ **else** $\mathsf{return}\ (X)$ **end_if**.

We show that $\mathsf{leader}(X)$ satisfies the properties of the class Ω^{n-1}_*.

Let us first consider the case where $X = \Pi$. Due to the properties of Ω^{n-1}, there is a time after which LEADER() always returns the same set L_X such that $|L_X| = n - 1$ and $L_X \cap Correct \neq \emptyset$. It trivially follows that $X \cap L_X \cap Correct \neq \emptyset$. Consequently, the eventual multi-leadership property is satisfied for the invocations $\mathsf{leader}(\Pi)$.

Given any set X such that $X \neq \Pi$, let us now consider the case of the invocations $\mathsf{leader}(X)$. The definition of $\mathsf{leader}(X)$ indicates that the set $L_X = X$ is then returned by these invocations, and we have $|X| = |L_X| \leq n - 1$. If X contains at least one correct process, we have $X \cap L_X \cap Correct \neq \emptyset$, and the eventual multi-leadership property is satisfied for the invocations $\mathsf{leader}(X)$. If X contains no correct process, the set returned by $\mathsf{leader}(X)$ can be arbitrary. $\square_{Theorem\ 2}$

4 Ω^k_*-Based k-Set Agreement

4.1 Wait-Free k-Set Agreement

The k-set agreement has been informally stated in the introduction. It has been defined in [4]. The parameter k of the set agreement can be seen as the degree of coordination associated with the corresponding instance of the problem. The smaller k, the more coordination among the processes: $k = 1$ means the strongest possible coordination (this is the consensus problem), while $k = n$ means no coordination at all. More precisely, in a set of n processes, each of a subset of $p \geq 1$ processes proposes a value. These processes are the *participating* processes. The wait-free k-set agreement is defined by the following properties:

- Termination (wait-free): a correct process that proposes a value decides a value (whatever the behavior of the other processes).
- Agreement: no more than k different values are decided.
- Validity: a decided value is a proposed value.

4.2 Principles and Description of the Algorithm

The k-set agreement algorithm is described in Figure 2. A process p_i that participates in the k-set agreement invokes the operation $\mathsf{kset\_propose}_k(v_i)$ where v_i is the value it proposes. If it does not crash, it terminates that operation when it executes the statement $\mathsf{return}(decided_i)$ (line 11) where $decided_i$ is the value it decides.

Shared objects The processes share three objects:

- A KA object. A process p_i accesses it by invoking $KA.\mathsf{alpha\_propose}_k(r_i, v_i)$ where r_i is a round number and the value v_i proposed by p_i (line 07). Due to the properties of the KA object, the value returned by such an invocation is a proposed value or $\perp$.
- An array of atomic single-writer/multi-reader boolean registers, $PART[1..n]$. The register $PART[i]$ can be updated only by p_i; it can read by all the processes. Each entry $PART[i]$ is initialized to $false$. $PART[i]$ is switched to $true$ to indicate that p_i is now participating in the k-set agreement (line 01). $PART[i]$ is updated at most once.
- An array of atomic single-writer/multi-reader registers denoted $DEC[1..n]$. $DEC[i]$ can be written only by p_i. It can read by all the processes. Each entry $DEC[i]$ is initialized to $\perp$ (a value that cannot be proposed by the processes). When it is updated to a non-$\perp$ value v, that value v can be decided by any process. It is updated (to such a value v or $\perp$) each time p_i invokes $KA.\mathsf{alpha\_propose}_k(r_i, v_i)$ to store the value returned by that invocation (line 07).

The algorithm. The behavior of a process is pretty simple. As in Paxos, it decouples the safety part from the wait-free/termination part. The safety is ensured thanks to the KA object, while the liveness rests on Ω^k_*.

After it has registered its participation (line 01), a process p_i executes a **while** loop (lines 03-09) until it finds a non-$\perp$ entry in the $DEC[1..n]$ array. When this occurs, p_i decides such a value (lines 03 and 10-11).

Each time it executes the **while** loop, p_i first computes its local view (denoted $part_i$) of the set of the processes it perceives as being the participating processes (line 04). It then uses this participating set to invoke $\mathsf{Leader}()$ (line 05). If it does not belong to the set returned by $\mathsf{Leader}(part_i)$, p_i continues looping. Otherwise (it then belongs to set of leaders), p_i invokes the KA object (line 07) to try to obtain a non-$\perp$ value from that object. The local variable r_i is used by p_i to define the round number it uses when it invokes the KA object. It is easy to see from the management of r_i at line 02 and line 06 that each process uses increasing round numbers, and that no two processes use the same round numbers (a necessary requirement for using the KA object). The properties of KA ensure that no more than k values are decided, while the properties of Ω^k_* ensure that all the correct participating processes do terminate.

```
operation kset_propose_k(v_i):
(1)   PART[i] ← true;
(2)   r_i ← (i − n);
(3)   while (∀j : DEC[j] = ⊥) do
(4)       part_i ← {j : PART[j] ≠ ⊥};
(5)       leaders_i ← Leader(part_i);
(6)       if (i ∈ leaders_i) then r_i ← r_i + n;
(7)                               DEC[i] ← KA.alpha_propose_k(r_i, v_i)
(8)       end_if
(9)   end_while;
(10)  let decided_i = any DEC[j] ≠ ⊥;
(11)  return(decided_i)
```

Fig. 2. An Ω^k_*-based k-set agreement algorithm (code for p_i)

4.3 Proof of the Algorithm

Theorem 3. *The algorithm described in Figure 2 wait-free solves the k-set agreement problem whatever the number p of participating processes in a set of n processes (this number p being a priori unknown).*

Proof

Validity. The validity property follows from the following observations:

- The value $\perp$ cannot decided (lines 03 and 10).
- The $DEC[1..n]$ array can contain only $\perp$ or values that have been proposed to the KA object (line 07).
- Any value v_i proposed to the KA object is a value proposed to the k-set agreement.

Agreement. The agreement property follows directly from the agreement property of the KA object (that states that at most k non-$\perp$ values can be returned from that object).

Termination (wait-free). If an entry of $DEC[1..n]$ is eventually set to a non-$\perp$ value, it follows from the test of line 03 that any correct participating process terminates. So, let us assume by contradiction that no entry of $DEC[1..n]$ is ever set to a non-$\perp$ value. Let us first observe that all the leader() invocations issued by the processes are meaningful.

If no correct participating process decides, there is a time τ_0 after which we have the following:

- All the participating processes have entered the algorithm, and consequently the array $PART[1..n]$ determines the whole set of participating processes. Let X be this set of processes.
- all the leader() invocations have X as input parameter.

It follows from the eventual multi-leadership property associated with X, that there is a time $\tau_X \geq \tau_0$ such that, for all the times $\tau \geq \tau_X$, all the invocations of leader(X) return the same set L_X of at most k processes, and this set includes at least one correct participating process.

As no process decides (assumption) and each alpha_propose$_k$() invocation issued by a correct process returns (termination property of the KA object), the correct participating processes of the set X execute KA.alpha_propose$_k$() infinitely often (lines 06-07). It then follows from the convergence property of the KA object that these processes obtain non-$\perp$ values, and deposit these values in the array $DEC[1..n]$. A contradiction.

$\square_{Theorem\ 3}$

5 Building a KA Object from Registers

This section presents an implementation of a KA object from single-writer/multi-readers atomic registers. As already indicated, this algorithm is inspired from Paxos-like algorithms [8,12].

5.1 Implementing KA

An algorithm constructing a KA object is described in Figure 3. It uses an array of single-writer/multi-reader atomic registers REG. As previously, $REG[i]$ can be written only by p_i. A register $REG[i]$ is made up of three fields $REG[i].lre$, $REG[i].lrww$ and $REG[i].val$ whose meaning is the following ($REG[i]$ is initialized to $< 0, 0, \perp >$):

- $REG[i].lre$ stores the number of the $l$ast $r$ound $e$ntered by p_i. It can be seen as the logical date of the last invocation issued by p_i.
- $REG[i].lrww$ and $REG[i].val$ constitute a pair of related values: $REG[i].lrww$ stores the number of the $l$ast $r$ound $w$ith a $w$rite of a value in the field $REG[i].val$. So, $REG[i].lrww$ is the logical date of the last write in $REG[i].val$.

(To simplify the writing of the algorithm, we consider that each field of a register can be written separately. This poses no problem as each register is single writer. A writer can consequently keep a copy of the last value it has written in each register field and rewrite it when that value is not modified.)

```
operation alpha_propose_k(r, v):
(1)  REG[i].lre ← r;
(2)  for j ∈ {1, ..., n} do reg_i[j] ← REG[j] end_do;
(3)  let value_i be reg_i[j].val where j is such that ∀x : reg_i[j].lrww ≥ reg_i[x].lrww;
(4)  if (value_i = ⊥) then value_i ← v end_if;
(5)  REG[i].(lrww, v) ← (r, value_i);
(6)  for j ∈ {1, ..., n} do reg_i[j] ← REG[j] end_do;
(7)  if (|{j | reg_i[j].lre ≥ r}| > k) then return(⊥)
(8)                                   else return(value_i) end_if
```

Fig. 3. A KA object algorithm (code for p_i)

The principle that underlies the algorithm is very simple: it consists in using a logical time frame (represented here by the round numbers) to timestamp the invocations, and answering $\perp$ when the timestamp of the corresponding invocation does not lie within the k highest dates (registered in $REG[1..n].lre$). To that end, the algorithm proceeds as follows:

- Step 1 (lines 01-02): Access the shared registers.
  - When a process p_i invokes alpha_propose$_k(r, v)$, it first informs the other processes that the KA object has attained (at least) the date r (line 01). Then p_i reads all the registers in any order (line 02) to know the last values (if any) written by the other processes.
- Step 2 (lines 03-05): Determination and writing of a value.
  Then, p_i determines a value. In order not to violate the agreement property, it selects the last value ("last" according to the round numbers/logical dates) that has been deposited in a register $REG[j]$. If there is no such value it considers its own value v. After this determination, p_i writes in $REG[i]$ the value it has determined, together with its round number (line 05.
- Step 3 (lines 06-08): Commit or abort.
  - p_i reads again the shared registers to know the progress of the other processes (measured by their round numbers), line 07. If it discovers it is "late", p_i aborts returning $\perp$. (Let us observe that this preserves the agreement property.) "To be late" means that the current date r of p_i does not lie within the window defined by the k highest dates (round numbers) currently entered by the processes (these round numbers/dates are registered in the field lre of each entry of the array $REG[1..n]$).
  - Otherwise, p_i is not late. It then returns ("commits") $value_i$ (line 08). Let us observe that, as the notion of "being late" is defined with respect to a window of k dates (round numbers), it is possible that up to k processes are not late and return concurrently up to k distinct non-$\perp$ values.

It directly follows from the code that the algorithm is wait-free. Moreover, in order to expedite the alpha_propose$_k()$ operation, it is possible to insert the statement

if $(|\{j | reg_i[j].lre \geq r\}| > k)$ **then** return$(\perp)$ **end_if**

between the line 02 and the line 03. This allows the invoking process to return $\perp$ when, just after entering the alpha_propose$_k()$ operation, it discovers it is late.

5.2 Proof of the KA Object

Theorem 4. *The algorithm described in Figure 3 wait-free implements a KA object.*

Proof

Termination (wait-free). This property follows directly from the code of the algorithm (the only loops are **for** loops that trivially terminate when the invoking process is correct).

Validity. Let us observe that if a value v is written in $REG[i].val$, that value has been previously passed as a parameter in an $\mathsf{alpha\_propose}_k()$ invocation. The validity property follows from this observation and the fact that only $\perp$ or a value written in a register $REG[i]$ can be returned from an $\mathsf{alpha\_propose}_k()$ invocation.

Convergence. Let τ be a time after which there is a set of $k' \leq k$ processes such that each of them invokes $\mathsf{alpha\_propose}_k()$ infinitely often. This means that, from τ, the values of $n - k'$ registers $REG[x]$ are no longer modified. Consequently, as the k' processes p_j repeatedly invoke $\mathsf{alpha\_propose}_k()$, there is a time $\tau' \geq \tau$ after which each $REG[j].lre$ becomes greater than any $REG[x].lre$ that is no longer modified. There is consequently a time $\tau'' \geq \tau'$ after which the k' processes are such that their registers $REG[j].lre$ contain forever the k greatest timestamp values. It follows from the test done at line 07 that, after τ'', no $\mathsf{alpha\_propose}_k()$ invocation by one of these k' processes can be aborted. Consequently, each of them returns a non-$\perp$ value at line 08.

Agreement. If all invocations returns $\perp$, the agreement property is trivially satisfied. So, let us consider an execution in which at least one $\mathsf{alpha\_propose}_k()$ invocation returns a non-$\perp$ value. To prove the agreement property we show that:

- Before the first non-$\perp$ value is returned by an invocation, there is a time at which the algorithm has determined a set V of at least one and at most k non-$\perp$ values[6].
- Any value $v \neq \perp$ returned by an invocation is a value of V.

To simplify the reasoning, and without loss of generality, we assume that a process that repeatedly invokes $\mathsf{alpha\_propose}_k()$, stops invoking that operation as soon as it returns a non-$\perp$ value at line 08.

1. Invariants. $\forall j \in \{1, \ldots, n\}$:
   - $REG[j].lre$ is increasing (assumption on the successive round numbers used by p_j).
   - $REG[j].lrww \leq REG[j].lre$ (because p_j executes line 05 after line 01).
2. Among all the invocations that execute the test of line 07, let $\mathcal{I}$ be the subset of invocations for which the predicate $\big|\{j | reg_i[j].lre \geq r\}\big| \leq k$ is true. (This means that any invocation of $\mathcal{I}$ either returns a non-$\perp$ value -at line 08-, or crashes after it has evaluated the predicate at line 07, and before it executes line 08.) Among the invocations of $\mathcal{I}$, let I be the invocation with the smallest round number. Let p_{j_1} be the process that invoked I and r the corresponding round number.

[6] According to the terminology introduced in [2], the set V defines the values that are *locked*. This means that from now on the set of non-$\perp$ values that can be returned is fixed forever: no value outside V can ever be returned.

3. Time instants (see Figure 4).
   - Let τ be the time at which I executes line 05 (statement $REG[j_1] \leftarrow< r, r, v >$).
   - Let τ' be the time just after I has finished reading the array $REG[1..n]$. Without loss of generality, we consider that this is the time at which I locally evaluates the predicate of line 07.
   - Let $\tau[j]$ be the time at which I reads $REG[j]$ at line 06. We have $\tau < \tau[j] < \tau'$.

Fig. 4. Time instants with respect to accesses to the registers $REG[1..n]$

4. From $\tau[j] < \tau'$, the fact that predicate $\big|\{j | reg_{j_1}[j].lre \geq r\}\big| \leq k$ is true at τ', and the monotonicity of $REG[j].lre$, we can conclude that a necessary requirement for the predicate $REG[j].lre \geq r$ to be true at τ is that it is true at τ'.
   Let $L = \{j_1, \ldots, j_x, \ldots, j_\ell\}$ be the set of processes p_j such that $REG[j].lre \geq r$ is true at τ. As the predicate $\big|\{j | reg_i[j].lre \geq r\}\big| \leq k$ is true at τ', we have $1 \leq \ell = |L| \leq k$.
5. From the previous item, we conclude that there are at least $n - \ell \geq n - k$ entries j of the array $REG[1..n]$ such that $REG[j].lre < r$ at time τ. Let $\overline{L}$ denote this set of processes (L and $\overline{L}$ define a partition of the whole set of processes).
6. Let the *τ-time* invocation of p_j be the invocation issued by p_j whose round number is the value of $REG[j].lre$ at time τ (assuming a fictitious initial invocation if needed).
7. The τ-time invocations of the processes p_j in L define a set, denoted V, including at most $\ell \leq k$ values, such that these values are written in $REG[1..n]$ with a write timestamp (value of the field $REG[j].lrww$) that is $\geq r$. This claim follows from the following observation.
   - The τ-time invocation by p_{j_1} (namely I) writes a value and the round number r in $REG[j_1]$.
   - Let $p_{j_x} \in L$, $p_{j_x} \neq p_{j_1}$. From the definition of L, it follows that the round number of the τ-time invocation issued by p_{j_x} is $REG[j_x].lre = r' > r$. When it executes that invocation, p_{j_x} atomically executes $REG[j_x] \leftarrow< r', r', v' >$ (if it does not crash before executing the line 05).
   - It is possible that, on one side, no process in L crashes before executing line 05, and, on another side, all the values that are written are different. It consequently follows that up to $\ell \leq k$ different values (with a write timestamp $lrww \geq r$) can be written in $REG[1..n]$. Hence, V can contain up to k values.
   - Moreover, it is also possible that each process in L returns at line 08 the value it has selected at line 05 (this depends on the value of the predicate evaluated at line 07). Consequently each value of V can potentially be returned.

8. Given an execution, the previous item has extracted a non-empty set V of at most k non-$\perp$ values that can be returned. We now show that (1) from τ, only values of V can be written in $REG[1..n]$ with a timestamp field ($lrww$) greater than r, and (2) a non-$\perp$ value returned by an invocation is necessarily a value of V.
   (a) The τ-time invocation issued by any $p_j \in \overline{L}$ has a round number $REG[j].lre$ that is smaller than $REG[j_1].lre = r$ (this follows from the definition of $\overline{L}$). As by definition, r is the smallest round number during which a process finds true the predicate of line 07, it follows that any $p_j \in \overline{L}$ needs to issue an invocation with a round number greater than r to have a chance to return a non-$\perp$ value.
   (b) Let $\mathcal{I}'$ be the set of all the invocations that have a round number greater than r. They are issued by the processes of $\overline{L}$ or the processes of L whose τ-time invocation has returned $\perp$ at line 07. Let us observe that any invocation of $\mathcal{I}'$ starts after τ.
   Let I' be the first invocation of $\mathcal{I}'$ that executes 05. I' (issued by some process p_j) selects (at line 03) a value $value_j$ from a register $REG[y]$ such that $REG[y].lrww \geq REG[j_1].lrww = r$. As up to now, only processes of L have written values in $REG[1..n]$ with a write timestamp ($lrww$) $\geq r$, it follows that I' selects a value from V [7]. Consequently, this invocation does not add a new value to V.
   Let I'' be the invocation of $\mathcal{I}'$ that is the second to execute line 05. The same reasoning (including now I') applies. Etc. It follows from this induction that a value written at line 05 by an invocation of $\mathcal{I}'$ is a value of V, which proves that only values of V can be written in the array $REG[1..n]$ with a write timestamp greater than r.
   (c) Finally, an invocation that returns a value at line 08, returns the value it has written at line 05. Due to the definition of r, its round number r' is $\geq r$. It follows that the non-$\perp$ value that is returned is a value of V. $\square_{Theorem\ 4}$

6 Conclusion

Considering asynchronous systems equipped with a failure detector object, this paper focused on the set agreement problem when only a subset of the processes participate, namely, the *wait-free set agreement* problem. Wait-free means here that a correct process has to decide a value, whatever the behavior of the other processes (that can be correct or not and participate or not).

A wait-free failure detector-based k-set agreement algorithm has been presented. Its design principles follows the Paxos approach, decoupling the way the safety and the termination properties are guaranteed. The algorithm safety is based on an object denoted KA that can be built from single-writer/multi-reader atomic registers. The liveness property is based on a leader failure detector class, denoted Ω^k_*, that takes into

[7] It is possible that, when I' reads the array $REG[1..n]$ at line 02, not all the values of V have yet been written in that array. The important points are here that (1) at least one value of V has already been written in the array (namely, $REG[j_1].val$ with the timestamp $REG[j_1].lrww = r$), and (2) any register $REG[x]$ that currently contains a value not in V, is such that $REG[x].lrww < r$.

account the participating processes. The very existence of the algorithm shows that Ω_*^k is sufficient to wait-free solve the k-set agreement problem. Showing that Ω_*^k is also necessary, or defining a class of weaker failure detectors solving the k-set agreement problem, remains one of the greatest research challenges for the fault-tolerant asynchronous computing theory community.

References

1. Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computation (STOC'93)*, San Diego (CA), pp. 91-100, 1993.
2. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
3. Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.
4. Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.
5. Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
6. Guerraoui R., Kapałka M. and Kouznetsov P., The Weakest Failure Detectors to Boost Obstruction-Freedom. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer Verlag LNCS #4167, pp. 376-390, Stockholm (Sweden), 2006.
7. Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers.* 53(4):453-466, 2004.
8. Guerraoui R. and Raynal M., The Alpha of Indulgent Consensus. *The Computer Journal*. To appear, 2006.
9. Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
10. Herlihy M.P. and Penso L. D., Tight Bounds for k-Set Agreement with Limited Scope Accuracy Failure Detectors. *Distributed Computing*, 18(2): 157-166, 2005.
11. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923,, 1999.
12. Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169; 1998.
13. Loui M.C., Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research*, JAI Press, 4:163-183, 1987.
14. Mostéfaoui A., Rajsbaum S., Raynal M. and Travers C., Irreducibility and Additivity of Set Agreement-oriented Failure Detector Classes. *Proc. 25th ACM Symposium on Principles of Distributed Computing PODC'06*, ACM Press, Denver (Colorado), 2006.
15. Mostéfaoui A. and Raynal M., k-Set Agreement with Limited Accuracy Failure Detectors. *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152, 2000.
16. Mostéfaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.
17. Mostéfaoui A., Raynal M. and Travers C., Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set agreement. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer Verlag LNCS #4167, pp. 1-15, Stockholm (Sweden), 2006.

18. Neiger G., Failure Detectors and the Wait-free Hierarchy. *Proc. 14th ACM Symp. on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, 1995.
19. Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.
20. Yang J., Neiger G. and Gafni E., Structured Derivations of Consensus Algorithms for Failure Detectors. *Proc. 17th ACM Symp. on Principles of Distributed Computing (PODC'98)*, ACM Press, pp.297-308, 1998.

A Topological Treatment of Early-Deciding Set-Agreement

Rachid Guerraoui[1,2], Maurice Herlihy[3], and Bastian Pochon[1]

[1] School of Computer and Communication Sciences, EPFL
[2] Laboratory of Computer Science and Artificial Intelligence, MIT
[3] Computer Science Department, Brown University

Abstract. This paper considers the k-set-agreement problem in a synchronous message passing distributed system where up to t processes can fail by crashing. We determine the number of communication rounds needed for all correct processes to reach a decision in a given run, as a function of k, the degree of coordination, and $f \leq t$ the number of processes that actually fail in the run. We prove a lower bound of $min(\lfloor f/k \rfloor + 2, \lfloor t/k \rfloor + 1)$ rounds. Our proof uses simple topological tools to reason about runs of a full information set-agreement protocol. In particular, we introduce a topological operator, which we call the *early deciding* operator, to capture rounds where k processes fail but correct processes see only $k-1$ failures.

Keywords: Set-agreement, topology, time complexity, lower bound, early global decision.

1 Introduction

This paper studies the inherent trade-off between the degree of coordination that can be obtained in a synchronous message passing distributed system, the time complexity needed to reach this degree of coordination in a given run of the system, and the actual number of processes that crash in that run. We do so by considering the time complexity of the k-set-agreement [3] (or simply set-agreement) problem. The problem consists for the processes of the system, each starting with its own value, possibly different from all other values, to agree on less than k among all initial values, despite the crash of some of the processes. The problem is a natural generalization of consensus [9], which correspond to the case where $k = 1$.

Most studies of the time complexity of k-set-agreement focused on *worst-case global decision* bounds. Chaudhuri et al. in [4], Herlihy et al. in [14], and Gafni in [10], have shown that, for any k-set agreement protocol tolerating at most t process crashes, there exists a run in which $\lfloor t/k \rfloor + 1$ communication rounds are needed for all correct (non-crashed) processes to decide. This (worst-case global decision) bound is tight and there are indeed protocols that match it, e.g., [4].

This paper studies the complexity of *early global decisions* [5]. Assuming a known maximum number of t processes that may crash, early-deciding protocols are those that takes advantage of the effective number $f \leq t$ of failures in any

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 20–35, 2006.

run. In particular, for runs where f is significantly smaller than t, such protocols are appealing for it is often claimed that it is good practice to optimize for the best and plan for the worst.

More specifically, assuming a maximum number t of failures in a system of $n+1$ processes, we address in this paper the question of how many communication rounds are needed for all correct (non-crashed) processes to decide (i.e., to reach a *global decision*) in any run of the system where f processes fail. Interestingly, there is a protocol through which all correct processes decide within $min(\lfloor f/k \rfloor + 2, \lfloor t/k \rfloor + 1)$ rounds in every run in which at most f processes crash [11].

We prove in this paper a lower bound of $min(\lfloor f/k \rfloor + 2, \lfloor t/k \rfloor + 1)$ on the round complexity needed to reach a global decision in any run in which at most f processes crash. The bound is thus tight. Our result generalizes, on the one hand, results on worst-case global decisions for set agreement [4,14], and on the other hand, results on early global decisions for consensus [16,2]. As we discuss in the related work section, our bound is also complementary to a recent result on early *local* decisions for set-agreement [11] with an unbounded number of processes.

To prove our lower bound result, we use the topological notions of *connectivity* and *pseudosphere*, as used in [14], and we combine them with a mathematical object which we introduce and which we call the *early-deciding* operator. This combination provides a convenient way to describe the topological structure of a bounded number of rounds of an early-deciding full information synchronous message-passing set-agreement protocol.

We prove our result by contradiction. Roughly speaking, we construct the *complex* (set of points in an Euclidean space) representing a bounded number of rounds of the protocol, where k processes crash in each round, followed by a single round in which k processes crash but no process sees more than $k-1$ crashes. In a sense, we focus on all runs where processes see a maximum of k failures in each round, except in the last round where they only see a maximum of $k-1$ failures. Interestingly, even if all failures are different, all correct processes need to decide in this round (to comply with the assumption, by contradiction, of $(\lfloor f/k \rfloor + 1)$. We prove nevertheless that the *connectivity* of the resulting complex is high enough, and this leads directly to show that not all correct processes can decide in that complex, without violating the safety properties of k-set-agreement.

Roadmap. The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 gives an overview of our lower bound proof. Section 4 presents our model of computation. Section 5 presents some topological preliminaries, used in our lower bound proof. Section 6 presents the actual proof. Section 7 concludes the paper with an open problem.

2 Related Work

The set-agreement problem was introduced in 1990 by Chaudhuri in [3]. Chaudhuri presented solutions to the problem in the asynchronous system model where $k-1$ processes may crash, and gave an impossibility proof for the case where at

least k processes might crash, assuming a restricted class of distributed protocols called *stable vector protocols*.

In 1993, three independent teams of researchers, namely Herlihy and Shavit [15], Borowsky and Gafni [1], and Saks and Zaharoglou [18], proved, concurrently, that k-set-agreement is impossible in an asynchronous system when k processes may crash. All used topological arguments for showing the results. (Herlihy and Shavit later introduced in [15] a complete topological characterization of asynchronous shared-memory runs, using the concept of algebraic spans [13] for showing the sufficiency of the characterization.)

Chaudhuri et al. in [4], and Herlihy et al. in [14], then investigated the k-set-agreement problem in the synchronous message-passing system, and established that, any k-set-agreement protocol tolerating at most t process crashes, has at least one run in which $\lfloor t/k \rfloor + 1$ rounds are needed for all processes to decide. This is a worst-case complexity bound for synchronous set-agreement.

Dolev, Reischuk and Strong were the first to consider early-stopping protocols (best-case complexity). In particular they studied in [5] the Byzantine agreement problem, for which they gave the first early-stopping protocol. Keidar and Rajsbaum in [16], and Charron-Bost and Schiper in [2], considered early-deciding consensus and proved that $f + 2$ rounds are needed in the synchronous message-passing system for all processes to decide, in runs with at most f process crashes.

Early-deciding k-set-agreement was first studied by Gafni et al. in [11]. An early-deciding k-set-agreement protocol was proposed, together with a matching lower bound. As we discuss now, the bound we prove in this paper and that of [11] are in a precise sense incomparable. On the one hand, the bound was given in [11] for the case where the number n of processes is unbounded. It is in this sense a *weaker* result than the one we prove here. Indeed, that lower bound does not generalize the results on consensus where $n + 1$ (the total number of processes), and t (the number of failures that may occur in any run) are fixed, nor on the (worst-case) complexity of k-set-agreement. In the present paper, we assume that n and t are fixed and known, and we present a *global* decision lower bound result that thus generalizes the results on the time complexity of early-deciding consensus and the worst-case time complexity of k-set-agreement [4,14,16,2]. All considered global decision with a fixed number of processes.

On the other hand, the bound of [11] states that *no* single process may decide within $\lfloor f/k \rfloor + 1$ rounds. In this sense, the result of [11] characterizes a *local decision* [7] bound and is in this sense *stronger* than the bound of this paper. Coming up with a bound on local decisions and a bounded number of processes is an open question that is out of the scope of this paper.

3 Overview of the Proof

Our lower bound proof relies on some notions of algebraic topology applied to distributed computing, following in particular the work of [15]. In short, an impossibility of solving set-agreement comes down to showing that the runs, or a subset of the runs, produced by a full-information protocol (a generic protocol

where processes exchange their complete local state in any round), gathered within a *protocol complex*, have a sufficiently high *connectivity*. Connectivity is an abstract notion of algebraic topology which, when used in the context of set-agreement, captures the fact that the processes are sufficiently *confused* so that they would violate set-agreement if they were to decide some value; i.e., they would decide on more than k values in at least one of the runs. Basically, 0-connectivity corresponds to the traditional graph connectivity, whereas $(k-1)$-connectivity means the absence of "holes" of dimension k.

Our proof proceeds by contradiction. We assume that all processes decide by the end of round $\lfloor f/k \rfloor + 1$ in any run with at most f failures, and we derive a contradiction in two steps. The first step concerns rounds 1 to $\lfloor f/k \rfloor$, whereas the second part concerns round $\lfloor f/k \rfloor + 1$. The second step builds on the result of the first part. In both steps, we show that that a full information protocol $\mathcal{P}$, remains highly connected, thus preventing processes from achieving k-set-agreement.

In both steps, we only focus on a subset of all possible runs. In the first step, we gather all the runs in which at most k processes crash in any round, starting from the set of all system states where $n+1$ processes propose different values from a value range V. The protocol complex corresponding to this subset of runs is $(k-1)$-connected, at the end of any round r [14]. Roughly speaking, the $(k-1)$-connectivity of the protocol complex at the end of round $\lfloor f/k \rfloor$ is made by those runs in which $k+1$ processes have $k+1$ distinct *estimate* values (potential decisions), and would thus decide on $k+1$ distinct values if these processes had to decide at the end of round $\lfloor f/k \rfloor$).

Then, in the second step, we focus on round $\lfloor f/k \rfloor + 1$, and we extend the protocol complex obtained at round $\lfloor f/k \rfloor$ with a round in which, as before, at most k processes crash, but now every process observes at most $k-1$ crashes. In other words, in this additional round $\lfloor f/k \rfloor + 1$, every process that reaches the end of the round receives a message from at least one process that crashes in round $r+1$. The intuition behind this round is to force processes to decide at the end of round $\lfloor f/k \rfloor + 1$, and then obtain the desired contradiction with the computation of the connectivity. Indeed, any process p_i that receives, in round $\lfloor f/k \rfloor + 1$, at least one message from one of the k processes that crash in round $\lfloor f/k \rfloor + 1$, decides at the end of round $\lfloor f/k \rfloor + 1$.

This is because the subset of runs that we consider is indistinguishable for any process at the end of round $\lfloor f/k \rfloor + 1$, from a run that has at most k crashes in the first $\lfloor f/k \rfloor$ rounds, and at most $k-1$ crashes in round $\lfloor f/k \rfloor + 1$: a total of $k \lfloor f/k \rfloor + (k-1)$ crashes. In this case, processes must decide at the end of round $\lfloor f/k \rfloor + 1$.

We finally obtain our contradiction by showing that extending the protocol complex obtained at the end of round $\lfloor f/k \rfloor$, with the round $\lfloor f/k \rfloor + 1$ described in the previous paragraph, i.e., where at most k processes crash but any process observes at most $k-1$ crashes, preserves the $(k-1)$-connectivity of the protocol complex, at the end of round $\lfloor f/k \rfloor + 1$. By applying the result relating high connectivity and the impossibility of set-agreement, formalized in Theorem 3, we derive the fact that not all processes may decide at the end of round $\lfloor f/k \rfloor + 1$.

The main technical difficulty is to prove that the connectivity of the complex obtained at the end of round $\lfloor f/k \rfloor + 1$ is high-enough. The approach here is similar to that of [14] in the sense that we compute connectivity by induction, using the topological notions of *pseudosphere* and union of pseudospheres. Basically, the protocol complexes of which we compute the connectivity can be viewed as a union of n-dimensional pseudospheres which makes it possible to apply (a corollary of) the Mayer-Vietoris theorem [17]. We also use here a theorem from [12], which itself generalizes Theorem 9 and Theorem 11 of [14].

The main originality in our work is the introduction of our *early-deciding* operator, which is key to showing that the connectivity is preserved from round $\lfloor f/k \rfloor$ to round $\lfloor f/k \rfloor + 1$, i.e., even if processes see less than k failures in the last round.

4 Model

Processes. We consider a distributed system made of a set Π of $n+1$ processes, $p_0, \ldots, p_n$. Each process is an infinite state-machine. The processes communicate via message passing though reliable channels, in synchronous rounds. Every round r proceeds in three phases: (1) first any process p_i sends a message to all processes in Π; (2) then process p_i receives all the messages that have been sent to it in round r; (3) at last p_i performs some local run, changes its state, and starts round $r + 1$.

Failures. The processes may fail by crashing. When a process crashes, it stops executing any step from its assigned protocol. If any process p_i crashes in the course of sending its message to all the processes, a subset only of the messages that p_i sends are received. We assume that at most t out of the $n + 1$ processes may crash in any run. The identity of the processes that crash vary from one run to another and is not known in advance. We denote by $f \leq t$ the effective number of crashes that occur in any run.

Problem. In this paper, we consider the k-set-agreement problem. In this problem, any process p_i is supposed to propose a value $v_i \in V$, such that $|V| > k$ (otherwise, the problem is trivially solved), and eventually decide on a value v'_i, such that the following three conditions are satisfied:

(Validity) Any decided value v'_i is a value v_j proposed by some process p_j.
(Termination) Eventually, every correct process decides.
(k-set-agreement) There are at most k distinct decided values.

5 Topological Background

This section recalls some general notions and results from basic algebraic topology from [17], together with some specific ones from [14] used to prove our result.

5.1 Simplexes and Complexes

It is convenient to model a global state of a system of $n+1$ processes as an *n-dimensional simplex* $S^n = (s_0, ..., s_n)$, where $s_i = \langle p_i, v_i \rangle$ defines local state v_i of process p_i [15]. We say that the vertexes $s_0, ..., s_n$ *span* the simplex S^n. We say that a simplex T is a *face* of a simplex S if all vertexes of T are vertexes of S. A set of global states is modeled as a set of simplexes, closed under containment, called a *complex*.

5.2 Protocols

A *protocol* $\mathcal{P}$ is a subset of runs of our model. For any initial state represented as an n-simplex S, a *protocol complex* $\mathcal{P}(S)$ defines the set of final states reachable from them through the runs in $\mathcal{P}$. In other words, a set of vertexes $\langle p_{i_0}, v_{i_0} \rangle, ..., \langle p_{i_n}, v_{i_n} \rangle$ span a simplex in $\mathcal{P}(S)$ if and only if (1) S defines the initial state of $p_{i_0}, ..., p_{i_n}$, and (2) there is a run in $\mathcal{P}$ in which $p_{i_0}, ..., p_{i_n}$ finish the protocol with states $v_{i_0}, ..., v_{i_n}$. For a set $\{S_i\}$ of possible initial states, $\mathcal{P}(\cup_i S_i)$ is defined as $\cup_i \mathcal{P}(S_i)$. If S^m is a face of S^n, then we define $\mathcal{P}(S^m)$ to be a subcomplex of $\mathcal{P}(S^n)$ corresponding to the runs in $\mathcal{P}$ in which only processes of S^m take steps and processes of $S^n \backslash S^m$ do not take steps. For $m < n - t$, $\mathcal{P}(S^m) = \emptyset$, since in our model, there is no run in which more than t processes may fail.

For any two complexes $\mathcal{K}$ and $\mathcal{L}$, $\mathcal{P}(\mathcal{K} \cap \mathcal{L}) = \mathcal{P}(\mathcal{K}) \cap \mathcal{P}(\mathcal{L})$: any state of $\mathcal{P}(\mathcal{K} \cap \mathcal{L})$ belongs to both $\mathcal{P}(\mathcal{K})$ and $\mathcal{P}(\mathcal{L})$, any state from $\mathcal{P}(\mathcal{K}) \cap \mathcal{P}(\mathcal{L})$ defines the final states of processes originated from $\mathcal{K} \cap \mathcal{L}$ and, thus, belongs to $\mathcal{P}(\mathcal{K} \cap \mathcal{L})$.

We denote by $\mathcal{I}$ a complex corresponding to a set of possible initial configurations. Informally, a protocol $\mathcal{P}$ solves k-set-agreement for $\mathcal{I}$ if there exists a map δ that carries each vertex of $\mathcal{P}(\mathcal{I})$ to a decision value in such a way that, for any $S^m = (\langle p_{i_0}, v_{i_0} \rangle, ..., \langle p_{i_m}, v_{i_m} \rangle) \in \mathcal{I}$ $(m \geq n - f)$, we have $\delta(\mathcal{P}(S^m)) \subseteq \{v_{i_0}, ..., v_{i_m}\}$ and $|\delta(\mathcal{P}(S^m))| \leq k$. (The formal definition of a *solvable task* is given in [15].)

Thus, in order to show that k-set-agreement is not solvable in r rounds, it is sufficient to find an r-round protocol $\mathcal{P}$ that cannot solve the problem for some $\mathcal{I}$. Such a protocol can be interpreted as a set of worst-case runs in which no decision can be taken.

5.3 Pseudospheres

To prove our lower bound, we use the notion of *pseudosphere* introduced in [14] as a convenient abstraction to describe the topological structure of a bounded number of rounds of distributed protocol in our model. To make the paper self-contained, we recall the definition of [14] here:

Definition 1. *Let $S^m = (s_0, ..., s_m)$ be a simplex and $U_0, ..., U_m$ be a sequence of finite sets. The* pseudosphere *$\psi(S^m; U_0, ..., U_m)$ is a complex defined as follows. Each vertex of $\psi(S^m; U_0, ..., U_m)$ is a pair $\langle s_i, u_i \rangle$, where s_i is a vertex of S^m and $u_i \in U_i$. Vertexes $\langle s_{i_0}, u_{i_0} \rangle, ..., \langle s_{i_l}, u_{i_l} \rangle$ define a simplex of $\psi(S^m; U_0, ..., U_m)$ if*

and only if all s_{i_j} $(0 \leq j \leq l)$ *are distinct. If for all* $0 \leq i \leq m$, $U_i = U$, *the pseudosphere is written* $\psi(S^m; U)$.

The following properties of pseudospheres follow from their definition:

1. If $U_0, ..., U_m$ are singleton sets, then $\psi(S^m; U_0, ..., U_m) \cong S^m$.
2. $\psi(S^m; U_0, ..., U_m) \cap \psi(S^m; V_0, ..., V_m) \cong \psi(S^m; U_0 \cap V_0, ..., U_m \cap V_m)$.
3. If $U_i = \emptyset$, then $\psi(S^m; U_0, ..., U_m) \cong \psi(S^{m-1}; U_0, ..., \widehat{U_i}, ..., U_m)$, where circumflex means that U_i is omitted in the sequence $U_0, ..., U_m$.

5.4 Connectivity

Computing the connectivity of a given protocol complex plays a key role in characterizing whether the corresponding protocol may solve k-set-agreement. Informally speaking, a complex is said to be k-connected if it has no holes in dimension k or less. Theorem 3 below states that a protocol complex that is $(k-1)$-connected cannot solve k-set-agreement.

Before giving a formal definition of connectivity, we briefly recall the standard topological notions of a *disc* and of a *sphere*. We say that a complex $\mathcal{C}$ is an m-disk if $|\mathcal{C}|$ (the convex hull occupied by $\mathcal{C}$) is homeomorphic to $\{x \in \mathbb{R}^m | d(x, 0) \leq 1\}$ whereas it is an $(m-1)$-sphere if $|\mathcal{C}|$ is homeomorphic to $\{x \in \mathbb{R}^m | d(x, 0) = 1\}$. For instance, the 2-disc is the traditional two-dimensional disc, whereas the 2-sphere is the traditional three-dimensional sphere.

We adopt the following definition of connectivity, given in [15]:

Definition 2. *For* $k > 0$, *a complex* $\mathcal{K}$ *is* k-connected *if, for every* $m \leq k$, *any continuous map of the* m*-sphere to* $\mathcal{K}$ *can be extended to a continuous map of the* $(m+1)$*-disk. By convention, a complex is* (-1)*-connected if it is non-empty, and every complex is* k*-connected for* $k < -1$.

The following corollary to the Mayer-Vietoris theorem [17] helps define the connectivity of the result of $\mathcal{P}$ applied to a union of complexes:

Theorem 1. *If* $\mathcal{K}$ *and* $\mathcal{L}$ *are* k*-connected complexes, and* $\mathcal{K} \cap \mathcal{L}$ *is* $(k-1)$*-connected, then* $\mathcal{K} \cup \mathcal{L}$ *is* k*-connected.*

The following theorem from [12] generalizes Theorem 9 and Theorem 11 of [14], and helps define the connectivity of a union of pseudospheres. The proof basically reuses the arguments from [14]. Later in the paper, we use Theorem 2 to compute the connectivity of a complex to which we apply our early-deciding operator.

Theorem 2. *Let* $\mathcal{P}$ *be a protocol,* S^m *a simplex, and* c *a constant integer. Let for every face* S^l *of* S^m, *the protocol complex* $\mathcal{P}(S^l)$ *be* $(l-c-1)$*-connected. Then for every sequence of finite sets* $\{A_{0_j}\}_{j=0}^m, ..., \{A_{l_j}\}_{j=0}^m$, *such that for any* $j \in [0, m]$, $\bigcap_{i=0}^{l} A_{i_j} \neq \emptyset$, *the protocol complex*

$$\mathcal{P}\left(\bigcup_{i=0}^{l} \psi(S^m; A_{i_0}, ..., A_{i_m})\right) \text{ is } (m-c-1)\text{-connected.} \qquad \text{(Eq. 1)}$$

Proof. Since for any sequence $V_0, ..., V_l$ of singleton sets, $\psi(S^l; V_0, ..., V_l) \cong S^l$, we notice that $\mathcal{P}(\psi(S^l; V_0, ..., V_l)) \cong \mathcal{P}(S^l)$ is $(l-c-1)$-connected.

(i) First, we prove that, for any m and any non-empty sets $U_0, ..., U_m$, the protocol complex $\mathcal{P}(\psi(S^m; U_0, ..., U_m))$ is $(m-c-1)$-connected. We introduce here the partial order on the sequences $U_0, ..., U_m$: $(V_0, ..., V_m) \prec (U_0, ..., U_m)$ if and only if each $V_i \subseteq U_i$ and for some j, $V_j \subset U_j$. We proceed by induction on m. For $m = c$ and any sequence $U_0, ..., U_m$, the protocol complex $\mathcal{P}(\psi(S^m; U_0, ..., U_m))$ is non-empty and, by definition, (-1)-connected.
Now assume that the claim holds for all simplexes of dimension less than m $(m > c)$. We proceed by induction on the partially-ordered sequences of sets $U_0, ..., U_m$. For the case where $(U_0, ..., U_m)$ are singletons, the claim follows from the theorem condition. Assume that the claim holds for all sequences smaller than $U_0, ..., U_m$ and there is an index i, such that $U_i = v \cup V_i$, where V_i is non-empty ($v \notin V_i$). $\mathcal{P}(\psi(S^m; U_0, ..., U_m))$ is the union of $\mathcal{K} = \mathcal{P}(\psi(S^m; U_0, ..., V_i, ..., U_m))$ and $\mathcal{L} = \mathcal{P}(\psi(S^m; U_0, ..., \{v\}, ..., U_m))$ which are both $(m-c-1)$-connected by the induction hypothesis. The intersection is:

$$\begin{aligned}\mathcal{K} \cap \mathcal{L} &= \mathcal{P}(\psi(S^m; U_0, ..., V_i \cap \{v\}, ..., U_m)) = \\ &= \mathcal{P}(\psi(S^m; U_0, ..., \emptyset, ..., U_m)) \cong \\ &\cong \mathcal{P}(\psi(S^{m-1}; U_0, ..., \widehat{\emptyset}, ..., U_m)).\end{aligned}$$

The argument of $\mathcal{P}$ in the last expression represents an $(m-1)$-dimensional pseudosphere which is $(m-c-2)$-connected by the induction hypothesis. By Theorem 1, $\mathcal{K} \cup \mathcal{L} = \mathcal{P}(\psi(S^m; U_0, ..., U_m))$ is $(m-c-1)$-connected.

(ii) Now we prove our theorem by induction on l. We show that for any $l \geq 0$ and any sequence of sets $\{A_{i_j}\}$ satisfying the condition of the theorem, Equation 1 is guaranteed. The case $l = 0$ follows directly from (i). Now assume that, for some $l > 0$,

$$\mathcal{K} = \mathcal{P}\left(\bigcup_{i=0}^{l-1} \psi(S^m; A_{i_0}, ..., A_{i_m})\right) \text{ is } (m-c-1)\text{-connected.} \qquad \text{(Eq. 2)}$$

By (i), $\mathcal{L} = \mathcal{P}(\psi(S^m; A_{l_0}, ..., A_{l_m}))$ is $(m-c-1)$-connected. The intersection is

$$\begin{aligned}\mathcal{K} \cap \mathcal{L} &= \mathcal{P}\left(\left(\bigcup_{i=0}^{l-1} \psi(S^m; A_{i_0}, ..., A_{i_m})\right) \cap \psi(S^m; A_{l_0}, ..., A_{l_m})\right) = \\ &= \mathcal{P}\left(\bigcup_{i=0}^{l-1} \psi(S^m; A_{i_0} \cap A_{l_0}, ..., A_{i_m} \cap A_{l_m})\right).\end{aligned}$$

By the initial assumption (Equation 2), for any j, $\bigcap_{i=0}^{l-1} (A_{i_j} \cap A_{l_j}) = \bigcap_{i=0}^{l} A_{i_j} \neq \emptyset$. Thus by the induction hypothesis,

$$\mathcal{K} \cap \mathcal{L} = \mathcal{P}\left(\bigcup_{i=0}^{l-1} \psi(S^m; A_{i_0} \cap A_{l_0}, ..., A_{i_m} \cap A_{l_m})\right) \text{ is } (m-c-1)\text{-connected.}$$

By Theorem 1, $\mathcal{K} \cup \mathcal{L}$ is $(m-c-1)$-connected.

5.5 Impossibility and Connectivity

The following theorem, borrowed from [14], is based on Sperner's lemma [17]: it relates the connectivity of a protocol complex derived from a pseudosphere, with the impossibility of k-set-agreement:

Theorem 3. *Let $\mathcal{P}$ be a protocol. If for every n-dimensional pseudosphere $\psi(p_0, ..., p_n; V)$, where V is non-empty, $\mathcal{P}(\psi(p_0, ..., p_n; V))$ is $(k-1)$-connected, and there are more than k possible input values, then $\mathcal{P}$ cannot solve k-set agreement.*

6 The Lower Bound

As we pointed out in Section 3, our lower bound proof proceeds by contradiction. We assume that there is a full information protocol $\mathcal{P}$using which all correct processes can decide by round $\lfloor f/k \rfloor + 1$. We construct a complex of $\mathcal{P}$that satisfies the precondition of Theorem 3: namely, for any pseudosphere $\psi(p_0, ..., p_n; V)$, where V is non-empty, $\mathcal{P}(\psi(p_0, ..., p_n; V))$ is $(k-1)$-connected. Basically, the $(k-1)$-connectivity of the protocol complex at the end of round $\lfloor f/k \rfloor + 1$ is made by those runs in which $k+1$ processes have $k+1$ distinct estimate values, and would thus decide on $k+1$ distinct values if these processes had to decide at the end of round $\lfloor f/k \rfloor + 1$. The protocol complex corresponding to the subset of runs of $\mathcal{P}$where, in every run, at most k processes are allowed to fail, is $(k-1)$-connected, at the end of any round r, in particular $\lfloor f/k \rfloor$: this follows from the use of the topological operator §, introduced in [14]. In round $\lfloor f/k \rfloor + 1$, we extend the protocol complex with a last round in which at most k process crash, but every process observes at most $k-1$ crashes. In other words, in this last round, every process that reaches the end of the round receives a message from at least one process that crashes in the round. We show that this extension still preserves the $(k-1)$-connectivity of the protocol complex at the end of round $r+1$. We use here a notion topological operator $\mathcal{E}$. We conclude by applying the result of Theorem 3, and derive the fact that not all processes may decide at the end of round $r + 1 = \lfloor f/k \rfloor + 1$.

6.1 Single Round and Multiple Round Operators

In the proof, we use the topological round operator §, which generates a set of runs in a synchronous message-passing model, in which at most k processes may crash in any round. Operator § was introduced in [14]. We recall some results about § that are necessary for presenting our lower bound proof.

The protocol complex $§^1(S^l)$ corresponds to all single-round runs of our model, starting from an initial configuration S^l, in which up to k processes can fail by crashing. We consider the case where $k \leq t$, otherwise the protocol complex is trivial. $§^1(S^l)$ is the union of the complexes $§^1_K(S^n)$ of single-round runs starting from S^n in which *exactly* the processes in K fail. Given a set of processes, let $S^n \backslash K$ be the face of S^n labeled with the processes *not* in K. Lemmas 1, 2 and 3

below, are Lemmas 18, 21 and 22 from [14]. The first lemma says that $\S^1_K(S^n)$ is a pseudosphere, which means that $\S^1(S^n)$ is a union of pseudospheres.

Lemma 1. $\S^1_K(S^n) \cong \psi(S^n \backslash K; 2^K)$.

Lemma 2. *If $n \geq 2k$ and for all l, then $\S^1(S^l)$ is $(l-(n-k)-1$-connected.*

The connectivity result over a single round is now used to compute the connectivity over runs spanning multiple rounds.

Lemma 3. *If $n \geq rk + k$, and $\S^r$ is an r-round, $(n+1)$-process protocol with degree k, then $\S^r(S^l)$ is $(l-(n-k)-1)$-connected for any $0 \leq m \leq n$.*

6.2 Early-Deciding Operator

So far, we have characterized runs in which at most k processes may crash in a round, without being interested in how many of these crashes other processes actually see. To derive our lower bound, we focus on runs where processes see less than k failures in the last round.

We introduce for that purpose a new round operator, $\mathcal{E}^1(S^n)$, which generates all single-round runs from the initial simplex S^n (obtained following the construction of the previous paragraph), in which at most k processes crash, and any process that does not crash misses at most $k-1$ messages from crashed processes (in other words, any process that does not crash receives a message from at least one crashed process). $\mathcal{E}^1(S^n)$ is the complex of one-round runs of an $(n+1)$-process protocol with input simplex S^n in which at most k processes crash and every non-crashed process misses at most $k-1$ messages. It is the union of complexes $\mathcal{E}^1_K(S^n)$ of one-round runs starting from S^n in which *exactly* the processes in K fail and any process that does not crash misses at most $k-1$ messages.

We first show that $\mathcal{E}^1_K(S^n)$ is a pseudo-sphere, which means that $\mathcal{E}^1(S^n)$ is a union of pseudo-spheres. In the following lemma, 2^K_k denotes the set of all subsets of K of size at most $k-1$.

Lemma 4. $\mathcal{E}^1_K(S^n) \cong \psi(S^n \backslash K; 2^K_k)$.

Proof. The processes that do not crash are those in $S^n \backslash K$. Each process that does not crash may be labeled with all messages from processes that do not crash (processes in $S^n \backslash K$), plus any combination of size at most $k-1$ of the messages from processes that crash, represented by the subsets in 2^K_k. Hence, for any $i \in ids(S^n \backslash K)$, then $label(i)$ concatenates $S^n \backslash K$, plus a particular subset of K.

To compute the union of all pseudo-spheres, we characterize their intersection and apply Theorem 2. We order the sets K in the lexicographic order of process ids, starting from the empty set, singleton sets, 2-process sets, etc. Let $K_0, \ldots, K_l$ be the ordered sequence of process ids less than or equal to K_l, listed in lexicographic order.

Lemma 5

$$\bigcup_{i=0}^{l-1} \mathcal{E}^1_{K_i}(S^n) \cap \mathcal{E}^1_{K_l}(S^n) \cong \bigcup_{j \in K_l} \psi(S^n \backslash K_l; 2_k^{K_l - \{j\}}).$$

Proof The proof proceeds in two parts, first for the $\subseteq$ inclusion, then for the $\supseteq$ inclusion.

For the $\subseteq$ inclusion, we show that any $\mathcal{E}^1_{K_i}(S^n) \cap \mathcal{E}^1_{K_l}(S^n)$ is included in $\psi(S^n \backslash K_l; 2_k^{K_l - \{j\}})$ for some j in K_l:

$$\begin{aligned}
\mathcal{E}^1_{K_i}(S^n) \cap \mathcal{E}^1_{K_l}(S^n) &\cong \psi(S^n \backslash K_i; 2_k^{K_i}) \cap \psi(S^n \backslash K_l; 2_k^{K_l}) && (1)\\
&\cong \psi((S^n \backslash K_i) \cap (S^n \backslash K_l); (2_k^{K_i}) \cap (2_k^{K_k})) && (2)\\
&\cong \psi(S^n \backslash (K_i \cup K_l); 2_k^{K_i \cap K_l}) && (3)\\
&\subseteq \psi(S^n \backslash K_l; 2_k^{K_l - \{j\}}). && (4)
\end{aligned}$$

Equation 1 follows from the definition. Equations 2 and 3 follow from basic properties of pseudo-spheres. Equation 4 follows from the following observation: since K_i precedes K_l in the sequence and $K_i \neq K_k$, then there exists at least one process $p_j \in K_l$ and $p_j \notin K_i$. Thus we have (i) $S^n \backslash (K_i \cup K_l) \subseteq S^n \backslash K_l$ and (ii) $2_k^{K_j \cap K_l} \subseteq 2_k^{K_l - \{j\}}$.

For the $\supseteq$ inclusion, we observe that for any process p_j, each set $K_l - \{j\}$ precedes K_l in the sequence. Hence for any process p_j, we have:

$$\begin{aligned}
\mathcal{E}^1_{K_l - \{j\}}(S^n) \cap \mathcal{E}^1_{K_l}(S^n) &\cong \psi(S^n \backslash K_l - \{j\}; 2_k^{K_l - \{j\}}) \cap \psi(S^n \backslash K_l; 2_k^{K_l}) && (5)\\
&\cong \psi((S^n \backslash K_l - \{j\}) \cap (S^n \backslash K_l); 2_k^{K_l - \{j\}} \cap 2_k^{K_l}) && (6)\\
&\cong \psi(S^n \backslash K_l; 2_k^{K_l - \{j\}}). && (7)
\end{aligned}$$

Equation 5 follows from the definition of the early-deciding operator. Equation 6 follows from basic properties of pseudo-spheres, presented in Section 5.3. Equation 7 follows from the fact that $K_l - \{j\} \cap K_l = K_l - \{j\}$.

We denote $\mathcal{E}^1(S^n)$ the protocol complex for a one-round synchronous $(n+1)$-process protocol in which no more than k processes crash, and every process that does not crash misses at most $k-1$ messages from processes that crash.

Lemma 6. *For $n \geq 2k$, then $\mathcal{E}^1(S^m)$ is $(k - (n-m) - 1)$-connected.*

Proof. We have three cases: (i) $m = n$, (ii) $n - k \leq m < n$, and (iii) $m < n - k$.

For case (i), let $K_0, \ldots, K_l$ be the sequence of sets of k processes that crash in the first round ordered lexicographically, that are less or equal to K_l. Let K_l be the maximal set of k processes, i.e., $K_l = \{p_{n-k+1}, \ldots, p_n\}$. Then we have:

$$\mathcal{E}^1(S^n) = \bigcup_{i=0}^{l} \mathcal{E}^1_{K_i}(S^n).$$

We inductively show on l that $\mathcal{E}^1(S^n)$ is $(k-1)$-connected. First, observe that for $l = 0$, then $\mathcal{E}^1_{K_0}(S^n) \cong \psi(S^n; \{\emptyset\}) \cong S^n$ which is $(n-1)$-connected. As $n \geq 2k$, $n-1 \geq k-1$, and $\mathcal{E}^1_{K_0}(S^n)$ is $(k-1)$-connected.

For the induction hypothesis, assume that:

$$\mathcal{K} = \bigcup_{i=0}^{l-1} \mathcal{E}^1_{K_i}(S^n)$$

is $(k-1)$-connected. Let the complex $\mathcal{L}$ be:

$$\mathcal{L} = \mathcal{E}^1_{K_l}(S^n) = \psi(S^n \backslash K_l; 2^{K_l}_k).$$

As $dim(S^n \backslash K_l) \geq n-k$, $\mathcal{L}$ is $(n-k-1)$-connected by Corollary 10 of [14]. As $n \geq 2k$, $\mathcal{L}$ is $(k-1)$-connected.

We want to show that $\mathcal{K} \cup \mathcal{L}$ is $(k-1)$-connected, and for that, we need to show that $\mathcal{K} \cap \mathcal{L}$ is at least $(k-2)$-connected. We have:

$$\mathcal{K} \cap \mathcal{L} = \bigcup_{i=0}^{l-1} \mathcal{E}^1_{K_i}(S^n) \cap \mathcal{E}^1_{K_l}(S^n) \tag{8}$$

$$= \bigcup_{j \in K_l} \psi(S^n \backslash K_l; 2^{K_l - \{j\}}_k). \tag{9}$$

Equation 8 follows from the definition of $\mathcal{K}$ and $\mathcal{L}$. Equation 9 follows from Lemma 5.

Now let $A_i = 2^{K_l - \{i\}}_k$. We know that:

$$\bigcap_{i \in K_l} A_i = \{\emptyset\} \neq \emptyset.$$

and $S^n \backslash K_l$ has dimension at least $n-k$, so Corollary 12 of [14] implies that $\mathcal{K} \cap \mathcal{L}$ is $(n-k-1)$-connected. As $n \geq 2k$, $\mathcal{K} \cap \mathcal{L}$ is $(k-1)$-connected.

For case (ii), $n-k \leq m < n$. Recall that $\mathcal{E}^1(S^m)$ is the set of runs in which only processes in S^m take steps. As a result, the corresponding protocol complex is equivalent to the complex made of runs of $m+1$ processes, out of which $k-n+m$ may be faulty. If we now substitute m for n, and $k-n+m$ for k, $\mathcal{E}^1(S^m)$ is $(k-(n-m)-1)$-connected.

For case (iii), $m < n-k$, $k-(n-m)-1 < -1$ and thus, $\mathcal{E}^1(S^m)$ is empty.

Combining our one-round operator $\mathcal{E}$ and the round operator $\mathcal{S}$ corresponding to the set of runs in which at most k processes crash in a round, we obtain the following:

Lemma 7. *If $n \geq (r+1)k+k$, $\mathcal{E}^1(\mathcal{S}^r(S^m))$ is an $(r+1)$-round, $(n+1)$-process protocol with degree k, then $\mathcal{E}^1(\mathcal{S}^r(S^m))$ is $(k-(n-m)-1)$-connected, for any $0 \leq m \leq n$.*

Proof. We prove the theorem by induction on r. For the base case $r = 0$, $n \geq 2k$ and thus in this case, Lemma 6 proves that $\mathcal{E}^1(S^m)$ is $(k-(n-m)-1)$-connected. For the induction hypothesis, assume the claim holds for $r-1$.

We first consider the case where $m = n$. We denote by $K_0, \ldots, K_l$ the sequence of all sets of processes less than or equal to K_l, listed in lexicographic order. The set of r-round runs in which *exactly* the processes in K_i fail in the first round can be written as $\S_i^{r-1}(\S^1_{K_i}(S^n))$, where $\S_i^{r-1}$ is the complex of for an $(r-1)$-round, $(t-|K_i|)$-faulty, $(n+1-|K_i|)$-process full-information protocol. The $\S_i^{r-1}$ are considered as different protocols because the $\S^1_{K_i}(S^n)$ have varying dimensions. We inductively show that if $|K_l| \leq k$, then:

$$\bigcup_{i=0}^{l} \mathcal{E}^1(\S_i^{r-1}(\S^1_{K_i}(S^n))) \text{ is } (k-1)\text{-connected.}$$

The claim then follows when K_l is the maximal set of size k.

For the base case, we have $l = 0$, $K_0 = \emptyset$, and thus $\S^1_{\emptyset}(S^n)$ is $\psi(S^n; 2^{\emptyset}) \cong S^n$, and $\mathcal{E}^1(\S^{r-1}(S^n))$ is $(k-1)$-connected by the induction hypothesis on r.

For the induction step on l, assume that:

$$\mathcal{K} = \bigcup_{i=0}^{l-1} \mathcal{E}^1(\S_i^{r-1}(\S^1_{K_i}(S^n))) \text{ is } (k-1)\text{-connected.}$$

By Lemma 1, we have:

$$\mathcal{L} = \mathcal{E}^1(\S_l^{r-1}(\S^1_{K_l}(S^n))) = \mathcal{E}^1(\S_l^{r-1}(\psi(S^n \backslash K_l; 2^{K_l}))).$$

We recall that $\mathcal{E}^1(\S_l^{r-1})$ is a rk-faulty, $(n+1-|K_l|)$-process, r-round protocol, where $n+1-|K_l| \geq rk$, so by the induction hypothesis, for each simplex $S^d \in \S^1_{K_l}(S^n) = \psi(S^n \backslash K_l; 2^{K_l})$, $\mathcal{E}^1(\S_l^{r-1}(S^d))$ is $(k-(n-|K_l|-d)-1)$-connected. By Theorem 2, $\mathcal{E}^1(\S_l^{r-1}(\psi(2\backslash K_l; 2^{K_l}))) = \mathcal{E}^1(\S_l^{r-1}(\S^1_{K_l}(S^n))) = \mathcal{L}$ is $(k-1)$-connected.

We claim the following property:

Claim

$$\begin{aligned}\mathcal{K} \cap \mathcal{L} &= \bigcup_{i=0}^{l-1} \mathcal{E}^1(\S_i^{r-1}(\psi(S^n \backslash K_i; 2^{K_i}))) \cap \mathcal{E}^1(\S_l^{r-1}(\psi(S^n \backslash K_l; 2^{K_l}))) \\ &= \mathcal{E}^1(\widetilde{\S}_l^{r-1}\left(\bigcup_{i \in K_l} \psi(S^n \backslash K_l; 2^{K_l - \{i\}})\right)),\end{aligned}$$

where $\widetilde{\S}_l^{r-1}$ is a protocol identical to $\S_l^{r-1}$ except that $\widetilde{\S}_l^{r-1}$ fails at most $k-1$ processes in its first round.

Proof. For the $\subseteq$ inclusion, in the exact same manner as we have seen in the proof of Lemma 5 and, for each i, there is some $j \in K_l$ such that:

$$\psi(S^n \backslash K_i \cap S^n \backslash K_l; 2^{K_i \cap K_l}) \subseteq \psi(S^n \backslash K_l; 2^{K_l - \{j\}}).$$

We still need to show how $\mathcal{E}^1(\S_i^{r-1})$ and $\mathcal{E}^1(\S_l^{r-1})$ intersect. Because p_j has already failed in $\mathcal{E}^1(\S_l^{r-1})$, the only runs $\mathcal{E}^1(\S_i^{r-1})$ that are also present in $\mathcal{E}^1(\S_l^{r-1})$ are ones in which p_j fails without sending any messages to non-faulty processes. But then $\mathcal{E}^1(\S_i^{r-1})$, and therefore $\mathcal{E}^1(\S_l^{r-1})$, can fail at most $k-1$ processes that do send messages to non-faulty processes. Any such run is also a run of $\mathcal{E}^1(\widetilde{\S}_l^{r-1})$.

For the reverse inclusion $\supseteq$, we have seen in Lemma 5 that for each $j \in K_l$:

$$\mathcal{E}^1_{K_l - \{j\}}(S^n) \cap \mathcal{E}^1_{K_l}(S^n) \cong \psi(S^n \backslash K_l; 2^{K_l - \{j\}}_k).$$

It turns out that the same argument also holds for the case:

$$\S^1_{K_l - \{j\}}(S^n) \cap \S^1_{K_l}(S^n) \cong \psi(S^n \backslash K_l; 2^{K_l - \{j\}}).$$

The set of runs in which the two protocols overlap are exactly those runs in which $\mathcal{E}^1(\S_i^{r-1})$ immediately fails p_j, and in which $\mathcal{E}^1(\S_l^{r-1})$ fails no more than $k-1$ processes. These runs comprise $\mathcal{E}^1(\widetilde{\S}_l^{r-1})$.

While $\S_l^{r-1}$ has degree k, $\widetilde{\S}_l^{r-1}$ has degree $k-1$. By the induction hypothesis on r, for any simplex S^{n-k}, $\widetilde{\S}_l^{r-1}(S^{n-k})$ is $(k-2)$-connected. Let $A_i = 2^{K_l - \{i\}}$, for $i \in K_l$. As $\cap_{i \in K_l} A_i = \{\emptyset\} \neq \emptyset$, $\mathcal{K} \cap \mathcal{L}$ is $(k-2)$-connected by Claim 6.2 and Theorem 2. The claim now follows from Theorem 1.

If $n > m \geq n-k$, $\mathcal{E}^1(\S^r(S^m))$ is equivalent to an m-process protocol in which at most $k-(n-m)$ processes fail in the first round, and k thereafter. This protocol has degree $k-(n-m)$, so $\mathcal{E}^1(\S^r(S^m))$ is $(k-(n-m)-1)$-connected.

When $m < n-k$, $k-(n-m)-1 < -1$ and $\mathcal{E}^1(\S^r(S^m))$ is empty, so the condition holds vacuously.

Theorem 4. *If $n \geq k\lfloor t/k \rfloor + k$, then in any solution to k-set-agreement , not all processes may decide earlier than within round $\lfloor f/k \rfloor + 2$ in any run with at most f failures, for $0 \leq \lfloor f/k \rfloor \leq \lfloor t/k \rfloor - 1$.*

Proof. Consider the protocol complex $\mathcal{E}^1(\mathcal{S}^{\lfloor f/k \rfloor}(S^m))$. We have $k(\lfloor f/k \rfloor + 1) + 1 \leq k\lfloor t/k \rfloor + k \leq n$, thus Lemma 7 applies. Hence $\mathcal{E}^1(\mathcal{S}^{\lfloor f/k \rfloor}(S^m))$ is $(k-(n-m)-1)$-connected for any f such that $\lfloor f/k \rfloor \leq \lfloor t/k \rfloor - 1$, and $0 \leq m \leq n$. The result now holds immediately from Theorem 3.

7 Concluding Remark

This paper establishes a lower bound on the time complexity of early-deciding set-agreement in a synchronous model of distributed computation. This lower bound also holds for synchronous runs of an eventually synchronous model [8] but we conjecture a larger lower bound for such runs. Determining such a bound, which would generalize the result of [6], is an intriguing open problem.

As we discussed in the related work section, although, at first glance, the local decision lower bound presented in [11] seems to imply a global decision on

k-set-agreement, the model in which early-deciding k-set-agreement was investigated in [11] relies on the fact that the number of processes is not bounded. In fact, the proof technique we used here is fundamentally different from [11]: in [11], the proof is based on a pure algorithmic reduction whereas we use here a topological approach. Unifying these results would mean establishing a local decision lower bound assuming a bounded number of processes. This, we believe, is an open challenging question that might require different topological tools to reason about on-going runs.

References

1. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computation. In *Proceedings of the* 25^{th} *ACM Symposium on the Theory of Computing (STOC'93)*, pages 91–100. ACM Press, 1993.
2. B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus. *Journal of Algorithms*, 51(1):15–37, 2004.
3. S. Chaudhuri. More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.
4. S. Chaudhuri, M. Herlihy, N. Lynch, and M. Tuttle. Tight bounds for k-set agreement. *Journal of the ACM*, 47(5):912–943, 2000.
5. D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.
6. P. Dutta and R. Guerraoui. The inherent price of indulgence. *Distributed Computing*, 18(1):85–98, 2005.
7. P. Dutta, R. Guerraoui, and B. Pochon. Tight lower bounds on early local decisions in uniform consensus. In *Proceedings of the* 17^{th} *International Symposium on Distributed Computing (DISC'03)*, Lecture Notes in Computer Science, pages 264–278. Springer-Verlag, October 2003.
8. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, 1988.
9. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
10. E. Gafni. Round-by-round fault detector — unifying synchrony and asynchrony. In *Proceedings of the* 17^{th} *ACM Symposium on Principles of Distributed Computing (PODC'98)*, 1998.
11. E. Gafni, R. Guerraoui, and B. Pochon. From a static impossibility to an adaptive lower bound: the complexity of early deciding set agreement. In *Proceedings of the* 37^{th} *ACM Symposium on Theory of Computing (STOC'05)*, May 2005.
12. R. Guerraoui, P. Kouznetsov, and B. Pochon. A note on set agreement with omission failures. *Electronic Notes in Theoretical Computing Science*, 81, 2003.
13. M. Herlihy and S. Rajsbaum. Algebraic spans. In *Proceedings of the* 14^{th} *ACM Symposium on Principles of Distributed Domputing (PODC'95)*, pages 90–99, New York, NY, USA, 1995. ACM Press.
14. M. Herlihy, S. Rajsbaum, and M. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the* 17^{th} *ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 133–142, 1998.
15. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

16. I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults – a tutorial. *SIGACT News, Distributed Computing Column*, 32(2):45–63, 2001.
17. J. R. Munkres. *Elements of Algebraic Topology*. Addison-Wesley, Reading MA, 1984.
18. M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, March 2000. A preliminary version appeared in the Proceedings of the 25^{th} ACM Symposium on the Theory of Computing (STOC'93).

Renaming with k-Set-Consensus: An Optimal Algorithm into $n + k - 1$ Slots

Eli Gafni

Department of Computer Science, UCLA, Los Angeles, CA., 90095 USA
eli@ucla.edu

Abstract. Recently Mostefaoui, Raynal, and Travers [1] showed that with the use of k-set-consensus they can strongly rename n processors into the range of $2n - \lceil n/k \rceil$. That is the overhead is $n - \lceil n/k \rceil$. Here we reduce the range to $n + k - 1$, i.e. we reduce the overhead to $k - 1$. For $k = c$ the improvement in the overhead is $O(n)$. We first argue that such an algorithm using topological embedding does exist. We then come-up with a novel explicit algorithm that does not require explicit embedding. The latter technique employed is of independent importance and interest.

1 Introduction

The Strong Renaming Problem, also called Adaptive Renaming, on n processors [2] is for all $q \leq n$ participating processors to output each a unique individual integer slot in the range $1, \ldots, M(q)$. It is known that if the system is read-write wait-free than $M(q) = 2q - 1$ is an upper and lower bound [2,21]. The Weak Renaming Problem on q processors, also called Identical or Comparison Renaming [2], imposes the same range constraint $F(n)$ on participating sets of any size $q \leq n$, while on the other hand it imposes symmetry over the computation. This paper deals only with Strong Renaming.

In [14] it was shown that given any k-set-consensus object then for $n > k$ the range can be reduced to $M(n) = 2n - 2$. On the flip side, if $n = k + 1$ and one could rename into $M(n) = 2n - 2$ than such a renaming object could be used to for $(k + 1, k)$-set-consensus.

While [14] was mainly concerned with showing the equivalence of tasks, in the case at hand between reduced range Strong Renaming, and set-consensus, recently Mostefaoui, Raynal, and Travers [1] went further and investigated the quantification question of how much can the renaming range be reduced given that any number of processors can reach k-set-consensus. They give a nice, simple, clean variation of the algorithm in [3], which reduces $M(n)$ to $2n - \lceil n/k \rceil$.

In this paper we observe that to achieve the result of [1] there is no need to use the full power of set-consensus, and indeed a close scrutiny of the technique in the paper reveals it was not used.

The k-set-consensus problem [12] is a variation of consensus, in which each processor wait-free elects a participating processor, and the number of elected processor should not exceed k. With $k = 1$ we get the standard consensus. A weaker problem, the k-test-and-set problem [9], is a variation of test-and-set in which each processor

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 36–44, 2006.

wait-free decides 0 or 1 and the number of participating processors which decide 0, is greater than 1 and less or equal to k. The availability of k-test-and-set is equivalent to being able to achieve k-set-consensus between $k + 1$ processors (denote by $(k + 1, k)$-set-consensus), but not between $k+2$ processors. That $(k+1, k)$ is strictly weaker than $(k + 2, k)$ is a simple consequence of the BG simulation [9,4].

It turns out that for the result of Mostefaoui, Raynal, and Travers [1] k-test-and-set is sufficient rather than k-set-consensus. Consequently, they employ the power of k-set-consensus for any number of processor, denoted by (∞, k), only for the case $n = k + 1$. Yet, for $n = k + 2$ they do not take advantage of the full power of the available object. Can one utilized that unused power? This paper answer the question positively by showing that with k-set-consensus for any number of processors one can get a strictly, order of magnitude, better result.

Mostefaoui, Raynal, and Travers [1] show that with k-set-consensus they can achieve $M(n) = 2n - \lceil n/k \rceil$. For say $k = 2$ this will yield $1.5n$. In contrast we reduce the range to $M(n) = n + k - 1$, which for $k = 2$ is $n + 1$, i.e., just one extra slot overhead. Resulting an $O(n)$ improvement.

We do this using ideas from the simplex-agreement algorithm in [10,21]. For a task to be solvable using k-set-consensus, the link of any simplex has to be $k - 1$-connected. If this is the case, the algorithm in [10] proceeds in round-by-round [6] phases. Implicitly all processes agree on a sub-simplex s of processors that has converged. They then using k-set-consensus decide on at most k starting vertices in the link of s, and they then reach epsilon-agreement [7] on the link since it is $k - 1$-connected. Consequently, at least one more processor can converge. In the case of renaming as we argue next, for a link of a simplex to be $k - 1$-connected we need an overhead of at most $k - 1$ slots, hence the result.

Doing epsilon-agreement [7] on the link of s in the renaming case, with at most $k - 1$ slots overhead, and at most k actual starting points, is a task A that is solvable if and only if the $k - 1$ overhead results in $k - 1$ connectivity. As we explain in the body, this task A is derived from the renaming task. We call this task the *c*olorless renaming task. To show that $k - 1$ overhead results in $k - 1$ connectivity, we use a rather novel approach. We define a task B on k processors whose first phase using the general simplex-agreement algorithm in [10] is actually the epsilon-agreement problem we want to solve. Thus if B is solvable then A is. To show that B is solvable we give an algorithm, which in our case is a variation of the renaming algorithm in [2].

Yet, this scheme that relies on the solvability of an epsilon-agreement in a task A, relies on some given embedding of a provably exiting protocol complex for the task B, thus we call the resulting algorithm, an *implicit* algorithm.

The key to the explicit algorithm is the is this read-write wait-free solution to the task B on k processors. While in the implicit algorithm we have used it existence to argue the exitence of a solution to the colorless version, here we use the algorithm itself.We then use the equivalence between regular shared-memory and Iterated Immediate Snapshots (IIS) [10], and the explicit transformation given there to arrive at an explicit algorithm for B in the IIS model. We then show how to solve the task A given an IIS solution to B, by showing how any number of processors can solve the colorless version of the task of Immediate Snapshots on k processors. The latter uses the idea

of "shepherding-tokens" down the ladders of Immediate-Snapshot implementation [3]. The idea of "shepherding-tokens" is not new and has been employed by the author numerous times in the past. Yet, combining the simulation of read-write by IIS, together with a solution to the colorless IS, to produce a phase in a convergence algorithm, is a new, major step, in deriving a general schema of solving tasks explicitly.

The paper is organized as follow: In the next section we outline the general scheme by which both the implicit and explicit algorithm evolve. We then outline the implicit algorithm followed by a section that outlines the explicit algorithm. Both sections are divided into two subsections. Since both the implicit and explicit algorithm evolve in round-by-round phases, we use the detailed explanation of the first simpler phase, as an introduction to the more involved phases that follows. The first phase is simpler since there a common-knowledge by all processors of the initial conditions of the phase - something which is lost in all but the first phase. We finish with concluding remarks.

2 The Scheme

The algorithm evolves in round-by-round phases [6]. At the beginning of a round processors select at most k distinct processors which have not been assigned a slot as yet, using k-set consensus. With each selected processor p_i there is an associated *converge set* CS_i whose elements are pairs (p_j, sl_j), each consisting of a processor name p_j and a slot sl_j. The interpretation is that p_i "suspects" that processors in tuples CS_i have terminated, each acquiring the associated slot. An invariant of the algorithm is that if p_j terminated acquiring sl_j then the pair (p_j, sl_j) will belong to any set CS_i. Similarly, if slot sl_j is suspected by some processor to have been taken by processor p_j, then that slot is not suspected by any processor to have been taken by any other processor other than p_j.

At each phase we are guaranteed the progress that at least one new processor will terminate acquiring a unique individual slot. The algorithm maintains the invariant that if cumulatively until that phase, m distinct processors have been selected, then the largest slot acquired is $m + k - 1$.

The first phase is easier than other phases since at the first phase all processors p_i start with $CS_i = \emptyset$. The detailed description and analysis of the first phase is a good introduction to the more involved generic phases that precede it, when the converge set of different processors might differ.

3 The Implicit Algorithm

3.1 The First Phase

Using k-set consensus each processor selects a participating processor and consequently the number of selected processor is at most k. At least one selected processor p_i will choose some slot sl_i and terminate in the first phase. All processors p_j which will not terminate and move to the next phase will have their suspected set CS_j containing the element (p_i, sl_i) and $sl_i \leq 2k-1$. To affect this outcome, we now face a new interesting task which we call *colorless family renaming*:

The task is as follows: Let all processor that select p_i be considered to belong to the p_i *family*. Thus we have the processors partitioned into at most k families. If a processor from the p_i family participates in the task let us say that the family p_i participates. Thus, wait free, each processors has to decide a unique slot sl_j for some participating family p_j, such that no other processor decides another slot for p_j, and no other family p_l is decided by any processor to slot sl_j. Moreover, if the number of families that participate is m then the highest slot assigned is $2m - 1$. This is similar to group-renaming [15], with two distictions. In group-renaming a processor from family p_i has to decide a slot for family p_i, while in colorless family renaming, it can decide a slot for any participating family p_j rather than its own, p_i. The second distinction is that while in group-renaming few slots may be decided for the same family, here a family if decided is decided to the same slot by all processor who decide for that family.

Is colorless family renaming read-write wait-free solvable?

We answer on the affirmative through a rather general result that for any task T one can define a new task called colorless family T, denoted by $CF(T)$, and that if T is read-write wait-free solvable then $CF(T)$ is. If we take the colorless family renaming task we can see it is the colorless family version of renaming. Since renaming is solvable, this will prove that the task of colorless family renaming is read-write wait-free solvable.

3.2 The Implicit r/w Wait-Free Solvability of $CF(T)$

Let T be a task. We define a new task $CF(T)$ where for each processor p_i in the task T, we now in $CF(T)$ have a family of processors, called family p_i. If P is the participating set of processors in $CF(T)$, we say that the family p_i participates if a processor that belong to the family p_i also belongs to P. A participating processor in $CF(T)$ has to output a pair (p_j, O_j) such that p_j is a participating family, and all the set of outputs together can be completed to a valid tuple in T, for the processors in T that correspond to the participating families in P.

Theorem 1. *If T is r/w wait-free solvable then $CF(T)$ is.*

Proof. This is a simple corollary of the simplex convergence algorithm in [10,21] and the HS conditions of wait-free solvability [21]. In the first phase of the convergence algorithm in [10] processors do epsilon-agreement, and the observation here. which is not new, is that epsilon agreement is solvable for groups rather than just individual processors.

3.3 Subsequent Phases

Above we have seen that there exist an algorithm for the first phase. We now describe the process of transiting from phase to phase. This description is specialization to renaming of the transition from phase to phase in the simplex convergence of [10].

After the first phase assume processor p_j has output (p_i, sl_i) in the colorless family renaming. It posts its output in shared-memory. It then snapshots the posted pairs to get the vector of pairs S_j. It then posts the vector S_j, and snapshot the posted vectors. Let S_{min} be the smallest vector it snapshots in the posted vectors, and S_{max} be the largest.

If a pair $(p_j, sl_j) \in S_{min}$ then processor p_j terminates and quits with an output sl_j. Else, it sets $CS_j = S_{max}$ and is ready to move to the second round. It is easy to see that any maximum vector contains any minimum vector. The set of pairs in CS_j are the processors it suspects might have quit taking the associated slots.

We now make some observations: 1. If processor p_j terminated with output sl_j, then the pair (p_j, sl_j) appears in CS_l for all l, and 2. A processor p_j which appears in a pair within some set CS_l, and does not appear in some smaller set $CS_{l'}$, has not quit.

In general, after solving the task within phase r assume p_j outputs from the task is (p_i, sl_i). But it also has a set CS_j carried over from phase $r-1$. At the end of the first phase, CS_j carried over from the previous set is the empty set. Thus together with choosing S_{max} and S_{min} as before, it also posts CS_j. It then takes the intersection of all CS_i posted to get its temporary CS_j. It then union this temporary CS_j with S_{max} to get it final CS_j at the end of round r.

In the second phase processors propose to k-set-consensus an id of a processor that has not terminated (say, their own), together with its CS snapshot. Thus, processors divide themselves again into families. But, unlike the first phase, we now face a more complex task. With each family p_i we have an associated set of processors CS_i with the potential slots they terminated with. We want each processor to assign some family to a slot as in phase 1. Yet now the assigned slot is not to contradict the possibility that some processor has already acquired this slot in the previous phase and quit with it. That is, the slot assigned has to be compatible with the intersection of all the CSs of the participating families. Moreover, if m families participate in the phase, then the slot assigned is at furtherest the $2m-1$ slot which is not suspected to have been taken in the previous phase.

Call this task colorless family renaming with initial compatibility constraints, $CFRICC$. We argue that $CFRICC$ is read-write wait-free solvable. This can be seen by looking at the single processor per family version, i.e. the standard renaming task with initial constrains. It is easy to see that a "raising flags and ranking" algorithm a la [2], solves the problem. Thus invoking the Colorless Family Theorem, we conclude that the task is solvable.

4 An Explicit Algorithm

All that we change in order to get an explicit rather than implicit algorithm is the way we solve the renaming subtask that needs to be solved in every phase. It was solved implicitly by assuming embedding of the protocol complex and now we outline explicit solution that does nor require embedding. The handling of the "suspect-set" CS remains as in the implicit algorithm. As before, we now describe first an explicit solution to the colorless family renaming without initial constraints. It will serve as a good introduction to the explicit algorithm with the initial constraints.

4.1 An Explicit Solution to the Colorless Family Renaming with No Initial Constraints

In [3] we have a renaming algorithm for m processors that uses at most m immediate snapshots. Thus, it will suffice if we can solve the Colorless Family Immediate Snapshot

problem. I.e. we have some m families of processors, and a processor has to output an immediate snapshot for some participating family.

The Immediate-Snapshot (IS) task for a participating set P is for processor p_i to output a set $S_i \subseteq P$ such that:

1. $p_i \in S_i$
2. S_i and S_j are related by containment, and
3. if $p_i \in S_j$ then $S_i \subseteq S_j$.

The algorithm for p_i to solve IS shepherding token $q = i$ is as follows:
$T[1, ..., m, 1, ..., m]$ initially $\perp$

1. For $L = m$ down to 1 do
   (a) $T[L, q] := 1$
   (b) $S_i = \{j | T[L, j] = 1\}$
   (c) if $|S_i| = L$ return S_i
2. od

To solve the colorless family version we allow many processors to shepherd the same token as follow:
$TW[1, ..., m, 1, ...m]$, $TR[1, ..., m, 1, ..., m]$ initially $\perp$

1. For $L = m$ down to 1 do
   (a) $TW[L, q] := 1$
   (b) $S_i = \{j | TW[L, j] = 1\}$
   (c) if $|S_i| = L$ then return $(p_j, S_j = S_i)$ where j is such that $j \in S_i \wedge TR[Ll, j] = \perp$
   (d) else
      i. $TR[L, q] := 1$
      ii. $S_i = \{j | TW[L, j] = 1\}$
      iii. if $|S_i| = L$ then return $(p_j, S_j = S_i)$ where j is such that $j \in S_i \wedge TR[L, j] = \perp$
2. od

The interpretation is that a processor from one family can push a token for any participating family. Renaming in [3] is a sequence of n immediate-snapshots. In the first immediate-snapshot a processor can push the token of its own family. Yet he may be able to determine a value only for another family rather than it own. Thus it goes to the next immediate-snapshot simulating a token for another family for which the value obtained in the first immediate-snapshot is known.

For a processor to get a value in an immediate snapshot all that we need is a full level L containing L token in the immediate snapshot. Many processors shepherd the same token down the level ladder. While one processor may observe a level full, another processor which observed the memory earlier may have not seen the level full and may take some token down. Thus, if a processor return a value for a token at some full level it has to be sure that this token will not be dragged down later.

To affect this we execute the level change in two sub-phases. The presence of a token q at level L or below is signified by setting $TW[L,q] := 1$. It is exactly at level L if $TW[L,q] = 1$ and $TW[L-1,q] = \perp$. A processor that observes token q at level L and wants to take it down since L is not full yet, signals this by setting $TR[L,q] := 1$. The observation we make is that were L to become full there will always be at least one token such that $TW[L,q] = 1$ and $TR[L,q] = \perp$. To see this, consider the last token q to arrive at level L. Then no processor saw q at L when L was not full. Thus no processor will set $TR[L,q] := 1$.

Using the observation it is then safe to return an immediate-snapshot for such q. Even though later $TR[L,q]$ may be set to 1, since we do not know really who the last one was, then the processor that set $TR[L,q] := 1$, will again observe the level, will find it full and not push the token down. Hence, anybody that will return a snapshot for q will return the same value.

4.2 An Explicit Solution with Initial Constraints

We want to solve the Colorless Family Renaming with Initial Constraints. That is family p_i has a set CS_i of suspected processors each with a slot that it allegedly acquired. The sets CS_j relate by containment, and we want processors to output each a unique slot for some participating family such that this slot does not contradict the smallest CS_j for p_j that participates.

We do not know of a simple variation of the renaming of [3] that solves $CFRIC$. That algorithm evolves in round-by-round phases. The algorithm we have eluded to before [2] to solve $CFRIC$ is a regular shared-memory algorithm that relies heavily on "side-effects." How can we constructively get an IIS algorithm for $CFRIC$ without embedding? For that purpose we invoke the general transformation from shared-memory to IIS in [10]. Thus we take the resulting algorithm in IIS, and now use the shepherding above to solve it.

5 Conclusions

We have presented a renaming algorithm using k-set-consensus that is optimal (if there are n processors and they rename into $n+k-2$ then we can solve $(k-1,n)$-test-and set which is equivalent to $(k-1,k)$-set-consensus, and k-set-consensus cannot implemet $(k-1,k)$-set-consensus). While the availability of consensus has no advantage over test-and-set when renaming is concerned, both result in a minimal range renaming possible, k-set-consensus is advantageous to k-test-and-set even for renaming. Like [1] in the full paper we extend the result to a failure detector Ω^k, by processor aborting a phase once it realizes the participating set is more than k. Also, since $k-1$-resilient system implement k-set-consensus, our algorithm is an alternative to show that l-resilient system renames into $n+l$ slots [2]. The same technique here can be easily seen to solve the open problem in [1] concerning t-resiliency affirmatively. The algorithms contains a host of new ideas to be exploited in other contexts. Finally, our explicit algorithm uses the transformation from shared-memory to IIS. It will be nice to get an IIS algorithm directly.

References

1. Mostefaoui A, Raynal M. and Travers C., *Exploring Gafni's reduction land: from Ω^k to wait-free adaptive $(2p - \lceil p/k \rceil)$-renaming via k-set agreement*, Private Communication., To appear in DISC 2006.
2. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., and Reischuk, R., *Renaming in an asynchronous environment*, J. ACM 37, 3 (Jul. 1990), 524-548.
3. Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming (Extended Abstract). *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, ACM Press, pp. 41-51, 1993.
4. Borowsky, E, Gafni, E., Lynch, N.A, and Rajsbaum,S, *The BG distributed simulation algorithm*, Distributed Computing 14(3): 127-146 (2001)
5. Herlihy, M. and Shavit, N., *The topological structure of asynchronous computability*, J. ACM 46, 6 (Nov. 1999), 858-923.
6. Gafni, E., *Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony*, In Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing (Puerto Vallarta, Mexico, June 28 - July 02, 1998). PODC '98. ACM Press, New York, NY, 143-152.
7. Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E. *Reaching approximate agreement in the presence of faults*, J. ACM 33, 3 (May. 1986), 499-516.
8. Afek Y., H. Attiya, Dolev D., Gafni E., Merrit M. and Shavit N., Atomic Snapshots of Shared Memory. *Proc. 9th ACM Symposium on Principles of Distributed Computing (PODC'90)*, ACM Press, pp. 1–13, 1990.
9. Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations *Proc. 25th ACM Symposium on the Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.
10. Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 189–198, August 1997.
11. Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG Distributed Simulation Algorithm. *Distributed Computing,* 14(3):127–146, 2001.
12. Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.
13. Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
14. Gafni E., DISC/GODEL presentation: R/W Reductions (DISC'04), 2004. http://www.cs.ucla.edu/~ eli/eli/godel.ppt
15. Gafni E., Group-Solvability. *Proc. 18th Int. Symposium on Distributed Computing (DISC'04)*, Springer Verlag LNCS #3274, pp. 30–40, 2004.
16. Gafni E. and Kouznetsov P., Two Front Agreement with Application to Emulation and Robustness. *to appear*.
17. Gafni E., Merritt M. and Taubenfeld G., The Concurrency Hierarchy, and Algorithms for Unbounded Concurrency. *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press, pp. 161–169, 2001.
18. Gafni E. and Rajsbaum S., Musical Benches. *Proc. 19th Int. Symposium on Distributed Computing (DISC'05)*, Springer Verlag LNCS # 3724, pp. 63–77, September 2005.
19. Gafni E. and Rajsbaum S., Raynal M., Travers C., The Committee Decision Problem. *Proc. Theoretical Informatics, 7th Latin American Symposium (LATIN'06)*, Springer Verlag LNCS #3887, Valdivia, Chile, March 20-24, 502-514, 2006.

20. Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on programming Languages and Systems*, 11(1):124-149, 1991.
21. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.
22. Mostefaoui A., Raynal M.and Tronel F., From Binary Consensus to Multivalued Consensus in Asynchronous Message-Passing Systems. *Information Processing Letters*, 73:207-213, 2000.
23. Merrit M., Taubenfeld G., Computing with infinitely many processes. *Proc. 14th Int. Symposium on Distributed Computing (DISC'00)*, Springer Verlag LNCS #1914, pp. 164–178, October 2000.
24. Saks, M. and Zaharoglou, F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.

When Consensus Meets Self-stabilization*

Self-stabilizing Failure-Detector, Consensus and Replicated State-Machine (Extended Abstract)

Shlomi Dolev[1], Ronen I. Kat[1], and Elad M. Schiller[2]

[1] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il, kat@cs.bgu.ac.il
[2] Division of Computer Science and Engineering, Chalmers University of Technology. Göteborg, Sweden
elad@chalmers.se.

Abstract. This paper presents a self-stabilizing failure detector, asynchronous consensus and replicated state-machine algorithm suite, the components of which can be started in an *arbitrary state* and converge to act as a virtual state-machine.

Self-stabilizing algorithms can cope with transient faults. Transient faults can alter the system state to an arbitrary state and hence, cause a temporary violation of the safety property of the consensus. New requirements for consensus that fit the on-going nature of self-stabilizing algorithms are presented. The wait-free consensus (and the replicated state-machine) algorithm presented is a classic combination of a failure detector and a (memory bounded) rotating coordinator consensus that satisfy both *eventual safety* and *eventual liveness*.

Several new techniques and paradigms are introduced. The *bounded memory failure detector* abstracts away synchronization assumptions using bounded heartbeat counters combined with a balance-unbalance mechanism. The *practically infinite* paradigm is introduced in the scope of self-stabilization, where an execution of, say, 2^{64} sequential steps is regarded as (practically) infinite. Finally, we present the first *self-stabilizing wait-free reset* mechanism that ensures eventual safety and can be used in other scopes.

Keywords: Failure Detector, Consensus, State-Machine, Wait-Free, Distributed Reset, Self-Stabilization.

1 Introduction

Self-stabilization. Self-stabilization [13,14] is a fundamental property of a system that ensures automatic recovery of the system following the occurrence of

* Partially supported by the Israeli Ministry of Science, the Lynn and William Frankel Center for Computer Sciences, and the Rita Altura Trust Chair in Computer Sciences.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 45–63, 2006.

faults. Self-stabilizing systems are designed to be started in an arbitrary state and to converge to exhibit a desired behavior of the system. Recovery oriented computing, autonomic computing and self-* computing, e.g., [24], are research and industrial terms used extensively nowadays. The research and industrial activities in these fields may greatly benefit from using the well-understood and rigorous fundamentals of self-stabilization.

Consensus and failure detectors. Consensus is a fundamental and, in a sense, a complete problem in distributed computing. A distributed task is reduced to a centralized task by agreeing on the current distributed inputs (and the system state) and by consequently computing the fitting outputs. Unfortunately, as proved in [21] (and in [10,32] for shared memory) there is no asynchronous consensus algorithm even in executions in which just one process may stop taking steps. Fortunately, there are consensus algorithms, e.g., [29], that preserve safety (i.e., processes never decide on different values). Liveness is achieved in well-behaved executions (excluding, for instance, executions chosen according to [21]). Failure detectors, e.g., [8], form a mechanism that captures the synchronization requirements to obtain consensus liveness.

The consensus task is defined as a one-shot task, where the distributed system is started with inputs for each process and every non-crashed process must decide[1] on a common value[2] that appeared in one of the inputs[3]. In the scope of self-stabilization it is possible that the processes are started in a state in which each of them has already decided on a different value and does not take any further steps. Hence, one-shot self-stabilizing consensus is impossible. The definition of the consensus task in the scope of self-stabilization should incorporate the need for repeated invocations of the consensus, for example as the means of implementing a replicated state-machine. In such a case, the requirements for the self-stabilizing consensus must ensure eventual termination of initialized or non-initialized execution, as well as, all the consensus requirements for the set of processes that initialized a new session of the consensus execution. For instance, when considering the elegant algorithm presented in [29], and allowing an arbitrary state (and counters values), it is unclear whether there is a set of executions starting in such a state that will ensure termination. This is simply because, wrap around of counters is not considered. One may argue that a counter of 64 bits is practically infinite. This argument does not hold in the scope of self-stabilization since a single transient-fault may cause the counter to reach its upper bound at once. However, as we discuss in the sequel, we do consider an execution of 2^{64} sequential steps as practically infinite.

Replicated state-machine. A bold application for consensus is an implementation of a fault-tolerant replicated state-machine, e.g., [29]. The abstraction of a replicated state-machine has been proven important in several domains, e.g., [7,16,17].

[1] This is a termination requirement for the largest possible set of executions.
[2] The agreement requirement must hold in every execution.
[3] The validity requirement must hold in every execution.

Related work. The literature on failure detectors is rich; see for example the recent surveys [22,35]. We focus on the eventual strong $\Diamond S$ failure detector that is known to be the weakest failure detector required to solve consensus [8,9] in message-passing systems, when the majority of the processes are non-crashed. The weakest failure detector for the case when more than half of the processes may fail is considered in [12]. We note that the restriction on the number of failed processes does not apply to the shared-memory settings. This is due to the fact that a message can be delayed and the sending process can take additional steps, but a write to a register cannot be hidden from the other processes when the writing process takes additional steps. The $\Diamond S$ failure detector is an unreliable mechanism for detecting crashes. The $\Diamond S$ failure detector guarantees that (1) eventually every process that has crashed is permanently suspected by every non-crashed process (strong completeness property), and (2) there is a time after which some non-crashed processes are never suspected by any of the non-crashed processes (eventual weak accuracy property). The few self-stabilizing failure detectors mentioned in the literature uses message-passing systems. In [6] a self-stabilizing failure detector is presented in partial synchronous settings. In addition to the fact that the failure detector of [6] is not designed for shared memory systems, [6] does not handle the case of $n-1$ crashed processes. The algorithm in [27] uses randomization to construct a self-stabilizing perfect failure detector. Recent research on failure detectors is focused in identifying the weakest synchrony requirements necessary for solving consensus. Aguilera et al [2] presented a leader election algorithm (i.e., the Ω failure detector) that requires $n-1$ eventual timely outgoing links for at least one non-crashed process. In [3] the requirements were weakened to f eventual timely outgoing links, where f is the number of crashed processes. In [34] it is assumed that the outgoing links may be moving (i.e., the destinations are not fixed). Note that [2,3,34] require unbounded counters while a self-stabilizing failure detector may use only bounded counters.

Consensus is also an extensively studied topic in distributed computing, see, for example, [5,33]. In the case of eventual (unreliable and non-randomized) failure detectors there is a known bounded memory implementation for specific settings. For example, in [23], the case of three processes, where only one may fail, is considered. However, for the general case, to the best of our knowledge, there is no memory bounded consensus algorithm for asynchronous systems. The known algorithms require unbounded counters to preserve safety. We note that a bounded consensus algorithm is known for the case in which failures are instantly detected. Such failure detectors are inherently not self-stabilizing. A self-stabilizing consensus algorithm eventually ensures safety (in the presence of crash failures) and liveness (under reasonable synchronization assumptions for the failure detector).

Using asynchronous reset with the purpose of circumventing unbounded values is discussed in [28]. This asynchronous reset is not wait-free and requires a complete knowledge regarding failures of links and processes. That is, the algorithm is notified when edges become either active or inactive. A resetable vector

clocks that provide non-blocking resets in the absence of faults is presented in [4]. In case faults occur, the algorithm requires one blocking reset (i.e., a global reset) before the system stabilizes. Another solution for circumventing unbounded values is self-stabilizing timestamps [1]. $O(n)$ invocations of weak timestamps procedures (each invocation requires $O(n)$ operations) are used in [1] in order to achieve bounded timestamps. We propose a wait-free reset that can be used for implementing self-stabilizing bounded timestamps which requires only $O(n)$ operations per invocation (following the convergence of the wait-free reset).

Our contribution. We present a new self-stabilizing failure detector, consensus and replicated state-machine algorithm suite in shared-memory settings. All components can be started in an *arbitrary state* and converge to act as a virtual state-machine. Thus, we gain a self-stabilizing infrastructure for the execution of self-stabilizing applications. In addition, we present the first wait-free reset technique.

• Failure detector. Our self-stabilizing failure detector does not use randomization and is designed for partial synchronous settings. In such settings, the interleaving order of (non-crashed processes) steps is eventually somewhat restricted. Roughly speaking, each process has a bounded heartbeat counter. The relative advances of the counter are compared to other processes. To avoid confusion that could arise due to the fact that the counters are bounded, we use wrap around flags that indicate when a process has wrapped its counter. We show that bounded flags are sufficient for computing of the relative speed of steps. This simple mechanism identifies crashed processes and allows the active set of processes to continue in their consensus task. Our failure detector uses synchronization assumptions that are analogous to the settings assumed in [2]. Note that our algorithm uses shared memory and not message-passing as in [2]. Moreover, our algorithm is memory bounded while [2] requires unbounded counters.

• Consensus. Our self-stabilizing algorithm is able to achieve eventual safety (and eventual liveness) using bounded memory. The consensus algorithm assumes the existence of the obtained self-stabilizing $\Diamond S$ failure detector. Generally speaking, the algorithm is a rotating coordinator algorithm that is based on the (memory unbounded) algorithm in [31]. The processes are sequentially assigned to be the consensus coordinator. A coordinating process that takes steps and is not suspected as crashed, will successfully bring the system to a *univalent configuration* (a configuration after which all decisions have the same value). A univalent configuration is reached, once the coordinating process succeeds in writing the proposed consensus value (which is one of the inputs) and at the moment that every process reads this proposed value prior to becoming current coordinator. The use of the eventual strong failure detector ensures that eventually one of the processes (i.e., p) is never suspected. It is possible that a univalent configuration is reached before the stage of the execution in which p is never suspected as crashed. Otherwise, when p becomes the current coordinator, no other process will become a new coordinator, and a univalent configuration is, therefore, imposed by p.

• Replicated state-machine. We use epochs of consensus invocations for deciding on the transition of the (distributed) state-machine. Note that due to the

self-stabilizing nature of our replicated state-machine, the application executed by the replicated state-machine should be self-stabilizing as well.

Our contribution is also in the modification made to the traditional replicated state-machine, where the machine has to decide on a common state as well. Thus, we tolerate an arbitrary initial configuration in which processes have different notions regarding the current state of the state-machine. We suggest the use of hash functions to avoid copying the entire state when possible.

• Practically infinite executions and self-stabilizing wait-free reset. We define a set of executions that we call *practically infinite executions*. Practically infinite executions are executions in which the time complexity measured by the longest *happened before chain* [30] is longer than, lets say 2^{64}. Note that even if each step takes a single nanosecond, no computer system will last such a long period of time, and no client will wait for the last step in this sequence. In case the suite of asynchronous consensus algorithm and failure detector does not reach a decision within a period of time that corresponds to 2^{64} sequential steps, the decision value will clearly be obsolete.

The above argument is used in the scope of self-stabilization for the first time. Rather than assuming that 2^{64} counter is infinite, we argue regarding the amount of time required for a chain of 2^{64} sequential steps of counter increments. If due to a transient fault the counter reaches its upper bound at once, a wait-free reset takes place ensuring a subsequent practically infinite resetless execution.

Paper organization. The rest of the paper is organized as follows: The system settings appear in Section 2; The self-stabilizing failure detector is described in Section 3; The self-stabilizing consensus and state-machine algorithm appear in Section 4; Concluding remarks appear in Section 5. Details and proofs are omitted from this extended abstract and can be found in [18].

2 System Settings

We consider a shared memory system with a set of Π communicating entities that we call *processes*. There are n processes in the system, with each process having a distinct identifier in the range of $0, \ldots, n-1$. Each process p is associated with a set of atomic multi-reader/single-writer registers. A process $p \in \Pi$ can write only in its associated register and can read any register. Process p writes the value v to register R, by using the command $\mathbf{write}(R, v)$. Process p_i reads the value of register R by using the command $\mathbf{read}(R)$. We use capital letters for register names, e.g., R_p. We use lower case letters, e.g., r_p, for denoting the local variables of processes that contain the values read by the process from the register, i.e., r_p contains the last value read from R_p.

Each process p is modeled by a state-machine. We use a program in pseudo code to describe the state space and the transition function of p. In every given instance, the state of a process p includes the process program counter, the values of p's local variables, and the values of the registers associated with p. A state transition of a process p is defined by an *(atomic) step*. A step consists of a sequence of internal (program) computations that ends in a single *read* or

write operation to a register. A *system configuration* consists of the states of all the processes. An *execution* is a sequence of configurations and steps $E = (c_1, a_1, c_2, a_2, \ldots)$, where configuration c_{i+1} is reached by executing a step a_i by one process. A *task* is defined by a set of executions called *legal executions* (LE). A configuration c is a *safe configuration* for an algorithm and task LE provided that any execution that starts in c is a legal execution (belongs to LE). An algorithm is *self-stabilizing* with relation to task LE if every infinite execution of the algorithm reaches a safe configuration with relation to the algorithm and the task. A process may fail by permanently stopping to execute steps. We say that such a process *crashed*. Hence, it does not execute any step in a suffix of the execution. Note that the output of a read command to a shared register of crashed process p is constant, as only p can write to its registers. A process that executes a step infinitely often is said to be *non-crashed*.

Our self-stabilizing consensus and wait-free reset algorithms are designed for asynchronous systems (with a failure detector) and our self-stabilizing failure detector algorithm assumes the existence of a (unknown to the processes) bound on the (relative) execution speed of the processes.

3 Self-stabilizing Failure Detector

Fischer, Lynch and Paterson [21], (and then [10,32]) have shown that consensus cannot be reached in message-passing (and then in shared memory) asynchronous environments. A failure detector [8] is an oracle that identifies crashed processes and helps to separate safety and liveness concerns in a way that may lead to a feasible safe solution for consensus in which liveness depends only on (synchronization or) scheduling of actions while safety always holds. We present a self-stabilizing $\Diamond S$ failure detector that satisfies the following properties:

Property 1 (Strong completeness). Every execution has a suffix in which (eventually) every process that has crashed is permanently suspected by every *non-crashed* process.

Property 2 (Eventual weak accuracy). Every execution has a suffix in which some non-crashed processes are never suspected by any of the non-crashed processes.

Partial Synchronous Settings and Requirements. We assume the existence of a global clock t that is unknown to the processes. We use real time to argue concerning progress during executions. We make no assumptions regarding local clocks or clock synchronization.

We say that a *source process* is a process that executes any two successive steps exactly δ time units (of the global clock) apart. Such a process is said to be executing steps *on time*. The definition is similar to the notion of eventual timely output links, in a message-passing model, of a process as in [2]. We also assume that no process executes two successive steps faster than a source process p. Namely, for every non-crashed process q and any two successive steps a_i and a_{i+1} of a process q, there are at least δ time units between the time in which the communication actions of a_i and a_{i+1} have taken place.

We say that an infinite execution is *admissible* if it has at least one source process p. Since no process is faster than p, process p executes at least one step between any two successive steps of any process in an admissible execution. The failure detector task is defined by a set of executions in which the strong completeness (Property 1) and eventual weak accuracy (Property 2) hold.

The Failure Detector. The heart-beat mechanism is used for detecting crashed processes. Processes that are not crashed signal the rest of the processes by repeatedly changing a value in a register, read by the other processes. The failure detector algorithm maintains a set of suspicious processes in the local variable *fd* (failed detection). If process p suspects process q to have crashed, then $q \in fd$. A non-crashed process p continuously advances a "heartbeat" counter (HB_p) in order to avoid being mistakenly suspected. Process p periodically compares the other processes' counters progress over time.

Since a crashed process q does not advance its heartbeat counter HB_q, every non-crashed process suspects q within a finite period of time. Process p uses a cyclic history array h_q for recording indications on the last k heartbeats (counter) of q. If q increased its counter, p records 1 in an entry of h_q. Otherwise, p records 0. Process q is suspected as crashed by p only if during the last k records of p, q did not advance its heartbeat. Note that the "fastest process" (e.g., the source process) is never suspected since every process marks the continuous progress of the source process. A process that executes steps slowly may be falsely suspected as crashed when the history size, k, is too small. A longer history provides a more robust and accurate indication of crashed processes. However, the need for accumulating a long history before making the decision delays failure indications.

Since a self-stabilizing failure detector has to use a bounded heartbeat counter, each counter is incremented modulo m. In some cases, a slow process may (incorrectly) consider a process that has wrapped its counter, as crashed. Therefore, in case the counter of process p wraps, p uses a balance/unbalance [15,20] protocol to ensure that p is not considered as crashed. That is, p keeps a flag $WR_{p,q}$ for every process q. Process q keeps a copy of p's flag in $LWR_{q,p}$ and makes sure that the copy always equals p's value. When p wraps its counter, p assigns q's flag ($WR_{p,q}$) a new value that is different from the value of $LWR_{q,p}$ (q's copy of $WR_{p,q}$). Therefore, when q reads p's flag ($WR_{p,q}$), q notices the fact that p's counter has wrapped. The counter size (m) is chosen in a way that optimizes the number of wraps around (to zero) with relation to balance/unbalance usage. The registers $WR_{p,q}$ and $LWR_{q,p}$ contain a value of $\{0, 1, 2\}$ in order to ensure that p can introduce a new value and make sure that q notices the change. The register has three values to ensure correct behavior given that processes p and q keep a local variable for both $WR_{p,q}$ and $LWR_{q,p}$. The value of the local variable may differ from the value of the register. We say that p *is unbalanced towards* q in case $WR_{p,q} \neq LWR_{q,p}$. Otherwise, p *is balanced towards* q. We use the predicate $unbalance(q)$ to describe p's view (indicated by the values of $wr_{p,q}$ and $lwr_{q,p}$) regarding its state (balanced/unbalanced) towards q. When p's counter is wrapped, p writes a value to $WR_{p,q}$ (for every q) in such a way that the predicate $unbalance(q)$ is $true$. In case $wr_{p,q} = lwr_{q,p}$, p increases $lwr_{q,p}$

by one modulo 3. A process q that notices the wrap indication (i.e., the predicate $unbalance(q)$ is $true$) copies $wr_{p,q}$'s value to $LWR_{q,p}$.

In Figure 1 we present the algorithm for the self-stabilizing failure detector. In every iteration of the program, process p increases the heartbeat counter HB_p (lines 1 and 2) and signals, by executing the procedure $unbalance\_all()$ (lines 3 and 4), to all other processes when the counter wraps around to zero. In lines $f1$ through $f5$ process p flags an unbalance indication towards every other process q. In lines $f2$ and $f3$ process p reads $LWR_{q,p}$ and $WR_{p,q}$. If p is balanced towards q, then process p unbalances and writes to register $WR_{p,q}$ (lines $f4$ and $f5$). A process p becomes balanced towards q by reading register $WR_{q,p}$ (line 9) and writing the value in register $LWR_{p,q}$ (line 18).

Process p records the history of process q in the cyclic history array h_q. Process p keeps the history of length k for each process. In line 5 process p moves to the next history entry. In line 6 process p initializes the list of new suspects in $newfd$. In lines 8 through 14 process p reads the heartbeat counter and the balance/unbalance indications of every process q as well as records q's progress. If q increases its counter (i.e., q executed enough steps since p last read q's counter), then p records 1; otherwise, p records 0 in the history entry.

```
Types:
HBvals = 0, ..., m − 1 heartbeat values.
WRvals = 0, ..., 2 (un)balanced values.
Dvals = 0, 1 process progress.
FDvals = 2^Π suspects list.
Shared variables:
atomic HB_p ∈ HBvals : p ∈ Π
atomic LWR_{p,q} ∈ WRvals : p, q ∈ Π
atomic WR_{p,q} ∈ WRvals : p, q ∈ Π
Local variables:
newhb_p, lhb_p, ∈ HBvals : p ∈ Π
lwr_{p,q}, wr_{p,q} ∈ WRvals : p, q ∈ Π
h_q[i] ∈ Dvals : q ∈ Π : 0 ≤ i < k
fd, newfd ∈ FDvals
procedure unbalance_all():
f1. ∀q ∈ (Π − {p}) do
f2.   lwr_{q,p} ← read(LWR_{q,p})
f3.   wr_{p,q} ← read(WR_{p,q})
f4.   if (wr_{p,q} = lwr_{q,p})
f5.     write(WR_{p,q}, lwr_{q,p} + 1 mod 3)
Macros:
unbalance(q) ≡ wr_{p,q} ≠ lwr_{q,p}
Do forever every Δ steps
1.  newhb_p ← newhb_p + 1 mod m
2.  write(HB_p, newhb_p)
3.  if (newhb_p = 0)
4.    unbalance_all()
5.  i ← i + 1 mod k
6.  newfd ← ∅
7.  ∀q ∈ (Π − {p}) do
8.    newhb_q ← read(HB_q)
9.    wr_{q,p} ← read(WR_{q,p})
10.   lwr_{p,q} ← read(LWR_{p,q})
11.   if ((newhb_q − lhb_q) mod m > 0 ∨
          unbalance(q))
12.     h_q[i] ← 1
13.   else
14.     h_q[i] ← 0
15.   if ( ∀i h_q[i] = 0 )
16.     newfd ← newfd ∪ {q}
17.   lhb_q ← newhb_q
18.   write(LWR_{p,q}, wr_{q,p})
19. fd ← newfd
```

Fig. 1. Failure Detector for process p

If every entry of q's history is 0 (i.e., q did not execute enough steps during the last k iterations), then q is suspected to have crashed (lines 15 and 16). Otherwise, p does not suspect q. In lines 17 and 18, p keeps the last value of q's counter and writes the value of $WR_{q,p}$ to $LWR_{p,q}$ (i.e., balances towards q). In line 19, process p updates its suspects list fd with the new computed suspicions.

Note that the requirement for the existence of a source process can be relaxed to, say, the existence of a set of processes that are "fast enough". A process p is considered "fast enough" when no process suspects p as being crashed. Thus, the proofs hold for a much larger set of executions, for example, executions where a process p exists for which the history associated with p by every other

process has at least one non-zero entry. That is, every other process increases its heartbeat at most k times between any two successive heartbeat increases of p. The following Theorem states the correctness of the algorithm. Details are omitted from this extended abstract.

Theorem 1. *Every admissible execution of the failure detector algorithm (Figure 1) has a suffix that satisfies the failure detector task. The suffix starts after every non-crashed process executes* $(k+3)\Delta$ *steps.*

4 Self-stabilizing Consensus and Replicated State Machine

We will now describe the self-stabilizing consensus and the replicated state-machine algorithm using the eventual strong failure detector of the previous section.

A self-stabilizing replicated state-machine is a collection of processes, each of which independently implements a state-machine. Every non-crashed process executes the same sequence of transitions and reaches the same state. In order to guarantee that each process, eventually, executes the same transitions, we employ a sequence of consensus instances, one for each transition. Each instance has an *epoch* number, in which the processes decide on a single value (i.e., the transition) from the possible transitions (i.e., inputs) suggested by each process. We assume that the inputs are provided to the algorithm. The origin of the inputs is outside the scope of our work. The self-stabilizing consensus satisfies the following properties in the presence of the self-stabilizing $\Diamond S$ failure detector:

Property 3 (Eventual termination). Every execution has a suffix in which every non-crashed process decides on a value in every epoch.

Property 4 (Eventual Validity). Every execution has a suffix in which every non-crashed process decides on the initial value of some non-crashed process in every epoch.

Property 5 (Eventual Agreement). Every execution has a suffix in which no two non-crashed processes decide on different values in every epoch.

We use the self-stabilizing consensus to implement a replicated state-machine. The self-stabilizing replicated state-machine guarantees the following properties:

Property 6 (Eventual Coordination). Every execution has a suffix starting with some epoch e such that for every epoch $e' > e$, no two processes execute a different transition.

Property 7 (Eventual Consistency). Every execution has a suffix starting with some epoch e in which no two processes differ in their machine state in every epoch.

Asynchronous Settings and Requirements. The system is assumed to be completely asynchronous, i.e., there are no timing assumptions. Moreover, processes may be as slow (or fast) as one may choose them to be or they may stop operating altogether. Processes cannot distinguish a slow process from a crashed process. Each process is associated with a single multi-reader/single-writer register R_p that holds $O(n \log n)$ bits.

The task of the self-stabilizing consensus is defined by a set of executions in which eventual termination (Property 3), validity (Property 4) and agreement (Property 5) hold. The task of the self-stabilizing replicated state-machine is defined by a set of executions in which eventual coordination (Property 6) and eventual consistency (Property 7) hold.

The Consensus and the State-Machine. Next we describe the consensus, the replicated state-machine and the wait-free reset that along with the failure detector form a bounded replicated state-machine that is self-stabilizing.

• Consensus with a rotating coordinator. The consensus ensures that no two processes will decide on different values in an epoch. Every process (from the set Π of processes) either follows the previous decision of others, or strives to make a decision that the other processes would follow. A process, which does not copy a decision value from a process that has already decided, must execute the following sequence of *scan* (read from the registers of all processes) and write operations: *announce*, *scan*, *propose*, *scan*, and *decide*. Each of such sequences is identified by a unique sequence number that is called a *round number*.

The algorithm is based on the observation that there is no possible interleaving of such atomic operations that allows different (transition) values for the decide write operation. Consider a process p with the smallest round number r that *proposes* and subsequently *decides* with round number r. The algorithm states that any process with a round number r' smaller than r will not decide with r'. According to the algorithm, every process q with a higher round number that *proposes* adopts the value proposed (and decided) by p. A process with a higher round number that does not adopt p's proposed value must *scan* prior to p's proposal. This implies that q *announced* before p *proposed*. In such a case, p cannot decide since p finds q's higher round number. The above observation (i.e., that implies safety) is based on the assumption that the round numbers are ever increasing. We assume that the round number counters do not wrap in an execution that starts with counters initialized to zero. We show how to achieve such an execution in the sequel.

The system achieves consensus when a single process p executes the above sequence of steps without crashing. A process that tries to execute such a sequence of steps is said to be *the coordinator*. The failure detector assists in moving the responsibility to the next process when a coordinator is suspected to have crashed. The *rotating coordinator* paradigm [8] states that in every round r there is a single coordinator p (i.e., $p = r \bmod n$) that carries out the above sequence of steps. Every other process q (i.e., q is not a coordinator in r) waits for p to decide within round r. Process q repeatedly performs *scans* until some process decides, or until it is obvious (for q) that the coordinator will not decide

within r. For example, the coordinator (of round r) is in a higher round number than r or q's failure detector suspects the coordinator of round r. In such cases, q moves to a new round number, by increasing r by one.

• Self-stabilizing replicated state-machine. The replicated state-machine ensures that every non-crashed process executes the same sequence of transitions and reaches the same state. Each transition is associated with an instance (i.e., epoch number) of the consensus. Every process computes the transition using the decision value (in an epoch) and a fixed, hardwired, transition function (of commands) in order to reach a new state.

Starting from an initial state, each process of the replicated state-machine executes the same transitions due to the fact that the consensus decision values in every epoch are the same. Periodically, process p compares its epoch number and state with other processes. When p finds another process with a higher epoch, p increases its epoch number to the highest epoch (p observed) and copies the state (p read) from the process with the smallest identifier among the processes in the highest epoch.

The state-machine for process p is represented by the tuple $\langle e_p, state_p \rangle$, where e_p is the current epoch number and $state_p$ is the state reached in the previous epoch. When a decision is reached in epoch e_p, p executes the *decide*() procedure, computes the new state and increases its epoch number to $e_p + 1$. We assume that the epoch and the round numbers are practically unbounded (i.e., the real-time needed to reach the highest value is practically infinite) when the system is started from an initial state. In the case that the system starts from an arbitrary state, we show that there is an execution suffix in which the epoch and the round numbers are practically unbounded. That is, we allow the epoch and the round numbers to be reinitialized to zero.

• Self-stabilizing wait-free reset. Next, we show how to resolve the contradiction between safety, which requires that epoch and round numbers are ever increasing, and the fact that the counters are bounded and may wrap around (to zero). The fact that a counter of 64-bits (or more) is practically infinite does not hold in the scope of self-stabilization. A single transient-fault (or incorrect initialization) may cause the counter to reach such a large value at once, not allowing the consensus algorithm to have enough rounds to reach a decision in an epoch. A similar argument applies to epoch numbers and the state-machine transitions. The goal of the reset mechanism is to set all epoch and round number counters (for every non-crashed process) to zero. Thus, following a reset, the consensus and the replicated state-machine algorithm will have practically unbounded number of rounds to reach a decision and an unbounded number of epochs to perform transitions.

We denote by *inf* the maximal value of the epoch and the round number counters (*inf* is related to the bounded size of the counter). In addition, we assume that, when a process increases its counter beyond *inf*, the counter value is not changed. We assume that the value *inf* is very big and is reached only when a counter is initialized (or changed due to a transient fault) to some non-zero (large) value.

Whenever the epoch or the round number of process p reaches *inf*, p attempts to set all counters to zero by performing a *reset* and by assigning zero to the epoch and the round numbers. We associate every counter wrap with a reset sequence number. When p performs a *reset*, p increases by one its reset sequence number *rseq*. We will show that the reset sequence number should be in the range of 0 to $2n$. Each process p keeps track of all the other processes reset sequence numbers in the array *rseq*, and p's reset sequence number is kept in $rseq[p]$. In addition to the reset sequence number, p uses a balance/unbalance protocol instance with every process q. The balance/unbalance protocol is used to identify slow or crashed processes. When p performs a reset, p ensures that for every process q, it holds that: $wr_{p,q} \neq lwr_{q,p}$ (unbalanced), by assigning a value to $wr_{p,q}$ if needed. That is, if $wr_{p,q} = lwr_{q,p}$, then p increases $lwr_{q,p}$ by 1 modulo 3. Process p evaluates the predicate $isreset(q)$ to be *true* if in a configuration c the local variables of process p indicate that the reset sequence number ($rseq_p[p]$ and $rseq_q[p]$) and/or the balance/unbalance flags ($wr_{p,q}$ and $lwr_{q,p}$) of p and q differ.

We say that q *is reset* or q *is flagged as reset* in case the predicate $isreset(q)$ is *true* for process q. The flag indicates that q needs to set its counters to zero. The goal of a process that performs a reset is to invalidate the values of the epoch and the round counters of other process and signal them to assign zero to these counters thereafter. A process q that is reset, does not strive to propose and decide in the consensus algorithm (and is ignored by other processes). When q notes that process p has flagged q as reset, process q *acknowledges* the reset by copying the values of p's flags and the reset sequence number and by resetting the epoch/round number to 0. The acknowledgment indicates that q has set its counters to zero according to the reset request.

Another possible function of the reset mechanism is to help maintaining the common state of the replicated state-machine. We suggest using the output of a hash function on the state instead of using the full state. Whenever a process finds a conflict of hash values the process invokes a reset that will set the machine state of each process to an initial, predefined state. In addition, we can enhance the probabilistic nature of hash functions collisions by communicating the full state (instead of the hash value) in every fixed number of epochs.

We now describe the consensus and the replicated state-machine algorithm. Figure 2 describes the registers, variables and macros. Figure 3 contains the procedures and Figure 4 describes the algorithm. The algorithm combines a replicated state-machine that executes transitions and a consensus that decides on the values of the transitions. A process p writes only to register R_p. The register consists of a tuple $\langle v, g, e, r, state, wr, lwr, rseq \rangle$. v is p's estimate of the decision value in epoch e. The value of g is the consensus phase tag, which can be either *announce*, *propose* or *decide*. e and r are the current epoch and round numbers. *state* represents the last state of the state-machine. wr and lwr are arrays (of size n) of the balance/unbalance values, where the value of $wr[q]$ is used for the unbalancing action by p and $lwr[q]$ is used for the balancing action by p towards q. Finally, *rseq* is an array of the reset sequence numbers (one reset sequence number for each process). In the sequel, we compare instances of $\langle e, r \rangle$.

We say that $\langle e_1, r_1\rangle > \langle e_2, r_2\rangle$ if $e_1 > e_2$, or if $e_1 = e_2 \wedge r_1 > r_2$. We say that a process has decided in epoch e if a non-reset process q with a higher epoch number exists, or if a register (of a non-reset process) contains the tag *decide* for epoch e. In such cases, the predicate $decisionExists(e)$ is *true*.

The procedure *scan*() (Figure 3) reads the registers of all the processes. The values that are associated with the consensus are stored in the set m_p and the values for determining if process q is reset are stored in wr_q, lwr_q and $rseq_q$. The values in m_p are tuples of $\langle q, v_q, e_q, r_q, state_q\rangle$, where q represents the process identifier, v_q is the current estimate of process q, e_q and r_q are the current epoch and round numbers of q, and $state_q$ is the state of the state-machine of process q.

Types:
$vals$ = consensus decision values.
$CSstate = announce, propose, decide.$
$CSnumber = -1, \ldots, inf - 1$ epoch/round number.
$STMvals$ = Hash value on the-machine state.
$WRvals = 0, \ldots, 2$ balance/unbalance.
$RSEQvals = 0, \ldots, 2n$ reset sequence number.
Shared variables:
atomic $R_p = \langle v_p, g_p, e_p, r_p, state_p, wr_p, lwr_p, rseq_p\rangle$
Local variables:
$wr_p[q], lwr_p[q], \in WRvals : p, q \in \Pi$
$e_p, r_p, r_{max} \in CSnumber$
$m_p \in \langle \mathcal{P}, vals, CSstate, CSnumber, CSnumber, STMvals\rangle$
$rseq_p[q] \in RSEQvals : p, q \in \Pi$
$v_p, init_p \in vals : p \in \Pi$
$g_p \in CSstate : p \in \Pi$
$state_p \in STMvals : p \in \Pi$
Macros:
$wr_{p,q} \equiv wr_p[q]$
$lwr_{p,q} \equiv lwr_p[q]$
$unbalanced(p, q) \equiv wr_{p,q} \neq lwr_{q,p} \vee rseq_u[u] \neq rseq_q[u]$
$isreset(q) \equiv \exists u \in \mathcal{P}\ unbalanced(u, q)$
$decisionExist(e) \equiv \exists \langle q, v_q, \text{decide}, e, r_q, *\rangle \in m_p :$
$\neg isreset(q) \vee (\exists \langle q, *, *, e', *, *\rangle \in m_p : e' > e \wedge$
$\neg isreset(q))$

Fig. 2. Definitions for process p

The procedure *next*() in Figure 3 advances p to a new round $r + 1$. First, p scans the registers (line $g1$). In case p is flagged as reset, p repeats the scan (line $g3$), reading the registers values after the reset. Then, p resets its epoch and round number to zero (line $g4$), obtains the estimate for the new epoch (line $g5$) and balances any balance/unbalanced flags and reset sequence numbers (lines $g6$ through $g8$). That is, p copies q's unbalanced flag ($wr_{q,p}$) to $lwr_{p,q}$ and copies q's reset sequence number ($rseq_q[q]$) to $rseq_p[q]$. In case no process flagged p as reset, p checks if a process has reached an epoch/round number higher than p's epoch/round number and verifies the state consistency of the replicated state-machine (lines $g9$ through $g14$). In lines $g9$ and $g10$ process p finds, using the function $index - max$, a non-reset process q with the smallest identifier that has the maximal epoch/round number $\langle e_q, r_q\rangle$. If $\langle e_q, r_q\rangle > \langle e_p, r_p\rangle$, then p copies $\langle e_q, r_q\rangle$ (lines $g11$ and $g12$) and the state of q's state-machine (line $g13$). In case p does not copy an epoch or a round number from another process, p checks if the epoch/round number reached *inf*. If e_p or r_p reached *inf*, then p performs a reset (lines $g15$ through $g19$), by unbalancing (if the balance/unbalance instance from p to q is not already unbalanced) all other processes and by increasing its reset sequence number $rseq_p[p]$ as well. Otherwise, in lines $g20$ and $g21$, process p increases its round number. The procedure *decide*() (Figure 3) computes the

new state, advances to the next epoch and obtains the input for the new epoch. Figure 4 describes the consensus and the replicated state-machine code. In lines 1 and 2, process p initializes the state-machine. In lines 3 through 5, process p advances to the next round and announces its estimate. In case p is the coordinator, process p executes lines 7 through 20. In lines 7 through 10 process p checks if process q has already decided. If so, p decides after coping q's decision value and the state of q's state-machine. If no process reached a higher round than p, then in lines 13 through 16, p adopts the latest estimate that a process has proposed. If no process has proposed, then p proposes its own estimate. In lines 17 through 20, p verifies once more that no other process has reached a higher round, and if so, p decides. In case p is not a coordinator (lines 22 through 29), p continuously reads the registers (i.e., the epoch/round numbers and the reset indications) and waits for the coordinator to reach p's round number. Due to the fact that it is possible for the coordinator not to reach p's round, p stops waiting if (1) p waits for itself, or (2) p is reset, (3) a process has decided within the epoch, or (4) the coordinator is suspected as a crashed process. In case p exits the loop after a process has decided with p's epoch number, then p copies the decision value (lines 27 through 29).

We use the following in our proofs: a process p, which executes steps in epoch e (i.e., $e_p = e$) and writes a tuple with the tag *decide* to register R_p, is said to be *decided* in epoch e. Note that a process is *decided* after executing the step in lines 9, 19 or 28. A *resetless execution* of the state-machine is a practically infinite execution, starting in an arbitrary configuration, in which no process flags another as reset.

```
procedure scan() :
f1.   m_p ← ∅
f2.   ∀q ∈ Π do
f3.     ⟨v_q, g_q, e_q, r_q, state_q, wr_q, lwr_q, rseq_q⟩ ← read(R_q)
f4.     m_p ← m_p ∪ {⟨q, v_q, g_q, e_q, r_q, state_q⟩}
procedure next() :
g1.   scan()
g2.   if (isreset(p))
g3.     scan()
g4.     ⟨e_p, r_p⟩ ← ⟨0, 0⟩
g5.     v_p ← new input(e_p)
g6.     ∀q ∈ P do
g7.       lwr_{p,q} ← wr_{q,p}
g8.       rseq_p[q] ← rseq_q[q]
g9.   else if (∃⟨q, ∗, ∗, e_q, r_q, state_q⟩ ∈ m_p :
            (⟨e_q, r_q⟩ ≥ ⟨e_p, r_p⟩) ∧ ¬isreset(q))
g10.    q ← index-max{⟨e_q, r_q⟩|⟨q, ∗, ∗, e_q, r_q, state_q⟩ ∈ m_p :
              ¬isreset(q)}
g11.    if (⟨e_q, r_q⟩ > ⟨e_p, r_p⟩)
g12.      ⟨e_p, r_p⟩ ← ⟨e_q, r_q⟩
g13.      state_p ← state_q
g14.    v_p ← new input(e_p)
g15.  if (r_p = inf ∨ e_p = inf)
g16.    ∀q ∈ P : wr_{p,q} = lwr_{q,p} do
g17.      wr_{p,q} ← lwr_{q,p} + 1 mod 3
g18.    rseq_p[p] ← rseq_p[p] + 1 mod 2n
g19.    ⟨e_p, r_p⟩ ← ⟨0, 0⟩
g20.  else
g21.    ⟨e_p, r_p⟩ ← ⟨e_p, r_p + 1⟩
procedure decide() :
h1.   state_p ← new state(state_p, v_p)
h2.   ⟨e_p, r_p⟩ ← ⟨e_p + 1, −1⟩
h3.   v_p ← new input(e_p)
```

Fig. 3. Procedures for process p

Proof overview. First, we show that there are infinitely many steps in which the round number is increased in a resetless execution. In the sequel, we show that in every execution a resetless execution exists. We prove that if there is a decision in a certain epoch e, then there is a subsequent configuration in which a process has

an epoch number that is at least $e + 1$ (assuming a resetless execution). Next, we prove that eventually a process decides, using the eventual completeness and eventual accuracy of the failure detector indications (that are guaranteed in admissible executions). Note that this concludes the liveness arguments. Next, we turn to prove the eventual safety property. We show that any resetless execution has a suffix in which no two processes decide on different values with the same epoch number.

The rest of the proof focuses on proving that practically resetless executions must exist. We prove that eventually when p invokes a reset, the values of the registers that are used for implementing the reset are not equal, and eventually, when q notices the reset and acknowledges (the reset), these values are equal. We use the behavior of the balance/unbalance protocol and the reset sequence numbers to show that eventually when a process resets the system, the reset is successful. A *successful reset* is the one after which every process uses 0 as its epoch/round number or uses a counter value that has been incremented from 0 due to steps, for which this reset has a happen-before relation. In fact, we show that a successful reset takes place, when a process p manages (when writing a reset indication) to introduce a new reset sequence number to every neighboring process q and the first step of q, in which q balances, is the one that also assigns 0 to the epoch and the round numbers. We identify crashed processes by the balance/unbalance protocol.

```
1.  ⟨e_p, r_p⟩ ← ⟨0, −1⟩
2.  v_p ← new input(e_p)
do forever
3.  next()
4.  c ← r_p mod n
5.  write(R_p, ⟨v_p, announce, e_p, r_p, state_p, wr_p, lwr_p, rseq⟩)
6.  if (p = c)
7.    scan()
8.    if (∃⟨q, v_q, decide, e_p, r_q, state_q⟩ ∈ m_p : ¬isreset(q))
9.      write(R_p, ⟨v_q, decide, e_p, r_p, state_q, wr_p, lwr_p, rseq⟩))
10.     decide()
11.   else
12.     if (∀⟨q, v_q, ∗, e_q, r_q, ∗⟩ ∈ m_p : ⟨e_p, r_p⟩ > ⟨e_q, r_q⟩∨
            isreset(q))
13.       if (∃⟨q, v_q, propose, e_p, r_q, ∗⟩ ∈ m_p : ¬isreset(q))
14.         r_max ← max{r_q | ⟨q, v_q, propose, e_p, r_q, state_q⟩ ∈ m_p :
                ¬isreset(q)}
15.         ⟨state_p, v_p⟩ ← ⟨state_q, v_q⟩ :
              ⟨q, v_q, propose, e_p, r_max, state_q⟩ ∈ m_p : ¬isreset(q)
16.       write(R_p, ⟨v_q, propose, e_p, r_p, state_p, wr_p, lwr_p, rseq⟩)
17.       scan()
18.       if (∀⟨q, v_q, ∗, e_q, r_q, ∗⟩ ∈ m_p : ⟨e_p, r_p⟩ > ⟨e_q, r_q⟩∨
            isreset(q))
19.         write(R_p, ⟨v_q, decide, e_p, r_p, state_p, wr_p, lwr_p, rseq⟩)
20.         decide()
21. else
22.   do
23.     scan()
24.     ⟨e_p, r_p⟩ ← ⟨e, r⟩ : ⟨p, ∗, ∗, e, r, ∗⟩ ∈ m_p
25.     c ← r_p mod n
26.   until {c = p ∨ isreset(p) ∨ decisionExist(e_p) ∨ c ∈ fd_p∨
          ∃⟨c, v_c, g_c, e_p, r_c, ∗⟩ ∈ m_p : (⟨e_c, r_c⟩ > ⟨e_p, r_p⟩∧
          ¬isreset(c))}
27.   if (∃⟨q, v_q, decide, e_p, r_q, state_q⟩ ∈ m_p : ¬isreset(q))
28.     write(R_p, ⟨v_q, decide, e_p, r_p, state_q, wr_p, lwr_p, rseq⟩)
29.     decide()
```

Fig. 4. Replicated state-machine and consensus for process p

An important property of a successful reset is that following such a reset, in every practically infinite execution, the epoch and the round numbers of every process are always (much) smaller than *inf*. The proof is based both on the

origin of any such epoch or round number, which is 0, and on the sequential increment operations. The time needed to execute these sequential operations is proportional to the value of the epoch/round numbers. Hence, reaching a value of *inf* takes practically infinite time. Next, we prove that following a successful reset of p, any process q may invoke reset at most once before acknowledging p's reset. We use the above small epoch and round numbers property to conclude that no process q invokes reset after q balances, following a successful reset by a process p. Then, we show that following a successful reset there are at most $n-1$ resets.

The final part of the proof shows that a successful reset takes place. The proof uses the eventual behavior of the balance/unbalance and the reset sequence numbers. The proof assumes to the contrary that there are (slightly less than) $2n^2$ unsuccessful resets. Every such reset, executed by a process p, is unsuccessful due to the fact that there is at least one process q, with a large epoch or round number, for which the unbalance attempt of p has not succeeded (i.e., q did not reset its epoch and round numbers following the unbalance attempt of p) and the other processes copied this large value. Note that this happens when p does not know the actual state of the registers (i.e., when the values of the registers and the local variables differ). When such a scenario occurs, we say that q *interferes* with the reset of p. The use of reset numbers that are incremented modulo $2n$ implies that q may interfere in the resets of p at most twice in every $2n$ sequential resets of p. Thus, when p executes $2n$ unsuccessful resets there is at least one neighbor q that interferes three times. Since three interferences of a process cannot happen, it holds that within at most $2n^2$ sequential resets (by any process) at least one reset is successful.

Theorem 2 uses the above observations about the existence of a successful reset to show that there is a practically infinite resetless execution. Moreover, assuming failure detector indications in such a resetless execution, both the consensus properties (eventual termination, validity and agreement) and the replicated state-machine properties (eventual coordination and consistency) hold. The following Theorem states the correctness of the algorithm. Details are omitted from this extended abstract.

Theorem 2. *Every execution E of the consensus and the replicated state-machine algorithm, with an eventual strong failure detector, has a practically infinite suffix after at most $2n^2 - n$ resets that satisfies the consensus and replicated state-machine tasks.*

5 Concluding Remarks

While the definition of the consensus task is a combination of the safety and the eventual liveness properties, a self-stabilizing consensus ensures *eventual safety* and *eventual liveness*. Moreover, the self-stabilizing consensus task is suitable for on-going long-lived systems, in which there are repeated invocations of consensus incarnations. The self-stabilizing consensus will ensure the safety and the eventual liveness requirement starting from a consensus incarnation (epoch). In

fact, when started in a predefined initial configuration (with epoch and round numbers zero, and no resets or unbalance actions) safety is ensured as long as no transient faults occur.

To the best of our knowledge our work is the first to introduce a complete solution for a self-stabilizing asynchronous bounded memory consensus. Our solution starts from the design of the self-stabilizing eventual strong failure detector. Then, we present the asynchronous bounded self-stabilizing consensus that assumes an eventual strong failure detector. Finally, we expose all the details required for using the self-stabilizing consensus algorithm for implementing the self-stabilizing replicated state-machine, including stabilization of the bounded consensus incarnation (epoch) numbers.

New consideration, namely unboundedness, is introduced and used in our algorithm. One application of this work is extending the results in [11,19] to ensure the eventual stability of the consensus output, and this time in asynchronous executions in the presence of transient faults and crashes.

Acknowledgments. We would like to thank Gregory Chockler and Seth Gilbert for their fruitful discussions during the first stage of the research.

References

1. U. Abraham, "Self-stabilizing timestamps," Theoretical Computer Science, Vol. 308(1-3), 449-515, 2003.
2. M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier and S. Toueg, "On implementing omega with weak reliability and synchrony assumptions," *In Proc. of the 22nd ACM symposium on Principles of distributed computing*, 306-314, July 2003.
3. M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Communication-efficient leader election and consensus with limited link synchrony," *In Proc. of the 23rd ACM symposium on Principles of distributed computing*, 328-337, July 2004.
4. A. Arora, S.S. Kulkarni, and M. Demirbas, "Resettable vector clocks," *In Proc. of the 19th ACM Symposium on Principles of Distributed Computing*, 269-278, August 2000.
5. H. Attiya and J.L. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics (2nd edition)*. John Wiley and Sons, Inc., 2004.
6. J. Beauquier and S. Kekkonen-Moneta, "Fault-tolerance and self-stabilizing: Impossibility results and solutions using self-stabilizing failure detectors," *International Journal of Systems Science*, Vol. 28(11), 1177-1187, 1997.
7. K.P. Birman, "Replication and Fault-Tolerance in the ISIS System," *ACM Symposium on Operating Systems Principles* (SOSP), 79-86, 1985.
8. T.D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, Vol. 43(2), 225-267, March 1996.
9. T.D. Chandra, V. Hadzilacos and S. Toueg, "The weakest failure detector for solving consensus," *In Proc. of the 11th ACM Symposium on Principles of Distributed Computing*, 147-158, August 1992.
10. B. Chor, A. Israeli and M. Li, "On processor coordination using asynchronous hardware," *In Proc. of the 6th ACM Symposium on Principles of Distributed Computing*, 86-97, August 1987.

11. L. Davidovitch, S. Dolev, and S. Rajsbaum, "Consensus continue? Stability of multi-valued continuous consensus!," *In Proc. of the Workshop on Geometry and Topology in Concurrency and Distributed Computing*, 21-24, October 2004.
12. C. Delporte-Gallet, H. Fauconnier and R. Guerraoui, "Failure Detection Lower Bounds on Registers and Consensus," *In Proc. of the 16th International Conference on Distributed Computing*,237-251, October 2002.
13. E.W. Dijkstra, "Self-Stabilizing Systems in spite of Distributed Control," *Communications of the ACM*, Vol. 1(11), 643-644, 1974.
14. S. Dolev, *Self-stabilization*. MIT press, 2000.
15. S. Dolev, A. Israeli and S. Moran, "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity," *In Proc. of the 9th ACM Symposium on Principles of Distributed Computing*, 103-117, August 1990.
16. S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte, "Virtual Stationary Automata for Mobile Networks," *Proc. of the 2005 International Conference On Principles Of Distributed Systems*, (OPODIS), 2005.
17. S. Dolev, S. Gilbert , N. Lynch, E. Schiller, A. Shvartsman, and J.L. Welch, "Virtual Mobile Nodes for Mobile Ad Hoc Networks," *International Conference on Principles of Distributed Computing*, (DISC 2004), 230-244, 2004.
18. S. Dolev, R.I. Kat and E.M. Schiller, "When Consensus Meets Self-Stabilization: Self-Stabilizing Failure-Detector, Consensus and Replicated State-Machine," *Computer Science, Ben-Gurion University of the Negev, TR #06-05*, May 2006.
19. S. Dolev and S. Rajsbaum, "Stability of long-lived consensus," *Journal of Computer and System Sciences*, 26-45, August 2003.
20. S. Dolev and J.L. Welch, "Wait-free clock synchronization," *Algorithmica*, Vol. 18, 486-511, 1997.
21. M.J. Fischer, N.A. Lynch and M.S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, Vol. 32(2), 374-382, April 1985.
22. F.C. Freiling, R. Guerraoui and P. Kouznetsov, "The Failure Detector Abstraction," *Department for Mathematics and Computer Science, University of Mannheim*, TR-2006-003, 2006.
23. F.C. Freiling and H. Völzer, "Illustrating the impossibility of crash-tolerant consensus in asynchronous systems," *ACM SIGOPS Operating Systems Review*, Vol. 40(2), 105-109, April 2006.
24. A. Fox and D. Patterson, "Self-Repairing Computers," *Scientific American*, June 2003.
25. E. Gafni and L. Lamport, "Disk Paxos," *Distributed Computing*, Vol. 16(1), 1-20, 2003.
26. M. Herlihy, "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13(1), 124-149, 1991.
27. M. Hutle and J. Widder, "Self-Stabilizing Failure Detector Algorithms," *Parallel and Distributed Computing and Networks Conference*, 485-490, February 2005.
28. S.S. Kulkarni and A. Arora, "Multitolerance in Distributed Reset," *Chicago Journal of Theoretical Computer*, CJTCS-1998-4, 1998.
29. L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, Vol. 16(2), 133-169, May 1998.
30. L. Lamport, "Time, Clocks, and Ordering of Events in a Distributed System," *Communication of the ACM*, Vol. 21(7), 558-565, 1978.
31. W. Lo and V. Hadzilacos, "Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems," *Lecture Notes in Computer Science (WDAG)*, Vol. 857, 280-295, October 1994.

32. M.C. Loui and H.H. Abu-Amara, "Memory requirements for agreement among unreliable asynchronous processes," Advances in Computing Research, Vol. 4, 163-183, 1987.
33. N. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.
34. D. Malkhi, F. Oprea and L. Zhou, "Omega Meets Paxos: Leader Election and Stability without Eventual Timely Links," *In Proc. of the 19th Symposium on Distributed Computing (DISC)*, 99-213, September 2005.
35. M. Raynal, "A Short Introduction to Failure Detectors for Asynchronous Distributed Systems," *SIGACT News* column, 36(1), March 2005.

On the Cost of Uniform Protocols Whose Memory Consumption Is Adaptive to Interval Contention

(Extended Abstract)

Burkhard Englert*

California State University Long Beach, Dept. of Comp. Engr. & Comp. Science
Long Beach, CA 90840
benglert@csulb.edu

Abstract. A distributed shared memory protocol is called *memory-adaptive*, if all writes to MWMR registers are "close to the beginning of shared memory", that is the indices of all MWMR registers processes write to when executing the protocol are functions of the contention. The notion of memory-adaptiveness captures what it means for a distributed protocol to most efficiently make use of its shared memory. We previously considered a *store/release* protocol where processes are required to store a value in shared MWMR memory so that it cannot be overwritten until it has been released by the process. We showed that there do not exist uniformly wait-free store/release protocols using only the basic operations read and write that are memory-adaptive to point contention. We further showed that there exists a uniformly wait-free store/release protocol using only the basic operations read, write, and read-modify-write that is memory-adaptive to interval contention and time-adaptive to total contention. This left a significant gap which we close in this paper. We show that no uniform store/release protocol can exist that is memory adaptive to interval contention and only uses read/write (no read-modify-write) registers. We furthermore illustrate the validity and practicality of the concept of memory adaptiveness by providing a uniform, memory-adaptive to interval contention store/release protocol for Network Attached Disks.

1 Introduction

Shared memory algorithms such as collect or renaming provide essential building blocks for many applications. Most often collect or renaming are designed based on an a priori knowledge of an upper bound n on the number of participating processes or of an upper bound N on the ids of participating processes. Algorithms such as collect or renaming, however, become inefficient if only few of the n processes are actually participating. This motivated researchers to look for *adaptive* algorithms whose step complexity only depends on the number of

* Research supported by the Dept. of Transportation under METRANS USC-111699.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 64–80, 2006.

participating processes. Besides a possibly inefficient use of *time*, inefficient use of *space* is also a potential drawback of many distributed algorithms. In particular many shared memory algorithms require memory space whose size is a function of N (or n) even if only few of the processes are actually participating. Hence to truly improve the efficiency of distributed algorithms the step complexity should be made adaptive to the number of participating processes, i.e. the contention, and the space requirements should (if possible) depend on the number of participating processes or the contention. Following this approach one obtains two possible kinds of adaptive algorithms: Algorithms where the *step complexity* adapts to the contention are traditionally called *adaptive*. We called such algorithms *time-adaptive* to distinguish them from algorithms where the memory space consumption adapts to the contention that we called *memory-adaptive* [27]. In *memory-adaptive* algorithms processes are only allowed to write to a shared MWMR register whose index is a function of the contention (possibly point-, interval- or total contention) during the processes previous shared memory access.

Time-adaptive algorithms have a worst case step complexity that is bounded by a function of the number of concurrently participating, or actually active processes [6]. Motivated by Lamport's MX algorithm [34], many such time-adaptive algorithms have since been designed [3,4,6,7,8,9,11,17,18,20,21,22,26,35].

With respect to memory consumption of time-adaptive renaming or collect algorithms Afek, Boxer and Touitou [5] showed that the number of Multi-Writer Multi-Reader (MWMR) registers used must be a function of N. They specifically show that for any constant d there is a large enough N_d such that every long-lived time-adaptive (to interval contention, and hence, point contention as well) read/write implementation of collect (and renaming) with N_d processes must use at least d MWMR registers. In their paper they use a simple object called weak test and set [15] to derive their impossibility results. More recently Attiya, Fich and Kaplan [19] significantly improved on [5]. They showed that if a collect algorithm is time-adaptive to total contention, namely, its step complexity is $f(k)$, where k is the number of processes that ever became active during the current execution, then it uses $\Omega(f^{-1}(N))$ MWMR registers, where N is the total number of processes in the system.

In this paper we will remove the assumption of a known upper bound on the number of participating processes and consider *uniform* protocols [16,29,33], i.e., protocols that do not require a priori knowledge of or an upper bound on the number of processes that may participate. At the same time we will assume that the number of participating processes is always finite.

The notion of *memory-adaptiveness* [27] requires that each(!) write operation that a process makes must be close to the "front" of shared memory. The idea here is that if protocols allow processes to write to registers whose index depends say on the processes id and no upper bound on the number of participating processes is known in advance then memory must be unpredictably large. On the other hand, if we can guarantee that the memory required by each protocol that runs on a shared memory system is a bounded function of the contention,

then a distributed operating system can allocate large memory blocks to each protocol on an ad hoc basis and, on rare occasions when needed, increase or decrease the individual allocations as necessary.

Also, if processes are allowed to write to registers with arbitrary indices in time-adaptive protocols they eventually must move the values they wrote close to the beginning of memory for the protocol to stay time-adaptive during solo executions. Hence ideally processes will want to register in a fixed finite subset of the infinite set of MWMR registers that we called "close to the beginning of shared memory" [27].

Consider the renaming problem [3,4,6,7,10,20,36] for example: processes are allowed to use any shared MWMR register during the execution of the protocol, even a register with an extremely large index, but the final result must lie within a bounded distance from the front of shared memory. In the definition of *memory-adaptiveness*, to capture the notion of having to write close to the front of shared memory *every* time, we require processes to write to a MWMR register whose index is a function of the contention during the previous operation of the same process.

In [27] we investigated simple tasks, *store* and *release*, that require a given process to store a value in shared MWMR memory that cannot be overwritten by any other process and then to erase the value when no longer needed, freeing the memory for other processes to use.

We studied whether these simple commands can be implemented memory-adaptively under different assumptions about the contention of the protocol.

We showed that in a system with *infinitely* many MWMR registers and *infinitely* many SWMR registers: 1. There is no uniform, long-lived memory-adaptive to *point-contention* implementation of store/release that uses only read/write registers. 2. There does exist a uniform, long-lived implementation of store and release in the read-write model that is memory-adaptive to *total contention*. 3. Allowing write-plus (read-modify-write) there exists a uniform, long-lived implementation of store and release in the read-write model that is memory-adaptive to *interval contention*.

The question remained, however, whether in this setting there exists a uniform, long-lived, memory-adaptive to *interval contention* store release protocol that uses **only** read/write registers. This is of particular interest because one could argue that with adaptiveness to interval contention "true adaptiveness" really starts, since adaptiveness to total contention allows for the memory requirements to grow independent of the contention during operations. Moreover, even though we were able to show that there exists a protocol memory-adaptive to interval contention using read-modify-write registers, we were not able to justify the use of these stronger primitives. In this paper we will close this gap and hence significantly strengthen our previous results.

1.1 Our Approach

To prove our impossibility result we will use a covering construction as in the memory-adaptiveness to point contention impossibility proof in [27]. Our proof,

however, will have to be more complex and requires greater care. In [27] we were able to select all potentially participating processes in advance and construct the run that produced the contradiction as a single run. Using the pigeonhole principle we simply reduced the set of participating processes in pseudo solo runs more and more until all MWMR registers at the beginning of shared memory (that are accessed in solo runs) were covered. All processes that were covering one of the MWMR registers at the end of the construction had been participating from the beginning. If we want to obtain a contradiction to memory-adaptiveness to **interval contention**, however, we cannot proceed in this manner. Every time a new register is covered we must choose a new set of processes that never acted before since otherwise processes could receive information about the increasing interval contention allowing them to write to more ("new") MWMR registers and making it impossible for us to get a contradiction. In other words, when we extend the covering from the first j to the first $j + 1$ MWMR registers that processes write to during their pseudo solo runs, we first eliminate the traces of the process that now covers the $j+1$st register. This is done by releasing covering writes to overwrite this process. In [27] we were simply able to cover each of the required MWMR registers with infinitely many processes and then release covering writes as need be. Here we are not able to do this anymore. Instead - to ensure that our construction is memory-adaptive to interval contention - we must rebuild our covering after each time the covering writes have been released. Otherwise it might be possible for processes to detect that some of the processes involved in these covering writes were concurrently active with them and hence allow them to write to MWMR registers outside the bounded (by a constant) range of MWMR registers at the "beginning of shared memory". As we would cover more registers, processes would be able to memory-adaptively write to more "new" MWMR registers outside the "beginning of shared memory" making it impossible to get a contradiction.

As a result, our construction is similar to the construction in [5], however we note that the result there does not directly imply our result since it assumes a finite number of available MWMR registers while we allow for infinitely many available MWMR registers. Therefore we must always ensure that at all important steps of our construction processes are only able to write to the bounded set of registers at the "beginning of shared memory". We will do this by making these processes believe that they execute the protocol solo. Also in [5] algorithms are assumed to be time-adaptive allowing to bound the execution length of each participating process since any time-adaptive protocol is by definition wait-free. Here we do not assume that our protocols are time-adaptive. Instead to bound the execution length (otherwise processes could simply keep reading each others SWMR registers) we assume the protocol to be uniformly wait-free, that is that the length of all executions is uniformly bounded. The covering techniques used in our impossibility proofs first appeared in [24] to show some bounds on the number of registers necessary for mutual exclusion. Similar covering arguments were used in many recent papers to prove space and time lower bounds. For example, see the survey by Fich and Ruppert [28].

1.2 Network Attached Disks

In the second part of the paper we will consider an important and natural application of memory-adaptive algorithms. Recent advances in storage technology [32] have enabled systems like Storage Area Networks [13,14,23,37,39], which have network attached disks or NADs. A NAD is a simple device that just executes requests to read and write blocks of data. It can be accessed by any process in the system, so that the NADs effectively become a shared storage medium that can be used to solve distributed problems such as consensus, as in Disk Paxos [1,25,30]. Unlike message-passing systems, which typically require a majority of processes to be correct to avoid partitioning, NADs allow protocols that can withstand the crash of any number of processes. Therefore - like conventional shared-memory models - the model allows uniform protocols. One difficulty of this model is that a NAD can fail by crashing and thereby become inaccessible.

In [12] Aguilera, Englert and Gafni showed that one cannot uniformly implement a MWMR register on a NAD with a finite number of fail-prone base registers, even if the implementation need not be wait-free. Therefore, one cannot use the standard technique of implementing a MWMR register by first implementing SWMR registers: doing so would blow up the space complexity.

These implies the need for infinitely many base registers. Since this is however an unrealistic assumption in real NAD's, memory-adaptive algorithms are of particular practical interest on such disks. If uniform protocols that require infinitely many base registers and that run on NAD are memory-adaptive to interval contention they will remain practical since they will allow us to efficiently bound the memory requirements based on this contention.

Based on our impossibility result, we will show how store/release can be - uniformly and memory-adaptively to interval contention - implemented on *Active Disks*. This is possible since read-modify-write objects are available on Active Disks [2,38].

1.3 Related Work

Uniform protocols have been studied (e.g., [16,33]), particularly in the context of ring protocols. Adaptive protocols, i.e. protocols whose step complexity is a function of the size of the participating set, have been studied in [6,7,8,26,35]. Long-lived adaptive protocols that assume some huge upper bound N on the number of processes, but require the complexity of the algorithm to be a function of the concurrency have been studied in [3,4,9,10,11,20,21,36].

1.4 Contributions

We summarize the contributions of our paper.

1. Interval contention: (Theorem 1) We show that in a system with *infinitely* many MWMR registers and *infinitely* many SWMR registers, for any constant d, there exists a number N_d such that if N_d processes are allowed to

participate then there does not exist a *memory-adaptive* (to INTERVAL contention) implementation of store/release. In other words we show that under these conditions processes cannot memory-adaptively store a value in shared memory. This closes a gap that remained open in [27] and implies the impossibility of uniform memory adaptive to point contention algorithms [27] with all its consequences. Moreover it justifies the use of read-modify write registers and similar stronger primitives to uniformly implement memory-adaptive to interval contention store/release [27].

2. We present a uniform implementation of memory-adaptive to interval contention store/release on Active Disks. While it was shown [12] that one cannot uniformly implement a MWMR register on a NAD with a finite number of fail-prone base registers, this results provides a realistic and practical building block for algorithms on NAD where an upper bound on the interval contention can be enforced.

Paper Organization: We will first in Section 2 review our model, followed by our impossibility proof in Section 3. We conclude with a transfer of our algorithm to NAD's (Section 4) and some final remarks (Section 5).

2 Model and Preliminaries

For our impossibility result we use the standard shared-memory model of distributed computation. There are infinitely many processes each modeled by infinite-state machines that are capable of unbounded computation. Processes participate in a distributed deterministic asynchronous protocol and are indexed by the positive natural numbers so that each process "knows" its own "name." There are two areas of memory: the single-writer multi-reader (SWMR) space and the multi-writer multi-reader (MWMR) space.

In the SWMR space each register is associated with a distinct process so that only this process is allowed to write to this register while all other processes are able to read it. Each SWMR register can store an unbounded number of bits. The MWMR registers have all the same properties as the SWMR registers with the exception that any process may both read *and* write to any register.

Processes access the memory space using basic atomic operations. The atomic operations we will allow in this paper are *read, write,* and *read-modify-write*.

- READ: To execute a read command, a process specifies a register to be read and upon completion of the read, the process has gained a snapshot of the contents of the specified register.
- WRITE: A process specifies which register to write to (in either private or shared memory) and the data to be written. Upon completion of the write command, all previous data is overwritten with the new data specified by the process. (Note that we do not allow a process to overwrite "part" of a register.)
- READ-MODIFY-WRITE (RMW): The RMW command allows the unbreakable execution of the following code (where X is a shared variable and f is a mapping):

```
function RMW(X,f)
begin
   temp←X;
   X←f(X);
   return(temp);
end
```

A *protocol* is an algorithm that accomplishes a task using basic operations. An *adaptive* protocol is one in which the resources consumed by the protocol are functions of the number of processes that actually participate in the protocol (a.k.a. active processes) rather than the total number of processes. In an adaptive protocols, the size of the resources (time or space) consumed is a function of the contention. The contention can be measured in three different ways, effecting the strength of adaptiveness of a protocol: *Total contention* refers to the total number of processes that become active during the entire execution of the protocol. *Interval contention* during a given processes protocol is defined to be the total number of processes that become active during the execution interval of a processes protocol. Finally, *point contention* during a given processes protocol refers to the maximum number of processes that are simultaneously active at any point during the execution interval of a processes protocol.

A protocol is *time-adaptive* to a particular type of contention if the maximum number of basic operations executed during the protocol by any given process is a bounded function of the contention type. This type of definition has been studied extensively [3,6,4,7,8,9,11,17,18,20,21,22,26,35].

We say that a basic operation is *memory-adaptive* [27] to a type of contention if and only if the following is true. Whenever a process executes a basic operation, if the next basic operation changes the state of a shared memory register, the index of the register at which this change occurs is a bounded function of the contention (point, interval or total) at the time of the previous basic operation. (In the asynchronous model, without loss of generality, we may assume that the first basic operation in any protocol is a read, which does not change the state of any register.) In other words, a process can read wherever it wants, but it can only write to places that are as close to the "front" of shared memory as possible. Most time-adaptive algorithms that were presented [3,6,4,7,8,9,11,17,18,20,21,22,26,35] are not memory-adaptive in this sense. They might however force that the final result of a computation lies within a bounded distance of the "front" of shared memory.

The three protocols that we will focus on in this paper are *store*, *release* and *Weak Test and Set*.

- STORE: A data value is specified in advance by the process. The goal is for the process to store the data value in some shared register in such a way that upon completion the process knows that the value will not be moved or erased by any other process until the register is explicitly released.
- RELEASE: Assumes execution of a previous store protocol. Upon completion, the shared register occupied by the process is released.

– WEAK TEST AND SET (WT&S): Based on a test&set object. WT&S object guarantees safety - no two processes are able to concurrently "set the bit". However liveness is only guaranteed in solo executions: if two or more processes access a WT&S object concurrently it is possible that none of them captures the bit (i.e. none of the participating processes reads 0 as the bits value). We model the behavior of a weak test and set object with the following program (Figure 1). Each process is in one of four possible states: thinking, WT & Set, eating and RESET.

```
TS: object of type WT&S
Process p:
repeat forever
    thinking_section_p
    tbit:= WT&SET_p(TS)
    if tbit = 0 {
            eating_section_p
            RESET_p(TS) }
end repeat forever
```

Fig. 1. Weak Test & Set algorithm

A WT&S object satisfies the following two properties:

- **Exclusion:** At most one process is eating at any system state of the execution.
- If a process becomes hungry, that is leaves the thinking state, while all other processes are thinking and it only takes steps then it must eventually start eating.

Note that STORE and RELEASE are fundamental building blocks useful for many distributed protocols (e.g. collect, mutual exclusion, consensus, approximate agreement, and so on). WEAK TEST AND SET on the other hand is useful in the proof of lower bounds.

We call a protocol *uniformly wait-free* if there exists a uniform bound applicable to all processes on the number of basic operations that the protocol requires before termination. All protocols considered in this paper will be uniformly wait-free. We make the following definitions:

– A system state consists of the state of all processes and the value of all registers in the system. A system has one or more initial system states in which the system starts its execution.
– We say that a system state s is an *idle-state*, if all the processes are thinking at s.
– A run α is a finite or infinite sequence of events, starting from an initial system state. If the sequence is finite, we say that the run is finite.
– A run segment x_α of a run α is a finite, continuous subsequence of events of α.
– A *solo run segment* of p is a run segment starting at an idle-state, in which only p takes steps.

- A run segment is called a *p-segment* if it starts in a state s in which p is thinking, and in which only p takes steps. Note that a *p-segment* is not necessarily a *p-solo-segment*.
- We say that a state s is *transparent* with respect to process p, if p is thinking in s, and there is a p-segment starting in s, that p cannot distinguish from some solo run segment of p starting at an idle-state and ending with p eating. State s is transparent with respect to a set of processes Q, if s is transparent $\forall p \in Q$.
- A *pseudo-solo run segment* of p denoted $solo_p$, is a p-segment starting at state s s.t., s is transparent with respect to p, and ending with p eating. Note that by the definition of transparent, p cannot distinguish a pseudo-solo run segment from some sole run segment starting at an idle-state and ending with p eating.
- A register r is *covered* by process p at the end of run α, if a write operation by p to r is enabled at the end of α.
- Given a run segment x and a state s, $s \cdot x$ denotes the concatenation of x after some run α ending at s, assuming that α exists.

3 Interval Contention

In [27] we showed that there is no uniform, long-lived and *memory-adaptive to point contention* store and release protocol. We will now strengthen this result by showing that there is no uniform and *memory-adaptive to interval contention* weak test and set protocol. We begin by showing that we can implement WT&S from memory-adaptive store and release. The reduction uses the fact that store/release is uniformly wait-free.

Reduction from memory-adaptive and uniformly wait-free store and release to memory-adaptive and uniformly wait-free WT&S: In the uniform memory-adaptive store and release problem, processes repeatedly store and release values in shared memory. The index of the MWMR registers to which they write must be in the range $\{1, ..., f(k)\}$ where k is the number of processes that are active concurrently with the process that is trying to store or release a value. So when a process runs solo the index of the MWMR registers it writes to must be bounded by some constant $f(1)$. In an implementation of WT&S from memory-adaptive store and release we use one copy of the memory adaptive store and release object. To perform the WT&S operation a process first attempts to memory adaptively store the value "active" in shared memory. If at any point in time during the execution of the algorithm it writes to a MWMR register with an index greater than $f(1)$ it fails the WT&S and - if it already stored a value - releases the value it stored. Otherwise it reads all other $f(1)$ MWMR registers to see if any other process was able to concurrently store a value in shared memory. If it sees any other process as active it fails WT&S and releases the value. Otherwise it wins the WT&S object. To release the WT&S object a process releases the value it stored in shared memory. This clearly satisfies the required properties and implements the desired object.

We furthermore assume that each process has only one single-writer, single-reader (SWMR) register: All SWMR registers of a process can always be replaced by a single SWMR register.

A condition or property holds in a run if it holds at the end of that run (unless we state otherwise).

It hence suffices to show that there is no uniform memory-adaptive to interval contention (uniformly wait-free) Weak Test And Set implementation using only read/write registers.

3.1 The Theorem

Clearly if an implementation of a weak test and set object is memory-adaptive then there exists a constant i such that, no *solo run segment* writes to a MWMR register with an index greater than i. We say that the algorithm is *"i-solo-memory-adaptive"*.

Theorem 1. *For any constant i there is no long-lived, uniformly wait-free, i-solo-memory-adaptive to interval contention implementation of Weak-Test & Set in a system with infinitely many processes and infinitely many MWMR and SWMR read/write registers.*

Note that in contrast to [5] we do not require our algorithms to be time-adaptive. Hence the result in [5] does not immediately imply our result. Moreover the number of available MWMR registers is now unbounded, that is when covering writes are released processes that detect contention can possibly write to more than the first k MWMR registers. After addressing such issues our proof proceeds in a similar manner as [5].

The proof is by way of contradiction. First, assume that there is a memory-adaptive to interval contention WT&S implementation with infinitely many MWMR registers for a system with infinitely many processes. Then we show that under these conditions there is a run in which two processes p and q are in the critical section, i.e. are eating at the same time.

1. We construct a run prefix α s.t., the state at the end of α is transparent with respect to p, and every MWMR register that p writes in its pseudo-solo run starting at α is covered. As in [5] we construct this cover inductively.
2. Let $solo_p$ be the pseudo-solo run segment of process p starting after α. Hence p is eating in $\alpha \cdot solo_p$. Let $\{r_1, ..., r_{i'}\}$ be the set of MWMR registers written by p in $solo_p$, where $i' \leq i$.
3. We now enable the covering writes and wait until all processes reach a thinking state. This is guaranteed by the fact that we are dealing with a uniformly wait-free WT&S implementation.
4. We ensure that processes that are active do not detect each other by selecting them in such a way that they do not read each others SWMR registers. (This also follows from the protocol being uniformly wait-free. We show later in detail that this is possible.)
5. We select a process q that does not read the SWMR register of p. This process will enter the critical section together with p, a contradiction.

3.2 Sketch of the Proof of the Main Lemma

The proof is based on [5]. We construct α by first, for explanatory reasons, making strong assumptions. We then remove these assumptions to obtain the claimed result.

We use the following notations. For an infinite set R_{MW} of MWMR registers, we consider W to be an i-solo-memory-adaptive implementation of WT&S in the Read/Write shared memory model. Note that by the definition of memory-adaptiveness i must be a constant.

Phase 1

Assumption A: There are no write operations to SWMR registers in all legal runs. That is, we assume for the moment that there is a uniformly wait-free WT&S protocol that is i-solo-memory-adaptive to interval contention and uses no SWMR registers.
Assumption B: If G is a set of processes and s is a state that is transparent with respect to G, then during their pseudo-solo runs starting at s all processes in G write in the same MWMR registers in the same order.

These assumptions will later be removed. We will be able to remove assumption B because of the i-solo-memory-adaptiveness of the algorithm, that is processes can only write to a fixed number of MWMR registers in pseudo-solo runs and the fact that our protocol is uniformly wait-free. Hence using a Ramsey theoretic argument we can find a large enough set of processes that will write in the same order into these registers.

In the following lemma α is denoted by $s \cdot \beta$ and satisfies the properties of α: Property 1: the state at the end of $s \cdot \beta$ is transparent with respect to some set of processes called G_e, and property 2: there is a cover on all the MWMR registers written by processes in G_e in their pseudo-solo run segments, starting after $s \cdot \beta$. The size of G_e is a parameter and is determined in the full proof.

Lemma 1. *Let W be a long-lived, uniformly wait-free, i-solo-memory-adaptive WT&S algorithm satisfying assumptions A and B. Then for any constant e there exists a constant n_e s.t., for any set of processes G, $|G| \geq n_e$ and for any state s transparent with respect to G, there is a run segment β and a set of processes $G_e \subseteq G$ s.t., the following holds: (1) $|G_e| \geq e$, (2) the state at the end of $s \cdot \beta$ is transparent with respect to G_e, and (3) all the MWMR registers written in the pseudo-solo run segments of processes in G_e, after $s \cdot \beta$, are covered in $s \cdot \beta$.*

Sketch of proof of lemma: The proof is by induction on i:

Lemma 2. *Let W be a long-lived, uniformly wait-free, i-solo-memory-adaptive WT&S algorithm satisfying assumption A and B. Then for every j, $0 \leq j \leq i$ and for every constant e there exists a constant $n_{e,j}$ s.t., for any set of processes G, $|G| \geq n_{e,j}$ and for any state S transparent with respect to G, there is a run segment β_j and a set of processes $G_j \subseteq G$ s.t., the following holds: (1) $|G_j| \geq e$, (2) the state at the end of run $s \cdot \beta_j$ is transparent with respect to G_j, and*

(3) there exists a set $R_j = \{r_1, ..., r_j\}$ of MWMR registers that are covered in $s \cdot \beta_j$ and $r_1, ..., r_j$ are the first j MWMR registers written (in this order) in the pseudo-solo run segments of processes in G_j, after $s \cdot \beta_j$, or, the pseudo solo runs starting at $s \cdot \beta_j$ have less than j writes to a set $R_{j'} \subset R_j$ (in the same order) of MWMR registers, where $R'_j = \{r_1, ..., r_{j'}\}$, $1 \leq j' < j$ and all these writes are covered in $s \cdot \beta_j$.

Proof. In the full proof we show that $n_e = n_{e,i}$ is a function of i, j and e. It is similar to [5] and can be found in the full version of the paper. □

Phase 2

We now relax assumption A. To do this we use techniques developed in [5]. The run constructed in the previous lemma may not be valid anymore, as processes are allowed to write to their SWMR registers. The argument presented above may collapse in one of the following two ways:

1. The participating processes in any *clean* run segment may read the SWMR registers of other active processes. In particular, they may read the SWMR register of the processes whose traces their writes are supposed to eliminate. They would then leave the system in a non-transparent state by writing about the value they read.
2. After a *clean* run segment, a process q might start its q-segment execution and may read the SWMR register of another concurrently active process p. Hence, q will not perform a pseudo-solo run anymore, that is it may write to a MWMR register with an index greater than i and it may stop without covering the MWMR registers. Moreover, q may decide "on the spot" to write into different MWMR registers than what we originally planned.

As in [5] or [12], we will avoid the two *dangerous* situations by not allowing processes, whose SWMR registers are later read to take part in the constructed run. So, if in any given state in the run, if process q reads the SWMR register of process p and p is active, we construct another run in which p is replaced by another process p'. Process q will still read the same SWMR registers. The behavior of p and p' is in some sense "equivalent". They both write and cover the same MWMR registers. All we need to do is to show that a process like p' always exists since (1): There is a large enough set of processes to select p' from s.t., p' did not participate in the run before and has the same general properties as p. We can do this since at any give point in time at most finitely many processes participate in the execution while infinitely many processes are available. (2): Process q can perform only a constant number of read operations, since the number of concurrently active processes in the run is a function of d and k and since the algorithm is uniformly wait-free.

We maintain a large enough set of 'equivalent' runs, which allows us to replace at any point in time at which we fail to reach a transparent state. This set will shrink as the construction progresses.

Definition 1. *Two runs β and β' are equivalent with respect to a set of processes G if (1) the state at the end of both runs β and β' is transparent with respect to*

G, (2) the sets of MWMR registers covered in β and β' are the same, and (3) if process p participates in both β and β' then p cannot distinguish between the two runs.

In our construction, whenever a process p that was previously selected to participate in the run is discovered by a covering process, we need to replace it with some other process p' that cannot be discovered. We achieve this by considering an equivalent run in which p' takes steps instead of p. This allows us to restate the central inductive lemma as follows:

Lemma 3. *Let W be a long-lived, uniformly wait-free, i-solo-memory-adaptive WT&S algorithm. Then for every j, $0 \leq j \leq i$ and for every constant e there exists a constant $n_{e,j}$ s.t., for any set of processes G, $|G| \geq n_{e,j}$ and for any state S transparent with respect to G, there is a run segment β_j and a set of processes $G_j \subseteq G$ s.t., the following holds: (1) $|G_j| \geq e$, (2) the state at the end of run $s \cdot \beta_j$ is transparent with respect to G_j, and (3) there exists a set $R_i = \{r_1, ..., r_j\}$ of MWMR registers that are covered in $s \cdot \beta_j$ and $r_1, ..., r_j$ are the first j MWMR registers written (in this order) in the pseudo-solo run segments of processes in G_j, after $s \cdot \beta$, or, the pseudo solo runs starting at $s \cdot \beta_j$ have less than j writes to a set $R_{j'} \subset R_j$ (in the same order) of MWMR registers, where $R'_j = \{r_1, ..., r_{j'}\}$, $1 \leq j' < j$ and all these writes are covered in $s \cdot \beta_j$. And in addition, there is a large enough set of runs equivalent to β_j with respect to a large enough set $G'_j \subseteq G_j$.*

Note that we also need to modify the proof of the Main Theorem along the lines of the proof of this lemma.

Proof. Similar to [5]. For lack of space we leave it to the full paper. □

It remains to remove Assumption B. During a WT&SET operation processes are now allowed to write to different MWMR registers in different orders. This means that the cover we constructed earlier might not be on the "correct" registers anymore since two processes p and q may write into the MWMR registers in different orders.

To overcome this difficulty we first recall that we are only interested in pseudo solo runs. We know, however, that processes executing such runs are only allowed to write to the first i MWMR registers in shared memory. Hence in pseudo-solo runs the number of MWMR under consideration is a constant. Second we recall that our algorithm is uniformly wait-free that is the length of every pseudo solo run is a constant. Hence we can consider the different sequences of write operations to MWMR registers by the different pseudo-solo run segments of processes in G. The number of these sequences is bounded by i and m where m is the uniform bound on the length of a solo execution. Each such sequence defines an equivalence class in G. Since G is infinite, we can always find a subset of processes that in pseudo solo runs performs the same sequence of writes to MWMR registers starting at s.

But since in two different states s and s' that are transparent with respect to G the sequence of MWMR registers that processes in G write to in pseudo-solo

runs need not be the same, the required subset of processes cannot be computed in advance. Instead it is computed iteratively in rounds as in [5].

We restate the main inductive lemma with assumption B removed.

Lemma 4. *Let W be a long-lived, uniformly wait-free, i-solo-memory-adaptive WT&S algorithm. Then for every j, $0 \leq j \leq i$ and for every constant e there exists a constant $n_{e,j}$ s.t., for any set of processes G, $|G| \geq n_{e,j}$ and for any state s transparent with respect to G, there is a run segment β_j and a set of processes $G_j \subseteq G$ s.t., the following holds: (1) $|G_j| \geq e$, (2) the state at the end of run $s \cdot \beta_j$ is transparent with respect to G_j, and (3) either the first j MWMR registers written in the pseudo-solo run segments of processes in G_j, after $s \cdot \beta_j$, are the same and covered in $s \cdot \beta_j$, or the pseudo solo runs starting at $s \cdot \beta_j$ have less than j writes to the same MWMR registers and all these writes are covered in $s \cdot \beta_j$. And in addition, there is a large enough set of runs equivalent to β_j with respect to a large enough set $G'_j \subseteq G_j$.*

Proof. We leave the complete proof to the full paper. It is similar to [5]. □

4 Uniform Memory-Adaptive Algorithms for NAD's

We will now discuss what our results imply for the design of memory adaptive algorithms (e.g. store/release) for NAD's. Earlier in this paper we showed that there is no uniformly wait-free, uniform store/release protocol memory-adaptive to interval contention that uses only read/write registers. In [12] it was shown that one cannot uniformly implement a MWMR register on a NAD with a finite number of fail-prone base registers, even if the implementation need not be wait-free. This implies the need for infinitely many base registers. Since this is however an unrealistic assumption, memory-adaptive algorithms are of particular practical interest in the uniform setting on NAD's. If uniform protocols that require infinitely many base registers and that run on NAD are memory-adaptive they will remain practical since they will allow us to efficiently bound the memory requirements based on the contention. Hence memory-adaptive algorithms are not only attractive but essential for uniform algorithms on NAD's.

In [27] we provided a uniform memory-adaptive to interval contention implementation of store/release using stronger primitives namely an operation we called *write-plus* which is weaker than the standard read-modify-write. The write-plus command is equivalent to specifying that the function f in the definition of read-modify-write (see the model section) is required to be a constant independent of X (the value read).

Active Disks [32] on the other hand are capable of supporting stronger semantics that are not normally provided by disk drives. In particular they can provide read-modify-write operations. Our results imply that to run realistic uniform algorithms on a NAD - that is algorithms that are memory adaptive to interval contention - read/write registers are not sufficient. Our results justify the use of Active Disks in the uniform setting. We will now show how to implement memory adaptive to interval contention store/release on active disks if disks and hence registers may fail.

Theorem 2. *There exists a long-lived, uniformly wait-free, uniform store/release protocol for Active Disks using only the operations read, write, and read-modify-write that is memory-adaptive to interval cont., time-adaptive to total cont.*

Proof. We first recall our memory-adaptive to interval contention algorithm using read-modify write registers from [27]:

We assume that memory is arranged in the form of a two-dimensional grid, this time indexed by $\mathcal{N} \times \mathcal{N}$. Whenever a process executes a read-modify-write into shared memory, it keeps a copy of what was previously written there in its private memory space along with whatever it writes into the register. As a result the process always has a complete record of all of its operations starting from the beginning till the current time in its private space along with the values that it overwrites. During each store and with each write, the process keeps track of the number of times it has stored a value in shared memory. Each write will contain a field with this parameter. The algorithm uses splitters [36]. We assume that splitters are able to hold values. Each process when executing the algorithm attempts to capture a splitter so that it can store its value in this splitter.

Using these assumptions we showed in [27] that a process has the ability to tell whether a splitter is "clean" or "dirty". In other words, the process is able to tell whether, given a splitter, there exists another process that has previously written into the splitter's slot #1 and yet has not either written into slot #2 or written into some other shared register. Based on this processes execute the following protocol: Whenever a process executes a store, it begins at splitter $(1, 1) = (i, j)$. If the splitter is taken with a value, then the process moves to $(i + 1, 1)$. If the splitter is dirty, it moves to $(i, j + 1)$. If the splitter is clean, it competes. It writes his name into slot #1 and checks slot #2. If there is a "new" name (i.e. a name that has been written in the splitter after the process started competing) in slot #2, the process moves to $(i + 1, 1)$. If there is no new name, then the process writes its name into slot #2 and checks slot #1. If there is a new name in slot #1, then the process moves to $(i, j + 1)$. If the process's name is still written in slot #1, then the process has won the splitter and the right to use its value register. It notes this in the register and writes its value.

In order to execute a release, the process simply indicates that the splitter is now clean. Also in [27] we showed that this protocol is time-adaptive to total contention and memory-adaptive to interval contention.

We now transfer this algorithm to Active Disks. To do so we use Active Disks that provide Read-Modify-Write registers. Active Disks however may fail. So to make this algorithm fault-tolerant assuming that at most t disks may fail we simply let each process execute a store on $2t + 1$ active disks. Each process is guaranteed to receive responses from a majority of disks so it suffices to wait for these responses when executing either store or release. □

5 Conclusion and Open Problems

In this paper we showed that there are no uniform, memory-adaptive to interval contention store/release protocols that use only read/write registers. This proves

that to implement protocols that are memory-adaptive to interval contention in this setting we must use stronger primitives such as read-modify-write registers, validating our uniform and memory adaptive to interval contention protocol from [27]. We furthermore show that Active Disks are an ideal environment for the employment of such a protocol. It would be interesting to closer investigate the relationship between time-adaptive and memory-adaptive protocols. What conditions must be met for a memory-adaptive protocol to be also time-adaptive? How about the reverse? Answering these questions will allow us to better understand the true cost of distributed protocols and if and when they can be made adaptive.

References

1. I. Abraham, G. Chockler, I. Keidar and D. Malkhi. Byzantine Disk Paxos: Optimal resilience with byzantine shared memory. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 226-235, 2004.
2. A. Acharya, M. Uysal and J. Saltz. Active Disks: Programming model, algorithms and evaluation. *Proc. of the 8th International Conference on ASPLOS-VIII*, pp. 81-91, 1998.
3. Y. Afek, H. Attiya, A. Fouren, G. Stupp and D. Touitou. Long-Lived Renaming made adaptive. *Proc. of 18th ACM Symp. on PODC*: 91-103, May 1999.
4. Y. Afek, H. Attiya, G. Stupp and D. Touitou. Adaptive long-lived renaming using bounded memory. *Proc. of the 40th IEEE Symp. on FOCS*, pp. 262-272, 1999.
5. Y. Afek, P. Boxer and D. Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. *Proc. 19th ACM Symp. on PODC*: 81-89, 2000.
6. Y. Afek, D. Dauber and D. Touitou. Wait-free made fast. *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*: 538-547, May 1995.
7. Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. of the 18th ACM Symp. on PODC*: 105-112, May 1999.
8. Y. Afek, M. Merritt, G. Taubenfeld and D. Touitou. Disentangling multi-object operations. In *Proc. of 16th ACM Symp. on PODC*: 111-120, 1997.
9. Y. Afek, G. Stupp and D. Touitou. Long-lived adaptive collect with applications. *Proc. of the 40th Ann. Symp. on Foundations of Computer Science*: 262-272, 1999.
10. Y. Afek, G. Stupp and D. Touitou. Long Lived Adaptive Splitter and Applications. *Distributed Computing*, 15(2): 67-86, 2002.
11. Y. Afek, G. Stupp and D. Touitou. Long lived and adaptive atomic snapshot and immediate snapshot. *Proc. of the 19th ACM Symp. on PODC*, pages 71-80, 2000.
12. M. Aguilera, B. Englert and E. Gafni. Uniform Solvability with a finite number of MWMR registers. In *Proc. 17th International Conference DISC*: 16-30, 2003.
13. K. Amiri, G. A. Gibson and R. Golding. Highly Concurrent Shared Storage.In *Proceedings of ICDCS*, pages 298-307, 2000.
14. T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang. Serverless Network File Systems. *ACM Trans. on Comp. Systems* 14(1), pp. 41-79, 1996.
15. J. Anderson and J-H. Yang. Time/contention trade-offs for multiprocessor synchronization. *Information and Computation*, 124(1):68-84, 1996.
16. D. Angluin. Local and global properties in networks of processes. In *Proceedings of the 12th ACM Symposium on Theory of Computing (STOC 1980)*, pages 82-93.
17. J. Aspnes, G. Shah and J. Shah. Wait free consensus with infinite arrivals. *Proc. of the 34th Annual ACM STOC*, pages 524-533, 2002.

18. H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th ACM Symposium on PODC*, pp. 91-100, 2000.
19. H. Attiya, F. Fich and Y. Kaplan. Lower bounds for adaptive collect and related objects. In *Proc. 23rd ACM Symp. on Principles of Distr. Comp.*: 60-70, 2004.
20. H. Attiya and A. Fouren. Algorithms adaptive to point contention. In *J. ACM*, 50(4): 444-468, July 2003.
21. H. Attiya, A. Fouren and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2): 87-96, 2002.
22. H. Attiya, F. Kuhn, M. Wattenhofer and R. Wattenhofer. Efficient Adaptive Collect using Randomization. *Proc. 18th Ann. Conf. on Distr. Comp. (DISC)*, 2004.
23. R. Burns. Data Managment in a distributed file system for Storage Area Networks. PhD Thesis. Department of Computer Science, UC Santa Cruz, 2000.
24. J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation* 107(2):171-184, December 1993.
25. G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. *Proc. of the 21st ACM Symp. on PODC*, pp. 78-87, 2002.
26. M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1), pages 1-17, 1994.
27. B. Englert and D. Goldstein. Can Memory be used adaptively by Uniform Algorithms? *Proc. 9th International Conference on Principles of Distributed Systems (OPODIS)*, pp. 25-35, 2005.
28. F. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3), pp. 121-163, 2003.
29. E. Gafni. A simple algorithmic characterization of uniform solvability. *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pp. 228-237, 2002.
30. E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*,16(1):pp. 1-20, 2003.
31. E. Gafni, M. Merritt and G. Taubenfeld. The concurrency hierarchy and algorithms for unbounded concurrency.In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC 2001)*, pages 161-169, 2001.
32. G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg and J. Zelenka. A cost-effective high-bandwidth storage architecture.In *ACM SIGOPS Operating Systems Review*, pp. 92-103, 1998.
33. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1): 60-87, 1990.
34. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1): 1-11. February 1987.
35. M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45: 137-142, 1993.
36. M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1): pp. 1-39, 1995.
37. National Storage Industry Consortium. HTTP://WWW.NSIC.ORG/NASD
38. E. Riedel, C. Faloutsos, G. A. Gibson and D. Nagle. Active Disks for large scale data processing. *IEEE Computer*, June 2001.
39. Chandramohan Thekkath, Timothy Mann and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224-237, New York, 1997.

Optimistic Algorithms for Partial Database Replication*

Nicolas Schiper[1], Rodrigo Schmidt[1,2], and Fernando Pedone[1]

[1] University of Lugano, Switzerland
[2] EPFL, Switzerland

Abstract. In this paper, we study the problem of partial database replication. Numerous previous works have investigated database replication, however, most of them focus on full replication. We are here interested in genuine partial replication protocols, which require replicas to permanently store only information about data items they replicate. We define two properties to characterize partial replication. The first one, *Quasi-Genuine Partial Replication*, captures the above idea; the second one, *Non-Trivial Certification*, rules out solutions that would abort transactions unnecessarily in an attempt to ensure the first property. We also present two algorithms that extend the Database State Machine [8] to partial replication and guarantee the two aforementioned properties. Our algorithms compare favorably to existing solutions both in terms of number of messages and communication steps.

1 Introduction

Database replication protocols based on group communication have recently received a lot of attention [5,6,8,13]. The main reason for this stems from the fact that group communication primitives offer adequate properties, namely agreement on the messages delivered and on their order, to implement synchronous database replication. Most of the complexity involved in synchronizing database replicas is handled by the group communication layer.

Previous work on group-communication-based database replication has focused mainly on full replication. However, full replication might not always be adequate. First, sites might not have enough disk or memory resources to fully replicate the database. Second, when access locality is observed, full replication is pointless. Third, full replication provides limited scalability since every update transaction should be executed by each replica. In this paper, we extend the Database State Machine (DBSM) [8], a group-communication-based database replication technique, to partial replication. The DBSM is based on the deferred update replication model [1]. Transactions execute locally on one database site and their execution does not cause any interaction with other sites. Read-only transactions commit locally only; update transactions are atomically broadcast to all database sites at commit time for certification. The certification

* The work presented in this paper has been partially funded by the Hasler Foundation, Switzerland (project #1899) and SNSF, Switzerland (project #200021-107824).

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 81–93, 2006.

test ensures *one-copy serializability*: the execution of concurrent transactions on different replicas is equivalent to a serial execution on a single replica [1]. In order to execute the certification test, every database site keeps the *writesets* of committed transactions. The certification of a transaction T consists in checking that T's *readset* does not contain any outdated value, i.e., no committed transaction T' wrote a data item x after T read x.

A straightforward way of extending the DBSM to partial replication consists in executing the same certification test as before but having database sites only process update operations for data items they replicate. But as the certification test requires storing the writesets of all committed transactions, this strategy defeats the whole purpose of partial replication since replicas may store information related to data items they do not replicate. We would like to define a property that captures the legitimacy of a partial replication protocol. Ideally, sites should be involved only in the certification of transactions that read or write data items they replicate. Such a strict property, however, would force the use of an atomic multicast protocol as the group communication primitive to propagate transactions. Since existing multicast protocols are more expensive than broadcast ones [4], this property restricts the performance of the protocol. More generally, we let sites receive and momentarily store transactions unrelated to the data items they replicate as long as this information is shortly erased. Moreover, we want to make sure each transaction is handled by a site at most once. If sites are allowed to completely forget about past transactions, this constraint cannot obviously be satisfied. We capture these two requirements with the following property:

– *Quasi-Genuine Partial Replication:* For every submitted transaction T, correct database sites that do not replicate data items read or written by T permanently store not more than the identifier of T.[1]

Consider now the following modification to the DBSM, allowing it to ensure Quasi-Genuine Partial Replication. Besides atomically broadcasting transactions for certification, database sites periodically broadcast "garbage collection" messages. When a garbage collection message is delivered, a site deletes all the writesets of previously committed transactions. When a transaction is delivered for certification, if the site does not contain the writesets needed for its certification, the transaction is conservatively aborted. Since all sites deliver both transactions and garbage collection messages in the same order, they will all reach the same outcome after executing the certification test. This mechanism, however, may abort transactions that would be committed in the original DBSM. In order to rule out such solutions, we introduce the following property:

– *Non-Trivial Certification:* If there is a time after which no two conflicting transactions are submitted, then eventually transactions are not aborted by certification.

[1] Notice that even though transaction identifiers could theoretically be arbitrarily large, in practice, 4-byte identifiers are enough to uniquely represent 2^{32} transactions.

In this paper we present two algorithms for partial database replication that satisfy Quasi-Genuine Partial Replication and Non-Trivial Certification. Both algorithms make optimistic assumptions to ensure better performance. Our first algorithm is simpler and assumes *spontaneous total order*: with high probability messages sent to all servers in the cluster reach all destinations in the same order, a property usually verified in local-area networks. As a drawback, it processes a single transaction at a time. Our second algorithm is able to certify multiple transactions at a time and, as explained in Section 4, does not assume spontaneous total order.

To the best of our knowledge, [5] and [12] are the only papers addressing partial database replication using group communication primitives. In [5], every read operation is multicast to the sites replicating the data items read; write operations are multicast together with the transaction's commit request. A final atomic commit protocol ensures transaction atomicity. In [12], the authors extend the DBSM for partial replication by adding an extra atomic commit protocol. Each replica uses as its vote for atomic commit the result of the certification test. Both of our algorithms compare favorably to those presented in [5] and [12]: they either have a lower latency or make weaker assumptions about the underlying model, i.e., they do not require perfect failure detection.

2 System Model and Definitions

We consider a system $\Pi = \{s_1, .., s_n\}$ of database sites. Sites communicate through message passing and do not have access to a shared memory or a global clock. We assume the crash-stop failure model. A site that never crashes is *correct*, and a site that is not correct is *faulty*. The system is asynchronous, i.e., message delays and the time necessary to execute a step can be arbitrarily large but are finite. Furthermore, the communication channels do not corrupt or duplicate messages, and are (quasi-)reliable: if a correct site p sends a message m to a correct site q, then q eventually receives m.

Throughout the paper, we assume the existence of a *Reliable Broadcast* primitive. Reliable Broadcast is defined by primitives *R-bcast(m)* and *R-deliver(m)*, and satisfies the following properties [2]: if a correct site R-bcasts a message m, then it eventually R-delivers m (*validity*), (ii) if a correct site R-delivers a message m, then eventually all correct sites R-deliver m (*agreement*) and (iii) for every message m, every site R-delivers m at most once and only if it was previously R-bcast (*uniform integrity*). Reliable Broadcast does not ensure agreement on the message delivery order, that is, two broadcast messages might be delivered in different orders by two different sites. In local-area networks, some implementations of Reliable Broadcast can take advantage of network hardware characteristics to deliver messages in total order with high probability [9]. We call such a primitive Weak Ordering Reliable Broadcast, WOR-Broadcast.

Our algorithms also use a *consensus* abstraction. In the consensus problem, sites propose values and must reach agreement on the value decided. Consensus is defined by the primitives *propose(v)* and *decide(v)*, and satisfies the following

properties: (i) every site decides at most once (*uniform integrity*), (ii) no two sites decide differently (*uniform agreement*), (iii) if a site decides v, then v was proposed by some site (*uniform validity*) and (iv) every correct site eventually decides (*termination*).

A database $\Gamma = \{x_1, .., x_n\}$ is a finite set of data items. Database sites have a partial copy of the database. For each site s_i, $Items(s_i) \subseteq \Gamma$ is defined as the set of data items replicated on s_i. A transaction is a sequence of read and write operations on data items followed by a commit or abort operation. For simplicity, we represent a transaction T as a tuple (id, rs, ws, up), where id is the unique identifier of T, rs is the readset of T, ws is the writeset of T and up contains the updates of T. More precisely, up is a set of tuples (x, v), where, for each data item x in ws, v is the value written to x by T. For every transaction T, $Items(T)$ is defined as the set of data items read or written by T. Two transactions T and T' are said to be conflicting, if there exists a data item $x \in Items(T) \cap Items(T') \cap (T.ws \cup T'.ws)$. We define $Site(T)$ as the site on which T is executed. Furthermore, we assume that for every data item $x \in \Gamma$, there exists a correct site s_i which replicates x, i.e., $x \in Items(s_i)$. Finally, we define $Replicas(T)$ as the set of sites which replicate at least one data item written by T, i.e., $Replicas(T) = \{s_i \mid s_i \in \Pi \wedge Items(s_i) \cap T.ws \neq \emptyset\}$.

3 The Database State Machine Approach

We now present a generalization of the Database State Machine approach. The protocol in [8] is an instance of our generalization in the fully replicated context. For the sake of simplicity, we consider a replication model where a transaction T can only be executed on a site s_i if $Items(T) \subseteq Items(s_i)$. Moreover, to simplify the presentation, we consider a client c that sends requests on behalf of a transaction T to $Site(T)$. In the following, we comment on the states in which a transaction can be in the DBSM.

- *Executing:* Read and write operations are executed locally at $Site(T)$ according to the strict two-phase locking rule (strict 2PL). When c requests to commit T, it is immediately committed and passes to the Committed state if it is a read-only transaction, event which we denote $Committed(T)_{Site(T)}$; if T is an update transaction, it is submitted for certification and passes to the Submitted state at $Site(T)$. We represent this event as $Submitted(T)_{Site(T)}$. In the fully replicated case, to submit T, sites use an atomic broadcast primitive; in a partial replication context, the algorithms of Section 4 are used.
- *Submitted:* When T enters the Submitted state, its read locks are released at $Site(T)$ and T is eventually certified. With full replication, the certification happens when T is delivered; Section 4 explains when this happens in a partially replicated scenario. Certification ensures that if a committed transaction T' executed concurrently with T, and T read a data i tem written by T' then T is aborted. T' is concurrent with T if it committed at $Site(T)$ *after* T entered the Submitted state at $Site(T)$. Therefore, T passes the

certification test on site s_i if for every T' already committed at s_i the following condition holds:

$$\begin{array}{c} Committed(T')_{Site(T)} \rightarrow Submitted(T)_{Site(T)} \\ \vee \\ T'.ws \cap T.rs = \emptyset, \end{array} \quad (1)$$

where $\rightarrow$ is Lamport's happened before relation on events [7].
In the fully replicated DBSM, transactions are certified locally by each site upon delivery. In the partially replicated DBSM, to ensure *Quasi-Genuine Partial Replication*, sites only store the writesets of committed transactions that wrote data items they replicate. Therefore, sites might not have enough information to decide on the outcome of all transactions. Hence, to satisfy *Non-trivial Certification*, we introduce a voting phase where each site sends the result of its certification test to the other sites. Site s_i can safely decide to commit or abort T when it has received votes from a *voting quorum* for T. Intuitively, a voting quorum VQ for T is a set of databases such that for each data item read by T, there is at least one database in VQ replicating this item. More formally, a quorum of sites is a voting quorum for T if it belongs to $VQS(T)$, defined as follows:

$$VQS(T) = \{VQ | VQ \subseteq \Pi \wedge T.rs \subseteq \bigcup_{s \in VQ} Items(s)\} \quad (2)$$

For T to commit, every site in a voting quorum for T has to vote *yes*. If a site in the quorum votes *no*, it means that T read an old value and should be aborted; committing T would make the execution non-serializable. Notice that $Site(T)$ is a voting quorum for T by itself, since for every transaction T, $Items(T) \subseteq Items(Site(T))$. If T passes the certification test at s_i, it requests the write locks for the data items it has updated. If there exists a transaction T' on s_i that holds conflicting locks with T's write locks, the action taken depends on T''s state on s_i and on T''s type, read-only or update:

1. *Executing:* If T' is in execution on s_i then one of two things will happen: if T' is a read-only transaction, T waits for T' to terminate; if T' is an update transaction, it is aborted.
2. *Submitted:* This happens if T' executed on s_i, already requested commit but was not committed yet. In this case, T's updates should be applied to the database before T''s. How this is ensured is implementation specific.[2]

Once the locks are granted, T applies its updates to the database and passes to the Committed state. If T fails the certification test, it passes to the Aborted state.

– *Committed/Aborted:* These are final states.

[2] For example, a very simple solution would be for s_i to abort T'; if T' later passes certification, its writes would be re-executed. The price paid for simplicity here is the double execution of T''s write operations.

4 Partially-Replicated DBSM

In this section, we present two algorithms for the termination protocol of the DBSM in a partial replication context. These protocols ensure both one-copy serializability [1] and the following liveness property: if a correct site submits a transaction T, then either $Site(T)$ aborts T or eventually all correct sites in $Replicas(T)$ commit T. The algorithms also satisfy Quasi-Genuine Partial Replication and Non-Trivial Certification. The proof of correctness can be found in [11].

4.1 The "One-at-a-time" Algorithm

Sites execute a sequence of *steps*. In each step, sites decide on the outcome of one transaction. A step is composed of two phases, a consensus phase and a voting phase. Consensus is used to guarantee that sites agree on the commit order of transactions. In the voting phase, sites exchange the result of their certification test to ensure that the commit of a transaction T in step K induces a serializable execution.

The naive way to implement the termination protocol is to first use consensus to determine the next transaction T in the serial order and then execute the voting phase for T. We take a different approach: Based on the observation that with a high probability messages broadcast in a local-area network are received in total order [9], we overlap the consensus phase with the voting phase to save one communication step. If sites receive the transaction to be certified in the same order, they vote for the transaction before proposing it to consensus. Luckily, by the time consensus decides on a transaction T, every site will already have received the votes for T and will be able to decide on the outcome of T.

Algorithm 1 is composed of three concurrent tasks. Each line of the algorithm is executed atomically. The state transitions of transactions are specified in the right margin of lines 10, 28, and 30. Notice that the state transition happens after the corresponding line has been executed. Every transaction T is a tuple $(id, site, rs, ws, up, past, order)$. We added three fields to the definition of a transaction (c.f. Section 2), namely *site*, *past*, and *order*: *site* is the database site on which T is executed; *past* is the order of T's submission; and *order* is T's commit order. The algorithm also uses five global variables: K stores the *step* number; *UNDECIDED* and *DECIDED* are (ordered) sequences of, respectively, pending transactions and transactions for which the outcome is known; *COMMITTED* is the set of committed transactions; and the set *VOTES* stores the votes received, i.e., the results of the certification test. We use the operators $\oplus$ and $\ominus$ for the concatenation and decomposition of sequences. Let seq_1 and seq_2 be two sequences of transactions. Then, $seq_1 \oplus seq_2$ is the sequence of transactions in seq_1 followed by all the transactions in seq_2, and $seq_1 \ominus seq_2$ is the sequence of transactions in seq_1 that are not in seq_2. Transactions are matched using their identifiers.

To take advantage of spontaneous total order, database sites use the WOR-Broadcast primitive to submit transactions (line 10). When no consensus instance is running and *UNDECIDED* is not empty, sites first execute the *Vote*

Algorithm 1. The "One-at-a-time" algorithm - Code of database site s

```
1: Initialization
2:    K ← 1, UNDECIDED ← ϵ, DECIDED ← ϵ,COMMITTED ← ∅, VOTES ← ∅

3: function Certify(T)
4:    return ∀(id, order, ws) ∈ COMMITTED : order < T.past ∨ ws ∩ T.rs = ∅

5: procedure Vote(T)
6:    if T.rs ∩ Items(s) ≠ ∅ then
7:       send(VOTE, T.id, K, Certify(T)) to all q in Replicas(T)

8: To submit transaction T                                          {Task 1}
9:    T.past ← K
10:   WOR-Broadcast(VOTE_REQ, T)                                   {Executing → Submitted}

11: When receive(VOTE, T.id, K', vote) from q                       {Task 2}
12:   VOTES ← VOTES ∪ (T.id, q, K', vote)

13: When WOR-Deliver(VOTE_REQ, T) ∧ T.id ∉ DECIDED                  {Task 3}
14:   UNDECIDED ← UNDECIDED ⊕ T

15: When UNDECIDED ≠ ϵ
16:   T ← head(UNDECIDED)
17:   Vote(T)
18:   Propose(K, T)
19:   wait until Decide(K, T')
20:   if T'.id ≠ T.id then Vote(T')
21:   UNDECIDED ← UNDECIDED ⊖ T'
22:   DECIDED ← DECIDED ⊕ T'.id
23:   if T'.ws ∩ Items(s) ≠ ∅ then
24:      wait until ∃VQ ∈ VQS(T') : ∀q ∈ VQ : (T'.id, q, K, −) ∈ VOTES
25:      if ∀q ∈ VQ : (T'.id, q, K, yes) ∈ VOTES then
26:         T'.order ← K
27:         COMMITTED ← COMMITTED ∪ (T'.id, T'.order, T'.ws ∩ Items(s))
28:         commit T'                                              {Submitted → Committed}
29:      else
30:         if s = T'.site then abort T'                           {Submitted → Aborted}
31:   K ← K + 1
32:   VOTES ← {(tid, q, K', v) ∈ VOTES | K' ≥ K}
```

procedure for T at the head of *UNDECIDED* (line 17) and then propose T (line 18). In the *Vote* procedure, T is certified and the result of the certification is sent in a message of type *VOTE*.

Notice that even though $Site(T)$ is a voting quorum for T by itself ($Items(T) \subseteq Items(Site(T))$), in the algorithm, all sites replicating a data item read by T vote. This is done to tolerate the crash of $Site(T)$. If only $Site(T)$ voted, the following undesirable scenario could happen: $Site(T)$ submits T and crashes just after executing line 10. Databases *WOR-Deliver* T, propose T and decide on T. In this execution, sites would wait forever at line 24, as $Site(T)$ crashed before voting for T.

Two further remarks concern the *Vote* procedure. First, to be able to certify transactions, we need to implement the precedence relation $\rightarrow$ between events. For two transactions T and T', this is done by comparing the value of their *past* and *order* fields. If $T.order < T'.past$, we are sure that T committed before T' was submitted, because K is incremented after transactions commit. Second,

notice that *VOTE* messages contain the *step number* K in which T was certified. This information is necessary because a transaction can be certified in different steps and the result of the certification test in steps K and K' might be different. This is precisely why sites wait for *VOTE* messages coming from step number K at line 24. Moreover, even if sites receive votes from different voting quorums, they will agree on the outcome of the transaction. Intuitively, this holds because we only take into account *voting quorums* that voted in *step* K, therefore they consider the same sequence of committed transactions. Finally, by verifying that transactions T and T' are the same at line 20, sites check if the spontaneous total order holds. If it is not the case, sites need to vote for the transaction decided by consensus.

4.2 The "Many-at-a-time" Algorithm

The previous algorithm certifies transactions sequentially. Thus, if many transactions are submitted, an ever-growing chain of uncommitted transactions can be formed. Algorithm 2 solves that problem by allowing a sequence of transactions to be proposed in consensus instances and by changing the certification test accordingly.

Algorithm 2 follows the same structure and uses the same global variables as Algorithm 1. The difference lies in Task 3 and the auxiliary procedures used. In the general case, when sites notice that there is a sequence of pending transactions that have not been committed or aborted ("$UNDECIDED \neq \epsilon$" at line 25), this sequence is voted for and proposed in consensus instance K (lines 26–27). In the *Vote* procedure, every pending transaction is certified considering only the previously committed transactions (lines 3–9). The results are gathered in a set and later sent to all sites that have data items updated by some transaction in the pending sequence (lines 10–12). The "$VOTES \neq \emptyset$" condition at line 25 is there for garbage collection purposes: it forces the proposal of empty sequences in case there are votes for undelivered vote requests (a possible situation due to failures that would violate *Quasi-Genuine Partial Replication*).

After the K-th instance of consensus has decided on a sequence *SEQ* of transactions (line 28), sites verify whether they have voted for all transactions in *SEQ*; if it is not the case, they vote for the sequence *SEQ* (lines 29–30). Then, sites replicating data items updated by one of the transactions in *SEQ* sequentially certify all transactions in *SEQ* following their order (lines 35–45). The certification of transaction T is divided into two parts. First, T is certified considering the transactions committed in steps lower than K by taking into account the votes of a voting quorum (line 37). Second, sites certify T considering committed transactions that have been decided in the same consensus instance (line 38). This is done by gathering committed transactions in a set called *LCOMMIT* and by verifying that there does not exist a transaction T' in this set that writes a data item read by T. If T passes both certifications and updates a data item in $Items(s)$, it is treated in exactly the same way as certified transactions in Algorithm 1 (lines 41–43).

Algorithm 2. The "Many-at-a-time" algorithm - Code of database site s

```
1:  Initialization
2:    K ← 1, UNDECIDED ← ϵ, DECIDED ← ϵ, COMMITTED ← ∅, VOTES ← ∅

3:  function Certify(SEQ)
4:    V ← ∅
5:    for all T ∈ SEQ do
6:      if ∀(id, order, ws) ∈ COMMITTED : order < T.past ∨ ws ∩ T.rs = ∅ then
7:        V ← V ∪ (T.id, yes)
8:      else V ← V ∪ (T.id, no)
9:    return V

10: procedure Vote(SEQ)
11:   if ∃T ∈ SEQ : T.rs ∩ Items(s) ≠ ∅ then
12:     send (Vote, Strip(SEQ), K,Certify(SEQ)) to {q | ∃T ∈ SEQ : q ∈ Replicas(T)}

13: function Strip(SEQ)
14:   RESULT ← ϵ
15:   for all T ∈ SEQ in order do
16:     RESULT ← RESULT ⊕ T.id
17:   return RESULT

18: To submit transaction T                                              {Task 1}
19:   T.past ← K
20:   R-bcast (Vote_Req, T)                                  {Executing → Submitted}

21: When receive (Vote, IDSEQ, K′, V) from q                             {Task 2}
22:   VOTES ← VOTES ∪ (IDSEQ, q, K′, V)

23: When R-deliver (Vote_Req, T) ∧ T.id ∉ DECIDED                        {Task 3}
24:   UNDECIDED ← UNDECIDED ⊕ T

25: When UNDECIDED ≠ ϵ ∨ VOTES ≠ ∅
26:   Vote(UNDECIDED)
27:   Propose(K, UNDECIDED)
28:   wait until Decide(K,SEQ)
29:   if ∃T : T ∈ SEQ ∧ T ∉ UNDECIDED then
30:     Vote(SEQ)
31:   DECIDED ← DECIDED ⊕ Strip(SEQ)
32:   UNDECIDED ← UNDECIDED ⊖ SEQ
33:   if ∃T ∈ SEQ : T.ws ∩ Items(s) ≠ ∅ then
34:     LCOMMIT ← ∅
35:     for all T ∈ SEQ in order do
36:       wait until
              ∃VQ ∈ VQS(T) : ∀q ∈ VQ : ∃(SEQ_q, q, K, V_q) ∈ VOTES : T ∈ SEQ_q
37:       if (∀q ∈ VQ : ∃(SEQ_q, q, K, V_q) ∈ VOTES : T ∈ SEQ_q ∧ (T.id, yes) ∈ V_q)
38:           ∧ (∄T′ ∈ LCOMMIT : T′.ws ∩ T.rs ≠ ∅) then
39:         LCOMMIT ← LCOMMIT ∪ {T}
40:         if T.ws ∩ Items(s) ≠ ∅ then
41:           T.order ← K
42:           COMMITTED ← COMMITTED ∪ (T.id, T.order, T.ws ∩ Items(s))
43:           commit T                                   {Submitted → Committed}
44:       else
45:         if s = T.site then abort T                   {Submitted → Aborted}
46:   K ← K + 1
47:   VOTES ← {(tid, q, K′, v) ∈ VOTES | K′ ≥ K}
```

Differently from Algorithm 1, Algorithm 2 does not rely on spontaneous total order. This is because sequences of transactions are used when voting and proposing values to a consensus instance, and the order of transactions in this sequence does not matter when it comes to voting. Recall that the vote phase in step K consists in independently certifying undecided transactions against

transactions committed in previous steps (line 11 and function *Certify* at lines 3–9). This phase does not take into consideration conflicts within the sequence itself since they are solved after the consensus instance is decided. Nevertheless, votes are still optimistic in Algorithm 2 as they are sent before the consensus instance has decided on its outcome.

The optimistic assumption that allows a transaction T to be certified as soon as consensus instance K decides on a sequence containing T is that every member of at least one correct voting quorum VQ for T has voted for any sequence containing T before consensus instance K (line 26). Notice that the sequences considered by different members of VQ do not have to be the same, the only requirement is that they all contain T.

We could further relax the optimistic assumptions required at the price of having a higher number of VOTE messages. In the way both algorithms are described, sites vote for a transaction only before it is proposed to the next consensus instance (line 17 of Algorithm 1, line 26 of Algorithm 2). Consider a scenario where the vote request for a transaction T is delivered by a site s right after s has proposed transaction(s) to consensus. Site s will therefore have to wait until the instance finishes to send its vote concerning T. However, T's vote request might have been delivered earlier by some other site and might even have been proposed to the current instance of consensus. If that is the case, and T is part of the consensus decision, the optimistic assumptions will not hold and the protocols might need an extra message step to certify T. This problem can be avoided if sites are allowed to vote while solving a consensus instance. In our example scenario, site s would vote for T even though it has already voted for its consensus proposal. Both votes would then be received by other sites and they would be used to decide on the outcome of T. This optimization relieves the need for spontaneous total order in Algorithm 1 and relaxes even more the optimistic assumption of Algorithm 2. As a secondary effect, it reduces the average latency of transaction certification since votes are sent right after the vote request is received.

5 Related Work and Final Remarks

In this section we compare our algorithms to the related work and conclude the paper. We focus here on the related works satisfying *Quasi-Genuine Partial Replication*.

In [5] the authors propose a database replication protocol based on group multicast. Every read operation on data item x is multicast to the group replicating x; writes are multicast along with the commit request. The delivered operations are executed on the replicas using strict two-phase locking and results are sent back to the client. A final atomic commit protocol ensures transaction atomicity. In the atomic commit protocol, every group replicating a data item read or written by a transaction T sends its vote to a *coordinator* group, which collects the votes and sends the result back to all participating groups. The protocol ensures *Quasi-Genuine Partial Replication* because a transaction operation on

data item x is only multicast to the group replicating x and the atomic commit protocol is executed among groups replicating data item read or written by the transaction. In [12] the authors extend the DBSM to partial replication. They use an optimistic atomic broadcast primitive and a variation of atomic commit, called resilient atomic commit. In contrast to atomic commit, resilient atomic commit may decide to commit a transaction even though some participants crash. When a transaction T is optimistically delivered, replicas certify T and execute a resilient atomic commit protocol using the result of the certification test as their vote. If the optimistic order of T corresponds to the final order, the protocol ends; otherwise when the final order is known, T is certified again and a second resilient atomic commit protocol is executed. The protocol ensures *Quasi-Genuine Partial Replication*, since only sites replicating data item written by T keep T in their committed transaction sequence.

We now compare the cost of the protocols in [5,12] with the two algorithms presented in this paper. We compare the number of communication steps and the number of messages exchanged during the execution of a transaction T. To simplify the analysis, we assume that all messages have a delay of δ. We consider two cases, one where the algorithms' respective optimistic assumptions hold and one where they do not (c.f. Section 4). In both cases, we consider the best achievable latency and the minimum number of messages exchanged, when neither failures nor failure suspicions occur, the most frequent case in practical settings. We first present in Figure 1 the cost of known algorithms used by the protocols compared in this section. Variable k is the total number of participants in the protocol.

Problem	steps	unicast msgs.	broadcast msgs.
Non-Uniform R. Broadcast (RBcast) [2]	1	$k(k-1)+1$	k
Uniform Consensus (Consensus) [10]	2	$2k(k-1)$	$2k$
Non-Blocking A. Commit (NBAC) [3] [3]	2	$2k(k-1)$	$2k$
Uniform A. Broadcast (ABcast) [2]	3	$3k(k-1)+1$	$3k$
Uniform A. Multicast (AMcast) [4]	4	$4k(k-1)+1$	$4k$

Fig. 1. Cost of different agreement problems

Figure 2 presents the cost of the different algorithms. To compute the cost of the execution of T, we consider that T consists of a read and a write operation on the same data item x. For all the protocols, we consider that d database sites replicate data item x and that n is the total number of database sites in the system.

In [5], one multicast is used to read x, d messages are sent containing the result of the read, one multicast is used to send the write along with the commit request and a final atomic commit protocol among d participants is executed. Notice

[3] This cost corresponds to the case where all participants spontaneously start the protocol. This assumption makes sense here because in [5] participants deliver a transaction's commit request before starting the atomic commit protocol.

Algorithm	steps	unicast msgs.	broadcast msgs.
[5]	11	$10d^2 - 9d + 2$	$11d$
[12]	3	$3n(n-1) + d(d-1) + 1$	$3n + d$
Algorithms 1 & 2	3	$3n(n-1) + d(d-1) + 1$	$3n + d$

(a) Optimistic assumption holds

Algorithm	steps	unicast msgs.	broadcast msgs.
[5]	11	$10d^2 - 9d + 2$	$11d$
[12]	4	$3n(n-1) + 2d(d-1) + 1$	$3n + 2d$
Algorithms 1 & 2	4	$3n(n-1) + 2d(d-1) + 1$	$3n + 2d$

(b) Optimistic assumption does not hold

Fig. 2. Comparison of the database replication protocols

that none of the optimistic assumptions assumed by the algorithms in this paper influence the cost of this protocol. In [12], the transaction is atomically broadcast and one communication step later it is optimistically delivered. A resilient atomic commit protocol is then executed among the d database sites. Resilient atomic commit is implemented in one communication step, in which all participants exchange their votes. To guarantee agreement on the outcome of a transaction, the implementation requires perfect failure detection, an assumption that we do not need in this paper. In the best-case scenario, i.e., spontaneous total order holds, the number of communication steps is equal to $max(2, steps(ABcast))$. If the optimistic order is not the final order of the transaction, another resilient atomic commit protocol is needed and therefore the number of communication steps becomes $steps(ABcast) + 1$.

For Algorithms 1 and 2, the cost is computed as follows. In the best-case scenario, the number of communication steps is equal to $steps(RBcast) + max(steps(Consensus), steps(vote\ phase))$, where *vote phase* corresponds to d broadcast messages. If the algorithms' respective optimistic assumptions do not hold, after deciding on T in consensus, another *vote phase* has to take place and therefore the number of communication steps becomes $steps(RBcast)+steps(Consensus)+steps(vote\ phase)$. For simplicity, we assume that in this second *vote phase*, all participants vote, generating an extra $d(d-1)$ messages.

Considering latency, Algorithms 1, 2, and [12] give the best results. However, to achieve such latency, [12] uses perfect failure detection. In terms of the number of messages generated, [5] is cheaper than the DBSM-based solutions if d is much smaller than x. Nonetheless, this protocol has a serious drawback: its number of communication steps highly depends on the number of read operations, as every read operation adds 5 message steps (4 for the multicast and 1 to send back the result). As a final remark, notice that in this analysis we consider the execution of only one transaction. The cost of the protocols might however change if we considered multiple transactions. In this scenario, the following observations can be made. First, even though Algorithms 1 and 2 have equal costs in Figure 2, the overhead might be higher for Algorithm 1 when multiple transactions are submitted. This stems from the fact that in Algorithm 2, the cost of running

consensus might be shared among a set of transactions, therefore reducing the number of generated messages. Second, in [5,12], each transaction requires a separate instance of atomic commit to decide on its outcome. In Algorithm 2, however, at most two voting phases are needed to decide on the outcome of the sequence of transactions decided in the same consensus instance. Therefore, the longer this sequence, the cheaper Algorithm 2 will be compared to [5,12].

References

1. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
2. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
3. J. Gray and L. Lamport. Consensus on transaction commit. Technical Report MSR-TR-2003-96, Microsoft Research, 2004.
4. R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS)*, pages 578–585, Baltimore, USA, May 1997.
5. Udo Fritzke Jr. and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 284–291, 2001.
6. B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 156–163, 1998.
7. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
8. F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98, 2003.
9. F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference (EDCC)*, October 2002.
10. André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.
11. N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. Technical Report 2006, University of Lugano, 2006.
12. A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Proceedings of the 1st International Symposium on Network Computing and Applications (NCA)*, October 2001.
13. I. Stanoi, D. Agrawal, and A. E. Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 148–155, 1998.

Optimal Clock Synchronization Revisited: Upper and Lower Bounds in Real-Time Systems

Heinrich Moser* and Ulrich Schmid

Technische Universität Wien
Embedded Computing Systems Group (E182/2)
A-1040 Vienna, Austria
{moser, s}@ecs.tuwien.ac.at

Abstract. This paper[1] introduces a simple real-time distributed computing model for message-passing systems, which reconciles the distributed computing and the real-time systems perspective: By just replacing instantaneous computing steps with computing steps of non-zero duration, we obtain a model that both facilitates real-time scheduling analysis and retains compatibility with classic distributed computing analysis techniques and results. As a by-product, it also allows us to investigate whether/which properties of real systems are inaccurately or even wrongly captured when resorting to zero step-time models. We revisit the well-studied problem of deterministic internal clock synchronization for this purpose, and show that, contrary to the classic model, no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model. We prove that optimal precision is only achievable with algorithms that take $\Omega(n)$ time in our model, and establish several additional lower bounds and algorithms.

1 Motivation

Executions of distributed algorithms are typically modeled as sequences of atomic computing steps that are executed in zero time. With this assumption, it does not make a difference, for example, whether messages arrive at a processor simultaneously or nicely staggered in time: The messages are processed instantaneously when they arrive. The zero step-time abstraction is hence very convenient for analysis, and a wealth of distributed algorithms, impossibility results and lower bounds have been developed for models that employ this assumption [1].

In real systems, however, computing steps are neither instantaneous nor arbitrarily preemptable: A computing step triggered by a message arriving in the middle of the execution of some other computing step is usually delayed until the current computation is finished. This results in queuing phenomenons, which depend not only on the actual message arrival pattern but also on the queuing/scheduling discipline employed. The real-time systems community has

* Corresponding author.

[1] This work is part of our project *Theta*, supported by the Austrian Science Foundation (FWF) under grant P17757 (*http://www.ecs.tuwien.ac.at/projects/Theta*).

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 94–109, 2006.

established powerful techniques for analyzing such effects [2], such that the resulting worst-case response times and end-to-end delays can be computed.

This paper introduces a real-time distributed computing model for message-passing systems, which reconciles the distributed computing and the real-time systems perspective: By just replacing the zero step-time assumption with non-zero step times, we obtain a real-time distributed computing model that admits real-time analysis without invalidating standard distributed computing analysis techniques and results.

Apart from making distributed algorithms amenable to real-time analysis, our model also allows to address the interesting question whether/which properties of real systems are inaccurately or even wrongly captured when resorting to classic zero step-time models. In this paper, we revisit the well-studied problem of deterministic internal clock synchronization [3,4] for this purpose. Clock synchronization is a particularly suitable choice here, since the achievable synchronization precision is known to depend on the end-to-end delay uncertainty (i.e., the difference between maximum and minimum end-to-end delay). Since non-zero computing step times are likely to affect end-to-end delays, one may expect that some results obtained under the classic model do not hold under the real-time model. Our analysis confirms that this is indeed the case: We show that no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model. Since such an algorithm has been given for the classic model [4], we have found an instance of a problem where the standard distributed computing analysis gives too optimistic results. Actually, we show that optimal precision is only achievable with algorithms that take $\Omega(n)$ time, even if they are provided with a constant-time broadcast primitive.

Lacking space does not allow us to present all our results and derivations here, which can be found in [5].

2 Classic Computing Model

In clock synchronization research [6,7,8,9,4], system models are considered where the uncertainty comes from varying message delays, failures, and drifting clocks. Denoted "Partially Synchronous Reliable/Unreliable Models" in [3], such models are nowadays called (non-lockstep) synchronous models in literature. In order to solely investigate the effects of non-zero step-times, our real-time computing model will be based on the simple failure- and drift-free synchronous model introduced in [4]. Here it will be referred to as the *classic computing model*.

We consider a network of n failure-free *processors*, which communicate by passing unique messages. Each processor p is equipped with a CPU, some local memory, a hardware clock $HC_p(t)$ running at the same rate as real time, and reliable, non-FIFO links to all other processors.

The CPU is running an algorithm, specified as a mapping from processor indices to a set of initial states and a transition function. The *transition function* takes the processor index p, one incoming message, receiver processor current local state and hardware clock reading as input, and yields a list of *states* and *messages to be sent*,

e.g. $[oldstate, int.st._1, int.st._2, \text{msg. } m \text{ to } q, \text{msg. } m' \text{ to } q', int.st._3, newstate]$, as output. A "message to be sent" is specified as a pair consisting of the message itself and the destination processor the message will be sent to. The intermediate states are usually neglected in the classic computing model, as the state transition from *oldstate* to *newstate* is instantaneous. We explicitly model these states to retain compatibility with our real-time computing model, where they will become important.

Every message reception immediately causes the receiver processor to change its state and send out all messages according to the transition function. Such a *computing step* will be called an *action* in the following. The complete action (message arrival, processing and sending messages) is performed in zero time.

Actions can actually be triggered by ordinary messages and timer messages: Ordinary messages are transmitted over the links. The *message delay*[2] $\underline{\delta}$ is the difference between the real time of the action sending the message and the real time of the action receiving the message. There is a lower bound $\underline{\delta}^-$ and an upper bound $\underline{\delta}^+$ on the message delay of every ordinary message.

Timer messages are used for modeling time(r)-driven execution in our message-driven setting: A processor setting a timer is modeled as sending a timer message (to itself) in an action, and timer expiration is represented by the reception of a timer message. Note that timer messages do not need to obey the message delay bounds, since they are received when the hardware clock reaches (or has already reached) the time specified in the timer message.

An execution in the classic computing model is a sequence of actions. An action ac occurring at real-time t at processor p is a 5-tuple, consisting of the processor index $proc(ac) = p$, the received message $msg(ac)$, the occurrence real-time $time(ac) = t$, the hardware clock value $HC(ac) = HC_p(t)$ and the state transition sequence $trans(ac) = [oldstate, \ldots, newstate]$ (including messages to be sent). A valid execution must satisfy obvious properties such as conformance with the transition function, timer messages arriving on time, and reliable message transmission. To trigger the first action of a processor in an execution, we allow one special *init message* to arrive at each processor from outside the system.

A *classic system* $\underline{s}$ is a system adhering to the classic computing model defined in this section, parameterized by the system size n and the interval $[\underline{\delta}^-, \underline{\delta}^+]$ specifying the bounds on the message delay.

Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system. An execution is $\underline{s}$-*admissible*, if the execution comprises n processors and the message delay for each ordinary message stays within $[\underline{\delta}^-, \underline{\delta}^+]$.

3 Real-Time Computing Model

Zero step-time computing models have good coverage in systems where message delays are much higher than message processing times. There are applications like

[2] To disambiguate our notation, systems, parameters like message delay bounds, and algorithms in the classic computing model are represented by underlined variables (usually $\underline{s}, \underline{\delta}^-, \underline{\delta}^+, \underline{\mathcal{A}}$).

high speed networks, however, where this is not the case. Additionally, and more importantly, the zero step-time assumption inevitably ignores message queuing at the receiver: It is possible, even in case of large message delays, that multiple messages arrive at a single receiver at the same time. This causes the processing of some of these messages to be delayed until the execution of their predecessors has been completed. Common practice so far is to take this queuing delay into account by increasing the upper bound $\underline{\delta}^+$ on the message delay. This approach, however, has two disadvantages: First, a-priori information about the algorithm's message pattern is needed to determine a parameter of the system model, which creates cyclic dependencies. Second, in lower bound proofs, the adversary can choose an arbitrary message delay within $[\underline{\delta}^-, \underline{\delta}^+]$ – even if this choice is not in accordance, i.e., not possible, with the current message arrival pattern. This could lead to overly pessimistic lower bounds.

3.1 Real-Time System Model

The system model in our real-time computing model is the same as in the classic computing model, except for the following change: A computing step in a real-time system is executed non-preemptively within a system-wide lower bound μ^- and upper bound μ^+. Note that we allow the processing time and hence the bounds $[\mu^-, \mu^+]$ to depend on the number of messages sent in a computing step. In order to clearly distinguish a computing step in the real-time computing model from a zero-time action in the classic model, we will use the term *job* to refer to the former. Interestingly, this simple extension has far-reaching implications, which make the real-time computing model more realistic but also more complex. In particular, queuing and scheduling effects must be taken into account:

- We must now distinguish two modes of a processor at any point in real-time t: *idle* and *busy*. Since computing steps cannot be interrupted, a *queue* is needed to store messages arriving while the processor is busy.
- When and in which order messages collected in the queue are processed is specified by some *scheduling policy*, which is, in general, independent of the algorithm. Formally, a scheduling policy is specified as an arbitrary mapping from the current queue state (= a sequence of messages), the hardware clock reading, and the current local processor state onto a single message from that message sequence. The scheduling policy is used to select a new message from the queue whenever processing of a job has been completed. To ensure liveness, we assume that the scheduling policy is *non-idling*.
- The delay of a message is measured from the real time of the *start of the job* sending the message to the arrival real time at the destination processor (where the message will be enqueued or, if the processor is idle, immediately causes the corresponding job to start). Analogous to the classic computing model, message delays of ordinary messages must be within a system-wide lower bound δ^- and an upper bound δ^+. Like the processing delay, the message delay and hence the bounds $[\delta^-, \delta^+]$ may depend on the number of messages sent in the sending job.

- We assume that the hardware clock can only be read at the beginning of a job. This restriction in conjunction with our definition of message delays will allow us to define transition functions in exactly the same way as in the classic computing model. After all, the transition function just defines the "logical" semantics of a transition, but not its timing.
- Contrary to the classic computing model, the state transitions $oldstate \rightarrow \cdots \rightarrow newstate$ in a single computing step need not happen at the same time: Typically, they occur at different times during the job, allowing an intermediate state to be valid on a processor for some non-zero duration.

Figure 1 depicts an example of a single job at the sender processor p, which sends one message m to receiver q currently busy with processing another message. Part (a) shows the major timing-related parameters in the real-time computing model, namely, *message delay* (δ), *queuing delay* (ω), *end-to-end delay* ($\Delta = \delta + \omega$), and *processing delay* (μ) for the message m represented by the dotted arrow. The bounds on the message delay δ and the processing delay μ are part of the system model, although they need not be known to the algorithm. Bounds on the queuing delay ω and the end-to-end delay Δ, however, are *not* parameters of the system model—in sharp contrast to the classic computing model (see Sect. 2), where the end-to-end delay always equals the message delay. Rather, those bounds (if they exist) must be derived from the system parameters (n, $[\delta^-, \delta^+]$, $[\mu^-, \mu^+]$) and the message pattern of the algorithm, by performing a real-time scheduling analysis. Part (b) of Fig. 1 shows the detailed relation between message arrival (enqueuing) and actual message processing.

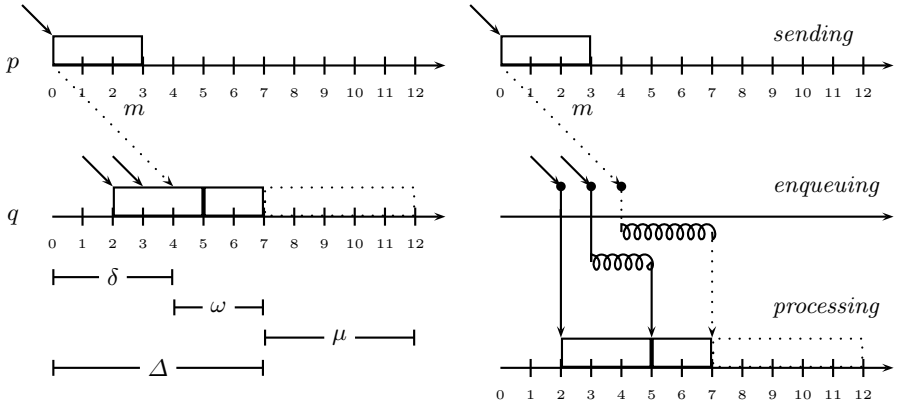
3.2 Real-Time Runs

This section defines a *real-time run* (*rt-run*), corresponding to an execution in the classic computing model. A *rt-run* is a sequence of receive events and jobs.

A *receive event* R for a message arriving at p at real-time t is a triple consisting of the processor index $proc(R) = p$, the message $msg(R)$, and the arrival real-time $time(R) = t$. Recall that t is the enqueuing time in Fig. 1(b).

A *job* J starting at real-time t on p is a 6-tuple, consisting of the processor index $proc(J) = p$, the message being processed $msg(J)$, the start time $begin(J) = t$, the job processing time $d(J)$, the hardware clock reading $HC(J) = HC_p(t)$, and the state transition sequence $trans(J) = [oldstate, \ldots, newstate]$.

Figure 1 provides an example of a rt-run, containing three receive events and three jobs on the second processor. For example, the dotted job on the second processor q consists of $(q, m, 7, 5, HC_q(7), [oldstate, \ldots, newstate])$, with m being the message received during the receive event $(q, m, 4)$. Neither the actual state transition times nor the actual sending times of the sent messages are recorded in a job. Measuring all message delays from the beginning of a job and knowing that the state transitions and the message sends occur in the listed order at arbitrary times during the job is sufficient for proving that a rt-run satisfies a given set of properties, as well as for performing time complexity analysis.



(a) Timing parameters for some msg. m (b) Relation of msg. arrival and msg. processing, with enqueuing shown explicitly

Fig. 1. Real-time computing model

In addition to the properties required for valid executions, a valid rt-run must also satisfy additional consistency constraints: Jobs on the same processor must not overlap, and the execution must conform to some arbitrary non-idling scheduling policy.

3.3 Systems and Admissible Real-Time Runs

A real-time system s is defined by an integer n and two intervals $[\delta^-, \delta^+]$ and $[\mu^-, \mu^+]$. Considering δ^-, δ^+, μ^- and μ^+ to be constants would give an unfair advantage to broadcast-based algorithms when comparing algorithms' time complexity: Computation steps would take between μ^- and μ^+ time units, independently of the number of messages sent. This makes it impossible to derive a meaningful time complexity lower bound for systems in which a constant-time broadcast primitive is not available. Corollary 15 will show an example.

Therefore, the interval boundaries δ^-, δ^+, μ^- and μ^+ can be either constants or non-decreasing functions $\{0, \ldots, n-1\} \to \mathbb{R}^+$, representing a mapping from the number of destination processors to which ordinary messages are sent during that computing step to the actual message or processing delay bound.[3]

Example: During some job, messages to exactly three processors are sent. The duration of this job lies within $[\mu^-_{(3)}, \mu^+_{(3)}]$. Each of these messages has a message delay between $\delta^-_{(3)}$ and $\delta^+_{(3)}$. The delays of the three messages need not be the same.

Sending ℓ messages at once must not be more costly than sending those messages in multiple steps. In addition, we assume that the message delay

[3] As message size is not bounded, we can make the simplifying assumption that at most one message is sent to every other processor during each job. $\delta^-_{(0)}$ and $\delta^+_{(0)}$ are assumed to be 0 because this allows some formulas to be written in a more concise form.

uncertainty $\varepsilon_{(\ell)} := \delta^+_{(\ell)} - \delta^-_{(\ell)}$ is also non-decreasing and, therefore, $\varepsilon_{(1)}$ is the minimum uncertainty. This assumption is reasonable, as usually sending more messages increases the uncertainty rather than lowering it.

Definition 1. *Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system. A rt-run is* s-admissible*, if the rt-run contains exactly n processors and satisfies the following timing properties: If ℓ is the number of messages sent during some job J,*

- *The message delay (measured from begin(J) to the corresponding receive event) of every message in trans(J) must be within $[\delta^-_{(\ell)}, \delta^+_{(\ell)}]$.*
- *The job duration $d(J)$ must be within $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$.*

3.4 Correctness and Impossibility

In the model presented so far, the scheduling policies are *adversary-controlled*, meaning that, in the game between player and adversary, the player chooses the algorithm and afterwards the adversary can decide on a scheduling policy that is most unsuitable for the algorithm. Thus, *correctness* proofs are *strong* (as the algorithm can defend itself against the most vicious scheduling policy), but *impossibility* proofs are *weak* (as the adversary has the scheduling policy on its side).

However, sometimes algorithms are designed for particular, a-priori-known scheduling policies, or the algorithm designer has the freedom to choose the scheduling policy which is most convenient for the algorithm. Thus, it is useful to define as *weak correctness* the correctness of an ⟨algorithm, scheduling policy⟩ pair and, analogously, as *strong impossibility* the absence of any such pair. In this paper, we only consider strong correctness and strong impossibility.

3.5 Transformations

The classic computing model and the real-time computing model are fairly equivalent from the perspective of solvability of problems. In [5], we present two transformations: One direction, simulating a real-time system on top of a classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$, is quite straightforward: It suffices to implement an artificial processing delay $\underline{\mu}$, the queuing of messages arriving during such a simulated job, and the scheduling policy. This simulation allows to run any real-time computing model algorithm $\mathcal{A}$ designed for a system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with $\delta^-_{(1)} \leq \underline{\delta}^-$, $\delta^+_{(1)} \geq \underline{\delta}^+$ and $\mu^- \leq \underline{\mu} \leq \mu^+$ on top of it, thereby resulting in a correct classic computing model algorithm. From this result (Theorem 3 in [5]), the following lemma can be derived directly:

Lemma 2. *Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system. If there exists an algorithm for solving some problem $\mathcal{P}$ in some real-time system $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with $\delta^-_{(1)} \leq \underline{\delta}^-$ and $\delta^+_{(1)} \geq \underline{\delta}^+$, then $\mathcal{P}$ can be solved in $\underline{s}$.*

The other direction, simulating a classic system $(n, [\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+])$ on top of a real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$, is more tricky: First, the class of classic computing model algorithms $\underline{\mathcal{A}}$ that can be transformed into a real-time

computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ must be somewhat restricted. Second, and more importantly, a *real-time scheduling analysis* must be conducted in order to break the circular dependency of algorithm $\underline{\mathcal{A}}$ and end-to-end delays $\Delta \in [\Delta^-, \Delta^+]$ (and vice versa): On one hand, the classic computing model algorithm $\underline{\mathcal{A}}$, run atop of the simulation, might need to know the *simulated* message delay bounds $[\underline{\delta}^-, \underline{\delta}^+]$, which are just the end-to-end delay bounds $[\Delta^-, \Delta^+]$ of the underlying simulation. Those end-to-end delays, on the other hand, involve the queuing delay ω and are thus dependent on (the message pattern of) $\underline{\mathcal{A}}$ and hence on $[\underline{\delta}^-, \underline{\delta}^+]$. This circular dependency is "hidden" in the parameters of the classic computing model, but necessarily pops up when one tries to instantiate this model in a real system.

In our setting, this circularity can be broken as follows: Given some classic computing model algorithm $\underline{\mathcal{A}}$ with assumed message delay bounds $[\underline{\delta}^-, \underline{\delta}^+]$, considered as unvalued parameters, a real-time scheduling analysis of the combined algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ ($\underline{\mathcal{A}}$ and simulation algorithm) must be conducted. This provides an equation for the resulting end-to-end delay bounds $[\Delta^-, \Delta^+]$ in terms of the real-time systems parameters $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ *and* the algorithm parameters $[\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+]$, i.e., a function F satisfying

$$[\Delta^-, \Delta^+] = F\big(n, [\delta^-, \delta^+], [\mu^-, \mu^+], [\Delta^-, \Delta^+]\big) \ . \tag{1}$$

We do not want to embark on the intricacies of advanced real-time scheduling analysis techniques here, see [2] for an overview. For the purpose of simple problems like terminating clock synchronization (see below), quite trivial considerations are sufficient: A trivial end-to-end delay lower bound Δ^- is $\delta^-_{(1)}$. An upper bound Δ^+ can be obtained easily if, for example, there is an upper bound on the number of messages a processor receives in total.

Anyway, if (1) provided by the real-time scheduling analysis can be solved for $[\Delta^-, \Delta^+]$, resulting in meaningful bounds $\Delta^- \le \Delta^+$, they can be assigned to the algorithm parameters $[\underline{\delta}^-, \underline{\delta}^+]$. Our transformation in fact guarantees that any timer-free algorithm designed for some classic system $\underline{s}$ solving some suitable problem $\mathcal{P}$ can also solve that problem in the corresponding real-time system s if such a feasible assignment for $[\underline{\delta}^-, \underline{\delta}^+]$ exists.

Clock Synchronization: In the classic computing model, a tight bound of $(1 - \frac{1}{n})\underline{\varepsilon}$ has been proved in [4] as the best achievable clock synchronization precision. We can use Lemma 2 to show that clock synchronization closer than $(1 - \frac{1}{n})\varepsilon_{(1)}$ is impossible in the

Theorem 3. *In the real-time computing model, no algorithm can synchronize the clocks of a system closer than* $(1 - \frac{1}{n})\varepsilon_{(1)}$.

Proof. Assume for a contradiction that there is some algorithm that can provide clock synchronization for some real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ to within $\gamma < (1 - \frac{1}{n})\varepsilon_{(1)}$. Applying Lemma 2 would imply that clock synchronization to within $\gamma < (1 - \frac{1}{n})(\underline{\delta}^+ - \underline{\delta}^-)$ can be provided for some classic system $(n, [\underline{\delta}^- = \delta^-_{(1)}, \underline{\delta}^+ = \delta^+_{(1)}])$. This, however, contradicts a well-known lower bound result [4].

4 Algorithms Achieving Optimal Precision

Theorem 3 raises the question whether the lower bound of $(1 - \frac{1}{n})\varepsilon_{(1)}$ is tight in the real-time computing model. In this section, we will answer this in the affirmative: We show how the algorithm presented in [4] can be modified to avoid queuing effects and thus provide optimal precision in a real-time system s. We will first present an algorithm achieving a precision of $(1-\frac{1}{n})\varepsilon_{(n-1)}$ (which is $(1-\frac{1}{n})\varepsilon_{(1)}$ if a constant-time broadcast primitive is available), and then describe how to extend this algorithm so that it achieves $(1 - \frac{1}{n})\varepsilon_{(1)}$ in the unicast case as well. Two lemmata from [4] can be generalized to our setting:

Lemma 4. *If q receives a timestamped message from p with end-to-end delay uncertainty ε_Δ, q can estimate p's hardware clock value within an error of at most $\frac{\varepsilon_\Delta}{2}$. (Proof: See Lemma 5 of [4] or see [5].)*

Lemma 5. *If every processor knows the difference between its own hardware clock and the hardware clock of every other processor within an error of at most $\frac{err}{2}$, clock synchronization to within $(1-\frac{1}{n})err$ is possible. (Proof: See Theorem 7 of [4].)*

4.1 Optimality for Broadcast Systems

Note: *To ease presentation, we use the abbreviations $\dot{\delta^-}$, $\dot{\delta^+}$, $\dot{\mu^-}$, $\dot{\mu^+}$ and $\dot{\varepsilon}$ to refer to $\delta^-_{(n-1)}$, $\delta^+_{(n-1)}$, $\mu^-_{(n-1)}$, $\mu^+_{(n-1)}$ and $\varepsilon_{(n-1)}$ in this subsection.*

The Lundelius-Lynch clock synchronization algorithm [4] works by sending one timestamped message from every processor to every other processor, and then computing the average of the estimated clock differences as a correction value. Any processor broadcasts its message as soon as its initialization message arrives. This algorithm can easily be modified to avoid queuing effects by "serializing" the information exchange, rather than sending all messages (possibly) simultaneously.

```
1   var estimates ← {}
2
3   function process_message(msg, time)
4     /* start alg. by sending (SEND) to proc. 0 */
5     if msg = (SEND)
6       send (TIME, time) to all other processors
7     elseif msg = (TIME, remote_time)
8       estimates.add(remote_time − time + (δ̇⁻ + δ̇⁺)/2)
9       if estimates.count = ID
10        send timer (SEND) for time + max(ε̇ − δ̇⁻ + μ̇⁺, μ̇⁺)
11      if estimates.count = n−1
12        set adjustment value to (∑ estimates)/n
```

Fig. 2. Clock-synchronization algorithm to within $\varepsilon_{(n-1)}$, code for processor ID

The modified algorithm, depicted in Fig. 2, works as follows: The n fully-connected processors have IDs $0, \ldots, n-1$. The first processor (0) sends its clock value to all other processors. Processor i waits until it has received the message from processor $i-1$, waits for another $\max(\dot{\varepsilon} - \dot{\delta^-} + \dot{\mu^+}, \dot{\mu^+})$ time units, and then broadcasts its own hardware clock value. That way, every processor receives the hardware clock values of all other processors with uncertainty $\dot{\varepsilon}$, provided that no queuing occurs (which will be shown below). This information suffices to synchronize clocks to within $(1 - \frac{1}{n})\dot{\varepsilon}$.

Lemma 6. *No queuing occurs when running the algorithm of Fig. 2.*

Proof. By the design of the algorithm, processor i only broadcasts its message after it has received exactly i messages. As processor 0 starts the algorithm and every processor broadcasts only once, this causes the processors to send their messages in the order of increasing processor number. For queuing to occur, some processor p must receive two messages within a time window smaller than $\dot{\mu^+}$. It can be shown, however, that the following invariant holds for all t: All receive events up to time t on the same processor i (a) occur in order of increasing (sending) processor number (including the timer message from i itself), and (b) are at least $\dot{\mu^+}$ time units apart.

Assume by contradiction that some message from processor $j > 0$ arrives on processor i at time t, although the message from processor $j-1$ has arrived (or will arrive) at time $t' > t - \dot{\mu^+}$. Choose t to be the first time the invariant is violated.

Case 1: $j = i$, i.e., the arriving message is i's timer message. This leads to a contradiction, as due to line 10, this message must not arrive earlier than $\dot{\mu^+}$ time units after $j-1$'s message, which has triggered the job sending the timer message.

Case 2: $j \neq i$. As j's broadcast arrived at t, it has been sent no later than $t - \dot{\delta^-}$. Process j's broadcast is triggered by a timer message sent by j's job starting $\max(\dot{\varepsilon} - \dot{\delta^-} + \dot{\mu^+}, \dot{\mu^+})$ time units earlier, i.e., no later than $t - \dot{\delta^-} - (\dot{\varepsilon} - \dot{\delta^-} + \dot{\mu^+}) = t - \dot{\varepsilon} - \dot{\mu^+}$. The job sending the timer message has been triggered by the arrival of $j-1$'s broadcast, which must have been sent no later than $t - \dot{\varepsilon} - \dot{\mu^+} - \dot{\delta^-}$. If $j - 1 = i$, we have the required contradiction, because i must have received its timer message at $t' \leq t - \dot{\varepsilon} - \dot{\mu^+} - \dot{\delta^-}$ long ago. Otherwise, if $j - 1 \neq i$, process $j-1$'s broadcast arrived at i no later than $t - \dot{\varepsilon} - \dot{\mu^+} - \dot{\delta^-} + \dot{\delta^+} = t - \dot{\mu^+}$, also contradicting our assumption.

Theorem 7 (Optimal broadcast algorithm). *The algorithm of Fig. 2 achieves a precision of* $(1 - \frac{1}{n})\varepsilon_{(n-1)}$*, which matches the lower bound in Theorem 3 if communication is performed by a constant-time broadcast primitive, i.e., if* $\varepsilon_{(n-1)} = \varepsilon_{(1)}$*. It performs exactly n broadcasts and has a time complexity that is at least* $\Omega(n)$*.*

Proof. On each processor, the set *estimates* contains the estimated differences between the local hardware clock and the hardware clocks of the other processes.

As no queuing occurs by Lemma 6, the end-to-end delays are just the message delays. Line 8 in the algorithm of Fig. 2 ensures that the estimates are calculated as specified in the proof of Lemma 4. Thus, the estimates have a maximum error of $\frac{\dot{\varepsilon}}{2}$. According to Lemma 5, this allows the algorithm to calculate an adjustment value (in line 12) ensuring clock synchronization to within $(1 - \frac{1}{n})\dot{\varepsilon}$.

With respect to message and time complexity, the algorithm obviously performs exactly n broadcasts, and the worst-case time between two subsequent broadcasts is $\max(\dot{\delta}^+, 2\dot{\varepsilon}) + \dot{\mu}^+$ (= the timer message delay $\max(\dot{\varepsilon} - \dot{\delta}^- + \dot{\mu}^+, \dot{\mu}^+)$ plus one message delay $\dot{\delta}^+$). Thus, the time complexity is at least linear in n, and depends on the complexity of $\delta^+_{(\ell)}$, $\varepsilon_{(\ell)}$ and $\mu^+_{(\ell)}$ w.r.t. ℓ.

4.2 Optimality for Unicast Systems

The algorithm of the previous section provides clock synchronization to within $(1 - \frac{1}{n})\varepsilon_{(n-1)}$. However, unless constant-time broadcast is available, $\varepsilon_{(1)}$ will usually be smaller than $\varepsilon_{(n-1)}$. The algorithm can be adapted to unicast sends as shown in Fig. 3, however: Rather than sending all $n - 1$ messages at once, they are sent in $n - 1$ subsequent jobs connected by "send" timer messages, each sending only one message. These messages are timestamped with their corresponding HC value, e.g. the message sent during the second job will be timestamped with the hardware clock reading of this second job.

Theorem 8 (Optimal unicast algorithm). *The algorithm of Fig. 3 achieves a precision of* $(1 - \frac{1}{n})\varepsilon_{(1)}$*, which matches the lower bound given in Theorem 3. It sends exactly* $n(n-1) = \mathrm{O}(n^2)$ *messages system-wide and has* $\mathrm{O}(n)$ *time complexity.*

Proof. Omitted due to size limitations; see [5].

```
1  var estimates ← {}
2
3  function process_message(msg, time)
4     /* start alg. by sending (SEND, 1) to proc. 0 */
5     if msg = (SEND, target)
6        send (TIME, time) to target
7        if target + 1 mod n ≠ ID
8           send timer (SEND, target + 1 mod n) for time + μ+(1)
9     elseif msg = (TIME, remote_time)
10       estimates.add(remote_time − time + (δ−(1) + δ+(1))/2)
11       if estimates.count = ID
12          send timer (SEND, ID + 1) for
13                time + max(ε(1) − δ−(1) + 2μ+(1), μ+(1))
14       if estimates.count = n−1
15          set adjustment value to (∑ estimates)/n
```

Fig. 3. Clock-synchronization algorithm to within $\varepsilon_{(1)}$, code for processor ID

5 Lower Bounds

In this section, we will establish lower bounds for message and time complexity of (close to) optimal precision clock synchronization algorithms.

In particular, for optimal precision, we will prove that at least $\frac{1}{2}n(n-1) = \Omega(n^2)$ messages must be exchanged, since at least one message must be sent over every link. This bound is asymptotically tight, since it is matched by Theorem 8. A strong indication for this result follows already from the work of Biaz and Welch [7]. They have shown that no algorithm can achieve a precision better than $\frac{1}{2}diam(G)$ for any communication network G, with $diam(G)$ being the diameter of the graph where the edges are weighted with the uncertainties: In the classic computing model, a fully-connected network with equal link uncertainty $\underline{\varepsilon}$ can achieve no better precision than $\frac{1}{2}\underline{\varepsilon}$, whereas removing one link yields a lower bound of $\underline{\varepsilon}$. Thus, after removing one link, the optimal precision of $(1-\frac{1}{n})\underline{\varepsilon}$ shown by [4] can no longer be achieved.

Unfortunately, the proof from [7] cannot be used directly in our context: While they show that $(1-\frac{1}{n})\underline{\varepsilon}$ cannot be achieved if the system forbids the algorithm to use one system-chosen link, we have to show that if the algorithm is presented with a fully-connected network and decides not to use one algorithm-chosen link (which can differ for each execution/rt-run) dynamically, this algorithm cannot achieve optimal precision. A shifting argument similar to the one used in the proof of Theorem 3 of [7] can be used, however. *Shifting* is a common technique in the classic computing model for proving clock synchronization lower bounds. Analogously, shifting a rt-run ru of n processors by $(x_0, \ldots, x_{n-1})$ results in another rt-run ru', where

- receive events and jobs on processor p_i starting at real-time t in ru start at real-time $t - x_i$ in ru',
- the hardware clock of p_i is shifted such that all receive events and jobs still have the same hardware clock reading as before, i.e. $HC'_i(t) := HC_i(t) + x_i$.

Environment: Let $c \in \mathbb{R}^+$ be a constant and $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system with $n > 2$. Let $\mathcal{A}$ be an algorithm providing clock synchronization to within $c \cdot \varepsilon_{(1)}$ in s. Let ru be an s-admissible rt-run of $\mathcal{A}$ in s, where the message delay of all messages is the arithmetic mean of the lower and upper bound. Thus, modifying the delay of any message by $\pm\varepsilon_{(1)}/2$ still results in a value within the system model bounds. The duration of all jobs sending ℓ messages is $\mu^+_{(\ell)}$.

Lemma 9. *The message graph of ru has a diameter of* $2c$ *or less.*

Proof. Let the *message graph* of a rt-run ru be defined as an undirected graph containing all processors as vertices and exactly those links as edges over which at least one message is sent in ru. Assume by contradiction that the message graph has a diameter $D > 2c$. Let p and q be two processors at distance D. Let Π_d be the set of processors at distance d from p. Let ru' be a new rt-run in which processors in Π_d are shifted by $d \cdot \varepsilon_{(1)}/2$, i.e., all receive events and jobs

on some processor in Π_d happen $d \cdot \varepsilon_{(1)}/2$ time units earlier although with the same hardware clock readings. As processors in Π_d only exchange messages with processors in Π_{d-1}, Π_d and Π_{d+1}, message delays are changed by $-\varepsilon_{(1)}/2$, 0 or $\varepsilon_{(1)}/2$. Thus, ru' is s-admissible.

Let Δ and Δ' be the final (signed) differences between the adjusted clocks of p and q in ru and ru', respectively. As both rt-runs are s-admissible and $\mathcal{A}$ is assumed to be correct, $|\Delta| \leq c \cdot \varepsilon_{(1)}$ and $|\Delta'| \leq c \cdot \varepsilon_{(1)}$.

By definition of shifting, $HC'_p(t) = HC_p(t)$ and $HC'_q(t) = HC_q(t) + D \cdot \varepsilon_{(1)}/2$. Thus, $\Delta' = HC'_p(t) + adj_p - (HC'_q(t) + adj_q) = HC_p(t) + adj_p - (HC_q(t) + D \cdot \varepsilon_{(1)}/2 + adj_q) = \Delta - D \cdot \varepsilon_{(1)}/2$.

Let ru'' be ru shifted by $-d \cdot \varepsilon_{(1)}/2$. The same arguments hold, resulting in $\Delta'' = \Delta + D \cdot \varepsilon_{(1)}/2$. As $|\Delta|$, $|\Delta'|$ and $|\Delta''|$ must all be $\leq c \cdot \varepsilon_{(1)}$, we have the inequalities $|\Delta| \leq c \cdot \varepsilon_{(1)}$, $|\Delta + D \cdot \varepsilon_{(1)}/2| \leq c \cdot \varepsilon_{(1)}$ and $|\Delta - D \cdot \varepsilon_{(1)}/2| \leq c \cdot \varepsilon_{(1)}$, which imply $c \geq D/2$ and hence contradict $D > 2c$.

5.1 Message Complexity

For clock synchronization to within some $\gamma < \varepsilon_{(1)}$, Lemma 9 implies that there is a rt-run with message graph diameter < 2, i.e., whose message graph is fully connected, and, therefore, has $\frac{n(n-1)}{2}$ edges. This leads to the following theorem:

Theorem 10. *Clock synchronization to within $\gamma < \varepsilon_{(1)}$ has a worst-case message complexity of at least $\frac{n(n-1)}{2} = \Omega(n^2)$.*

Section 4 presented an algorithm achieving optimal precision of $(1 - \frac{1}{n})\varepsilon_{(1)}$ with $n(n-1) = \mathrm{O}(n^2)$ messages, cp. Theorem 8. Theorem 10 reveals that this bound is asymptotically tight. Obviously, a smaller lower bound can be given for suboptimal clock synchronization. We will use the following simple graph-theoretical lemma:

Lemma 11. *In an undirected graph with $n > 2$ nodes and diameter D or less, there is at least one node with degree $\geq \sqrt[D+1]{n}$.*[4]

Proof. Assume by contradiction that all nodes have a maximum degree of some non-negative integer $d < \sqrt[D+1]{n}$. As $n > 2$, $d = 0$ or $d = 1$ would cause the graph to be disconnected, thereby contradicting the assumption of bounded diameter. Thus, we can assume that $d > 1$.

Fix some node p. Clearly, after D hops, the maximum number of nodes reachable from p (including p at distance 0) is $\sum_{i=0}^{D} d^i = \frac{d^{D+1}-1}{d-1} \leq d^{D+1} < \sqrt[D+1]{n}^{D+1} = n$. As we cannot reach n nodes after D hops, we are done.

Combining Lemmata 9 and 11 shows that there is at least one processor in ru which exchanges (= sends or receives) at least $\lceil \sqrt[2c+1]{n} \rceil$ messages. This implies the following results:

[4] A result with similar order of magnitude can be derived from the Moore bound.

Theorem 12. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$ in some real-time system s, there is an s-admissible rt-run in which at least one processor exchanges $\lceil \sqrt[2c+1]{n} \rceil$ messages.*

Corollary 13. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$, there is no constant upper bound on the number of messages exchanged per processor.*

Note, however, that it is possible to constantly bound either the number of received *or* the number of sent messages per processor (but not both): Section 6 presents an algorithm synchronizing clocks to within $\varepsilon_{(1)}$ where every processor receives exactly one message, and an algorithm provided in [5] achieves this precision with sending just 3 messages per processor.

5.2 Time Complexity

In the case of optimal precision, the following Theorem 14 states a time complexity lower bound of $\Omega(n)$. This bound is asymptotically tight, since it is matched by the optimal algorithm underlying Theorem 8.

Theorem 14. *Clock synchronization to within $\gamma < \varepsilon_{(1)}$ has a worst-case time complexity of at least $\Omega(n)$.*

Proof. Theorem 10 revealed that n processors need to exchange at least $\frac{n(n-1)}{2}$ messages. Therefore, no algorithm can achieve a run time better than $\max\left(\frac{n-1}{2}\mu^+_{(0)}, \delta^+_{(\frac{n-1}{2})}\right)$, assuming optimal concurrency. This implies a time complexity lower bound of $\Omega(n)$ as asserted.

On the other hand, Theorem 12 implies a lower bound on the worst-case time complexity of any algorithm that synchronizes clocks to within $c \cdot \varepsilon_{(1)}$: Some process p must exchange $\lceil \sqrt[2c+1]{n} \rceil$ messages, some k of which are received and the remaining ones are sent by p. Recalling $\delta^+_{(\ell)} \leq \ell\delta^+_{(1)}$ from Sect. 3.3, the algorithm's time complexity must be at least $\min_{k=0}^{\lceil \sqrt[2c+1]{n} \rceil}(k \cdot \mu^+_{(0)} + \delta^+_{(n-k)})$. Clearly, $k\mu^+_{(0)}$ is linear in k, so the interesting term is $\delta^+_{(n-k)}$. Consequently:

Corollary 15. *If multicasting a message in constant time is impossible, clock synchronization to within a constant factor of the message delay uncertainty cannot be done in constant time.*

6 Achievable Precision for Less than $\Omega(n^2)$ Messages

Sometimes, $\Omega(n^2)$ messages may be too costly if a precision of $(1 - \frac{1}{n})\varepsilon_{(1)}$ is not required. Clearly, every clock synchronization algorithm requires a minimum of $n - 1$ messages to be sent system-wide; otherwise, at least one processor would not participate. Interestingly, $n - 1$ messages (plus one external init message) already suffice to achieve a precision of $\varepsilon_{(1)}$ by using a simple star topology-based algorithm: Figure 4 is actually a simpler version of the algorithm presented in

```
1  function s−process_message(msg, time)
2     /* start alg. by sending (INIT) to some proc. */
3     if msg = (INIT)
4        send time to all other processors
5        set adjustment value to 0
6     else
7        set adjustment value to (msg − time + (δ⁻_(n−1) + δ⁺_(n−1))/2)
```

Fig. 4. Star Topology-based Clock Synchronization Algorithm

Sect. 4. Rather than collecting the estimated differences to all other processors and then calculating the adjustment value, this algorithm just sets the adjustment value to the estimated difference to one designated master processor, the one receiving (INIT). Lemma 4 shows that the error of these estimates is bounded by $\frac{\varepsilon_{(n-1)}}{2}$. Thus, setting the adjustment value to the estimated difference causes all clocks to be synchronized to within $\varepsilon_{(n-1)}$.

If δ^-, δ^+, μ^- and μ^+ are independent of n (i.e., if a constant-time broadcasting primitive is available), $\varepsilon_{(n-1)} = \varepsilon_{(1)}$ and the algorithm achieves this precision in constant time (w.r.t. n). Otherwise, the following modification puts the precision down to $\varepsilon_{(1)}$ in the unicast case as well:

- Do not send all messages during the same job but during subsequent jobs on the "master" processor.
- Replace $\delta^-_{(n-1)}$ and $\delta^+_{(n-1)}$ in Line 7 with $\delta^-_{(1)}$ and $\delta^+_{(1)}$.

The algorithm still exchanges only $n-1$ messages and has linear time complexity w.r.t. n. As Theorem 10 has shown, $\varepsilon_{(1)}$ is the best precision that can be achieved with less than $\Omega(n^2)$ messages. As Corollary 15 has shown, this precision cannot be achieved in constant time in the unicast case.

7 Conclusions and Future Work

We presented a real-time computing model, which just adds non-zero computing step times to the classic computing model. Since it explicitly incorporates queuing effects, our model makes distributed algorithms amenable to real-time scheduling analysis, without, however, invalidating classic algorithms, analysis techniques, and impossibility/lower bound results. General transformations based on simulations between both models were established for this purpose.

Revisiting the problem of optimal deterministic clock synchronization in the drift- and failure-free case, we showed that the best precision achievable in the real-time computing model is $(1-\frac{1}{n})\varepsilon_{(1)}$. This matches the well-known result in the classic computing model; it turned out, however, that there is no constant-time algorithm achieving optimal precision in the real-time computing model. Since such an algorithm is known for the classic model, this is an instance of a problem where the classic analysis gives too optimistic results. We also established algorithms and lower bounds for sub-optimal clock synchronization in the

real-time computing model. For example, we showed that clock synchronization to within a constant factor of the message delay uncertainty can be achieved in constant time only if a constant-time broadcast primitive is available.

Part of our current research is devoted to extending our real-time computing model to failures and, in particular, drifting clocks. Clearly, all our lower bound results also hold for the drifting case. As time complexity influences the actual precision achievable with drifting clocks, however, a simpler, less precise algorithm might in fact yield some better overall precision than a more complex optimal algorithm, depending on the system parameters. Apart from this, we are looking out for problems and algorithms that involve more intricate real-time scheduling analysis techniques.

References

1. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
2. Sha, L., Abdelzaher, T., Arzen, K.E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real time scheduling theory: A historical perspective. Real-Time Systems Journal **28**(2/3) (2004) 101–155
3. Simons, B., Lundelius-Welch, J., Lynch, N.: An overview of clock synchronization. In Simons, B., Spector, A., eds.: Fault-Tolerant Distributed Computing, Springer Verlag (1990) 84–96 (Lecture Notes on Computer Science 448).
4. Lundelius, J., Lynch, N.: An upper and lower bound for clock synchronization. Information and Control **62** (1984) 190–240
5. Moser, H., Schmid, U.: Optimal clock synchronization revisited: Upper and lower bounds in real-time systems. Research Report 71/2006, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (2006) `http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=1973&viewmode=paper&year=2006`.
6. Lundelius, J., Lynch, N.: A new fault-tolerant algorithm for clock synchronization. In: Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC). (1984) 75–88
7. Biaz, S., Welch, J.L.: Closed form bounds for clock synchronization under simple uncertainty assumptions. Information Processing Letters **80**(3) (2001) 151–157
8. Patt-Shamir, B., Rajsbaum, S.: A theory of clock synchronization (extended abstract). In: STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, New York, NY, USA, ACM Press (1994) 810–819
9. Attiya, H., Herzberg, A., Rajsbaum, S.: Optimal clock synchronization under different delay assumptions. In: PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM Press (1993) 109–120

Distributed Priority Inheritance for Real-Time and Embedded Systems*

César Sánchez[1], Henny B. Sipma[1],
Christopher D. Gill[2], and Zohar Manna[1]

[1] Computer Science Department
Stanford University, Stanford, CA 94305, USA
{cesar, sipma, manna}@CS.Stanford.EDU
[2] Department of Computer Science and Engineering
Washington University, St. Louis, MO 63130, USA
cdgill@CSE.wustl.EDU

Abstract. We study the problem of priority inversion in distributed real-time and embedded systems and propose a solution based on a distributed version of the priority inheritance protocol (PIP). Previous approaches to priority inversions in distributed systems use variations of the priority ceiling protocol (PCP), originally designed for centralized systems as a modification of PIP that also prevents deadlock. PCP, however, requires maintaining a global view of the acquired resources, which in distributed systems leads to high communication overhead.

This paper presents a distributed PIP built on top of a deadlock avoidance schema that requires much less communication than PCP. Since the system is already deadlock free and priority inversions can be detected locally, we obtain an efficient dynamic resource allocation system that prevents deadlocks and handles priority inversions.

Keywords: Priority Scheduling, Deadlock Avoidance, Distributed Resource Allocation, Distributed Real-Time and Embedded Systems.

1 Introduction and Related Work

Modern distributed real-time and embedded systems (DRE) are built using a real-time middleware that extends conventional middleware services with real-time QoS capabilities. It is common in real-time systems to assign priorities to processes to achieve a higher confidence that the more critical tasks will be accomplished in time. A priority inversion is produced when a high priority process is blocked by a lower priority one. State-of-the-art middleware solutions, based for example on CORBA ORBs, may suffer priority inversions [24].

* César Sánchez, Henny B. Sipma and Zohar Manna are supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, and by NAVY/ONR contract N00014-03-1-0939. Christopher D. Gill is partially supported by NSF CAREER Award CCF-0448562.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 110–125, 2006.

In centralized systems, priority inversions are usually handled using a protocol from the priority inheritance family [26,15]. However, priority inheritance, and synchronization in general, is significantly affected by concurrent execution. Even though some variations of these centralized protocols have been proposed for multiprocessors [15] and distributed systems [15,13], there is not yet a widely accepted general scheme. We propose here a mathematically sound method to deal with priority inversions in DRE architectures.

In this paper we consider DRE systems that consist of a set of *sites*, each of which is capable of executing a predefined set of computations or methods, connected using an asynchronous network. Processes, consisting of local and remote method calls, are created dynamically. The relevant resources are the threads (or execution contexts) to run the methods. We assume a *WaitOnConnection* [25] thread-allocation policy, that is, each method requires its own thread, including nested up-calls, and methods hold on to their thread until they finish.

Since resources are finite and we impose no restriction on the number of processes, deadlocks are reachable unless there is a mechanism in place to control those allocations. It is important to distinguish between two different kinds of deadlocks: intra-resource (caused by parallel access to a single resource) and inter-resource deadlocks (caused by interference across different allocations).

Intra-resource deadlocks: Absence of intra-resource deadlock is one of the requirements of a solution to mutual exclusion, together with exclusive access and, sometimes, lack of starvation. Several algorithms have been proposed for distributed mutual exclusion, which can be classified (see [28,29,17]) into:

- Permission based: A process can access a resource if there is unanimity [19] between the participants about its safety. Unanimity can be relaxed to majority quorums [8,31,12], or even majorities in coteries [7,1]. The message complexities range from $2(N-1)$ in the original Ricart-Agrawala algorithm [19] to $O(log\ N)$ in the best case (with no failures) in the modern coterie-based approaches.
- Token-based: The system is equipped with a single token per resource, which is held by the process that accesses it. A distributed data-type is maintained to select the next recipient. For example, Suzuki-Kasami's approach [30] exhibits a complexity of $O(N)$ messages per access, and Raymond's [16] and Naimi-Tehel's [14] approaches, use spanning trees to obtain an average case complexity of $O(log\ N)$, still exhibiting a $O(N)$ worst case.

However, the most efficient way (in terms of message complexity) to achieve distributed mutual exclusion is to use a centralized algorithm, like a primary site approach [2]. For every resource, a distinguished site arbitrates the accesses, reducing the problem of distributed mutual exclusion to the centralized case. Thus, allocations can be resolved with one message per request. This comes at a price of lower resiliency because, if the site managing the resource fails, the resource becomes inaccessible. However, in some cases, like DRE systems and Flexible Manufacturing Systems [6,18] resources are indeed tightly coupled to the sites where they reside, and therefore it is natural to use this site as the

primary means to resolve each request contention. This is the basic approach that we use for intra-resource arbitration.

Inter-resource deadlocks: A different kind of deadlock can be produced due to the interleavings between accesses to different resources, when a set of processes is waiting in a circular chain in which a process holds a resource needed by the next process in the chain. The two mechanisms commonly used to ensure deadlock-free assignment of resource allocation are deadlock prevention and deadlock avoidance. A third mechanism, deadlock detection and resolution, is common in databases, but not used in DRE systems, because it may lead to unbounded worst-case execution times.

Deadlock prevention methods eliminate one of the necessary causes of the circular contention at design time, at the price of decreasing the concurrency. For example, "monotone locking", widely used in practice [3], determines an arbitrary total order on the set of resources that is followed at run-time to acquire resources.

Deadlock avoidance methods check the current resource allocation at runtime and grant a resource only if it is *safe*, that is, if there is a way for all processes to eventually finish. This check is made possible by having processes that enter the system announce their maximum resource usage, an approach first proposed by Dijkstra in his Banker's algorithm [5,9,10]. When resources are distributed across multiple sites, however, deadlock avoidance is harder, because different sites may have to consult each other to determine whether a particular allocation is safe. Because of this need for distributed agreement, a general solution to distributed deadlock avoidance is considered impractical [27]. Efficient solutions do exist, however, for the type of systems considered here, namely DRE systems with a *WaitOnConnection* thread-allocation policy. In [23,22] we demonstrated an efficient distributed deadlock avoidance method for systems for which all the possible sequences of invocations are known and available to analysis a priori. In this paper we build on this algorithm to construct an efficient distributed priority inheritance protocol.

Priority Inheritance Protocols: It is common in real-time systems to assign priorities to processes. A priority inversion is produced when a process with high priority is unnecessarily blocked by a process with lower priority. To bound priority inversions, the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) were introduced, primarily applicable to hard real-time systems with shared resources and static priorities [26]. Later, these methods were extended to dynamic priority scheduling algorithms such as Earliest Deadline First (EDF) [4]. PIP can bound blocking times if a bound on the running time of each process and all its critical sections is known. PIP does not, however, prevent deadlocks, and therefore PCP was introduced to prevent inter-resource deadlocks, at the price of some concurrency.

A distributed version of PCP was proposed in [13] to deal with priority inversion and inter-resource deadlock in distributed systems. This protocol, however, suffers from a high communication overhead: before a request is granted, the

ceiling of each resource that is (globally) used must be queried. This requires maintaining global views of the system. Having more information about the system in the form of call graphs, however, we can use the simpler and more efficient priority inheritance protocol (PIP) to deal with priority inversions and use our deadlock avoidance algorithm to guarantee absence of inter-resource deadlocks. Our priority inheritance protocol allows more concurrency and, more importantly, it involves *no communication overhead when priority inversions are not present*, which in our setting is locally testable (it requires no communication to detect a priority inversion). Moreover, when priority inversions do exist, our protocol involves only one-way communication, without the need for return messages. Once an inversion is detected and the priority inheritance protocol is run—which may inject messages into the network—the local processes can proceed immediately without compromising deadlock freedom. This leads to a more efficient solution, especially in scenarios where latencies are significant compared to the running times of methods.

Our PIP protocol enables the computation of a bound on the number of lower priority processes that can block a higher priority one. Consequently, the blocking time of a process can also be bound if the maximum running time of each method and the latency of each message is known. This solution enables the following design methodology for DRE systems. A distributed system with periodic and sporadic tasks with deadlines can be analyzed for schedulability: (1) computing initial priorities of processes statically, and (2) showing a proof that deadlines are met in all possible executions. In this paper we prove the correctness of the distributed priority inheritance protocol, and leave distributed schedulability analysis for future research.

The rest of this paper is structured as follows. Section 2 reviews our distributed deadlock avoidance algorithms. Section 3 includes the distributed priority inheritance protocol and proves its correctness, and Section 4 presents our conclusions and describes future work.

2 Distributed Deadlock Avoidance

We model a distributed system $\mathcal{S} : \langle \mathcal{R}, \mathcal{G} \rangle$ by a set of *sites* and a *call graph specification*. The sites $\mathcal{R} : \{r_1, \ldots, r_{|\mathcal{R}|}\}$ model distributed devices that perform computations and handle a necessary and scarce local resource, such as a finite pool of threads. A call graph specification $\mathcal{G} : \langle N, \rightarrow, I \rangle$ consists of a directed acyclic graph $\langle N, \rightarrow \rangle$, which captures all the possible sequences of remote calls that processes can perform. The set of initial nodes $I \subseteq N$ contains those methods that can be invoked when a process is spawned.

A call graph node abstracts both the computation to be performed at a site and the resource needed. Each node has a unique name (the method name) and a site associated with it, the site where the method will be executed at run-time. If node n describes resource $(f : r)$ we say that n executes computation f and *resides* in site r. We use the predicate $n \equiv_{\mathcal{R}} m$ to represent that nodes n and m reside in the same site. To simplify notation, if the method name is unimportant

we use $n : r$ instead of $n = (f : r)$. We use $r, s, r_1, r_2, \ldots$ to refer to sites and $n, m, n_1, m_1, \ldots$ to refer to call graph nodes.

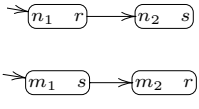
An edge $n \rightarrow m$ in the call graph denotes a possible remote invocation; in order to complete the computation modeled by n the result of a call to m may be needed. If this call is performed, the resource associated with n will be locked at least until the invocation of m returns. Note how this call semantics is not equivalent to synchronous calls since we do not assume that the caller needs a response to continue the computation but only needs a response to *complete* its execution. For example, after initiating a remote method call, the caller can immediately continue and perform more remote invocations before waiting for the reply. All our results immediately cover synchronous semantics as a particular case (in which callers do wait for a response before proceeding) and can be easily adapted to totally asynchronous semantics (where processes are even allowed to terminate without waiting for responses).

A run of a system consists of the execution of processes, created dynamically. When a new process is spawned it announces an initial call graph node whose outgoing paths capture the remote calls that the process may perform. Incoming invocations require a new resource to run, while call returns release a resource. We use the following terminology: every new method invocation is called a *process*, and the context will disambiguate between subprocesses (created by remote calls) and proper processes (corresponding to new instances entering the system). Once a process has received a resource, it holds onto it until completion, that is, there is no preemption once a resource is acquired. We also assume that all computations terminate, if their demanded resources are granted.

Even though in principle our computational model can be regarded as non-preemptive, methods that assume preemptive scheduling (the setting where classical priority inheritance with rate-monotonic scheduling was introduced [11]) are also relevant. This is because in our model, all computations occur inside critical sections (resources are nested).

Deadlocks can be reached if all resource requests are immediately granted, since resources are finite and fixed a priori in each site and—in principle—we impose no restriction on the topology of the call graph specification or the number of process instances. We use T_r for the total number of resources in site r, and t_r for a variable that keeps track of the number of resources available in r at every instant. Initially, $t_r = T_r$.

Example 1. Consider a system with sites $\mathcal{R} = \{r, s\}$, nodes $N = \{n_1, n_2, m_1, m_2\}$ with n_1 and m_1 initial nodes, and call graph

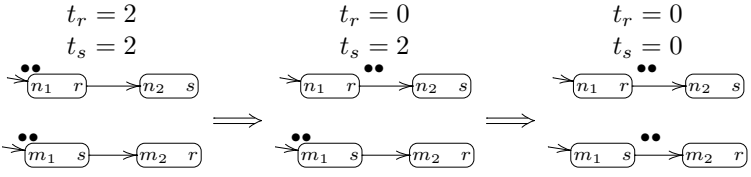


This system has reachable deadlocks if no controller is used. Let sites s and r handle exactly two threads each. If four processes are spawned, two instances of n_1 and two of m_1, all resources in the system will be locked after each process starts executing its initial node. Consequently, the allocation attempts for n_2

$$n :: \left[\begin{array}{l} \left[\begin{array}{c}\textbf{when } \alpha(n) \leq t_r \textbf{ do} \\ t_r\textbf{--}\end{array}\right] \\ f() \\ t_r\textbf{++}\end{array}\right]$$

Fig. 1. The deadlock avoidance protocol Basic-P

and m_2 will be blocked indefinitely, so no process will terminate or return a resource. A possible allocation sequence is shown below,



where a • represents an existing process that tries to acquire a resource at a node (if • precedes the node) or has just been granted the resource (if • appears after the node). It is easy to see that a deadlock can still occur even if the number of threads in r and s is increased. We can simply spawn the corresponding number of processes.

In our solution to deadlock avoidance, the assignment of resources is controlled by two cooperating components: the *allocation manager* and the *scheduler*. The allocation manager decides which subset of pending requests is safe (in the sense that no continuation of the execution will reach a deadlock if granted); these requests are called *enabled*. The scheduler then chooses a process among the enabled ones, which receives the resource. This interaction between the allocation manager and the scheduler is repeated until the set of enabled processes is empty. Processes whose request is disabled are called *waiting*, while processes that hold a resource are called *active*.

A deadlock avoidance algorithm is an allocation manager that guarantees that no deadlock can be reached, independently of the scheduler used. Our deadlock avoidance algorithms consist of two parts:

1. A computation of *annotations* of call graph nodes, carried out statically. We consider maps from nodes to natural numbers $\alpha : N \mapsto \mathbb{N}$ as annotations.
2. A *protocol*: a piece of code that ensures, at runtime, that allocations and deallocations are safe. It consists of two stages: one that runs when the resource is requested, and another when the resource is released. We are seeking protocols that only inspect and modify local variables of the site.

The deadlock avoidance protocol Basic-P [23] for node $n = (f : r)$ is shown in Fig. 1. The *entry section* that precedes the access to the method call $f()$ consists of a guard and an action that operate on local variables of site r. The guard captures the enabling condition of the request for node n. We assume that the entry section is executed atomically, as a test-and-set operation.

A requesting process executing node $n : r$ is enabled if there are at least $\alpha(n)$ resources available in r. Note, however, that only one resource is acquired. The *exit section* is executed when the method terminates and consists of an action that updates local variables, in this case increasing t_r. The execution of the exit section triggers the allocation manager to re-evaluate the entry condition of the waiting processes. Those requests for resources found safe, if any, are handed over to the scheduler, which chooses one to be granted.

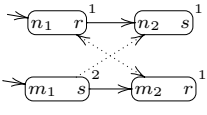
The most important property that protocols must enforce is freedom from deadlock. The following is a characterization of deadlock:

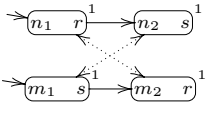
Definition 1 (Deadlock). *A deadlock is a global state in which there is a non-empty set of disabled processes that continue to be disabled even if all the other processes in the system return their acquired resources.*

Basic-P avoids deadlocks if the annotation is acyclic in the following sense. Given a system $\langle \mathcal{R}, \mathcal{G} \rangle$ and an annotation α, the annotated call graph $(N, \rightarrow, \dashrightarrow)$ adds to $\mathcal{G}$ one edge $n \dashrightarrow m$ for every pair of nodes n and m that reside in the same site and $\alpha(n) \geq \alpha(m)$. A node n depends on a node m, represented as $n \succ m$, if there is a path in the annotated graph from n to m that follows at least one $\rightarrow$ edge. The annotated graph is acyclic if no node depends on itself, in which case we say that the annotation is acyclic.

Theorem 1 (Annotation Theorem for Basic-P [23]). *Given a system and an acyclic annotation, if* Basic-P *is used in every node to control resource allocations then all executions of the system are deadlock free.*

Example 2. Reconsider the system from Example 1. The left diagram below shows an annotated call graph with $\alpha(n_1) = \alpha(n_2) = \alpha(m_2) = 1$ and $\alpha(m_1) = 2$. It is acyclic, and thus by Theorem 1, if Basic-P is used with these annotations, the system is deadlock free.





Let us compare this with Example 1 where a resource is granted simply if it is available. This corresponds to using Basic-P with the annotated call graph above on the right, with $\alpha(n) = 1$ for all nodes. In Example 1 we showed that a deadlock is reachable, and indeed this annotated graph is not acyclic; it contains dependency cycles, for example $n_1 \rightarrow n_2 \dashrightarrow m_1 \rightarrow m_2 \dashrightarrow n_1$. Therefore Theorem 1 does not apply. In the diagram on the left all dependency cycles are broken by the annotation $\alpha(m_1) = 2$. Requiring the presence of at least two resources for granting a resource at m_1 ensures that the last resource available in s can only be obtained at n_2, which breaks all possible circular waits.

The priority inheritance protocol presented in the next section is based on Basic-P. Its correctness relies on the following property.

Theorem 2 (Reachable state space [20]). *The set of global states reachable by* Basic-P *contains precisely those states in which, for all sites r and annotations k*

$$\varphi : \quad A_r[k] \leq T_r - (k-1) \ ,$$

where $A_r[k]$ denotes the number of active processes in site r executing call graph nodes with annotation k or higher.

The global states that satisfy φ are called φ-states. Since Basic-P guarantees deadlock free operation, Theorem 2 implies, for example, that if a system is in a φ-state and Basic-P is used as an allocation manager in every site, then there is some enabled process. In [21,20] we introduced more efficient protocols (Efficient-P, k-Efficient-P and Live-P) and proved annotation theorems similar to Theorem 1, but Basic-P is simpler and it is enough to illustrate the present discussion. In the remainder of the paper we will assume that Basic-P is used to allocate the resources.

3 Priority Inheritance

In this section we develop a priority inheritance mechanism and show how it helps to alleviate priority inversions. A priority specification extends a system specification with a description of the possible priorities at which processes can run. The fixed set $\mathcal{P}$ of priorities is a finite and totally ordered set. Without loss of generality, we take $\mathcal{P} = \{1, \ldots, p_m\}$, where lower value means higher priority: 1 represents the highest priority and p_m the lowest.

Definition 2. *Given a system $\langle \mathcal{R}, \mathcal{G} \rangle$ and set of priorities $\mathcal{P}$, a* priority assignment *is a map from initial nodes I to sets of priorities:*

$$\Pi : I \to 2^{\mathcal{P}}.$$

A priority specification $\langle \mathcal{R}, \mathcal{G}, \Pi \rangle$ *equips the system with a priority assignment.*

In the prioritized setting, when a process is created, it declares both the initial node i—as in the unprioritized case before— and its initial priority from $\Pi(i)$, called the *nominal priority* of the process. Informally, a process L will run at its nominal priority, and "accelerate" to a higher priority when some higher priority process H is waiting for some resource that L holds. This will prevent processes running at intermediate priorities from making L wait and blocking H indirectly. We now define the distributed priority inheritance protocol:

(PI.1) A process maintains a *running priority*, which is initially its nominal priority.

(PI.2) Let P be a process, running at priority p, that is denied access to a resource in site r, and let Q be an active process in r running at a priority lower than p. Q and all its subprocesses set their priority to p or their current running priority, whatever is higher. We say that Q is *accelerated* to p.

(PI.3) When a (sub)process is accelerated it does not decrease its running priority until completion.

(PI.2) may require sending acceleration messages for subprocesses running in remote sites. (PI.2) does not require changing the priority of ancestor processes since the acceleration of a caller cannot help the callee to finish earlier. (PI.3) states that a (sub)process cannot decelerate. In general, decelerations compromise deadlock freedom.

It is easy to see that a subprocess either runs at priority at least as high as any of its ancestors, or there are undelivered acceleration messages.

We now calculate the sets of possible priorities at which a call graph node can be executed. A node $n:r$ can be executed at a priority p either if $p \in \Pi(i)$ for some initial ancestor of n, or if a process can execute n running at priority lower than p and block another process running at p. This block can be produced either if there is some node in r that can be executed at p, or if some ancestor of n can block some process executing at p. Formally, the set of pairs (n, p) representing priorities p at which a node n can run is the smallest set $N^{pr} \subseteq N \times \mathcal{P}$ containing:

1. (n, p) for every n that descends from $i \rightarrow^* n$, $i \in I$ and $p \in \Pi(i)$.
2. (n, p) for every $(m, p) \in N^{pr}$ with $n \equiv_{\mathcal{R}} m$, and $(n, q) \in N^{pr}$ for some $q \geq p$, and
3. (n, p) for some ancestor $m \rightarrow^+ n$ that can also run at p, $(m, p) \in N^{pr}$.

Example 3. Consider the call graph

and the priority assignment $\Pi(n) = \{1\}, \Pi(m) = \{2\}, \Pi(o_1) = \{3\}$. The set of potential priorities is:

n	m	o_1	o_2
$\{1\}$	$\{1, 2\}$	$\{1, 3\}$	$\{1, 2, 3\}$

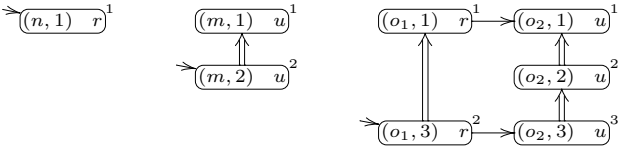
Node o_1 can run at priority 1 because o_1 resides in the same site as n and n can run at 1. Since o_2 is a descendant of o_1, o_2 can also run at 1, and since m resides in the same site as o_2, m can also run at 1. Moreover, m can run at 2, higher than o_2 running at 3, so o_2 can also run at 2. Thus $N^{pr} = \{(n, 1), (m, 1), (m, 2), (o_1, 1), (o_1, 3), (o_2, 1), (o_2, 2), (o_2, 3)\}$.

We extend the state transition system that models the global behavior of our model of distributed systems [23] with a new transition called *acceleration*. When an acceleration transition is taken, a process P running at priority p accelerates to higher priority q ($q < p$). It is easy to see that, if the priority inheritance protocol is used, and a process running in n can accelerate from priority p to q, then both (n, p) and (n, q) are in N^{pr}.

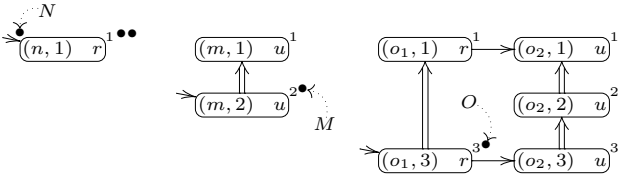
The notion of annotation can be adapted to prioritized specifications. A prioritized annotation α is a map from N^{pr} to the natural numbers. It *respects priorities* if for every two pairs (n, p) and (m, q) in N^{pr}, with $n \equiv_{\mathcal{R}} m$, $\alpha(n, p) > \alpha(m, q)$ whenever $p > q$, that is, if *higher priorities receive lower annotations*. As with unprioritized call graphs, we define an annotated call graph

by adding an edge relation $\overset{pr}{\dashrightarrow}$ connecting $(n,p) \overset{pr}{\dashrightarrow} (m,q)$ whenever n and m reside in the same site and $\alpha(n,p) \geq \alpha(m,q)$. If there is a path from (n,p) to (m,q) that contains at least a $\overset{pr}{\rightarrow}$ edge we say that (n,p) depends on (m,q), and we write $(n,p) \succ (m,q)$. An annotation is *acyclic* if no pair depends on itself.

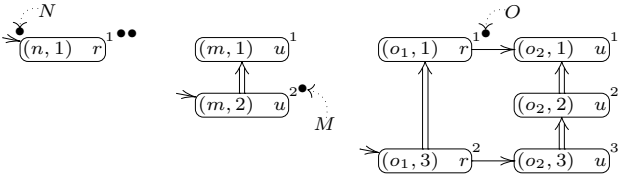
Example 4. The following diagram represents the annotated call graph of Example 3 with $\Rightarrow$ arrows representing accelerations. This annotated call graph is acyclic.



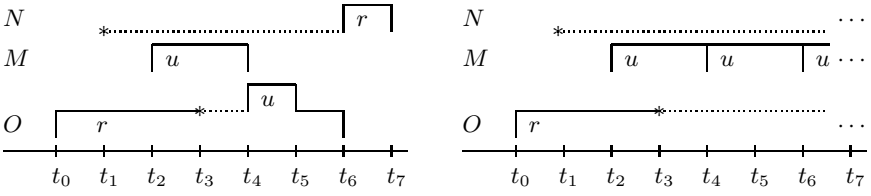
Example 5. This example shows how priority inheritance bounds the blocking time caused by priority inversions. Reconsider the annotated call graph of Example 4. Let the total number of resources be $T_r = 3$ and $T_u = 3$. Let σ be a state in which two active processes are running in n at priority 1, one active process M is running in m at priority 2, and one active process O is running in o_1 at priority 3. Thus the available resources in σ are given by $t_r = 0, t_u = 2$. Let N be a new process spawned to run in n with nominal priority 1. The resulting state is shown below.



N is blocked trying to access $(n,1)$. There is a priority inversion since O holds an r resource in o_1, while running at lower priority 3. If no acceleration is performed, then the remote call of O to o_2 is blocked until M completes, so N will be blocked indirectly by M (see Fig. 2 (a)). Even worse, if there are several processes waiting in $(m,2)$, all these processes will block O and indirectly N, causing an unbounded blocking delay (see Fig. 2 (b)). With priority inheritance in place, O inherits the priority 1 from N, and the resulting state after the acceleration is:

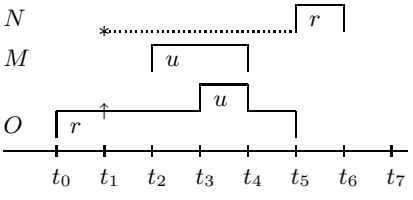


In this state, O will be granted the resource in o_2 in spite of M (and potentially other priority 2 processes waiting at m) and O will terminate, freeing the resource demanded by N. In this case the blocking time of N is bounded by the running time of O at priority 1, as shown in Fig. 3.



(a) Blocking due to Priority Inversions (b) Unbounded Blocking

Fig. 2. Time diagram (a) shows an execution of the scenario in Example 5 with blocking priority inversions. Diagram (b) shows an execution with unbounded blocking time due to priority inversions. In both diagrams, at instant t_0, process O is created, and its request for an r-resource is granted. At t_1, process N is created, but due to the existence of O and two active processes in n_1, its request is denied, indicated by $*$. At t_2, M is spawned to run m and its request for a u-resource is granted. This causes the request of O to execute o_2 at t_3 to be denied. O can only execute o_2 when some resource in u is freed. Therefore M is blocking N indirectly. In diagram (a) this blocking is restricted to the interval (t_3, t_4), while in diagram (b) the blocking delay is unbounded.



(a) No Blocking with Priority Inheritance

Fig. 3. At time t_1, process O inherits priority 1 from process N, depicted by ↑. This allows O to acquire the u-resource and execute o_2 at t_3, in spite of the existence of M or other processes trying to execute m at priority 2. Consequently, N can start executing n at t_5, with no blocking delay.

The following results hold in spite of when and how accelerations are produced:

Lemma 1. *If an annotation respects priorities, then accelerations preserve* φ.

Proof. Let P be a process that accelerates from priority p to q. If P is waiting, the result holds immediately since the global state does not change. If P is active at a node $n:r$, its annotation $\alpha(n,p) > \alpha(n,q)$ decreases. Therefore, all terms $A_r[k]$ are either maintained or decreased, and φ is preserved. □

Corollary 1. *The set of reachable states of a prioritized system that uses* Basic-P *as an allocation manager with an acyclic annotation that respects priorities is a subset of the* φ*-states.*

In the rest of this section we show that if Basic-P is used to control allocations, and accelerations are produced according to the priority inheritance protocol,

deadlocks are not reachable. When a process inherits a new priority, all its existing subprocesses, including those in other sites produced as a result of remote invocations, must accelerate as well. A message is sent to all sites where a subprocess may be running. When the message is received, if the process exists, then it is accelerated. If it does not, the acceleration is recorded as a future promise. We first show deadlock-freedom if all acceleration requests are delivered immediately with global atomicity. Then, we complete the proof for asynchronous delivery in general.

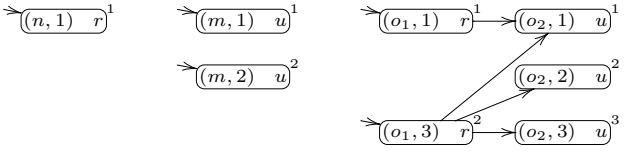
3.1 Priority Inheritance with Global Atomic Accelerations

Given a prioritized system $\mathcal{S}$ we build an (unprioritized) system and show that if $\mathcal{S}$ has reachable deadlocks so does the derived one.

Definition 3 (Flat call graph). *Given a priority specification $\langle \mathcal{R}, \mathcal{G}, \Pi \rangle$, a flat call graph is $\mathcal{G}^\flat : \langle N^{pr}, \xrightarrow{pr}, I^{pr} \rangle$, where N^{pr} is the set of potential priorities, there is an edge $(n,p) \xrightarrow{pr} (m,q)$ if $p \geq q$ and $n \to m$ occurs in the original call graph, and $(i,p) \in I^{pr}$ if i is initial in $\mathcal{G}$.*

We use $\mathcal{S}^\flat : \langle \mathcal{R}, \mathcal{G}^\flat \rangle$ for the (unprioritized) flat system that results from the flat call graph. It is easy to see that the set of reachable states of a process (the resources and running priorities of the process and each of its active subprocesses) is the same in a system and in its flat version. Moreover, if an annotation α of a prioritized specification is acyclic and respects priorities then α, when interpreted in the flat call graph, is acyclic.

Example 6. The flat call graph for the annotated specification in Example 3 is



Theorem 3. *Given a prioritized system $\mathcal{S}$ and an acyclic annotation that respects priorities, every global state reachable by $\mathcal{S}$ is also reachable by $\mathcal{S}^\flat$, if* Basic-P *is used as an allocation manager.*

Proof. Follows directly from Corollary 1 and Theorem 2. □

It is important to note that Theorem 3 states that for every sequence of requests and accelerations that leads to state σ in $\mathcal{S}$, there is a—possibly different but also legal— sequence that leads to σ in $\mathcal{S}^\flat$. Theorem 3 does not imply, though, that every transition in $\mathcal{S}$ can be mimicked in $\mathcal{S}^\flat$, which is not the case in general for accelerations. A consequence of Theorem 3 is that deadlocks are not reachable in $\mathcal{S}$, since the same deadlock would be reachable in $\mathcal{S}^\flat$, which is deadlock free by the Annotation Theorem.

Corollary 2. *If α is an acyclic annotation that respects priorities, and* Basic-P *is used as a resource allocation manager in every node, then all runs of $\mathcal{S}$ when accelerations are executed in global atomicity are deadlock free.*

The following section shows that the requirement for global atomicity in the previous corollary is actually not necessary.

3.2 Priority Inheritance with Asynchronous Communication

When an arbitrary asynchronous communication subsystem is assumed, with no guarantees of globally atomic delivery of messages, the proof of deadlock freedom is more challenging. In this case, the flat system does not directly capture the reachable states of the system with priorities, since subprocesses may accelerate later than their ancestors.

Theorem 4 (Annotation Theorem for Prioritized Specifications). *If α is an acyclic annotation that respects priorities, and* Basic-P *is used as a resource allocation manager for every call graph node, then all runs of $\mathcal{S}$ are deadlock free.*

Proof (sketch). Assume, by contradiction, that deadlocks are reachable, and let σ be a state in which a set $\{P\}$ of processes forms a deadlock. Note that σ need not be a reachable state of the flat system $\mathcal{S}^\flat$. Consider an arbitrary continuation of the run, and let σ' be the first state in which there is no undelivered message containing an acceleration of a process in the set $\{P\}$. Such a state exists since the set of possible accelerations is finite and all messages are eventually delivered. In σ' if all the processes not involved in the deadlock (i.e., not in $\{P\}$) return their resources the resulting state becomes a φ-state, and therefore some process (in $\{P\}$) can progress if Basic-P is used as an allocation manager. This contradicts the assumption that σ is a deadlock state. □

4 Conclusions and Further Work

We have presented a distributed priority inheritance protocol built using a deadlock avoidance mechanism, and proved its correctness. This protocol involves less communication overhead than a distributed PCP, since inversions can be detected locally, while PCP requires a global view of the resources allocated. Our approach enables the calculation of bounds on the maximum blocking times, which is necessary for schedulability analysis.

Message Complexity: The message complexity of a naïve implementation of the priority inheritance protocol described here is given by the number of different sites of the set of descendant nodes, which in the worst case is $|\mathcal{R}|$. However, this communication is one-way, in the sense that once the message is sent to the network, the local process can immediately accelerate, increasing its running priority. Moreover, broadcast can be used when available. Also, under certain

semantics for remote calls this worst case bound can be improved. For example, with synchronous remote calls (the caller is blocked until the remote invocation returns), one can build, using a pre-order traversal of the descendant sub-tree, an order on the visited sites. Then, a binary search on this order can be used to find the active subprocess where the nested remote call is executing. This gives a worst case ($log\ |\mathcal{R}|$) upper-bound on the number of messages needed for each priority inheritance.

Dynamic Priorities: Most dynamic priority scheduling algorithms, like EDF, require querying for the current status of existing processes to define their relative priorities. Our priority inheritance mechanism can be used with dynamic priorities if there is some static discretization of the set of priorities that processes may run at. To ensure the correctness of the priority inheritance protocol shown here, subprocesses must only increase (never decrease) their priorities while running. Note that in this kind of scheduling algorithm, accelerations would not only be caused by a priority inversion but also by the decision of a process to increase its priority to meet a deadline.

References

1. Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, February 1991.
2. Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering (ICSE '76)*, pages 562–570, Los Alamitos, CA, 1976. IEEE Computer Society Press.
3. Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.
4. Giorgio C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, New York, NY, 2005.
5. Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
6. Maria Pia Fanti and MengChu Zhou. Deadlock control methods in automated manufacturing systems. *IEEE Transactions on Systems, Man and Cybernetics–Part A: Systems and Humans*, 34(1):347–363, January 2004.
7. Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.
8. David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles (SOSP '79)*, pages 150–162, Pacific Grove, CA, 1979. ACM Press.
9. Arie N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12:373–377, 1969.
10. James W. Havender. Avoiding deadlock in multi-tasking systems. *IBM Systems Journal*, 2:74–84, 1968.
11. C. J. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

12. Mamory Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.
13. Frank Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *Proceedings of 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 340–349, Phoenix, AZ, December 1999. IEEE Computer Society.
14. Mohamed Naimi, Michel Trehel, and André Arnold. A $log(n)$ distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.
15. Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A priority inheritance approach*. Kluwer Academic Publishers, 1991.
16. Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, February 1989.
17. Michel Raynal. *Algorithms for mutual exclusion*. The MIT Press, 1986.
18. Spyros A. Reveliotis. *Real-time Management of Resource Allocation Systems: a Discrete Event Systems Approach*. International Series In Operation Research and Management Science. Springer, 2005.
19. Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.
20. César Sánchez and Henny B. Sipma. Reachable state spaces of distributed deadlock avoidance algorithms. Technical Report REACT-TR-2006-01, Stanford Computer Science, REACT Group, June 2006. Available from `http://theory.stanford.edu/~cesar`.
21. César Sánchez, Henny B. Sipma, Zohar Manna, and Christopher Gill. Efficient distributed deadlock avoidance with liveness guarantees. In *To appear in the Proceedings of the 6th Annual ACM Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, 2006. ACM Press.
22. César Sánchez, Henny B. Sipma, Zohar Manna, Venkita Subramonian, and Christopher Gill. On efficient distributed deadlock avoidance for distributed real-time and embedded systems. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodas, Greece, 2006. IEEE Computer Society Press.
23. César Sánchez, Henny B. Sipma, Venkita Subramonian, Christopher Gill, and Zohar Manna. Thread allocation protocols for distributed real-time and embedded systems. In Farn Wang, editor, *25th IFIP WG 2.6 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'05)*, volume 3731 of *LNCS*, pages 159–173, Taipei, Taiwan, October 2005. Springer-Verlag.
24. Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha S. Gokhale. Alleviating priority inversion and non-determinism in real-time CORBA ORB core architectures. In *Proc. of the Fourth IEEE Real Time Technology and Applications Symposium (RTAS'98)*, pages 92–101, Denver, CO, June 1998. IEEE Computer Society Press.
25. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.
26. Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
27. Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. McGraw-Hill, Inc., New York, NY, 1994.

28. Julien Sopena, Luciana Arantes, Marin Bertier, and Pierre Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *Proceedings of Euro-Par'05*, volume 3648 of *LNCS*, pages 654–663, Lisboa, Portugal, 2005. Spring-Verlag.
29. Pradip K. Srimani and Sunil R. Das, editors. *Distributed Mutual Exclusion Algorithms*. IEEE Computer Society Press, 1992.
30. Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, November 1985.
31. Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

Safe Termination Detection in an Asynchronous Distributed System When Processes May Crash and Recover

Neeraj Mittal[1], Kuppahalli L. Phaneesh[1,⋆], and Felix C. Freiling[2]

[1] Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA
[2] Department of Computer Science, University of Mannheim, D-68131 Mannheim, Germany

Abstract. The termination detection problem involves detecting whether an ongoing distributed computation has ceased all its activities. We investigate the termination detection problem in an asynchronous distributed system under crash-recovery model. It has been shown that the problem is impossible to solve under crash-recovery model in general. We identify two conditions under which the termination detection problem can be solved in a safe manner. We also propose algorithms to detect termination under the conditions identified.

1 Introduction

The termination detection problem arises when a distributed computation terminates *implicitly*, that is, once the computation ceases all its activities, no single process knows about the termination [1]. Therefore a separate algorithm has to be run to detect termination of the computation. To abstract from concrete applications in message-passing systems, the distributed computation is typically modeled using the following four rules. First, a process is either *active* or *passive*. Second, a process can send a message only if it is active. Third, an active process may become passive at any time. Fourth, a passive process may become active only on receiving a message. Intuitively, an active process is involved in some local activity, whereas a passive process is idle. Roughly speaking, a termination detection algorithm must detect termination once the computation which follows these rules has ceased all its activities.

Termination detection has been studied quite extensively for the last few decades, initially under the failure-free model (*e.g.*, [2,3,4,5,6], see [7] for a survey). When both processes and channels are reliable, the termination condition for a distributed computation can be defined as follows [2,3]: A computation is said to have terminated if all processes have become passive and all channels have become empty.

Termination detection has been studied relatively well in the crash-stop model as well (*e.g.*, [8,9,10,11,12]). In the crash-stop model, once a process crashes, it

⋆ The author is currently working at Microsoft Inc., Seattle, Washington, USA.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 126–141, 2006.

ceases all its activities. Moreover, any message in-transit towards a crashed process can be ignored because the message cannot initiate any new activity. Therefore, the termination condition for a distributed computation can be defined as follows [8,9]: A computation is said to have terminated if all up processes have become passive and all channels towards up processes have become empty.

Wu *et al.* [13] establish that, to be able to detect termination in the crash-stop model, it must be possible to *flush* the incoming channel of an up process with a down process. A channel can be flushed using either *return-flush* [8] or *fail-flush* [9] primitive. Both primitives allow an up process to ascertain that its incoming channel with the crashed process has become empty. In the absence of the two primitives, Tseng suggests *freezing* the channel from a down process to an up process [10]. When an up process freezes its channel with a down process, any message that arrives after the channel has been frozen is ignored. (A process can freeze a channel only after detecting that the process at the other end of the channel has crashed.) In this case, a computation is said to have terminated if all up processes have become passive, all channel between up processes have become empty, all channels from down processes to up processes have been frozen [10,11,12].

In this paper, we investigate the termination detection problem under a more severe failure model, namely *crash-recovery* model. In the crash-recovery model, processes can crash and later recover from a predefined state. To our knowledge, Majuntke [14] was the first to give a definition of termination in the crash-recovery model. Majuntke [14] shows that, if processes can restart in active state on recovery, then it is impossible to detect termination without the ability to predict future behavior of processes (*e.g.*, whether a crashed process will remain crashed forever or will recover in the future). The impossibility result holds even if a process can restart in an active state only if it crashed in an active state. Majuntke [14] also presents a *stabilizing* termination detection algorithm under the condition that there is no process that crashes and recovers an infinite number of times. The algorithm is stabilizing in the sense that it may falsely announce termination and revoke it later. However, false termination announcements and revocations can happen only a finite number of times even if the underlying computation never terminates [14].

Our focus is on developing non-stabilizing safe termination detection algorithms in the crash-recovery model, that is, unlike in [14], our termination detection algorithms are not allowed to revoke a termination announcement (even if the revocation occurs only a finite number of times). We identify two conditions under which termination of a computation can be detected in a *safe* manner, that is, it is possible to devise a termination detection algorithm that never announces false termination.

1. The first condition requires every process to be *eventually reliable*, that is, every process eventually stays up permanently.
2. The second condition requires a crashed process to *always restart in a passive* state (and rejoin the computation via a recovery operation). Further, a process can deliver an application message only if it is sent to its *current*

incarnation, that is, only if the sender is aware of all restarts of the destination process. We ensure the latter by allowing a process to deliver an application message only if the message is exchanged between current incarnations of the two processes (source and destination).

We present an algorithm to detect termination under each of the conditions. The second algorithm uses a new failure detector suitable to solve the termination detection problem in the crash-recovery model. Due to lack of space, proofs of all lemmas and theorems, and formal descriptions of the two algorithms can be found elsewhere [15].

The paper is organized as follows. We present our system model and notations in Sect. 2, derive a definition of a perfect failure detector for the crash-recovery model in Sect. 3 and formally define the termination detection problem in Sect. 4. We identify the two conditions for safe termination detection in Sect. 5. The algorithms for termination detection under the two conditions are described in Sect. 6 and Sect. 7. Finally, we present our conclusions and outline directions for future research in Sect. 8.

2 Model and Notation

2.1 Distributed System

We assume an asynchronous distributed system consisting of a set of processes, given by $\Pi = \{p_1, p_2, \ldots, p_N\}$, in which processes communicate by exchanging messages with each other over a communication network. A process changes its state by executing an *event*. The system is asynchronous in the sense that there is no bound on the amount of time a process may take to execute an event or a message may take to arrive at its destination. We do not assume any global clock or shared memory.

There are three kinds of events in the system: *internal event*, *send event* and *receive event*. An event at a process causes the state of the process to be updated. Additionally, a send event causes one or more messages to be sent, whereas a receive event causes a message to be received. Sometimes, we refer to the state of a process as *local state* and the state of a system as *global state*.

We assume that a process executes events sequentially. Therefore events on a process are totally ordered. However, events on different processes are only ordered partially. The partial order between events in the system is given by the *Lamport's happened-before relation* [16] defined as follows. An event e is said to have happened-before an event f, denoted by $e \rightarrow f$, if

- e and f are events on the same process and e was executed before f, or
- e and f are send and receive events, respectively, of the same message, or
- there exists an event g such that $e \rightarrow g$ and $g \rightarrow f$.

We use $\underline{\rightarrow}$ to denote the reflexive closure of $\rightarrow$.

2.2 Failure Model

We assume that processes are unreliable and may fail by crashing. Further, a crashed process may subsequently recover and resume its operation. While a process is crashed, it does not execute any events. This failure model is referred to as *crash-recovery model*.

In the crash-recovery model, a process may be either *stable* or *unstable*. A process is said to be stable if it crashes (and possibly recovers) only a finite (including zero) number of times; otherwise it is unstable. A stable process can be further classified into two categories: *eventually-up* or *eventually-down* [17]. A process is said to be eventually-up if the process eventually stays up after crashing and recovering a finite number of times; otherwise it is eventually-down. An eventually-up process is said to be *always-up* if it never crashes. Sometimes, eventually-up processes are referred to as *good processes*, and eventually-down and unstable processes are referred to as *bad processes* [17].

A process that is currently operational is called an *up* process, whereas a process that is currently crashed is called a *down* process. We use the phrases "up process" and "live process" interchangeably. Likewise, we use the phrases "down process" and "crashed process" interchangeably.

In the crash-recovery model, in addition to processes, typically, channels are also assumed to be unreliable. We assume *eventually-reliable channels with finite duplication* in this paper. Such channels satisfy the following properties:

- *No creation:* p_j delivers a message m only if m was sent earlier by p_i,
- *Finite duplication:* p_j delivers a message only a finite number of times, and
- *Eventual-reliability:* If p_i sends a message m to p_j, and neither p_i nor p_j crashes, then p_j eventually delivers m.

An eventually-reliable channel with finite duplication can be implemented on top of a *fair-lossy channel* [18]—a type of unreliable channel providing very weak guarantees—using retransmissions and acknowledgments. We refer to a channel as eventually-reliable if it satisfies no creation, no duplication and eventual-reliability properties. Unless otherwise stated, we assume all channels to be eventually-reliable with finite duplication.

2.3 Volatile and Stable Storage

We assume that each process has access to two types of storage mediums: *volatile storage* and *stable storage*. Any data that a process maintains in volatile storage, such as main memory, is lost once the process crashes. On the other hand, data stored in stable storage, such as magnetic disk, is persistent and survives any crashes. However, this persistence comes at the expense of speed. Accessing (reading/writing) stable storage is much slower than accessing volatile storage. As a result, it is desirable to minimize access to stable storage so as to avoid slowing down the system significantly.

2.4 Process Incarnations

When a crashed process recovers, we say that the process has a *new incarnation*. At the very least, we use stable storage to distinguish between various incarnations of the same process. Each process maintains an integer in its stable storage that keeps track of its incarnation number, that is, the number of times the process has crashed and recovered. The integer is initially set to 0 for all processes. Whenever a process recovers from a crash, before taking any other action, it reads the value of the integer from its stable storage, increments the value and writes the incremented value back to its stable storage. Observe that it is possible that a process may crash before it is able to write the incremented value back to its stable storage. Clearly, such a recovery is useless for all practical purposes. Therefore we consider a process to be down until it is able to successfully update its incarnation number in its stable storage.

If a process p_i crashes and the incarnation number of p_i immediately before the crash was x, then we say that "incarnation x of p_i has crashed". It is convenient to view process crash and recovery as special kinds of events, namely *crash event* and *recovery event*. We use $crash_i(x)$ (respectively, $recovery_i(x)$) to denote the crash event (respectively, recovery event) for incarnation x of process p_i. We refer to crash and recovery events as *operational events* (as opposed to *program events* that processes execute to change their states). The happened-before relation can be extended to include operational events as well.

We denote the *operational state* of a process (as opposed to *program state* which captures the values of all program variables on the process) using a tuple $\langle s, x\rangle$ containing two components. The first component, given by s, indicates the status of the process, that is whether the process is up or down. The second component, given by x, indicates the most recent incarnation number of the process. The formal interpretation of the tuple $\langle s, x\rangle$ is as follows:

- If $s = \mathsf{up}$, then the process is currently up and its current incarnation number is x.
- If $s = \mathsf{down}$, then the process is currently down and the most recent incarnation of the process to have crashed is x.

We assume that $\mathsf{up} < \mathsf{down}$. We can now define a less-than relation on operational states of a process as follows: $\langle s, x\rangle < \langle t, y\rangle$ if either (1) $x < y$, or (2) $x = y$ and $s < t$. Observe that the less-than relation as defined totally orders all operational states of a process. As before, $\leq$ is a reflexive closure of $<$. For an operational state $u = \langle s, x\rangle$, we use $u.status$ to refer to the status s and $u.number$ to refer to the incarnation number x. Let $opstate_i(t)$ denote the operational state of process p_i at time t.

Note that it is possible to avoid delivering duplicate messages during an incarnation without using stable storage by logging received messages in volatile storage only.

3 Failure Detector for Termination Detection

To solve many important distributed computing problems such as consensus, atomic broadcast and termination detection in an unreliable asynchronous distributed system, it is sometimes necessary for an up process to know the current status (up or down) of other processes in the system. However, in an asynchronous distributed system, it is not possible to distinguish between a down process and a slow process. To overcome this problem, many solutions to these problems assume the existence of a special device known as *failure detector* [19]. Using a failure detector, a process can maintain its view about the current status (up or down) of other processes in the system. This view might be unreliable and, at any given time, the views at different processes may be different as well. For a failure detector to be useful, these views should eventually be "error-free" and "converge" at good processes. A failure detector can be implemented by making timing assumptions about speeds of processes and delays of messages [19,20]. The notion of failure detector was originally defined for the crash-stop model (once a process crashes, it never recovers) [19] but has been extended to the crash-recovery model as well (see for example [17]). In this paper, we focus on *realistic failure detectors* which are not capable of predicting the future behavior of a process (*e.g.*, whether a process will stay up forever) [21].

One of the termination detection algorithms we describe in this paper uses a *perfect failure detector* [19] adapted to the crash-recovery model. Informally, a perfect failure detector for the crash-recovery model is responsible for detecting crashes of process incarnations. It satisfies the following properties: (1) *Strong Accuracy:* a process suspects a process incarnation to have crashed only after the incarnation has crashed, and (2) *Strong Completeness:* if a process incarnation has crashed, then eventually every good process permanently suspects the incarnation to have crashed.

The completeness property as stated above is hard to implement in practice. A process may crash immediately after it has updated its incarnation number in the stable storage (but before sending any messages) and no other process in the system will know about the recovery (and hence about the incarnation). Clearly, it is *unreasonable* to expect another process to be able to detect crash of such an incarnation. To address this problem, we define what it means for a process to *know-about* an incarnation. We say a process p_i knows-about the incarnation x of process p_j if there exists an event e on p_i such that $recovery_j(x) \underline{\rightarrow} e$. We assume that each process knows-about incarnation 0 of every other process.

Based on the above discussion, we modify the completeness property as follows. It now consists of two parts. First, if some always-up process knows-about a process incarnation and the incarnation has crashed, then eventually every good process permanently suspects the incarnation to have crashed. Second, if some good process permanently suspects a process incarnation to have crashed, then eventually every good process permanently suspects the incarnation to have crashed. We formally model this behavior as follows. The local failure detector at each process p_i maintains a list, denoted by $crash\text{-}list_i$, that contains all process incarnations it suspects to have crashed. Each entry in the list is of the

form $\langle i, x\rangle$, which means that incarnation x of process p_i has crashed. Observe that, in practice, it is sufficient for the local failure detector to maintain at most one entry in the list for every process in the system, which corresponds to the *latest* incarnation of the process that it suspects to have crashed. We assume that $crash\text{-}list_i$ is prefix-closed, that is, if $\langle j, x\rangle \in crash\text{-}list_i$ and $x \geq 1$, then $\langle j, x-1\rangle \in crash\text{-}list_i$.

Let $crash\text{-}list_i(t)$ denote the list at process p_i at time t. We assume that if p_i is down at t, then $crash\text{-}list_i(t) = \emptyset$. A perfect failure detector satisfies the following properties:

- *Strong Accuracy:* A process suspects a process incarnation to have crashed only if the incarnation has actually crashed. Formally, for all processes p_i and p_j,
$$\langle j, x\rangle \in crash\text{-}list_i(t) \;\Rightarrow\; \langle \mathsf{down}, x\rangle \leq opstate_j(t)$$
- *Strong Completeness:* It consists of two parts:
  1. If at least one always-up process knows-about a process incarnation, and the incarnation has crashed, then eventually every good process permanently suspects the incarnation to have crashed. Formally, for every good process p_i and for every process p_j,
$$\begin{gathered}(\text{if some always-up process knows about } recovery_j(x)) \wedge \\ (\langle \mathsf{down}, x\rangle \leq opstate_j(t)) \\ \Rightarrow \\ \langle \exists u :: \langle \forall v : v \geq u : \langle j, x\rangle \in crash\text{-}list_i(v)\rangle\rangle\end{gathered}$$
  2. If some good process permanently suspects a process incarnation to have crashed, then eventually every good process permanently suspects the incarnation to have crashed. Formally, for all good process p_i and p_k and for every process p_j,
$$\begin{gathered}\langle \forall w : w \geq t : \langle j, x\rangle \in crash\text{-}list_k(w)\rangle \\ \Rightarrow \\ \langle \exists u :: \langle \forall v : v \geq u : \langle j, x\rangle \in crash\text{-}list_i(v)\rangle\rangle\end{gathered}$$

The accuracy and completeness properties guarantee that if there are no unstable processes in the system, then eventually all good processes agree on which process incarnations have crashed. Our definition of a perfect failure detector allows a process to "lose" its knowledge about crashes of other processes, especially due to its own crash. We do assume, however, that, once a process suspects a process incarnation to have crashed, it continues to do so until it crashes. This can be easily achieved using volatile storage only.

4 The Termination Detection Problem

There are many distributed programs which, when executed, generate distributed computations that do not terminate explicitly but rather terminate *implicitly* [1].

In other words, when the computation terminates, it is possible that no process in the system knows that the computation has terminated. In this case, a separate *termination detection algorithm* has to be run to detect termination of the distributed computation. The distributed computation whose termination has to be detected is typically modeled using the states *active* and *passive* for processes and the rules mentioned in the introduction.

The termination detection problem involves determining whether the computation has ceased all its activities. In other words, no process is currently involved in any activity, and, moreover, no process can become involved in any activity in the future. Any termination detection algorithm should satisfy the following properties:

- *No false termination announcement (safety):* If the termination detection algorithm announces termination, then the computation has indeed terminated.
- *Eventual termination announcement (liveness):* Once the computation terminates, the termination detection algorithm eventually announces termination.

For every failure model, it is necessary to define what it means that a computation has terminated. In the crash-recovery model, a process may recover after crashing and resume its activity. Clearly, if a process, on recovery, can restart in any state—active or passive, then, once the termination condition becomes true, no process can crash thereafter. Otherwise, the termination condition can be simply falsified by a process crash and its subsequent recovery in an active state. This definition is too restrictive. Therefore, we assume that a process can restart in an active state on recovery only if it crashed in active state; otherwise, it restarts in a passive state. A process is said to be *forever-down* if it is currently crashed and never recovers from the crash. The termination condition for a distributed computation can be defined as [14]:

Definition 1 (termination in crash-recovery model). *A computation is said to have terminated in the crash-recovery model if every process that is not forever-down has become passive, and every channel towards such a process has become empty.*

Observe that the termination condition in the crash-recovery model, as stated above, requires a failure detector to be able to predict the future behavior of a down process, namely whether a down process will recover in the future or stay down permanently. In fact, Majuntke shows in [14] that it is impossible to detect termination of a computation in the crash-recovery model without using a non-realistic failure detector. However, the definition is still reasonable since Majuntke [14] also proves that the above definition of termination is equivalent to the condition that termination is a *stable property.*

In the next section, we investigate conditions under which it is possible to detect termination using only a realistic failure detector such as the one defined in Sect. 3. In contrast to [14], our focus is on deriving termination detection

algorithms that are (perpetually) safe and not eventually safe. Specifically, if the termination detection algorithm announces termination, then the computation has, in fact, terminated.

To avoid confusion, we refer to messages exchanged by a distributed computation as *application messages* and those exchanged by a termination detection algorithm as *control messages*.

5 Conditions for Safe Termination Detection

One of the reasons why detecting termination in the crash-recovery model is hard is because a crashed process, on recovery, may restart in an active state.

Lemma 1. *Assume that: (1) a crashed process, on recovery, may restart in an active state provided it failed in an active state, and (2) at most one process in the system is bad. Then there is no termination detection algorithm that can detect termination of every computation in a safe and live manner.*

Lemma 1 implies that, to be able to detect termination of a computation in a safe manner, we have to weaken at least one of two assumptions, that is, either (1) a crashed process, on recovery, always restarts in a passive state, or (2) all processes in the system eventually stay up permanently. We consider the two one by one. First, assume that processes in the system eventually stay up forever, that is, all processes are eventually reliable. In this case, the termination condition for a distributed computation in the crash-recovery model becomes equivalent to that in the failure-free model. Therefore the first condition under which we investigate the termination detection problem is:

Condition 1 (eventually reliable processes). *All processes in the system are good processes.*

Next, assume that a process, on recovery, always restarts in a passive state. Intuitively, this means that a process, on recovery, cannot start any activity on its own but has to wait to receive an application message from another process. Therefore the second condition under which we investigate the termination detection problem is:

Condition 2 (passive recovery). *A crashed process, on recovery, always restarts in a passive state.*

However, the above condition, by itself, does not solve the problem completely. For a message m, let $snd(m)$ and $rcv(m)$ denote the send and receive events, respectively, of m. Suppose process p_i sends an application message m to process p_j. We say that m is *old* with respect to incarnation x of p_j, where $x \geq 1$, if $recovery_j(x) \not\rightarrow snd(m)$. In other words, when p_i sent m, it did not know-about incarnation x of p_j. We show that such an old application message may create a problem for a termination detection algorithm.

Lemma 2. *Assume that: (1) a crashed process, on recovery, always restarts in a passive state, (2) at most two process in the system are bad, and (3) a process can accept an old application message. Then there is no termination detection algorithm that can detect termination of every computation in a safe and live manner.*

The main idea behind the proof of Lem. 2 is as follows. To tolerate eventually-down processes, it is not sufficient to ensure that all channels towards up processes are empty before announcing termination. It may also be necessary to ensure that all channels between down processes are empty (unless, of course, all down processes stay down permanently which requires knowledge about the future). Clearly, it is reasonable to assume that the channel from p_i to p_j can be tested for emptiness only by either p_i or p_j and not by any third process. To address this problem, we take an approach that is analogous to freezing of a channel in the crash-stop model.

The main difference is that instead of freezing channels between processes, we now freeze channels between *process incarnations*. Specifically, if a process suspects a process incarnation to have crashed, then it stops accepting application messages from that incarnation. Further, it only accepts those application messages that are sent to its current incarnation. To implement freezing of a channel between process incarnations, each process has to maintain its view of the most recent incarnation of other processes in the system. This can be accomplished by maintaining a vector analogous to Fidge/Mattern's vector clock [22,23]. We refer to this vector as *view vector*. The vector for process p_i, denoted by $view_i$, maintains the operational states of all processes in the system *as per* p_i*'s view*. The vector is piggybacked on every message (application as well as control) a process sends. As in the case of vector clock, a process, on receiving a message, updates its vector by taking a component-wise maximum of its vector and the vector received. Additionally, a process updates its vector on recovery and on detecting a crash. Like vector clocks, two view vectors are compared component-wise. Figure 1 describes the actions for modifying view vector.

For a program event e on process p_i, we use $e.view$ to denote the view vector value on p_i immediately after executing e. Note that, since a process has up-to-date knowledge about its own operational state, $e.view[i]$ represents the operational state of p_i immediately after executing e. If e is not a program event (that is, it is a crash or recovery event of p_i), we define the i^{th} entry of $e.view$, given by $e.view[i]$, as the operational state of p_i immediately after executing e. For instance, if $e = crash_i(x)$ for some x, then $e.view[i] = \langle \mathsf{down}, x \rangle$. Likewise, if $e = recovery_i(x)$ for some x, then $e.view[i] = \langle \mathsf{up}, x \rangle$. All other entries of $e.view$ are assumed to be set to their lowest values. Specifically, the j^{th} entry of $e.view$ with $j \neq i$ has the value $\langle \mathsf{up}, 0 \rangle$. Clearly, the i^{th} entry of the view vector of process p_i is monotonically non-decreasing even across crashes and recoveries.

We assume that the view vector of a process is stored in volatile storage but may be flushed to stable storage periodically while the process is up. Therefore, the view vector of a process is monotonically non-decreasing as long as the process does not crash.

Rules for updating view vector on process p_i:

Variables:
$view_i$: vector $[1..n]$ of operational states;

(A0) Initialization:
for each j in $[1, n]$ **do** $view_i[j] := \langle \mathsf{up}, 0 \rangle$; **endfor**;

(A1) On sending a message m:
piggyback $view_i$ on m;

(A2) On receiving a message m carrying view vector:
for each j in $[1, n]$ **do**
$view_i[j] := \max\{view_i[j], m.view[j]\}$;
endfor;

(A3) On detecting crash of incarnation x of process p_j:
$view_i[j] := \max\{view_i[j], \langle \mathsf{down}, x \rangle\}$;

(A4) On starting new incarnation x after recovery:
$view_i[i] := \langle \mathsf{up}, x \rangle$;
// other entries of $view_i$ may be initialized using stable storage, if applicable

Fig. 1. Rules for updating view vector on a process

For a message m, let $m.view$ denote the vector piggybacked on m. We say that p_i believes p_j to be currently up if $view_i[j].status = \mathsf{up}$. We now formally define what it means to freeze a channel between two process incarnations.

Condition 3 (channel freezing). *Consider an application message m sent by process p_i to process p_j. Then p_j accepts m if and only if both the following conditions hold:*

1. $view_j[j] = m.view[j]$ *and*
2. $view_j[i] \leq m.view[i]$.

We present two algorithms for safe termination detection. The first algorithm detects termination when Cond. 1 holds. The second algorithm detects termination when Cond. 2 and Cond. 3 hold.

6 Termination Detection with Eventually Reliable Processes

In this section, we present a termination detection algorithm assuming eventually reliable processes.

As explained before, when all processes are eventually reliable, detecting termination of a distributed computation becomes equivalent to detecting that all processes are passive and all channels are empty. In the crash-free model, testing whether a channel is empty is relatively easy. To test whether a channel from process p_i to process p_j is empty, it is sufficient to test that the number of messages that p_i has sent to p_j so far *is equal to* the number of messages that p_j has

received from p_i so far. However, in the crash-recovery model, a message that p_i sends to p_j may arrive at p_j while p_j is down and is, therefore, lost. As a result, when comparing p_i and p_j's states, if p_j is missing a message sent to it by p_j, we cannot distinguish between the case when the message has been lost and the case when the message has been simply delayed.

Therefore, to detect termination, we need some other mechanism to test for emptiness of a channel. To that end, we define a special operation on a channel, which we refer to as *flush*. A flush operation is defined using two events: start_flush and end_flush. A process p_i initiates a flush operation on its outgoing channel with another process, say process p_j, by executing the start_flush event. A flush operation initiated by p_i ends when p_i executes a matching end_flush event. A flush operation should satisfy the following two properties:

- *No old message delivery after flush (safety):* Once p_i executes an end_flush event, p_j does not deliver any application message that p_i sent before executing the corresponding start_flush event.
- *Eventual flush completion (liveness):* If neither p_i nor p_j crashes, then eventually p_i executes a matching end_flush event.

We provide an implementation of the flush operation later in this section. We now describe a scheme that enables a process to test if the underlying computation has terminated. The scheme consists of two phases. In the first phase, the process, which is testing for termination, requests all processes to flush their outgoing channels and also send their local states to it. A process sends a local state of passive if it is passive at the time of receiving the request and stays passive until all its outgoing channels have been flushed; otherwise it sends a local state of active. If local states of all processes indicate that all processes are passive, then the scheme proceeds to the second phase. In the second phase, the process again contacts all processes to determine if any one of them became active since sending its previous response. If no such process exists and no process fails during the entire execution of the scheme, then the process infers that the computation has terminated. We prove that the scheme is safe, that is, a process detects termination only if the computation has terminated.

To ensure liveness, a process uses an instance of the scheme to test whether the computation has terminated whenever it becomes passive or recovers from a crash. We show that once the computation terminates, some process eventually detects termination. Different instances of the scheme are differentiated using an *instance identifier*, which consists of (1) the identifier of the initiating process, (2) its incarnation number and (3) a sequence number. The sequence number helps differentiate between various instances of the scheme initiated by the same incarnation of a process. The sequence number can be stored in the volatile storage. We refer to the termination detection algorithm described in this section as TDA-ER. We show that:

Theorem 1 (TDA-ER is safe and live). *If TDA-ER announces termination, then the computation has already terminated. Further, once the computation terminates, TDA-ER eventually announces termination.*

Let R denote the sum of (1) the number of active-to-passive transitions in the computation and (2) the number of recovery events in the execution. Then there are at most R invocations of the testing scheme in total.

6.1 Implementing Flush Operation

To implement flush operation, we assume that all channels are eventually reliable (no duplication) and, moreover, satisfy FIFO property. On initiating a flush operation on an outgoing channel (that is, on executing a start_flush event), a process sends a flush message to the neighbor of the channel. The neighbor, on receiving the flush message, sends an acknowledgment message back to the process. On receiving the acknowledgment message, the process executes the end_flush event.

Another way to implement the flush operation is to use stable storage. A process logs every application message it sends and receives in stable storage. Further, it periodically retransmits every message in stable storage until it receives an acknowledgment for it. When a flush operation is initiated, it executes the end_flush event once all messages sent before the start_flush event have been acknowledged.

7 Termination Detection with Passive Recovery and Channel Freezing

In this section, we present a termination detection algorithm assuming passive recovery and channel freezing. Unlike in the previous case, in this case, a process may eventually crash and never recover. Therefore, as is usually the case, we need some kind of a failure detector to aid processes in determining the current status of other processes in the system. Specifically, we use a perfect failure detector defined in Sect. 3 to solve the termination detection problem. We do assume, however, that there is *at least one always-up process* in the system.

Due to passive recovery and channel freezing, when a crashed process recovers, it has to execute a recovery operation to rejoin the computation. Otherwise, it can never become active again. Intuitively, as part of the recovery operation, a process informs other operational processes in the system about its recovery. This serves two purposes. First, other processes can start sending it application messages which can now be accepted by the process since they will carry its latest incarnation number. Second, if the process crashes again, then the failure detector is obligated to detects its crash due to the strong completeness property.

We use the following recovery operation. A crashed process, on recovery, broadcasts a restart message to all processes in the system. It then waits to receive an acknowledgment from all those processes that it believes have not crashed even once. This ensures that at least one always-up process knows about the recovery. Note that all messages exchanged in the recovery operation (namely, restart and acknowledgment) are piggybacked with the incarnation vector of the sending process. Any application message received before the recovery operation has completed is buffered and processed later.

As in the previous algorithm, we now describe a scheme that enables a process to test if the underlying computation has terminated. The process, which is testing for termination, requests all processes in the system to send their current local states to it. The local state of a process includes: (1) the view vector, (2) the state with respect to the application, (3) the number of application messages it has sent to the latest incarnation of each process, and (4) the number of application messages it has received from the latest incarnation of each process. The process waits until it has received a local state from each process that it believes to be currently up. It then infers that the computation has terminated if both the following conditions hold:

1. all processes currently up in its view have identical view vectors, and
2. all processes currently up in its view are passive and all channels between them are empty.

We show that the scheme is safe, that is, a process detects termination only if the computation has terminated. To ensure liveness, a process uses an instance of the scheme to test whether the computation has terminated whenever it becomes passive or its view vector changes. We show that once the computation terminates, some process eventually detects termination. As before, different instances of the scheme can be differentiated using an appropriate instance identifier.

Theorem 2 (TDA-CF is safe and live). *If TDA-CF announces termination, then the computation has already terminated. Further, once the computation terminates, TDA-CF eventually announces termination.*

Let R_c denote the number of active-to-passive transitions in the computation and R_o denote the number of crash and recovery events in the execution. Then there are at most $R_c + NR_o$ invocations of the testing scheme in total.

8 Conclusions

We have identified two conditions under which the termination detection problem can be solved in a safe manner when processes can crash and recover. We have also proposed a termination detection algorithm to solve the problem under each of the two conditions.

Our algorithm for the second condition uses a perfect failure detector which is strictly stronger than the failure detector used to solve consensus in the crash-recovery model [17]. When processes do not recover after crashing, the set of assumptions for our second algorithm become *identical* to those under crash-stop model. Since a perfect failure detector is necessary to detect termination in the crash-stop model [13,12], we believe that a perfect failure detector is necessary to detect termination in the crash-recovery model as well when two or more processes may be bad. We plan to prove this rigorously in the future.

References

1. Tel, G.: Distributed Control for AI. Technical Report UU-CS-1998-17, Information and Computing Sciences, Utrecht University, The Netherlands (1998)
2. Dijkstra, E.W., Scholten, C.S.: Termination Detection for Diffusing Computations. Information Processing Letters (IPL) **11**(1) (1980) 1–4
3. Francez, N.: Distributed Termination. ACM Transactions on Programming Languages and Systems (TOPLAS) **2**(1) (1980) 42–55
4. Stupp, G.: Stateless Termination Detection. In: Proceedings of the 16th Symposium on Distributed Computing (DISC), Toulouse, France (2002) 163–172
5. Khokhar, A.A., Hambrusch, S.E., Kocalar, E.: Termination Detection in Data-Driven Parallel Computations/Applications. Journal of Parallel and Distributed Computing (JPDC) **63**(3) (2003) 312–326
6. Mittal, N., Venkatesan, S., Peri, S.: Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies. In: Proceedings of the 18th Symposium on Distributed Computing (DISC), Amsterdam, The Netherlands (2004) 290–304
7. Matocha, J., Camp, T.: A Taxonomy of Distributed Termination Detection Algorithms. The Journal of Systems and Software **43**(3) (1998) 207–221
8. Venkatesan, S.: Reliable Protocols for Distributed Termination Detection. IEEE Transactions on Reliability **38**(1) (1989) 103–110
9. Lai, T.H., Wu, L.F.: An $(N-1)$-Resilient Algorithm for Distributed Termination Detection. IEEE Transactions on Parallel and Distributed Systems (TPDS) **6**(1) (1995) 63–78
10. Tseng, Y.C.: Detecting Termination by Weight-Throwing in a Faulty Distributed System. Journal of Parallel and Distributed Computing (JPDC) **25**(1) (1995) 7–15
11. Hélary, J.M., Murfin, M., Mostefaoui, A., Raynal, M., Tronel, F.: Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. IEEE Transactions on Parallel and Distributed Systems (TPDS) **11**(9) (2000) 897–909
12. Mittal, N., Freiling, F.C., Venkatesan, S., Penso, L.D.: Efficient Reduction for Wait-Free Termination Detection in a Crash-Prone Distributed System. In: Proceedings of the 19th Symposium on Distributed Computing (DISC). (2005) 93–107
13. Wu, L.F., Lai, T.H., Tseng, Y.C.: Consensus and Termination Detection in the Presence of Faulty Processes. In: Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), Hsinchu, Taiwan (1992) 267–274
14. Majuntke, M.: Termination Detection in Systems Where Processes May Crash and Recover. Master's thesis, RWTH Aachen University (2006)
15. Mittal, N., Phaneesh, K.L., Freiling, F.C.: Safe Termination Detection in an Asynchronous Distributed System when Processes may Crash and Recover. Technical Report UTDCS-41-06, Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA (2006)
16. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM (CACM) **21**(7) (1978) 558–565
17. Aguilera, M.K., Chen, W., Toueg, S.: Failure Detection and Consensus in the Crash Recovery Model. Distributed Computing (DC) **13**(2) (2000) 99–125
18. Basu, A., Charron-Bost, B., Toueg, S.: Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. In: Proceedings of the, Workshop on Distributed Algorithms (WDAG), Bologna, Italy (1996) 105–122

19. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM **43**(2) (1996) 225–267
20. Larrea, M., Fernández, A., Arévalo, S.: On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems. IEEE Transactions on Computers **53**(7) (2004) 815–828
21. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A Realistic Look At Failure Detectors. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), Washington, DC, USA (2002) 345–353
22. Mattern, F.: Virtual Time and Global States of Distributed Systems. In: Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG). (1989) 215–226
23. Fidge, C.J.: Logical Time in Distributed Computing Systems. IEEE Computer **24**(8) (1991) 28–33

Lock-Free Dynamically Resizable Arrays

Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup

Texas A&M University
College Station, TX 77843-3112
{dechev, peter.pirkelbauer}@tamu.edu, bs@cs.tamu.edu

Abstract. We present a first lock-free design and implementation of a dynamically resizable array (vector). The most extensively used container in the C++ Standard Template Library (STL) is *vector*, offering a combination of dynamic memory management and constant-time random access. Our approach is based on a single 32-bit word atomic compare-and-swap (CAS) instruction. It provides a linearizable and highly parallelizable STL-like interface, lock-free memory allocation and management, and fast execution. Our current implementation is designed to be most efficient on multi-core architectures. Experiments on a dual-core Intel processor with shared L2 cache indicate that our lock-free vector outperforms its lock-based STL counterpart and the latest concurrent vector implementation provided by Intel by a large factor. The performance evaluation on a quad dual-core AMD system with non-shared L2 cache demonstrated timing results comparable to the best available lock-based techniques. The presented design implements the most common STL vector's interfaces, namely random access read and write, tail insertion and deletion, pre-allocation of memory, and query of the container's size. Using the current implementation, a user has to avoid one particular ABA problem.

Keywords: lock-free, STL, C++, vector, concurrency, real-time systems.

1 Introduction

The ISO C++ Standard [18] does not mention concurrency or thread-safety (though it's next revision, C++0x, will [4]). Nevertheless, ISO C++ is widely used for parallel and multi-threaded software. Developers writing such programs face challenges not known in sequential programming: notably to correctly manipulate data where multiple threads access it. Currently, the most common synchronization technique is to use mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system by diminishing its parallelism. The behavior of mutual exclusion locks can sometimes be optimized by using fine-grained locks [17] or context-switching. However, the interdependence of processes implied by the use of locks – even efficient locks – introduces the dangers of deadlock, livelock, and

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 142–156, 2006.

priority inversion. To many systems, the problem with locks is one of difficulty of providing correctness more than one of performance.

The widespread use of multi-core architectures and the hardware support for multi-threading pose the challenge to develop practical and robust concurrent data structures. The main target of our design is to deliver good performance for such systems (Section 4). In addition, many real-time and autonomous systems, such as the Mission Data Systems Project at the Jet Propulsion Laboratory [8], require effective fine-grained synchronization. In such systems, the application of locks is a complex and challenging problem due to the hazards of priority inversion and deadlock. Furthermore, the implementation of distributed parallel containers and algorithms such as STAPL [2] can benefit from a shared lock-free vector. The use of non-blocking (lock-free) techniques has been suggested to prevent the interdependence of the concurrent processes introduced by the application of locks [13]. By definition, a lock-free concurrent data structure guarantees that when multiple threads operate simultaneously on it, *some* thread will complete its task in a *finite* number of steps despite failures and waits experienced by other threads. The vector is the most versatile and ubiquitous data structure in the C++ STL [26]. It is a dynamically resizable array that provides automatic memory management, random access, and tail element insertion and deletion with an amortized cost of O(1).

This paper presents the following contributions:

(a) A first design and practical implementation of a lock-free dynamically resizable array. Our lock-free vector provides a set of linearizable [15] STL vector operations, which allow disjoint-access parallelism for random access read and write.
(b) A portable algorithm. Our design is based on the word-size compare-and-swap (CAS) instruction available on a large number of hardware platforms.
(c) A fast and space-efficient implementation. On a variety of tests executed on an Intel dual-core architecture it outperforms its lock-based STL counterpart and the concurrent vector provided by Intel by a factor of 10. The same tests executed on an 8-way AMD architecture with non-shared L2 cache indicate performance comparable to the best available lock-based techniques.
(d) An effective incorporation of non-blocking memory management and memory allocation schemes.

The rest of the paper is structured like this: 2: Background, 3: Implementation, 4: Performance Evaluation, and 5: Conclusion.

2 Background

As defined by Herlihy [13] [14], a concurrent object is *non-blocking* if it guarantees that *some* process in the system will make progress in a *finite* number of steps. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free*. Non-blocking (lock-free) and wait-free algorithms do not apply mutual exclusion locks. Instead, they rely on a set of

atomic primitives such as the word-size CAS instruction. Common CAS implementations require three arguments: a memory location, Mem, an old value, V_{old}, and a new value,V_{new}. The instruction atomically exchanges the value stored in Mem with V_{new}, provided that its current value equals V_{old}. The architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (e.g.: IA-32 [16]). The use of a hardware lock does not violate the non-blocking property as defined by Herlihy. Common locking synchronization methods such as semaphores, mutexes, monitors, and critical sections utilize the same atomic primitives to manipulate a control token. Such application of the atomic instructions introduces interdependencies of the contending processes. In the most common scenario, lock-free systems utilize CAS in order to implement a speculative manipulation of a shared object. Each contending process speculates by applying a set of writes on a local copy of the shared data and attempts to CAS the shared object with the updated copy. Such an approach guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps.

2.1 Pragmatic Lock-Free Programming

The practical implementation of lock-free containers is known to be difficult: in addition to addressing the hazards of race conditions, the developer must also use non-blocking memory management and memory allocation schemes [14]. As explained in [1] and [6], a single-word CAS operation is inadequate for the practical implementation of a non-trivial concurrent container. The use of a double-compare-and-swap primitive (DCAS) has been suggested by Detlefs et al. in [6], however it is rarely supported by the hardware architecture.

A software implementation of a multiple-compare-and-swap (MCAS) algorithm, based on CAS, has been proposed by Harris et al. [11]. This software-based MCAS algorithm has been effectively applied by Fraser towards the implementation of a number of lock-free containers such as binary search trees and skip lists [7]. The cost of this MCAS operation is relatively expensive requiring $2M+1$ CAS instructions. Consequently, the direct application of this MCAS scheme is not an optimal approach for the design of lock-free algorithms. However, the MCAS implementation employs a number of techniques (such as pointer bit marking and the use of descriptors) that are useful for the design of practical lock-free systems. As discussed by Harris et al., a descriptor is an object that allows an interrupting thread to help an interrupted thread to complete successfully.

2.2 Lock-Free Data Structures

Recent research into the design of lock-free data structures includes linked-lists [10], [20] double-ended queues [19], [27], stacks [12], hash tables [20], [25] and binary search trees [7]. The problems encountered include excessive copying, low parallelism, inefficiency and high overhead. Despite the widespread use of the STL vector in real-world applications, the problem of the design and implementation of a lock-free dynamic array has not yet been discussed. The vector's random access, data locality, and dynamic memory management poses serious

challenges for its non-blocking implementation. Our goal is to provide an efficient and practical lock-free STL-style vector.

2.3 Design Principles

We developed a set of design principles to guide our implementation:

(a) *thread-safety*: all data can be shared by multiple processors at all times.
(b) *lock-freedom*: apply non-blocking techniques for our implementation.
(c) *portability*: do not rely on uncommon architecture-specific instructions.
(d) *easy-to-use interfaces*: offer the interfaces and functionality available in the sequential STL vector.
(e) *high level of parallelism*: concurrent completion of non-conflicting operations should be possible.
(f) *minimal overhead*: achieve lock-freedom without excessive copying [1], minimize the time spent on CAS-based looping and the number of calls to CAS.

The lock-free vector's design and implementation provided follow the syntax and semantics of the ISO STL vector as defined in ISO C++ [18].

3 Algorithms

In this section we define a semantic model of the vector's operations, provide a description of the design and the applied implementation techniques, outline a correctness proof based on the adopted semantic model, address concerns related to memory management, and discuss some alternative solutions to our problem. The presented algorithms have been implemented in ISO C++ and designed for execution on an ordinary multi-threaded shared-memory system supporting only single-word read, write, and CAS instructions.

3.1 Implementation Overview

The major challenges of providing a lock-free vector implementation stem from the fact that key operations need to atomically modify two or more non-colocated words. For example, the critical vector operation `push_back` increases the size of the vector and stores the new element. Moreover, capacity-modifying operations such as `reserve` and `push_back` potentially allocate new storage and relocate all elements in case of a dynamic table [5] implementation. Element relocation must not block concurrent operations (such as `write` and `push_back`) and must guarantee that interfering updates will not compromise data consistency. Therefore, an update operation needs to modify up to four vector values: size, capacity, storage, and a vector's element.

The UML diagram in Fig. 1 presents the collaborating classes, their programming interfaces and data members. Each vector object contains the memory locations of the data storage of its elements as well as an object named "`Descriptor`" that encapsulates the container's size, a reference counter required by the applied memory management scheme (Section 3.4) and an optional reference to a

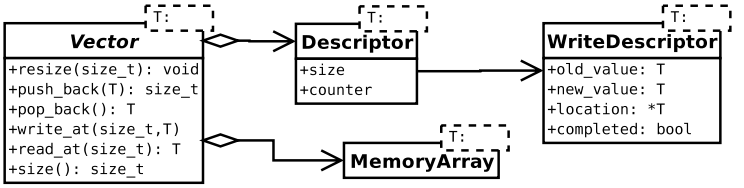


Fig. 1. Lock-free Vector. T denotes a data structure parameterized on T.

"`Write Descriptor`". Our approach requires that data types bigger than word size are indirectly stored through pointers. Like Intel's concurrent vector [24], our implementation avoids storage relocation and its synchronization hazards by utilizing a two-level array. Whenever `push_back` exceeds the current capacity, a new memory block twice the size of the previous one is added.

The semantics of the `pop_back` and `push_back` operations are guaranteed by the "`Descriptor`" object. The use of a "`Descriptor`" and "`WriteDescriptor`" (Barnes-style announcement [3]) allows a thread-safe update of two memory locations thus eliminating the need for a DCAS instruction. An interrupting thread intending to change the descriptor will need to complete any pending operation. Not counting memory management overhead, `push_back` executes *two successful CAS* instructions to update *two memory locations*.

3.2 Operations

Table 1 illustrates the implemented operations as well as their signatures, descriptor modifications, and runtime guarantees.

Table 1. Vector - Operations

	Operations	Descriptor (Desc)	Complexity
push_back	$Vector \times Elem \rightarrow void$	$Desc_t \rightarrow Desc_{t+1}$	$O(1) \times congestion$
pop_back	$Vector \rightarrow Elem$	$Desc_t \rightarrow Desc_{t+1}$	$O(1) \times congestion$
reserve	$Vector \times size\_t \rightarrow Vector$	$Desc_t \rightarrow Desc_t$	$O(1)$
read	$Vector \times size\_t \rightarrow Elem$	$Desc_t \rightarrow Desc_t$	$O(1)$
write	$Vector \times size\_t \times Elem \rightarrow Vector$	$Desc_t \rightarrow Desc_t$	$O(1)$
size	$Vector \rightarrow size\_t$	$Desc_t \rightarrow Desc_t$	$O(1)$

The remaining part of this section presents the generalized pseudo-code of the implementation and omits code necessary for a particular memory management scheme. We use the symbols `^`, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access respectively. The function $HighestBit$ returns the bit-number of the highest bit that is set in an integer value. On modern x86 architectures $HighestBit$ corresponds to the BSR assembly instruction. `FBS` is a constant representing the size of the first bucket and equals eight in our implementation.

Push_back (add one element to end). The first step is to complete a pending operation that the current descriptor might hold. In case that the storage capacity has reached its limit, new memory is allocated for the next memory bucket. Then, `push_back` defines a new "`Descriptor`" object and announces the current write operation. Finally, `push_back` uses CAS to swap the previous "`Descriptor`" object with the new one. Should CAS fail, the routine is re-executed. After succeeding, `push_back` finishes by writing the element.

Pop_back (remove one element from end). Unlike `push_back`, `pop_back` does not utilize a "`Write Descriptor`". It completes any pending operation of the current descriptor, reads the last element, defines a new descriptor, and attempts a CAS on the descriptor object.

Non-bound checking Read and Write at position i. The random access `read` and `write` do not utilize the descriptor and their success is independent of the descriptor's value.

Algorithm 1. pushback $vector, elem$

```
1: repeat
2:    desc_current ← vector.desc
3:    CompleteWrite(vector, desc_current.pending)
4:    bucket ← HighestBit(desc_current.size + FBS) − HighestBit(FBS)
5:    if vector.memory[bucket] = NULL then
6:       AllocBucket(vector, bucket)
7:    writeop ← new WriteDesc(At(desc_current.size)^, elem, desc_current.size)
8:    desc_next ← new Descriptor(desc_current.size + 1, writeop)
9: until CAS(&vector.desc, desc_current, desc_next)
10: CompleteWrite(vector, desc_next.pending)
```

Algorithm 2. AllocBucket $vector, bucket$

```
1: bucketsize ← FBS^(bucket+1)
2: mem ← new T[bucketsize]
3: if not CAS(&vector.memory[bucket], NULL, mem) then
4:    Free(mem)
```

Algorithm 3. Size $vector$

```
1: desc ← vector.desc
2: size ← desc.size
3: if desc.writeop.pending then
4:    size ← size − 1
5: return size
```

Reserve (increase allocated space). In the case of concurrently executing `reserve` operations, only one succeeds per bucket, while the others deallocate the acquired memory.

Algorithm 4. Read $vector, i$

1: **return** $At(vector, i)\hat{}$

Algorithm 5. Write $vector, i, elem$

1: $At(vector, i)\hat{} \leftarrow elem$

Algorithm 6. popback $vector$

1: **repeat**
2: $desc_{current} \leftarrow vector.desc$
3: $CompleteWrite(vector, desc_{current}.pending)$
4: $elem \leftarrow At(vector, desc_{current}.size - 1)\hat{}$
5: $desc_{next} \leftarrow new\ Descriptor(desc_{current}.size - 1, NULL)$
6: **until** $CAS(\&vector.desc, desc_{current}, desc_{next})$
7: **return** $elem$

Algorithm 7. Reserve $vector, size$

1: $i \leftarrow HighestBit(vector.desc.size + \texttt{FBS} - 1) - HighestBit(\texttt{FBS})$
2: **if** $i < 0$ **then**
3: $i \leftarrow 0$
4: **while** $i < HighestBit(size + \texttt{FBS} - 1) - HighestBit(\texttt{FBS})$ **do**
5: $i \leftarrow i + 1$
6: $AllocBucket(vector, i)$

Algorithm 8. At $vector, i$

1: $pos \leftarrow i + \texttt{FBS}$
2: $hibit \leftarrow HighestBit(pos)$
3: $idx \leftarrow pos\ xor\ 2^{hibit}$
4: **return** $\&vector.memory[hibit - HighestBit(\texttt{FBS})][idx]$

Algorithm 9. CompleteWrite $vector, writeop$

1: **if** $writeop.pending$ **then**
2: $CAS(At(vector, writeop.pos), writeop.value_{old}, writeop.value_{new})$
3: $writeop.pending \leftarrow false$

Size (read number of elements). The `size` operations returns the size stored in the "`Descriptor`" minus a potential pending write operation at the end of the vector.

3.3 Semantics

The semantics of the vector's operations is based on a number of assumptions. We assume that each processor can execute a number of the vector's operations.

This establishes a *history* of invocations and responses and defines a *real-time order* between them. An operation o_1 is said to precede an operation o_2 if o_2's invocation occurs after o_1's response. Operations that do not have real-time ordering are defined as *concurrent*. The vector's operations are of two types: those whose progress depends on the vector's descriptor and those who are independent of it. We refer to the former as *descriptor-modifying* and to the latter as *non-descriptor modifying operations*. All of the vector's operations in the set of concurrent descriptor-modifying operations S_1 are thread-safe and lock-free. The non-descriptor modifying operations such as random access read and write are implemented through the direct application of atomic read and write instructions on the shared data. In the set of non-descriptor modifying operations S_2, all operations are thread-safe and wait-free. In this section, we omit the discussion of ABA problem related issues, typical to all CAS-based implementations. There are a number of well known techniques to overcome these problems, see section 3.5.

Correctness. The main correctness requirement of the semantics of the vector's operations is linearizability [15]. A concurrent operation is linearizable if it appears to execute instantaneously in some moment of time between the time point t_{inv} of its invocation and the time point t_{resp} of its response. Firstly, this definition implies that each concurrent history yields responses that are equivalent to the responses of some legal sequential history for the same requests. Secondly, the order of the operations within the sequential history must be consistent with the real-time order. Let us assume that there is an operation $o_i \in S_{vec}$, where S_{vec} is the set of all the vector's operations. We assume that o_i can be executed concurrently with n other operations $\{o_1, o_2..., o_n\} \in S_{vec}$. We outline a proof that operation o_i is linearizable.

Linearization Points. For all non-descriptor-modifying operations the linearization point is at the time instance t_a when the atomic `read` (Algorithm 4, line 1) or `write` (Algorithm 5, line 1) of the element is executed. Assume o_i is a descriptor-modifying operation. It is carried out in two stages: modify the "`Descriptor`" variable and then update the data structure's contents. Let us define time points t_{desc} (Algorithm 1, line 10; Algorithm 6, line 6) and $t_{writedesc}$ (Algorithm 9, line 2) denote the instances of time when o_i executes an atomic update to the vector's "`Descriptor`" variable and when o_i's "`Write Descriptor`" is completed by o_i itself or another concurrent operation $o_c \in \{o_1, o_2..., o_n\}$, respectively. Similarly, time point $t_{readelem}$ (Algorithm 1, line 7; Algorithm 6, line 4) defines when o_i reads an element. o_i is either a `pop_back` or `push_back` operation. The linearization point is either $t_{readelem}$ or t_{desc} for the former case and $t_{readelem}$, t_{desc}, or $t_{writedesc}$ for the latter case.

Sequential Semantics. Let S_c be the set of all concurrent operations $\{o_1, ..., o_n\}$ in a time interval $[t_\alpha, t_\beta]$. If $\forall o_i \in S_c, DescriptorModifying(o_i)$, the linearization point for each operation is $t_{desc}(o_i)$. Similarly, if $\forall o_i \in S_c$, $NonDescriptorModifying(o_i)$, the linearization point for each operation is $t_a(o_i)$. In these cases, the resulting sequential histories are directly derived from the temporal order of the linearization points. In the remaining cases, the

derivation of a sequential history is significantly more complex. It is possible to transform all non-descriptor modifying operations into descriptor modifying in order to simplify the vector's sequential semantics. Given our current implementation, this can be achieved in a straightforward manner. We have chosen not to do so in order to preserve the efficiency and wait-freedom of the current non-descriptor modifying operations. Table 2 determines the linearization points for each pair of concurrent operations (o_1, o_2) where $DescriptorModifying(o_1)$ and $NonDescriptorModifying(o_2)$.

Table 2. Linearization Points of o_1, o_2

$o_1 \backslash o_2$	read	write
push_back	$t_{writedesc}(o_1), t_a(o_2)$	$t_{readelem}(o_1), t_a(o_2)$
pop_back	$t_{desc}(o_1), t_a(o_2)$	$t_{readelem}(o_1), t_a(o_2)$

We emphasize that the presented ordering relations are not transitive. Consider an example with three operations o_1 (`push_back`), o_2 (`write`), and o_3 (`read`), which access the same element. We assume that time points $t_a(o_2), t_a(o_3)$ occur between $t_{readelem}(o_1)$ and $t_{writedesc}(o_1)$ as well as that o_2 returns before the invocation of o_3. The resulting sequential history is o_1, o_2, o_3. It is derived from the real-time ordering between o_2 and o_3, and the pair-wise ordering relation between `push_back` and `write` in Table 2. A thorough linearizability proof for even the simplest data structure is non trivial and a further detailed elaboration is beyond the scope of this presentation.

Non-blocking. We prove the non-blocking property of our implementation by showing that out of n threads at least one makes progress. Since the progress of non-descriptor modifying operations is independent, they are wait-free. Thus, it suffices to consider an operation o_1, where o_1 is either a `push_back` or `pop_back`. A "`Write Descriptor`" can be simultaneously read by n threads. While one of them will successfully perform the "`Write Descriptor`"'s operation (o_2), the others will fail and not attempt it again. This failure is insignificant for the outcome of operation o_1. The first thread attempting to change the descriptor will succeed, which guarantees the progress of the system.

3.4 Memory Management

Our algorithms do not require the use of a particular memory management scheme. A garbage collected environment would have significantly reduced the complexity of the implementation (by moving key implementation problems inside the GC implementation). However, we do not know of any available general lock-free garbage collector for C++.

Object Reclamation. Our concrete implementation uses reference counting as described by Michael and Scott [23]. The major drawback of this scheme is that a timing window allows objects to be reclaimed while a different thread is about to increase the counter. Consequently, objects cannot be freed but only

recycled. Alternatives such as Michael's hazard pointers [21] and Herlihy et al.'s pass the buck [14] overcome the problem.

Allocator. Recent research by Michael [22] and Gidenstam [9] presents implementations of true lock-free memory allocators. Due to its availability and performance, we selected Gidenstam's allocator for our performance tests.

3.5 The ABA Problem

The ABA problem is fundamental to all CAS-based systems [21]. The semantics of the lock-free vector's operations can be corrupted by the occurrence of the ABA problem. Consider the following execution: assume a thread T_0 attempts to perform a `push_back`; in the vector's "`Descriptor`", `push_back` stores a write-descriptor announcing that the value of the object at position i should be changed from A to B. Then a thread T_1 interrupts and reads the write-descriptor. Later, after T_0 resumes and successfully completes the operation, a third thread T_2 can modify the value at position i from B back to A. When T_1 resumes its CAS is going to succeed and erroneously execute the update from A to B. There are two particular instances when the ABA problem can affect the correctness of the vector's operations:

(1) the user intends to store a memory address value A multiple times.
(2) the memory allocator reuses the address of an already freed object.

A universal solution to the ABA problem is to associate a version counter to each element on platforms supporting CAS2. However, because of hardware requirements of our primary application domain, we cannot currently assume availability of CAS2.

To eliminate the ABA problem of (2) (in the absence of CAS2), we have incorporated a variation of Herlihy et al.'s pass the buck algorithm [14] utilizing a separate thread to periodically reclaim unguarded objects.

The vector's vulnerability to (1) (in the absence of CAS2), can be eliminated by requiring the data structure to copy all elements and store pointers to them. Such behavior complies with the STL value-semantics [26], however it can incur significant overhead in some cases due to the additional heap allocation and object construction. In a lock-free system, both the object construction and heap allocation can execute concurrently with other operations. However, for significant applications, our vector can be used because the application programmer can avoid ABA problem (1). For example, a vector of unique elements (e.g. a vector recording live or active objects) does not suffer this problem. Similarly, a vector that has a "growth phase" (using `push_back`) that is separate from a "write phase" (using assignment to elements) (e.g., an append-only vector) is safe.

3.6 Alternatives

In this section we discuss several alternative designs for lock-free vectors.

Copy on Write. Alexandrescu and Michael present a lock-free map, where every write operation creates a clone of the original map, which insulates modifications from concurrent operations [1]. Once completed, the pointer to the

map's representation is redirected from the original to the new map. The same idea could be adopted to implement a vector. Since the complexity of any write operation deteriorates to $O(n)$ instead of $O(1)$, this scheme would be limited to applications exhibiting read-often but write-rarely access patterns.

Using Software DCAS. Harris et al. present a software multi-compare and swap (MCAS) implementation based on CAS instructions [11]. While convenient, the MCAS operation is expensive (requiring $2M + 1$ CAS instructions). Thus, it is not the best choice for an effective implementation.

Contiguous storage. Techniques similar to the ones used in our vector implementation could be applied to achieve a vector with contiguous storage. difference is that the storage area can change during lifetime. This requires `resize` to move all elements to the new location. Hence, storage and its capacity should become members of the descriptor. Synchronization between `write` and `resize` operations is what makes this approach difficult. A straightforward solution is to apply descriptor-modifying semantics as discussed in section 3.3.

We discussed the descriptor- and non-descriptor modifying writes in the context of the two-level array and the contiguous storage vector. However, these write properties are not inherent in these two approaches. In the two-level array, it is possible to make each write operation descriptor-modifying, thus ensure a write within bounds. In the contiguous storage approach, element relocation could replace the elements with marked pointers to the new location. Every access to these marked pointers would get redirected to the new storage.

4 Performance Evaluation

We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC OS 10.4.6 operating system. In our performance analysis, we compare the lock-free approach (with its integrated lock-free memory management and memory allocation) with the most recent concurrent vector provided by Intel [17] as well as an STL vector protected by a lock. For the latter scenario we applied different types of locking synchronizations - an operating system dependent mutex, a reader/writer lock, a spin lock, as well as a queuing lock. We used this variety of lock-based techniques to contrast our non-blocking implementation to the best available locking synchronization technique for a given distribution of operations. We utilize the locking synchronization provided by Intel [17].

Similarly to the evaluation of other lock-free concurrent containers [7] and [20], we have designed our experiments by generating a workload of various operations (`push_back`, `pop_back`, random access `write`, and `read`). In the experiments, we varied the number of threads, starting from 1 and exponentially increased their number to 32. Every active thread executed 500,000 operations on the shared vector. We measured the CPU time (in seconds) that all threads needed in order to complete. Each iteration of every thread executed an operation with a certain probability; `push_back` (`+`), `pop_back` (`-`), random access `write` (`w`), random access `read` (`r`). We use per-thread linear congruential random number

generators where the seeds preserve the exact sequence of operations within a thread across all containers. We executed a number of tests with a variety of distributions and found that the differences in the containers' performances are generally preserved. As discussed by Fraser [7], it has been observed that in real-world concurrent application, the read operations dominate and account to about 70% to 75% of all operations. For this reason we illustrate the performance of the concurrent vectors with a distribution of `+`:15%, `-`:5%, `w`:10%, `r`:70% on Figure 2A. Similarly, Figure 2C demonstrates the performance results with a distribution containing predominantly writes, `+`:30%, `-`:20%, `w`:20%, `r`:30%. In these diagrams, the number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. Both axes use logarithmic scale.

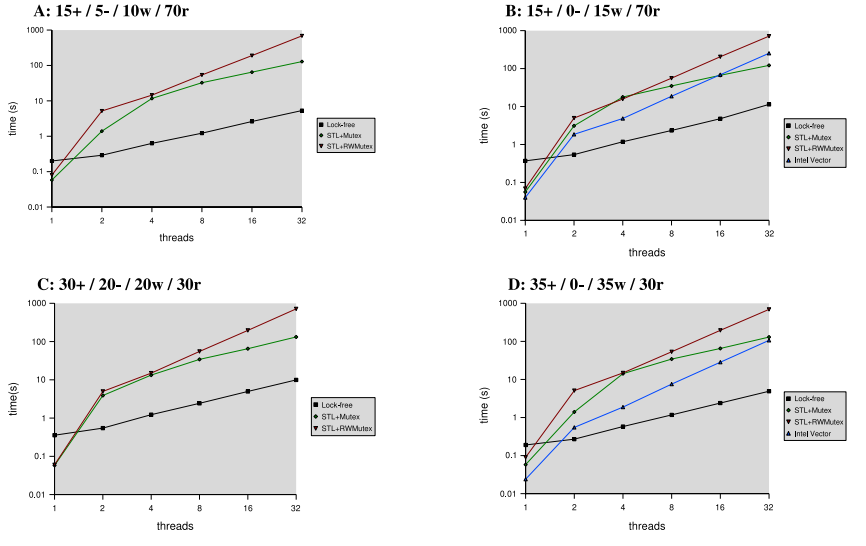


Fig. 2. Performance Results - Intel Core Duo

The current release of Intel's concurrent vector does not offer `pop_back` or any alternative to it. To include its performance results in our analysis, we excluded the `pop_back` operation from a number of distributions. Figure 2B and 2D present two of these distributions. For clarity we do not depict the results from the QueuingLock and SpinLock implementations. According to our observations, the QueuingLock performance is consistently slower than the other lock-based approaches. As indicated in [17], SpinLocks are volatile, unfair, and not scalable. They showed fast execution for the experiments with 8 threads or lower, however their performance significantly deteriorated with the experiments conducted with 16 or more active threads. To find a lower bound for our experiments we timed the tests with a non-thread safe STL-vector with pre-allocated

memory for all operations. For example, in the scenario described in figure 2D, the lower bound is about a $\frac{1}{10}$ of the lock-free vector.

Under contention our non-blocking implementation consistently outperforms the alternative lock-based approaches in all possible operation mixes by a significantly large factor. It has also proved to be scalable as demonstrated by the performance analysis. Lock-free algorithms are particularly beneficial to shared data under high contention. It is expected that in a scenario with low contention, the performance gains will not be as considerable.

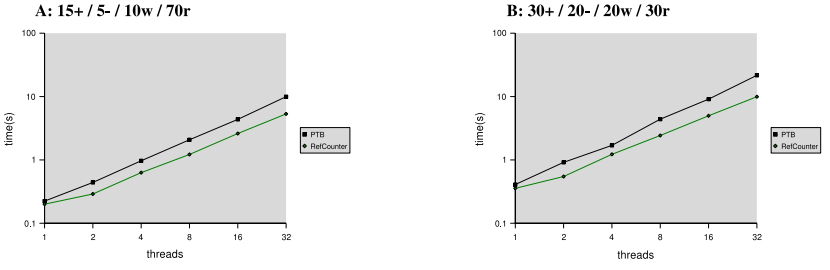


Fig. 3. Performance Results - Alternative Memory Management

As discussed in section 3.4, we have incorporated two different memory management approaches with our lock-free implementation, namely Michael and Scott's reference counting scheme (RefCount) and Herlihy et al.'s pass the buck technique (PTB). We have evaluated the vector's performance with these two different memory management schemes (Fig. 3).

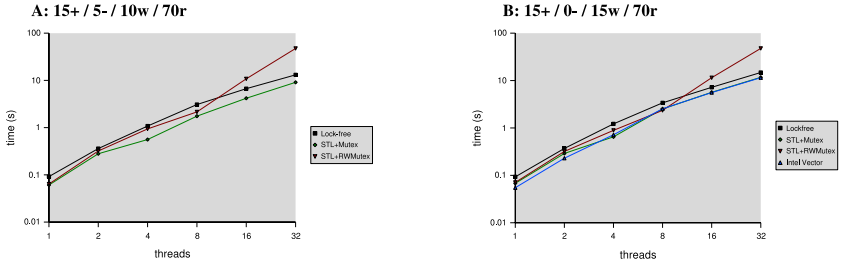


Fig. 4. Performance Results - AMD 8-way Opteron

On systems without shared L2 cache, shared data structures suffer from performance degradation due to cache coherency problems. To test the applicability of our approach on such architecture we have performed the same experiments on an AMD 2.2GHz quad dual core Opteron architecture with 1 MB L2 cache and 4GB shared RAM running the MS Windows 2003 operating system. (Fig.4). The applied lock-free memory allocation scheme is not available for MS Windows.

For the sake of our performance evaluation we applied a regular lock-based memory allocator. The experimental results on this architecture lack the impressive performance gains we have observed on the dual-core L2 shared-cache system. However, the graph (Fig.4) demonstrates that the performance of our lock-free approach on such architectures is comparable to the performance of the best lock-based alternatives.

5 Conclusion

We presented a first practical and portable design and implementation of a lock-free dynamically resizable array. We developed an efficient algorithm that supports disjoint-access parallelism and incurs minimal overhead. To provide a practical implementation, our approach integrates non-blocking memory management and memory allocation schemes. We compared our implementation to the best available concurrent lock-based vectors on a dual-core system and have observed an overall speed-up of a factor of 10. An essential direction in our future work is to further optimize the effectiveness of our approach on various hardware architectures and to eliminate the remaining ABA problem. In addition, it is our goal to precisely specify the concurrent semantics of the remaining STL vector interface and incorporate them in our implementation.

Acknowledgements

We thank Kirk Reinholtz, Herb Sutter, Olga Pearce, Yuriy Solodkyy, Luke Wagner, and the anonymous referees for their helpful suggestions as well as the Adobe Photoshop team for providing us with access to their multi-core machines.

References

1. A. Alexandrescu and M. Michael. Lock-free data structures with hazard pointers. *C++ User Journal*, November 2004.
2. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *LCPC '01*, pages 193–208, Cumberland Falls, Kentucky, Aug 2001.
3. G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, New York, NY, USA, 1993. ACM Press.
4. P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21N2009, April 2006.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.
6. D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele. Even better DCAS-based concurrent deques. In *DISC '00*, pages 59–73, 2000.
7. K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

8. D. Garlan, W. K. Reinholtz, B. Schmerl, N. D. Sherman, and T. Tseng. Bridging the gap between systems design and space systems software. In *SEW '05*, pages 34–46, Washington, DC, USA, 2005. IEEE Computer Society.
9. A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. In *ESA 2005: LNCS, volume 3669*, pages 329–342, 2005.
10. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01*, pages 300–314, London, UK, 2001. Springer-Verlag.
11. T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC '02*, 2002.
12. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, 2004. ACM Press.
13. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.
14. M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.
15. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
16. Intel. Ia-32 intel architecture software developer's manual, volume 3: System programming guide, 2004.
17. Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.
18. ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.
19. M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. In *Euro-Par '03, LNCS volume 2790*, pages 651–660, 2003.
20. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM Press.
21. M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.
22. M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conf. on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2004. ACM Press.
23. M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, 1995.
24. A. Robison, Intel Corporation. Personal communication, April 2006.
25. O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 102–111, New York, NY, USA, 2003. ACM Press.
26. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
27. H. Sundell and P. Tsigas. Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap. In *OPODIS 2004: Principles of Distributed Systems, 8th Int. Conf., LNCS, volume 3544*, pages 240–255, 2005.

Distributed Spanner Construction in Doubling Metric Spaces

Mirela Damian[1], Saurav Pandit[2], and Sriram Pemmaraju[2]

[1] Department of Computer Science, Villanova University, Villanova, PA 19085
mirela.damian@villanova.edu.
[2] Department of Computer Science, The University of Iowa, Iowa City, IA 52242-1419
{spandit, sriram}@cs.uiowa.edu.

Abstract. This paper presents a distributed algorithm that runs on an n-node unit ball graph (UBG) G residing in a metric space of constant doubling dimension, and constructs, for any $\varepsilon > 0$, a $(1+\varepsilon)$-spanner H of G with maximum degree bounded above by a constant. In addition, we show that H is "lightweight", in the following sense. Let Δ denote the aspect ratio of G, that is, the ratio of the length of a longest edge in G to the length of a shortest edge in G. The total weight of H is bounded above by $O(\log \Delta) \cdot wt(MST)$, where MST denotes a minimum spanning tree of the metric space. Finally, we show that H satisfies the so called *leapfrog property*, an immediate implication being that, for the special case of Euclidean metric spaces with fixed dimension, the weight of H is bounded above by $O(wt(MST))$. Thus, the current result subsumes the results of the authors in PODC 2006 that apply to Euclidean metric spaces, and extends these results to metric spaces with constant doubling dimension.

1 Introduction

A *unit ball graph* (UBG) is a graph whose vertices reside in some metric space and whose edges connect pairs of vertices at distance at most 1. The *doubling dimension* of a metric space is the smallest ρ such that any ball in this metric space can be covered by 2^ρ balls of half the radius. It is easy to verify that the d-dimensional Euclidean space, equipped with any of the L_p norms, has doubling dimension $\Theta(d)$. If ρ is a fixed constant (independent of the size of the UBG), then we call the UBG a *doubling UBG*. A t-spanner of a graph G is a spanning subgraph H of G such that, for all pairs of vertices $u, v \in V$, the length of a shortest uv-path in H is at most t times the length of a shortest uv-path in G. In this paper we present a distributed algorithm for constructing a low-weight, $(1+\varepsilon)$-spanner of bounded degree for doubling UBGs.

Precisely stated, our result is this: for any fixed $\varepsilon > 0$, our algorithm runs in $O(\log^* n)$ communication rounds on an n-node doubling UBG G, to construct a $(1+\varepsilon)$-spanner H of G with maximum degree bounded above by a constant. This constant depends on ε and ρ, the doubling dimension of the metric space in which G resides. Recall that $\log^* n = \min\{t \mid \log^{(t)} n \leq 2\}$, where $\log^{(0)} n = n$ and $\log^{(i)} n = \log(\log^{(i-1)} n)$ for any positive integer i. In addition, we show

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 157–171, 2006.

that H is "lightweight," in the following sense. Let Δ denote the aspect ratio of G, that is, the ratio of the length of a longest edge in G to the length of a shortest edge in G. We show that the total weight of H is bounded above by $O(\log \Delta) \cdot wt(MST)$, where MST denotes a minimum spanning tree of G. Thus we obtain a spanner that provides an $O(\log \Delta)$-approximation to a spanner of G of minimum weight. Finally, we also show that H satisfies the so called *leapfrog property* [8], which informally says that any uv-path in H that does not include $\{u, v\}$ must have length greater than $\{u, v\}$ by a constant factor. An immediate implication of this property is that, for the special case of Euclidean metric spaces with fixed dimension, the weight of H is bounded above by $O(wt(MST))$ [7]. Thus, our current result subsumes the results in [6] that apply to Euclidean metric spaces, and extends these results to metric spaces with constant doubling dimension.

1.1 Topology Control

Our result is motivated by the *topology control* problem in wireless ad-hoc networks. For an overview of topology control, see the survey by Rajaraman [16]. Since an ad-hoc network does not come with fixed infrastructure, there is no topology to start with and informally speaking, the topology control problem is one of selecting neighbors for each node so that the resulting topology has a number of useful properties such as sparseness, small weight, or maximum vertex degree bounded above by a constant. Most topology control protocols that provide worst case guarantees on the quality of the topology assume that the network is modeled by a unit disk graph (UDG) (see [14] for a recent example). The results in this paper apply to the more general model of doubling unit ball graphs (UBG). Doubling metric spaces have received a great deal of attention recently [4,11,12,13,17], partly because they are thought to capture real-world phenomena such as latencies in peer-to-peer networks and in the Internet. Also, doubling metrics are robust in the sense that the doubling dimension is roughly preserved under distortion (see Proposition 3 in [17]). Thus distorted versions of low dimensional Euclidean space also have small doubling dimension. Consequently, doubling UBGs can model wireless networks in which nodes have non-uniform transmission ranges or have erroneous perception of distances to other nodes. Finally, doubling metrics imply the following "bounded growth" phenomenon that seems to be characteristic of large scale wireless ad-hoc and sensor networks: the number of nodes that are far away from each other and yet are all in the vicinity of a particular node, is small. In other words, no node can have an arbitrarily large independent set in its neighborhood.

1.2 Net Trees

Let (V, d) be a metric space with $|V| = n$ and doubling dimension ρ. In a recent paper, Chan, Gupta, Maggs, and Zhou [2] show how to construct, via a sequential, polynomial-time algorithm, a $(1+\varepsilon)$-spanner of (V, d) with maximum degree bounded above by $\left(\frac{1}{\varepsilon}\right)^{O(\rho)}$. We will refer to this algorithm as the CGMZ

algorithm. The problem of constructing a spanner for a metric space can be thought of as a special case of our problem, in which the given UBG is a complete graph. Underlying the result in [2] is the notion of *net trees*, independently proposed by Har-Peled and Mendel [10]. Let $B(u, r)$ denote the ball of radius r centered at point u. A subset $U \subseteq V$ is an *r-net* of V if it satisfies two properties:

r-packing: For every u and v in U, $d(u, v) > r$.
r-covering: The union $\cup_{u \in U} B(u, r)$ covers V.

Such nets always exist for any $r > 0$, and can be easily computed using a greedy algorithm. Assume without loss of generality that the largest pairwise distance in V is exactly 1 (this can be achieved by appropriate scaling). Let α, with $\sqrt{1+\varepsilon} \leq \alpha$, and $\gamma = \frac{2\alpha}{\alpha - 1}\left(1 + \frac{4\alpha}{\varepsilon}\right)$ be constants (we use the fact that $\sqrt{1+\varepsilon} \leq \alpha$ in the proof of Lemma 7). Let h be the smallest positive integer such every pairwise distance is greater than $\frac{1}{\alpha^h}$. Let $r_0 = \frac{1}{\alpha^h}$ and let $r_i = \alpha \cdot r_{i-1}$, for $i > 0$. A *net tree* is a sequence of subsets $\langle V_0, V_1, V_2, \ldots, V_h \rangle$, such that $V_0 = V$ and V_i is an r_i-net of V_{i-1}, for $i > 0$. Note that every V_i, including V_0, is an r_i-packing. Also note that V_h, which is a 1-net of V_{h-1}, is a singleton, since the maximum separation between any pair of points is 1. To view the sequence $\langle V_0, V_1, V_2, \ldots, V_h \rangle$ as a tree, let $i(v) = \max\{i \mid v \in V_i\}$ for each $v \in V$. Then, for each $v \in V$, $i(v) + 1$ copies of v appear as nodes in the tree. These are denoted $(0, v), (1, v), \ldots, (i(v), v)$, where (i, v) represents the occurrence of v in V_i. For each $0 \leq i < i(v)$, the parent of node (i, v) is $(i+1, v)$. Node $(i(v), v)$ has no parent and is the root of the net tree, if $i(v) = h$; otherwise, vertex $v \notin V_{i(v)+1}$ and there is some vertex $u \in V_{i(v)+1}$ such that $B(u, r_{i(v)+1})$ contains v. Arbitrarily pick one such u and let $(i(v) + 1, u)$ be the parent of $(i(v), v)$. Informally speaking,

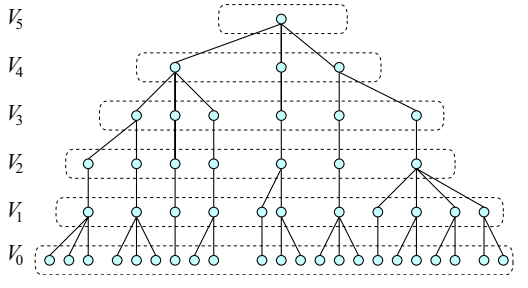


Fig. 1. A net tree with six levels

higher levels in the net tree (leaves are at level 0) represent the structure of V at lower resolution. Figure 1 shows an example of a net tree with 6 levels. Below we present the CGMZ algorithm [2]. For any two points $u, v \in V$, we use $d(u, v)$ to denote the distance between u and v in the underlying metric space.

The CGMZ Algorithm

1. Build a net tree $\langle V_0, V_1, \ldots, V_h \rangle$ of V.
2. Let $\gamma = \frac{2\alpha}{\alpha - 1}\left(1 + \frac{4\alpha}{\varepsilon}\right)$. Construct the edge sets

$$E_0 = \{\{u, v\} \in V_0 \times V_0 \mid \ d(u, v) \leq \gamma \cdot r_0\},$$

and

$$E_i = \{\{u, v\} \in V_i \times V_i \mid \ \gamma \cdot r_{i-1} < d(u, v) \leq \gamma \cdot r_i\},$$

for each $i = 1, \ldots, h$ and let $\widehat{E} = \cup_i E_i$.
3. Replace some edges in $\widehat{E}$ by other edges to obtain a new edge set $\widetilde{E}$.

Chan and coauthors [2] work with the version of the algorithm for $\alpha = 2$. They show that the graph $H = (V, \widehat{E})$ obtained after Step (2) is a $(1 + \varepsilon)$-spanner of the metric space and has linear number of edges, but may not satisfy the bounded degree requirement. Short paths in H can be obtained from the net tree in a natural manner. A uv-path in H whose length is at most $(1+\varepsilon) \cdot d(u, v)$ can be obtained by traveling up the net tree from the leaf u and from the leaf v until some level i is reached, such that the ancestors of u and v at level i are connected by an edge in H. In Step (3), a subset of the edges in $\widehat{E}$ is considered and each edge in this subset is replaced by at most one new edge. This step, which will be described in detail in Section 2.2, redistributes the edges so that all vertex-degrees are bounded above by a constant. The techniques used by Chan and coauthors for bounding vertex degrees play a critical role in this paper as well. In [6] we also describe an algorithm for constructing a bounded-degree $(1 + \varepsilon)$-spanner for Euclidean UBGs, but our results rely on purely geometric arguments to bound the vertex degree of the constructed spanner. Chan and coauthors [2] obtain the following theorem.

Theorem 1. [Chan, Gupta, Maggs, Zhou] *Let (V, d) be a finite metric with doubling dimension bounded by ρ. For any $\varepsilon > 0$, there is a $(1 + \varepsilon)$-spanner for (V, d), with maximum degree bounded above by $\left(\frac{1}{\varepsilon}\right)^{O(\rho)}$.*

Our algorithm is a modification of the CGMZ algorithm [2] that takes into account the fact that pairs of points separated by a distance greater than 1 are not connected by an edge and therefore such edges cannot be used in the spanner. A high level view of our algorithm is that it uses a slightly modified version of the CGMZ algorithm and constructs a graph H that may contain some *virtual edges*, that is, edges of length more than 1. H has all the desired properties with respect to the input UBG G. Subsequently, we show how to replace each virtual edge in H by at most one *real edge*, that is, an edge of length at most 1. The resulting graph is a $(1 + \varepsilon)$-spanner of G with degree bounded above by a constant.

To obtain a distributed implementation of the above idea in $O(\log^* n)$ rounds, we use an algorithm due to Kuhn, Moscibroda, and Wattenhofer [13]. For a given doubling n-node UBG G, the algorithm in [13] deterministically computes a $(1, O(1))$-network decomposition, that is, a partition of G into clusters such

that each cluster has diameter 1 and the resulting cluster graph has chromatic number $O(1)$. We use the same algorithm to compute a net tree. After computing the net tree, we require a constant number of additional rounds to construct the spanner.

2 Spanners for Doubling UBGs

Let (V, d) be a metric space with doubling dimension ρ. Let $G = (V, E)$ be the UBG induced by this metric space. Thus, for all $u, v \in V$, $u \neq v$, $\{u, v\} \in E$ if and only if $d(u, v) \leq 1$. For a fixed $\varepsilon > 0$, let the quantities h, r_i, α and γ be defined as in Section 1.2. Run Steps (1) and (2) of the CGMZ Algorithm to construct a set of edges $\widehat{E}$. Let $H = (V, \widehat{E})$. Note that V_h may not be a singleton since V may contain points whose pairwise distance is more than 1. So the sequence $\langle V_0, V_1, \ldots, V_h \rangle$ should be viewed as a forest of net trees, rooted at points in V_h. Recall that $\widehat{E} = \cup_{i=0}^{h} E_i$ and further recall that for $i > 0$, E_i consists of edges connecting all pairs of points $u, v \in V$ such that $d(u, v) \in (\gamma \cdot r_{i-1}, \gamma \cdot r_i]$. Note that there are values of i for which the right endpoint of the interval $(\gamma \cdot r_{i-1}, \gamma \cdot r_i]$ may be greater than 1 and for such values of i, E_i may contain edges that are not in E. Thus H is not necessarily a subgraph of G. Let $\delta = \lceil \log_\alpha \gamma \rceil$. It is easy to verify that for $0 \leq i \leq h - \delta$, $E_i \subseteq E$; for $i = h - \delta + 1$, the edge-set E_i may contain some edges in E and some edges not in E; and for $i > h - \delta + 1$, all edges in E_i are outside E. We call edges in H that also belong to E, *real edges*. Any edge in H that is not real is a *virtual edge*. Clearly, a spanner for G may not contain virtual edges, however virtual edges in H do carry important proximity information that will provide clues on how to replace them with real edges.

2.1 Properties of H

We will now prove some important properties of H. Let d_H be the distance metric induced by shortest paths in H. Specifically, we will show that H satisfies the following three properties:

(1) For every $\{u, v\} \in E$, $d_H(u, v) \leq (1 + \varepsilon) \cdot d(u, v)$ (Lemma 4).
(2) Edges of H can be oriented in such a way that the out-degree of H is bounded by $\left(\frac{1}{\varepsilon}\right)^{O(\rho)}$ (Lemma 5).
(3) The weight of H is $wt(H) = O(\log \Delta) \cdot \left(\frac{1}{\varepsilon}\right)^{O(\rho)} \cdot wt(MST)$ (Lemma 6).

Property (1) implies that H is connected, since G is assumed to be connected. Property (2) implies that H has a linear number of edges, though it does not imply that H has bounded maximum degree. In Section 2.2 we describe a method to alter H so as to bound the in-degree of H as well, while maintaining all the properties listed above. The proofs of these properties are based on some intermediate results, that we now establish. Proofs of Lemma 4 and Lemma 5 are similar to those in [3]. The next observation follows immediately from the definition of the doubling dimension of a metric space.

Proposition 1. *If (X, d) is a metric with doubling dimension ρ and $Y \subseteq X$ is a subset of points with aspect ratio Δ, then $|Y| \leq 2^{\rho \cdot \lceil \log_2 \Delta \rceil}$.*

For any point $u \in V_i$, let $N_i(u) = \{v \in V_i \mid \{u, v\} \in E_i\}$ denote the set of points connected to u by edges in E_i. We now show an upper bound on the size of $N_i(u)$.

Lemma 1. *For each $u \in V_i$, $|N_i(u)| \leq \left(\frac{1}{\varepsilon}\right)^{O(\rho)}$.*

Proof. That the aspect ratio of $N_i(u)$ is bounded by 2γ follows from two observations: (1) any two points in $N_i(u)$ are more than distance r_i apart, and (2) any point in $N_i(u)$ is at distance at most $\gamma \cdot r_i$ from u and therefore, by using the triangle inequality, any two points in $N_i(u)$ are at most $2\gamma \cdot r_i$ apart. Then Proposition 1 implies the lemma.

Lemma 2. *Suppose $u, v \in V_i$ and $d(u, v) \leq \gamma \cdot r_i$. Then $\{u, v\} \in \widehat{E}$.*

Proof. If $\gamma \cdot r_{i-1} < d(u, v) \leq \gamma \cdot r_i$, then by definition of E_i, $\{u, v\} \in E_i$. Otherwise, (a) $d(u, v) \leq \gamma \cdot r_0$ or (b) for some $j < i$, $\gamma \cdot r_{j-1} < d(u, v) \leq \gamma \cdot r_j$. Since $V_i \subseteq V_j$ for all $0 \leq j \leq i$, in case (a), $\{u, v\} \in E_0$ and in case (b), $\{u, v\} \in E_j$.

Lemma 3. *For each $u \in V$ and for each i, there exists $v \in V_i$ such that $d_H(u, v) \leq \frac{\alpha}{\alpha - 1} \cdot r_i$.*

Proof. The proof is by induction on i. For $i = 0$, $u \in V_0 = V$ and $d_H(u, u) = 0 < \frac{\alpha}{\alpha-1} \cdot r_0$, proving this case true. For $i > 0$, apply the inductive hypothesis to infer that there exists $w \in V_{i-1}$ such that $d_H(u, w) \leq \frac{\alpha}{\alpha-1} \cdot r_{i-1}$. Furthermore, since V_i is an r_i-net of V_{i-1}, there exists $v \in V_i \subseteq V_{i-1}$ such that $d(w, v) \leq r_i \leq \gamma \cdot r_{i-1}$. This along with Lemma 2 shows that $\{w, v\} \in \widehat{E}$ and therefore $d_H(w, v) = d(w, v) \leq r_i$. By the triangle inequality we have that $d_H(u, v) \leq d_H(u, w) + d_H(w, v) \leq \frac{\alpha}{\alpha-1} \cdot r_{i-1} + r_i = \frac{\alpha}{\alpha-1} \cdot r_i$.

In addition to proving the existence of a vertex v at each level i, Lemma 3 implies a certain path from vertex u to $v \in V_i$. Start from node $(0, u)$ in the tree (that is, the copy of u corresponding to a leaf) and follow the path through a sequence of parents, until a level-i node (i, v) is reached. Lemma 3 shows that the distance in H along this path is at most $\frac{\alpha}{\alpha-1} \cdot r_i$.

Lemma 4. **[Property 1]** *For any edge $\{u, v\} \in E$, $d_H(u, v) \leq (1 + \varepsilon) \cdot d(u, v)$.*

Proof. For ease of presentation, let $\lambda = \frac{\alpha}{\alpha-1}$. Let q be the smallest integer such that $\frac{4\lambda}{\alpha^q} \leq \varepsilon < \frac{8\lambda}{\alpha^q}$. Thus $q = \lceil \log_\alpha \frac{4\lambda}{\varepsilon} \rceil$. Let i be such that $r_i \leq d(u, v) < r_{i+1}$, and assume first that $i \leq q - 1$. Then $d(u, v) < \alpha^q \cdot r_0 \leq \frac{8\lambda}{\varepsilon} \cdot r_0 \leq \gamma r_0$, since $\gamma = 2\lambda\left(1 + \frac{4\alpha}{\varepsilon}\right) > \frac{8\lambda}{\varepsilon}$. Also since both u and v belong to V_0, by Lemma 2, we have that $\{u, v\} \in \widehat{E}$. This implies that $d_H(u, v) = d(u, v)$, proving the lemma true for this case. Assume now that $i \geq q$ and let $s = i - q \geq 0$. Note that $r_i = \alpha^q \cdot r_s$. By Lemma 3, there exist $x,\ y \in V_s$ such that $d_H(u, x) \leq \lambda \cdot r_s$ and $d_H(v, y) \leq \lambda \cdot r_s$. By the triangle inequality,

$$\begin{aligned}
d(x,y) &\leq d(x,u)+d(u,v)+d(v,y) \\
&\leq \lambda\cdot r_s + d(u,v) + \lambda\cdot r_s && (d(x,u)\leq d_H(x,u),\ d(v,y)\leq d_H(v,y)) \\
&< \lambda\cdot r_s + \alpha\cdot r_i + \lambda\cdot r_s && (\text{since } d(u,v) < r_{i+1}) \\
&= r_s(2\lambda+\alpha\cdot\alpha^q) && (\text{since } r_i=\alpha^q\cdot r_s) \\
&\leq r_s(2\lambda+\alpha\tfrac{8\lambda}{\varepsilon}) \\
&= \gamma\cdot r_s
\end{aligned}$$

Hence, by Lemma 2, $\{x,y\}\in\widehat{E}$ and therefore $d_H(x,y)=d(x,y)$. Using the triangle inequality again, we get

$$\begin{aligned}
d_H(u,v) &\leq d_H(u,x)+d_H(x,y)+d_H(y,v) \\
&\leq 2\lambda\cdot r_s + d(x,y) \\
&\leq 4\lambda\cdot r_s + d(u,v) && (\text{from the upper bound derivation of } d(x,y)) \\
&\leq (1+\tfrac{4\lambda}{\alpha^q})\cdot d(u,v) && (\text{since } r_i=\alpha^q\cdot r_s\leq d(u,v)) \\
&\leq (1+\varepsilon)\cdot d(u,v)
\end{aligned}$$

This completes the proof.

Lemma 4 also identifies a uv-path in H of length at most $(1+\varepsilon)\cdot d(u,v)$. Simply follow the sequence of parents, starting at the node $(0,u)$ in the tree and similarly, starting at the node $(0,v)$. At a certain level (denoted s in the proof), the ancestor of u and the ancestor of v at that level are connected by an edge in H.

We now prove Property (2) of H. Recall the notation: for each point u, $i(v)=\max\{i \mid v\in V_i\}$. For each edge $\{u,v\}\in\widehat{E}$, direct $\{u,v\}$ from u to v, if $i(u)<i(v)$. If $i(u)=i(v)$, pick an arbitrary orientation. This edge orientation is identical to the one used in [2]. Call the resulting digraph $\overrightarrow{H}$.

Lemma 5. [Property 2] *The out-degree of $\overrightarrow{H}$ is bounded above by $(\frac{1}{\epsilon})^{O(\rho)}$.*

Proof. Let $\{u,v\}\in\widehat{E}$ be an arbitrary edge directed from u to v, and let i be such that $\{u,v\}\in E_i$. Then $d(u,v)\leq\gamma\cdot r_i$. Now note that $r_{i+\delta}=\alpha^\delta\cdot r_i\geq\gamma\cdot r_i$ (recall that $\delta=\lceil\log_\alpha\gamma\rceil$). This, along with the fact that $V_{i+\delta}$ is an $r_{i+\delta}$-net, implies that it is not possible for both u and v to exist in $V_{i+\delta}$. Since $i(u)\leq i(v)$ (by our assumption), it follows that $i(u)\leq i+\delta$. On the other hand, $u\in V_i$ and so $i(u)\geq i$.

Summarizing, we have that $i(u)-\delta\leq i\leq i(u)$. This tells us that there are at most $\delta+1=O(\log_\alpha\gamma)$ values of i for which E_i may contain an edge outgoing from u. For each such i, by Lemma 1 there are at most $|N_i(u)|\leq\left(\frac{1}{\varepsilon}\right)^{O(\rho)}$ edges in E_i outgoing from u. It follows that the total number of edges in $\widehat{E}$ outgoing from u is $\left(\frac{1}{\varepsilon}\right)^{O(\rho)}\cdot O(\log_\alpha\gamma)=\left(\frac{1}{\epsilon}\right)^{O(\rho)}$.

We now prove Property (3) of H, showing that H has bounded weight.

Lemma 6. [Property 3] *The total weight of H is $wt(H)=O(\log\Delta)\cdot\left(\frac{1}{\epsilon}\right)^{O(\rho)}\cdot wt(MST)$, where MST is a minimum spanning tree of V, and Δ is the aspect ratio of G.*

Proof. We show that, for each i, $wt(E_i) = \left(\frac{1}{\epsilon}\right)^{O(\rho)} \cdot wt(MST)$. This along with the fact that there are $h+1 = \log_\alpha \frac{1}{r_0} + 1 = O(\log_\alpha \Delta)$ levels i, proves the claim of the lemma.

Let $U_i \subseteq V_i$ be the points in V_i incident to edges in E_i, and let $t = |U_i|$. Recall that any edge $\{u, v\} \in E_i$ satisfies $r_i < d(u, v) \leq \gamma \cdot r_i$. Thus, any spanning tree of a set of points containing U_i has weight at least $(t-1) \cdot r_i$, implying that $wt(MST) \geq (t-1) \cdot r_i$. Also note that the weight of E_i is bounded by $\Sigma_{u \in U_i} |N_i(u)| \cdot \gamma \cdot r_i \leq \left(\frac{1}{\varepsilon}\right)^{O(\rho)} \cdot t \cdot \gamma \cdot r_i$, using the upper bound on $|N_i(u)|$ given by Lemma 1. Using the lower bound on $wt(MST)$, we see that the weight of E_i is bounded above by $\left(\frac{1}{\varepsilon}\right)^{O(\rho)} \cdot \gamma \cdot (wt(MST) + r_i)$. Summing this expression over all E_i, yields the upper bound claimed in the lemma.

2.2 Altering H for Bounded Degree

In this section we show how to modify H so as to bound the degree of each vertex by a constant. Lemma 5 shows that an oriented version of H, namely $\overrightarrow{H}$, has bounded out-degree. Next we describe a method that carefully replaces some directed edges in $\overrightarrow{H}$ by others so as to guarantee constant bound on the in-degree as well, without increasing the out-degree. The replacement procedure is similar to the one used in [2], slightly adjusted to work with UBGs. Assume without loss of generality that $\varepsilon \leq \frac{1}{2}$; otherwise, if $\varepsilon > \frac{1}{2}$, we proceed with $\varepsilon = \frac{1}{2}$. We use the fact that $\varepsilon \leq \frac{1}{2}$ in the proof of Lemma 9. Let ℓ be the smallest positive integer such that $\frac{1}{\alpha^{\ell-1}} \leq \varepsilon$. Thus $\ell = O(\log_\alpha \frac{1}{\varepsilon})$.

Edge Replacement Procedure. Let u be an arbitrary point in V and let $M(u, i)$ be the set of all vertices $v \in V_i$ such that $\{v, u\}$ is an edge in E_i directed from v to u in $\overrightarrow{H}$. Let $I(u) = \langle i_1, i_2, \ldots \rangle$ be the increasing sequence of all indices i_k for which $M(u, i_k)$ is nonempty. For $1 \leq k \leq \ell$, we do not disturb any of the edges from points in $M(u, i_k)$ to u. For each $k > \ell$ such that $i_k \leq h - \delta - 2$, edges $\{v, u\}$ connecting $v \in M(u, i_k)$ to u are replaced by other edges. Specifically, an edge $\{v, u\}$, with $v \in M(u, i_k)$, is replaced by an edge $\{v, w\}$, where w is an arbitrary vertex in $M(u, i_{k-\ell})$. The replacement can be equivalently viewed as happening in either H or its oriented version $\overrightarrow{H}$. In $\overrightarrow{H}$, we replace the directed edge (v, u) by the directed edge (v, w). In the next two lemmas, our arguments will use $\overrightarrow{H}$ or H, as convenient.

Let $\widetilde{E}$ be the resulting set of edges. By our construction, $|\widetilde{E}| \leq |\widehat{E}|$. An important observation here is that the replacement procedure above is carried out only for edges in E_i, with $i \leq h - \delta - 2$ (that is, only edges of length no greater than $1/\alpha^2$). This is to ensure that only real edges get replaced and no virtual edges get added, a guarantee that is shown in the following lemma.

Lemma 7. *$\widetilde{E} \setminus \widehat{E}$ contains no virtual edges.*

Proof. Let $\{v, u\}$ be an edge that gets replaced by $\{v, w\}$, with $v \in M(u, i_k)$ and $w \in M(u, i_{k-\ell})$. Recall that $k > \ell$ and $i_k \leq h - \delta - 2$. Using the definitions of

E_{i_k} and $E_{i_{k-\ell}}$ and the fact that $\frac{1}{\alpha^{\ell-1}} \leq \varepsilon$, it follows that $d(w,u) \leq \varepsilon \cdot d(v,u)$. By the triangle inequality, $d(v,w) \leq d(v,u) + d(w,u) \leq (1+\varepsilon)d(v,u)$. Now note that $d(v,u) \leq 1/\alpha^2$. This is because edges in E_{i_k} have length no greater than $\gamma \cdot r_{i_k} \leq 1/\alpha^2$, for any $i_k \leq h - \delta - 2$. Therefore $d(v,w) \leq (1+\varepsilon)/\alpha^2 \leq 1$, for any $\alpha^2 \geq (1+\varepsilon)$.

Let $J = (V, \widetilde{E})$. First we show that J indeed has bounded degree (Lemma 8). Second we show that the metric distance d_J induced by shortest paths in J is a good approximation of d_H (Lemma 9). A consequence of this is that J remains connected, and maintains spanner paths between endpoints of real edges.

Lemma 8. *Every vertex in $J = (V, \widetilde{E})$ has degree bounded by $(\frac{1}{\epsilon})^{O(\rho)}$.*

Proof. Let A be the maximum out-degree of a vertex of $\overrightarrow{H}$. By Lemma 5, $A \leq (\frac{1}{\epsilon})^{O(\rho)}$. Let B be the largest of $|N_i(u)|$, for all i and all u. By Lemma 1, $B \leq (\frac{1}{\epsilon})^{O(\rho)}$. The edge-replacement procedure replaces a directed edge (v,u) by a directed edge (v,w). So the out-degrees of vertices remain unchanged by the edge-replacement procedure, and continue to be bounded above by $(\frac{1}{\epsilon})^{O(\rho)}$. Thus, we can simply focus on the in-degrees of vertices. We bound these by accounting for the in-degree of an arbitrary vertex x with respect to old edges (in $\widetilde{E} \cap \widehat{E}$) and with respect to new edges (in $\widetilde{E} \setminus \widehat{E}$); we show that both in-degrees are bounded above by $(\frac{1}{\epsilon})^{O(\rho)}$.

In-degree of x with respect to $\widetilde{E} \cap \widehat{E}$. Out of the edges in $\overrightarrow{H}$ that come into x, at most $B(\ell + \delta + 2)$ remain in $\widetilde{E}$. More specifically, at most B edges at each of the first ℓ levels $i_1, i_2, \ldots, i_\ell$ in $I(x)$, plus at most B edges in each of E_i, $i = h - \delta - 1, h - \delta, \ldots, h$, remain in $\widetilde{E}$. Any other edge directed into x gets replaced by an edge not incident to x. We end this case by noting that $B(\ell + \delta + 2) = (\frac{1}{\epsilon})^{O(\rho)}$.

In-degree of x with respect to $\widetilde{E} \setminus \widehat{E}$. Vertex x has a new in-coming edge whenever it plays the role of w in the edge-replacement procedure. Recall that in the edge-replacement procedure, w and v are both in-neighbors of u. For each edge (w,u), there are at most B edges (v,u) directed into u that may get replaced by (v,w). Furthermore, there are A edges (w,u) outgoing from w. This gives an upper bound of $AB = (\frac{1}{\epsilon})^{O(\rho)}$ on the in-degree of x.

It remains to show that d_J is a good approximation of d_H. Intuition for this is provided by the proof of Lemma 7. In that proof, it is shown that when $\{v,w\}$ replaces $\{v,u\}$, $d(w,u) \leq \varepsilon \cdot d(v,u)$ and $d(v,w) \leq (1+\varepsilon) \cdot d(v,u)$. Thus, if the path $\langle v,w,u \rangle$ existed in $\widetilde{E}$, this path would have length at most $(1+2\varepsilon) \cdot d(v,u)$. However, edge $\{w,u\}$ may not exist in $\widetilde{E}$, since it may itself have been replaced. Thus a shortest path from w to u in $\widetilde{E}$ may be longer than $d(w,u)$. However, since $d(w,u) \leq \varepsilon \cdot d(v,u)$, the extra cost of replacing $\{w,u\}$ is marginal and the eventual sum of all of these lengths is still bounded above by $(1+2\varepsilon) \cdot d(v,u)$. Thus we have the following lemma, whose proof we skip (due to space restrictions).

Lemma 9. $d_J \leq (1+2\varepsilon)d_H$.

2.3 Eliminating Virtual Edges

The only impediment in having $J = (V, \widetilde{E})$ serve as a spanner for the input UBG G is the presence of virtual edges in J. Recall that these are edges of length greater than 1 and clearly do not exist in G. In this section we show that there exist real edges that can take over the role of virtual edges in J, without violating the properties J is expected to have.

Let $\{u, v\} \in E$ be an arbitrary edge and let i be such that $r_i \leq d(u, v) < r_{i+1}$. Let q be as in the proof of Lemma 4: the smallest integer such that $\frac{\alpha}{\alpha-1} \cdot \frac{4}{\alpha^q} \leq \varepsilon < \frac{\alpha}{\alpha-1} \cdot \frac{8}{\alpha^q}$. As mentioned before, the proof of Lemma 4 implies a certain uv-path of length at most $(1+\varepsilon) \cdot d(u, v)$ in $H = (V, \widehat{E})$. If $i \leq q-1$, this path is just the edge $\{u, v\}$, because $\{u, v\}$ is guaranteed to exist in $\widehat{E}$. The Edge Replacement Procedure (Section 2.2) ensures that only real edges are replaced, and each real edge is replaced by a path consisting only of real edges. This along with Lemma 9 ensures that even in $\widetilde{E}$ there is a uv-path of length at most $(1 + 2\varepsilon) \cdot d(u, v)$, consisting of real edges only. If $i \geq q$, the uv-path in H implied by Lemma 4 may have more than one edge. Let $s = i - q$ and (s, u^*) (respectively, (s, v^*)) be the level-s ancestor of the leaf $(0, u)$ (respectively, the leaf $(0, v)$) in the net tree $\langle V_0, V_1, \ldots, V_h \rangle$. Then the edge $\{u^*, v^*\}$ is guaranteed to be present in $\widehat{E}$ and the uv-path implied by Lemma 4 starts at $(0, u)$, goes up to the net tree via parents to (s, u^*), then to (s, v^*), and then follows the unique path down the tree from (s, v^*) to $(0, v)$. It is easy to check that of all the edges in this path, only $\{u^*, v^*\}$ may be virtual. Specifically, when the edge $\{u, v\}$ is long enough to guarantee that $i \geq h - \delta + 1 + q$, then $s = i - q \geq h - \delta + 1$ and the edge $\{u^*, v^*\}$ may belong to E_s. Recall that for $j \geq h - \delta + 1$, edges in E_j may not be real and in particular $\{u^*, v^*\}$ may be a virtual edge. Since the uv-path implied by Lemma 4 passes through edge $\{u^*, v^*\}$, one has to be careful in replacing $\{u^*, v^*\}$ by a real edge. Our virtual edge replacement procedure is given below.

For any node (i, v) in the net tree, let $T(i, v)$ denote the set of all vertices $u \in V$, such that the subtree of the net tree rooted at (i, v) contains a copy of u. In other words, $T(i, v) = \{u \in V \mid (i, v) \text{ is an ancestor of } (j, u) \text{ for some } j \leq i\}$.

Virtual Edge Replacement Procedure. For a virtual edge $\{u, v\} \in E_i$, if there is a real edge $\{x, y\}$ already in the spanner H, with $x \in T(i, u)$ and $y \in T(i, v)$, then simply delete $\{u, v\}$. Similarly, if there is no such real edge $\{x, y\}$ in the input graph G with $x \in T(i, u)$ and $y \in T(i, v)$ then simply delete $\{u, v\}$. Otherwise, find a real edge $\{x, y\} \in E$, $x \in T(i, u)$ and $y \in T(i, v)$, and replace $\{u, v\}$ by $\{x, y\}$.

The reason why this replacement procedure works can be intuitively explained as follows. A virtual edge $\{u, v\} \in E_i$ is important for pairs of vertices $\{a, b\}$, with $a \in T(i, u)$ and $b \in T(i, v)$, for which all ab-paths of length at most $(1+\varepsilon) \cdot d(a, b)$ pass through $\{u, v\}$. Replacing $\{u, v\}$ by $\{x, y\}$ provides the following alternate ab-path that is short enough: starting at the leaf a, go up the tree rooted at (i, u) via parents until an ancestor common to a and x is reached, then come down to x, take edge $\{x, y\}$, go up the tree rooted at (i, v) until an ancestor common to

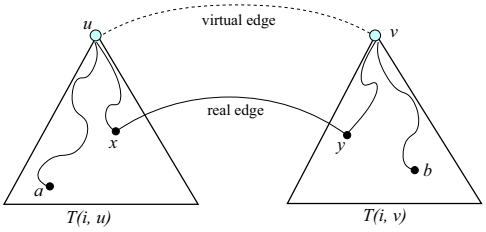


Fig. 2. A short ab-path passes through virtual edge $\{u, v\}$. After replacing virtual edge $\{u, v\}$ by real edge $\{x, y\}$, there is a short ab-path through $\{x, y\}$.

b and y is reached, and finally go down to b. Figure 2 illustrates this alternate path. Note that this entire path consists only of real edges.

We finally state our main result. Let G' be the graph obtained from J by replacing virtual edges using the Virtual Edge Replacement Procedure.

Theorem 2. *$G' = (V, E')$ is a $(1+\varepsilon)$-spanner of G with degree bounded above by $(\frac{1}{\varepsilon})^{O(\rho)}$ and weight bounded above by $O(\log \Delta) \cdot (\frac{1}{\varepsilon})^{O(\rho)} \cdot wt(MST)$.*

A proof similar to that of Lemma 4 can be used to show the spanner property of G'. The fact that G' is lightweight simply follows from the fact that a virtual edge of length greater than 1 in J, either gets eliminated, or gets replaced by at most one real edge of length at most 1 in G'. The constant degree bound follows from the observation that, for a vertex x to acquire a new incident edge, there is an ancestor of x in the net tree at level $h - \delta + 1$ or higher, that loses an incident edge at that level. There are a constant number of such ancestors and from Lemma 1, we know that any vertex has a constant number of incident edges at any particular level.

We conclude this section with a summary of our algorithm.

`Algorithm SPANNER(`(V, d), ε`)`

Let $\sqrt{1+\varepsilon} < \alpha$ be a constant, $\gamma = \frac{2\alpha}{\alpha - 1}\left(1 + \frac{4\alpha}{\varepsilon}\right)$, and $\delta = \lceil \log_\alpha \gamma \rceil$.
Let h be the smallest such that $\frac{1}{\alpha^h}$ is smaller than the minimum inter-point distance.
Let $r_0 = \frac{1}{\alpha^h}$ and let $r_i = \alpha \cdot r_{i-1}$, for all $i > 0$.

Constructing a linear size $(1+\varepsilon)$-spanner $H = (V, \widehat{E})$.

1. Construct the net tree $\langle V_0, V_1, \ldots, V_h \rangle$.
   [Let $i(u) = \max\{i \mid u \in V_i\}$.]
2. Construct the sets
   $$E_0 = \{\{u, v\} \in V_0 \times V_0 \mid \ d(u, v) \leq \gamma \cdot r_0\},$$
   $$E_i = \{\{u, v\} \in V_i \times V_i \mid \gamma \cdot r_{i-1} < \ d(u, v) \leq \gamma \cdot r_i\}, \text{ for } 1 \leq i \leq h.$$
   [Let $\widehat{E} = \cup_i E_i$ and $H = (V, \widehat{E})$.]

Replacing edges to obtain a constant degree bound.

3. Orient each edge $\{u, v\} \in \widehat{E}$ from u to v if $i(u) \leq i(v)$, breaking ties arbitrarily.
   [Let $M(u, i)$ denote the set of vertices $v \in V_i$, with $\{v, u\} \in \widehat{E}$.

4. For each $u \in V$, construct the increasing sequence $I(u) = \langle i_1, i_2, \ldots, \rangle$ of all i_k with $M(u, i_k) \neq \emptyset$. [Let ℓ be the smallest integer with $\frac{1}{\alpha^{\ell-1}} \leq \varepsilon$.]
5. For each $u \in V$ and each $i_k \in I(u)$, with $k > \ell$ and $i_k \leq h - \delta - 2$, do
6. Replace directed edge (v, u), $v \in M(u, i_k)$ by edge (v, w), for arbitrary $w \in M(u, i_{k-\ell})$.

[Let $J = (V, \widetilde{E})$ be the resulting graph, with distance metric d_J.]

Replacing virtual edges by real ones.

[Let $T(i, v) = \{x \in V \mid (i, v)$ is an ancestor of (j, x) for some $j \leq i\}$.]

7. For each $i \geq h - \delta + 1$ and each virtual edge $\{u, v\} \in E_i$ do
8. If there is a real edge $\{x, y\} \in \widetilde{E}$, $x \in T(i, u)$ and $y \in T(i, v)$, then do nothing.
9. Otherwise, if there is a real edge $\{x, y\} \in E$, with $x \in T(i, u)$ and $y \in T(i, v)$, replace $\{u, v\}$ by $\{x, y\}$.

[Let E' be the set of resulting edges. Output is $G' = (V, E')$.]

---

3 Leapfrog Property

In Lemma 6, we showed that $H = (V, \widehat{E})$ has total weight bounded above by $O(\log \Delta) \cdot \left(\frac{1}{\varepsilon}\right)^{O(\rho)} \cdot wt(MST)$, where Δ is the aspect ratio of G. Thus, for fixed ε and constant doubling dimension ρ, the upper bound is within $O(\log \Delta)$ times the optimal value. In an attempt to show a bound that is within $O(1)$ times the optimal value, we use a tool that is widely used in the computational geometry literature [8,5,9]. In the context of building lightweight $(1 + \varepsilon)$-spanners for Euclidean spaces, Das and Narasimhan [8] have shown that, if the set of edges in the spanner satisfy a property known as the *leapfrog property*, then the total weight of the spanner is bounded above by $O(wt(MST))$. Below we state the leapfrog property precisely.

Leapfrog Property. For any $t \geq t' > 1$, a set F of edges has the (t', t)-leapfrog property if, for every subset $S = \{\{u_1, v_1\}, \{u_2, v_2\}, \ldots, \{u_s, v_s\}\}$ of F,

$$t' \cdot d(u_1, v_1) < \sum_{i=2}^{s} d(u_i, v_i) + t \cdot \Big(\sum_{i=1}^{s-1} d(v_i, u_{i+1}) + d(v_s, u_1) \Big). \tag{1}$$

Informally, this definition says that, if there exists an edge between u_1 and v_1, then any u_1v_1-path not including $\{u_1, v_1\}$ must have length greater than $t' \cdot d(u_1, v_1)$. Das and Narasimhan [8] show the following connection between the leapfrog property and the weight of the spanner.

Lemma 10. *Let $t \geq t' > 1$. If the line segments F in d-dimensional space satisfy the (t', t)-leapfrog property, then $wt(F) = O(wt(MST))$, where MST is a minimum spanning tree connecting the endpoints of line segments in F. The constant in the asymptotic notation depends on t, t' and d.*

It is well known that, if a spanner is built "greedily", then the set of edges in the spanner satisfies the leapfrog property [8,5,9]. In [6] we showed that even a "relaxed" version of the greedy algorithm would ensure that the spanner edges

have the leapfrog property. This was critical to showing that the spanner constructed in a distributed manner for UBGs in Euclidean spaces [6] had total weight bounded above by $O(wt(MST))$. Here we ask if it is possible to do the same for UBGs in metric spaces with constant doubling dimension. In an attempt to answer this question we show that, using a variant of the SPANNER algorithm (outlined at the end of Section 2), we can build, for a given doubling UBG G, a $(1+\varepsilon)$-spanner with degree bounded above by a constant and with the (t, t')-leapfrog property, for some constants $t \geq t' > 1$. Note that this does not give us the desired $O(wt(MST))$ bound on the weight of the constructed spanner because we do not know if the equivalent of Lemma 10 holds for non-Euclidean metric spaces. The proof of Lemma 10 in [8] is quite geometric and does not suggest an approach to its generalization to metric spaces of constant doubling dimension.

To guarantee that the output spanner satisfies the (t', t)-leapfrog property, we need to make two modifications to the SPANNER algorithm. Let H_i denote the spanning subgraph of G induced by $E_0 \cup E_1 \cup \cdots \cup E_i$.

1. We modify Step (2) of the algorithm and place an edge $\{u, v\}$ into E_i only if $\{u, v\} \in V_i \times V_i$, $\gamma \cdot r_{i-1} < d(u, v) \leq \gamma \cdot r_i$, *and there is not already a* uv*-path of length at most* $(1+\varepsilon) \cdot d(u, v)$ *in* H_{i-1}.
2. Two edges $\{u, v\}$ and $\{u', v'\}$ in E_i are said to be *mutually redundant* if both of the following conditions hold:
   (a) $d_{H_{i-1}}(v, u') + d(u', v') + d_{H_{i-1}}(v', u) \leq (1+\varepsilon) \cdot d(u, v)$
   (b) $d_{H_{i-1}}(v', u) + d(u, v) + d_{H_{i-1}}(v, u') \leq (1+\varepsilon) \cdot d(u', v')$
   Note that these conditions imply that $H_i \setminus \{u, v\}$ contains a uv-path of length at most $(1+\varepsilon) \cdot d(u, v)$ and $H_i \setminus \{u', v'\}$ contains a $u'v'$-path of length at most $(1+\varepsilon) \cdot d(u', v')$. Thus, one of these can potentially be eliminated from H_i, without compromising the $(1+\varepsilon)$-spanner property of H_i. In fact, such mutually redundant pairs of edges need to be eliminated from H_i in order to show that H satisfies the leapfrog property.

These ideas and how they lead to a spanner that has the leapfrog property are discussed in detail in [6].

4 Distributed Implementation

In this section, we show that the SPANNER algorithm (Section 2) and its variant that ensures the leapfrog property (Section 3), both have distributed implementations that run in $O(\log^* n)$ rounds of communication. Here we focus on the SPANNER algorithm. It turns out that Step (1) of this algorithm takes $O(\log^* n)$ rounds, whereas the remaining steps take $O(1)$ additional rounds. We first examine Steps (2)-(9) of the algorithm.

It is easy to verify that in Steps (2)-(9), a node u needs to communicate only with other nodes that are either neighbors of u in G, or to which u is connected by a virtual edge. The main difficulty here is that the endpoints of a virtual edge $\{u, v\}$ may not be neighbors in the network. Consider a virtual edge $\{u, v\} \in E_i$.

By definition of E_i, $d(u,v) \leq \gamma \cdot r_i \leq \gamma$. Even though the distance between u and v in the underlying metric space is bounded above by a constant, it is not necessary that the hop distance between u and v in G be similarly bounded above. Let us call a virtual edge $\{u,v\} \in E_i$, *useful*, if there exist $x \in T(i,u)$ and $y \in T(i,v)$ such that $\{x,y\} \in E$. Notice that only useful virtual edges need to be considered by our algorithm. If a virtual edge $\{u,v\}$ is not useful, then even though it is added in Step (2), it is eliminated in Steps (7)-(9). In the following lemma we show that the hop distance between endpoints of useful virtual edges is small.

Lemma 11. *The hop distance in G between the endpoints of any useful virtual edge $\{u,v\} \in E_i$ is at most $2(\frac{2\alpha}{\alpha-1}+1)$.*

Proof. By definition of a useful virtual edge, there are points $x \in T(i,u)$ and $y \in T(i,v)$ such that $\{x,y\}$ is an edge in G. Thus a path in G between u and v is the following: start at node (i,u) in the net tree and travel down to a copy of x, follow the edge $\{x,y\}$, and then travel up to node (i,v). Note that the edge $\{x,y\} \in E$, but it may not belong to $\widetilde{E}$. The length of this path is at most $2(1+\frac{1}{\alpha}+\frac{1}{\alpha^2}+\ldots)+1$, implying that $d_G(u,v) \leq \frac{2\alpha}{\alpha-1}+1$. Now consider a shortest uv-path in G, say $\langle w_0 = u, w_1, \ldots, w_{k+1} = v\rangle$. Because G is a UBG and due to the triangle inequality, $d(w_i, w_{i+2}) > 1$ for all $0 \leq i \leq k-1$ (otherwise w_{i+1} could be eliminated from the uv-path to obtain an even shorter uv-path in G). This yields a lower bound of $k/2$ on $d(u,v)$, and since $d_G(u,v) \geq d(u,v)$, we have that $d_G(u,v) \geq k/2$. Combining this with the upper bound of $\frac{2\alpha}{\alpha-1}+1$, we obtain that $k \leq 2(\frac{2\alpha}{\alpha-1}+1)$.

Thus, in Steps (2)-(9) of the `SPANNER` algorithm, a node only needs to communicate with nodes that are at most $O(\frac{1}{\alpha-1})$ hops away. This suggests a simple way of implementing Steps (2)-(9): after Step (1) is completed, each node u gathers neighborhood information and the values of $i(v)$ from all nodes v that are $O(\frac{1}{\alpha-1})$ hops away. After this, node u can do all of its computation with no further communication.

The fact that Step (1) can be implemented in $O(\log^* n)$ rounds of communication follows from a clever argument in [13]. Suppose that we have computed the set V_{i-1}. The computation of the set V_i, which is an r_i-net of V_{i-1}, reduces to a maximal independent set (MIS) computation on a degree-bounded graph. To see this, create a graph, say G_i, whose vertex set is V_{i-1} and whose edges connect any pair of vertices $u, v \in V_{i-1}$, if $d(u,v) \leq r_i$. Then it is easy to see that an MIS in G_i is an r_i-net of V_{i-1}. Furthermore, the fact that G_i has bounded degree follows from the fact that the underlying metric space has bounded doubling dimension. There is a well-known deterministic algorithm due to Linial [15] for computing an MIS, that runs in $O(\log^* n)$ communication rounds on graphs with bounded degree. Using this algorithm, one can compute the r_i-net V_i of V_{i-1} in $O(\log^* n)$ rounds. Since there are $h+1 = O(\log \Delta)$ such sets to compute, it seems like this approach will take $O(\log \Delta \cdot \log^* n)$ rounds. However, in [13] it is shown that in this algorithm, each node uses information only from nodes that are at most $O(\log^* n)$ hops away in G. Therefore, this algorithm has a $O(\log^* n)$-round implementation in which each node u first gathers information

from nodes that are at most $O(\log^* n)$ hops away and then performs all steps of the SPANNER algorithm locally, using the collected information.

References

1. B. Awerbuch, A. Goldberg, M. Luby, and S. Plotkin. Network decomposition and locality in distributed computation. In *IEEE Symposium on Foundations of Computer Science*, pages 364–369, 1989.
2. Hubert T-H. Chan, Anupam Gupta, Bruce M. Maggs, and Shuheng Zhou. On hierarchical routing in doubling metrics. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 762–771, 2005.
3. T-H. Hubert Chan. Personal Communication, 2006.
4. T-H. Hubert Chan and Anupam Gupta. Small hop-diameter sparse spanners for doubling metrics. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 70–78, 2006.
5. A. Czumaj and H. Zhao. Fault-tolerant geometric spanners. *Discrete & Computational Geometry*, 32(2):207–230, 2004.
6. Mirela Damian, Saurav Pandit, and Sriram Pemmaraju. Local approximation schemes for topology control. In *PODC '06: Proceedings of the twenty-fifth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2006.
7. G. Das, P. Heffernan, and G. Narasimhan. Optimally sparse spanners in 3-dimensional euclidean space. In *ACM Symposium on Computational Geometry*, pages 53–62, 1993.
8. G. Das and G. Narasimhan. A fast algorithm for constructing sparse euclidean spanners. *Int. J. Comput. Geometry Appl.*, 7(4):297–315, 1997.
9. J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Fast greedy algorithms for constructing sparse geometric spanners. *SIAM J. Comput.*, 31(5):1479–1500, 2002.
10. S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. In *SCG'05: Proceedings of the 21st annual symposium on Computational geometry*, pages 150–158, 2005.
11. R. Krauthgamer, A. Gupta, and J.R. Lee. Bounded geometries, fractals, and low-distortion embeddings. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 534–543, 2003.
12. R. Krauthgamer and J.R. Lee. Navigating nets: simple algorithms for proximity search. In *SODA '04: Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms*, pages 798–807, 2004.
13. Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. On the locality of bounded growth. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 60–68, 2005.
14. Xiang-Yang Li and Yu Wang. Efficient construction of low weighted bounded degree planar spanner. *International Journal of Computational Geometry and Applications*, 14(1–2):69–84, 2004.
15. Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.
16. R. Rajaraman. Topology control and routing in ad hoc networks: A survey. *SIGACT News*, 33:60–73, 2002.
17. K. Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *STOC'04: Proceedings of the 36th annual ACM symposium on Theory of computing*, pages 281–290, 2004.

Verification Techniques for Distributed Algorithms

Anna Philippou and George Michael

Department of Computer Science, University of Cyprus,
Kallipoleos Street, 1678 Nicosia, Cyprus
{annap, gmichael}@cs.ucy.ac.cy

Abstract. A value-passing, asynchronous process calculus and its associated theory of confluence are considered as a basis for establishing the correctness of distributed algorithms. In particular, we present an asynchronous version of value-passing CCS and we develop its theory of confluence. We show techniques for demonstrating confluence of complex processes in a compositional manner and we study properties of confluent systems that can prove useful for their verification. These results give rise to a methodology for system verification which we illustrate by proving the correctness of two distributed leader-election algorithms.

1 Introduction

Distributed systems present today one of the most challenging areas of research in computer science. Their high complexity and dynamic nature and features such as concurrency and unbounded nondeterminism, render their construction, description and analysis a challenging task. The development of formal frameworks for describing and associated methodologies for reasoning about distributed systems has been an active area of research for the last few decades and is becoming increasingly important as a consequence of the great success of worldwide networking and the vision of ubiquitous computing.

Process calculi, otherwise referred to as *process algebras*, such as CCS [8], the π-calculus [10], and others, are a well-established class of modeling and analysis formalisms for concurrent and distributed systems. They can be considered as high-level description languages consisting of a number of operators for building processes including constructs for defining recursive behaviors. They are accompanied by semantic theories which give precise meaning to processes, translating each process into a mathematical object on which rigorous analysis can be performed. In addition, they are associated with axiom systems which prescribe the relations between the various constructs and can be used to reason algebraically about processes. During the last two decades, they have been extensively studied and they have proved quite successful in the modeling and reasoning about system correctness. They have been extended for modeling a variety of aspects of process behavior including mobility, distribution, value-passing and asynchronous communication.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 172–186, 2006.

Confluence arises in a variety of forms in computation theory. It was first studied in the context of concurrent systems by Milner in [8]. Its essence, to quote [9], is that "of any two possible actions, the occurrence of one will never preclude the other". As shown in the mentioned papers, for pure CCS processes confluence implies determinacy and semantic-invariance under internal computation, and it is preserved by several system-building operators. These facts make it possible to reason compositionally that a system is confluent and to exploit this fact while reasoning about its behavior. Work on the study of confluence in concurrent systems was subsequently carried out in various directions. In [2] various notions of confluence were studied and the utility of the ideas was illustrated for state-space reduction and protocol analysis. Furthermore, the theory of confluence was developed for value-passing CCS in [16,18]. In the context of mobile process calculi, such as the π-calculus and extensions of it, notions of confluence and *partial* confluence were studied and employed in a variety of contexts including [5,11,13,14,15].

The aims of this paper is to study the notion of confluence in an asynchronous, value-passing process calculus and to develop useful techniques for showing that complex asynchronous processes are confluent. Based on this theory, we develop a methodology for the analysis of distributed algorithms. We illustrate its utility via the verification of two distributed leader-election algorithms.

The extension of the theory of confluence to the asynchronous setting is at most places pretty straightforward. Thus, due to the lack of space, we omit the proofs of the results and draw the attention only to significant points pertaining to the ways in which the presence of asynchrony admits a larger body of compositional results. We then take a step from the traditional definition of confluence in value-passing process calculi, and we introduce a new treatment of input actions, motivated by the notion of "input-enabledness" often present in distributed systems. Our main result shows that the resulting notion allows an interesting class of complex processes to be shown as confluent by construction. These compositional results and other confluence properties are exploited in the verification of both of the algorithms we consider. We illustrate the application of the theory for the algorithm verification in detail.

The remainder of the paper is structured as follows. In the next section we present our asynchronous extension of value-passing CCS while, in Sect. 3, the theory of confluence is developed. Section 4 contains application of our verification methodology for establishing the correctness of two distributed leader-election algorithms and Sect. 5 concludes the paper.

2 The Calculus

In this section we present the CCS_v process calculus, an amalgamation of value-passing CCS [8,18], with features of asynchronous communication from the asynchronous π-calculus [1,4] and a kind of conditional agents.

We begin by describing the basic entities of the calculus. We assume a set of *constants*, ranged over by u, v, including the positive integers and the boolean

values and a set of functions, ranged over by f, operating on these constants. Moreover, we assume a set of *variables* ranged over by x, y. Then, the set of *terms* of CCS_v, ranged over by e, is given by (1) the set of constants, (2) the set of variables, and (3) function applications of the form $f(e_1, \ldots e_n)$, where the e_i are terms. We say that a term is *closed* if it contains no variables. The evaluation relation $\twoheadrightarrow$ for closed terms is defined in the expected manner. We write $\tilde{r}$ for a tuple of syntactic entities $r_1, \ldots, r_n$.

Moreover, we assume a set of *channels*, $\mathcal{L}$, ranged over by a, b. Channels provide the basic communication and synchronization mechanisms in the language. A channel a can be used in *input position*, denoted by a, and in o*utput position*, denoted by $\overline{a}$. This gives rise to the set of *actions* Act of the calculus, ranged over by α, β, containing

- the set of *input actions* which have the form $a(\tilde{v})$ representing the input along channel a of a tuple $\tilde{v}$,
- the set of *output actions* which have the form $\overline{a}(\tilde{v})$ representing the output along channel a of a tuple $\tilde{v}$, and
- the *internal action* τ, which arises when an input action and an output action along the same channel are executed in parallel.

For simplicity, we write a and $\overline{a}$ for $a(\langle\rangle)$ and $\overline{a}(\langle\rangle)$, where $\langle\rangle$ is the empty tuple. We say that an input and an output action on the same channel are *complementary* actions, and, for a non-internal action α, we denote by $\ell(\alpha)$ the channel of α. Finally, we assume a set of process constants $\mathcal{C}$, ranged over by C.

The syntax of CCS_v processes is given by the following BNF definition:

$$P ::= \mathbf{0} \;\mid\; \overline{a}(\tilde{v}) \;\mid\; \sum_{i \in I} a_i(\tilde{x_i}).P_i \;\mid\; P_1 \parallel P_2 \;\mid\; P \backslash L$$
$$\mid\; \mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n) \;\mid\; C\langle \tilde{v} \rangle$$

Process $\mathbf{0}$ represents the inactive process. Process $\overline{a}(\tilde{v})$ represents the asynchronous output process. It can output the tuple $\tilde{v}$ along channel a. Process $\sum_{i \in I} a_i(\tilde{x_i}).P_i$ represents the nondeterministic choice between the set of processes $a_i(\tilde{x_i}).P_i$, $i \in I$. It may initially execute any of the actions $a_i(\tilde{x_i})$ and then evolve into the corresponding continuation process P_i. It is worth noting that nondeterministic choice is only defined with respect to input-prefixed processes. This follows the intuition adopted in [1,4] that, if an output action is enabled, it should be performed and not precluded from arising by nondeterministic choice. Further, note that, unlike the summands $a_i(\tilde{x_i}).P_i$, the asynchronous output process does not have a continuation process due to the intuition that an output action is simply emitted into the environment and execution of the emitting process should continue irrespectively of when the output is consumed.

Process $P \| Q$ describes the concurrent composition of P and Q: the component processes may proceed independently or interact with one another while executing complementary actions. The conditional process $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$ offers a conditional choice between a set of processes: assuming that all e_i are closed terms, it behaves as P_i, where i is the smallest integer for which $e_i \twoheadrightarrow \mathsf{true}$.

In $P\backslash F$, where $F \subseteq \mathcal{L}$, the scope of channels in F is restricted to process P: components of P may use these channels to interact with one another but not with P's environment. This gives rise to the free and bound channels of a process. We write $\mathsf{fc}(P)$ for the free channels of P, and $\mathsf{fic}(P)$, $\mathsf{foc}(P)$ for the free channels of P which are used in input and output position within P, respectively.

Finally, process constants provide a mechanism for including recursion in the process calculus. We assume that each constant C has an associated definition of the form $C\langle\tilde{x}\rangle \stackrel{\text{def}}{=} P$, where process P may contain occurrences of C, as well as other process constants.

Each operator is given precise meaning via a set of rules which, given a process P, prescribe the possible transitions of P, where a transition of P has the form $P \xrightarrow{\alpha} P'$, specifying that process P can perform action α and evolve into process P'. The rules themselves have the form

$$\frac{T_1, \ldots, T_n}{T} \quad \phi$$

which is interpreted as follows: if transitions $T_1, \ldots, T_n$, can be derived, and condition ϕ holds, then we may conclude transition T. The semantics of the CCS_v operators are given in Table 1.

Table 1. The operational semantics

(Sum)	$\sum_{i\in I} a_i(\tilde{x_i}).P_i \xrightarrow{a_i(\tilde{v}_i)} P_i\{\tilde{v}_i/\tilde{x_i}\}$	(Out)	$\overline{a}(\tilde{v}) \xrightarrow{\overline{a}(\tilde{v})} \mathbf{0}$
(Par1)	$\dfrac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \| P_2 \xrightarrow{\alpha} P_1' \| P_2}$	(Par2)	$\dfrac{P_2 \xrightarrow{\alpha} P_2'}{P_1 \| P_2 \xrightarrow{\alpha} P_1 \| P_2'}$
(Par3)	$\dfrac{P_1 \xrightarrow{a(\tilde{v})} P_1',\ P_2 \xrightarrow{\overline{a}(\tilde{v})} P_2'}{P_1 \| P_2 \xrightarrow{\tau} P_1' \| P_2'}$	(Res)	$\dfrac{P \xrightarrow{\alpha} P',\ \ell(\alpha), \notin F}{P\backslash F \xrightarrow{\alpha} P'\backslash F}$
(Cond)	$\dfrac{P_i \xrightarrow{\alpha} P_i'}{\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n) \xrightarrow{\alpha} P_i}$	$e_i \twoheadrightarrow \mathsf{true}, \forall j < i, e_j \twoheadrightarrow \mathsf{false}$	
(Const)	$\dfrac{P\{\tilde{v}/\tilde{x}\} \xrightarrow{\alpha} P'}{C(\tilde{v}) \xrightarrow{\alpha} P}$	$C(\tilde{x}) \stackrel{\text{def}}{=} P$	

We discuss some of the rules below:

– (Sum). This axiom employs the notion of *substitution*, a partial function from variables to values. We write $\{\tilde{v}/\tilde{x}\}$ for the substitution that maps variables $\tilde{x}$ to values $\tilde{v}$. Thus, for all $i \in I$, the input-prefixed summation process can receive a tuple of values $\tilde{v}_i$ along channel a_i, and then continue as process P_i, with the occurrences of the variables $\tilde{x}_i$ in P_i substituted by values $\tilde{v}_i$. For example: $a(x,y).\overline{b}(x) + c(z).\mathbf{0} \xrightarrow{a(2,5)} \overline{b}(2)$ and $a(x,y).\overline{b}(x) + c(z).\mathbf{0} \xrightarrow{c(1)} \mathbf{0}$.

- (Par1). This axiom (and its symmetric version (Par2)) expresses that a component in a parallel composition of processes may execute actions independently. For example, since $\overline{a}(3) \xrightarrow{\overline{a}(3)} \mathbf{0}$, $\overline{a}(3) \parallel a(v).\overline{b}(c) \xrightarrow{\overline{a}(3)} \mathbf{0} \parallel a(v).\overline{b}(c)$.
- (Par3). This axiom expresses that two parallel processes executing complementary actions may synchronize with each other producing the internal action τ: $\overline{a}(3) \parallel a(v).\overline{b}(v) \xrightarrow{\tau} \overline{b}(3)$.
- (Cond). This axiom formalizes the behavior of the conditional operator. An example of the rule follows: $\mathsf{cond}(2 = 3 \rhd \overline{b}(3), \mathsf{true} \rhd \overline{c}(4)) \xrightarrow{\overline{c}(4)} \mathbf{0}$.
- (Const). This axiom stipulates that, given a process constant and its associated definition $C\langle \tilde{x} \rangle \stackrel{\text{def}}{=} P$, its instantiation $C\langle \tilde{v} \rangle$ behaves as process P with variables $\tilde{x}$ substituted by $\tilde{v}$. For example, if $C\langle x, y \rangle \stackrel{\text{def}}{=} \mathsf{cond}(x = y \rhd \overline{b}(x), \mathsf{true} \rhd \overline{c}(y))$, then $C(2,2) \xrightarrow{\overline{b}(2)} \mathbf{0}$.

An additional form of process expression derivable from our syntax and used in the sequel is the following: $!\,P \stackrel{\text{def}}{=} P \parallel\ !\,P$, usually referred to as the *replicator* process, represents an unbounded number of copies of P running in parallel.

We recall some useful definitions. We say that Q is a *derivative* of P, if there are $\alpha_1, \ldots, \alpha_n \in Act$, $n \geq 0$, such that $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$. Moreover, given $\alpha \in Act$ we write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, $\stackrel{\alpha}{\Longrightarrow}$ for the composition $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$, and $\stackrel{\hat{\alpha}}{\Longrightarrow}$ for $\Longrightarrow$ if $\alpha = \tau$ and $\stackrel{\alpha}{\Longrightarrow}$ otherwise.

We conclude this section by presenting a notion of process equivalence in the calculus. Observational equivalence is based on the idea that two equivalent systems exhibit the same behavior at their interfaces with the environment. This requirement was captured formally through the notion of *bisimulation* [8,12]. Bisimulation is a binary relation on states of systems. Two processes are bisimilar if, for each step of one, there is a matching (possibly multiple) step of the other, leading to bisimilar states. Below, we introduce a well-known such relation on which we base our study.

Definition 1. *Bisimilarity* is the largest symmetric relation, denoted by $\approx$, such that, if $P \approx Q$ and $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P' \approx Q'$.

Note that bisimilarity abstracts away from internal computation by focusing on *weak* transitions, that is, transitions of the form $\stackrel{\hat{a}}{\Longrightarrow}$ and requires that bisimilar systems can match each other's *observable* behavior. We also point out that, while two bisimilar processes have the same traces, the opposite does not hold.

Bisimulation relations have been studied widely in the literature. They have been used to establish system correctness by modeling a system and its specification as two process-calculus processes and discovering a bisimulation that relates them. The theory of bisimulation relations has been developed into two directions. On one hand, axiom systems have been developed for establishing algebraically the equivalence of processes. On the other hand, proof techniques that ease the task of showing two processes to be equivalent have been proposed. The results presented in the next section belong to the latter type.

3 Confluence

In [8,9], Milner introduced and studied a precise notion of *determinacy* of CCS processes. The same notion carries over straightforwardly to the CCS_v-calculus:

Definition 2. P is *determinate* if, for every derivative Q of P and for all $\alpha \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q''$, then $Q' \approx Q''$.

This definition makes precise the requirement that, when an experiment is conducted on a process, it should always lead to the same state up to bisimulation. As in pure CCS, a CCS_v process bisimilar to a determinate process is determinate, and determinate processes are bisimilar if they may perform the same sequence of visible actions. The following lemma summarizes conditions under which determinacy is preserved by the CCS_v operators.

Lemma 1

1. $\mathbf{0}$ and $\overline{a}(\tilde{v})$ are determinate processes.
2. If P is determinate so is $P\backslash F$.
3. If, for all $i \in I$, each P_i is determinate and the a_i are distinct channels, $\sum_{i\in I} a_i(\tilde{x}_i).\, P_i$ is also determinate.
4. If, for all $i \in I$, each P_i is determinate so is $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$.
5. If P_1 and P_2 are determinate and $\mathsf{fic}(P_1) \cap \mathsf{fc}(P_2) = \emptyset$, $\mathsf{fic}(P_2) \cap \mathsf{fc}(P_1) = \emptyset$, then $P_1 \| P_2$ is also determinate.

Proof: The interesting case is Clause (5). The proof consists of a case analysis on all pairs of actions that can be taken from a derivative of $P_1 \| P_2$ and it takes advantage of the asynchronous-output mechanism. Intuitively, since output actions have no continuation, if two identical outputs are concurrently enabled within a system, it does not matter which one is fired first. □

Note that, in the case of parallel composition, previous results in CCS and the π-calculus apply a stronger side-condition than the one of Clause (5) above. Namely, these side-conditions require that the parallel components P_1 and P_2 have no channels in common. Here, however, the asynchronous nature of output actions allows us to weaken the condition as shown.

According to the definition of [9], a CCS process P is *confluent* if it is determinate and, for each of its derivatives Q and distinct actions α, β, given the transitions to Q_1 and Q_2, the following diagram can be completed.

$$\begin{array}{ccc} Q & \xrightarrow{\alpha} & Q_1 \\ {\scriptstyle\beta}\,\Downarrow & & \Downarrow\,{\scriptstyle\hat{\beta}} \\ Q_2 & \stackrel{\hat{\alpha}}{\Longrightarrow} & Q_2' \sim Q_1' \end{array}$$

Let P be the CCS_v-calculus process $P \stackrel{\text{def}}{=} a(x).\, \overline{b}(x).\, \mathbf{0}$ and consider the transitions $P \xrightarrow{a(2)} \overline{b}(2).\, \mathbf{0}$ and $P \xrightarrow{a(3)} \overline{b}(3).\, \mathbf{0}$. Clearly, the two transitions cannot be 'brought together' in order to complete the diagram above. Despite this fact, it appears natural to classify P as a confluent process. Indeed, investigation of

confluence in the context of value-passing calculi resulted in extending the CCS definition above to take account of substitution of values [16,18]. The definitions highlight the asymmetry between input and output actions by considering them separately. Here we express this separation as follows:

Definition 3. A CCS_v process P is *confluent* if it is determinate and, for each of its derivatives Q and distinct actions α, β, where α and β are not input actions on the same channel, if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\beta} Q_2$ then, there are Q_1' and Q_2' such that $Q_2 \stackrel{\hat{\alpha}}{\Longrightarrow} Q_2'$, $Q_1 \stackrel{\hat{\beta}}{\Longrightarrow} Q_1'$ and $Q_1' \approx Q_2'$. □

We may see that bisimilarity preserves confluence. Furthermore, confluent processes possess an interesting property regarding internal actions. We define a process P to be τ-*inert* if, for each derivative Q of P, if $Q \xrightarrow{\tau} Q'$, then $Q \approx Q'$. By a generalization of the proof in CCS, we obtain:

Lemma 2. If P is confluent then P is τ-inert.

As observed in [2], τ-inertness implies confluence for a certain class of processes. An analogue result also holds in our setting, as stated below.

Lemma 3. Suppose P is a fully convergent process. Then P is confluent *iff* P is τ-inert and for all derivatives Q of P

1. if $\alpha \in Act$ and $P \xrightarrow{\alpha} P_1$, $P \xrightarrow{\alpha} P_2$, then $P_1 \approx P_2$, and
2. if α, β are distinct actions and are not input actions on the same channel, if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\beta} Q_2$ then, there are Q_1' and Q_2' such that $Q_2 \stackrel{\hat{\alpha}}{\Longrightarrow} Q_2'$, $Q_1 \stackrel{\hat{\beta}}{\Longrightarrow} Q_1'$ and $Q_1' \approx Q_2'$.

Note that this is an alternative characterization of confluence for fully convergent systems which is useful in that the original transitions to be matched are single transitions. The proof of the result is a simple modification of the one found in [2]. We proceed with a result on the preservation of confluence by CCS_v operators.

Lemma 4

1. $\mathbf{0}$ and $\overline{a}(\tilde{v})$ are confluent processes.
2. If P is confluent so are $P\backslash F$ and $a(\tilde{x}).P$.
3. If, for all $i \in I$, each P_i is confluent so is $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$.
4. If P_1 and P_2 are confluent and $\mathsf{fic}(P_1) \cap \mathsf{fc}(P_2) = \emptyset$, $\mathsf{fic}(P_2) \cap \mathsf{fc}(P_1) = \emptyset$, then $P_1 \| P_2$ is also confluent.
5. if P is confluent and $\mathsf{fic}(P) \cap \mathsf{foc}(P) = \emptyset$, then $!P$ is confluent.

Of course, here, the guarded summation clause is missing.

A main motivation in [8] for studying confluence was to strengthen determinacy to an interesting property preserved by a wider range of process-calculus operators. Here, we are also interested in such compositional results in the setting of asynchronous processes. To achieve this, we observe that, despite the rational behind the treatment of input actions in the definition of confluence, it

is often the case that distributed systems are *input-enabled*. This notion, which has been fundamental in the development of the I/O-Automata of Lynch and Tuttle [7], captures that input actions of a system are not under the control of the system and are always enabled. This suggests that the execution of an input along a certain channel does not preclude the execution of another input along the same channel. As such, a confluence-type property can be expected to hold for input actions which we formulate as follows:

Definition 4. A CCS_v process P is F^i*-confluent*, where $F \subseteq \mathcal{L}$, if, for all derivatives Q of P and for all $a \in F$, if $Q \xrightarrow{a(\tilde{v})} Q_1$ and $Q \xrightarrow{a(\tilde{u})} Q_2$ then, there are Q_1' and Q_2' such that $Q_2 \stackrel{a(\tilde{v})}{\Longrightarrow} Q_2'$, $Q_1 \stackrel{a(\tilde{u})}{\Longrightarrow} Q_1'$ and $Q_1' \approx Q_2'$. □

We may see that F^i-confluence implies that, if at some point during execution of a process an input action becomes enabled, then it remains enabled. For the case that $F = \mathcal{L}$ we simply write i-confluence for $\mathcal{L}^i$-confluence. F^i-confluence is preserved by the following operators.

Lemma 5

1. $\mathbf{0}$ and $\overline{a}(\tilde{v})$ are i-confluent processes.
2. If P is F^i-confluent and $a \in F$, $P\backslash L$, $!P$, $a.\,P$ and $!a(\tilde{x}).\,P$ are also F^i-confluent.
3. If P_i, $i \in I$, are F^i-confluent so is $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$.
4. If P_1 and P_2 are F^i-confluent so is $P_1 \| P_2$.

We conclude with our main result:

Theorem 1. Suppose $P = (P_1 \parallel \ldots \parallel P_n)\backslash L$, where (1) each P_j is confluent, (2) each P_j is F_j^i-confluent, where $F_j = \mathsf{fic}(P_j) \cap (\bigcup_{k \neq i} \mathsf{foc}(P_k))$, and $F_j \subseteq L$, and (3) $\mathsf{fic}(P_i) \cap \mathsf{fic}(P_j) = \emptyset$, for all $i \neq j$. Then P is confluent.

Proof: The proof, which is too long to include here in its full technical detail, employs Lemma 3. We show that any derivative Q of P is τ-inert by a case analysis on the possible internal actions of Q. Suppose that this arises by a communication of the form $Q_i \xrightarrow{a(\tilde{v})} Q_i'$ and $Q_j \xrightarrow{\overline{a}(\tilde{v})} Q_j'$. Then, the $\{a\}^i$-confluence of Q_i and the confluence of Q_j imply that any action enabled by Q_i and Q_j is still possible by Q_i' and Q_j'. Further, since $a \notin \mathsf{fic}(P_k)$, for all $k \neq i$, this transition cannot be precluded from arising. Then, Clause (1) of the lemma is easy to establish using the assumption that $\mathsf{fic}(P_i) \cap \mathsf{fic}(P_j) = \emptyset$, for all $i \neq j$, whereas Clause (2), employs similar arguments and the fact that $F_j \subseteq L$ for all j. □

4 Two Applications

We proceed to illustrate the utility of the CCS_v framework and its theory of confluence via the analysis of two distributed algorithms for leader-election in a distributed ring. We assume that n processes with distinct identifiers, chosen

from a totally-ordered set, are arranged around a ring. They are numbered 1 to n in a clockwise direction and they can communicate with their immediate neighbours in order to elect as the leader the node with the maximum identifier.

Our verification methodology consists of the following steps. First, we describe an algorithm and its specification as CCS_v processes, our aim being to establish that the two processes are bisimilar. We achieve this as follows: (1) We show that the process representing the algorithm contains at least one specification-respecting execution. (2) We show that this process is confluent. By confluence properties we then obtain the required result. We point out that the first of the algorithms we consider here was also proved correct in [6] using the I/O-Automata framework.

4.1 The LCR Algorithm

The LCR algorithm [6] is a simple, well-known algorithm for distributed leader-election with time complexity $O(n^2)$. Communication between the nodes of the ring can only take place in a clockwise direction. Figure 1 presents the algorithm architecture. Each node of the ring executes the following:

> It sends its identifier to its right neighbour. Concurrently, it awaits to receive messages from its left neighbour. For each incoming message, if it contains an identifier greater than its own, it forwards the message in a clockwise direction. If it is smaller it discards it and, if it is equal, it declares itself to be the leader.

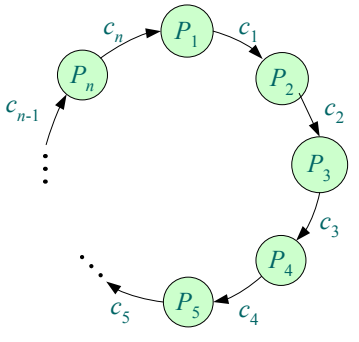


Fig. 1. The LCR-algorithm architecture

This informal description can be formalized in CCS_v: We assume a set of channels $c_1, \ldots, c_n$, where channel c_i connects processes P_i and P_{i+1}, for $1 \leq i < n$, and c_n connects processes P_n and P_1. For simplicity, hereafter, we write $i+1$ for 1 if $i = n$, and $i+1$ otherwise, and, similarly, $i-1$ for n if $i = 1$, and $i-1$, otherwise. The behavior of a node-process is described as follows:

$$
\begin{aligned}
P_i\langle u_i\rangle &\stackrel{\text{def}}{=} S_i\langle u_i\rangle \parallel R_i\langle u_i\rangle \\
S_i\langle u_i\rangle &\stackrel{\text{def}}{=} \overline{c_i}(u_i) \\
R_i\langle u_i\rangle &\stackrel{\text{def}}{=} !\, c_{i-1}(x).\, \mathsf{cond}(x < u_i \;\; \triangleright \;\; \mathbf{0},
\end{aligned}
$$

$$\begin{array}{lcl} x > u_i & \triangleright & \overline{c_i}(x), \\ \mathsf{true} & \triangleright & \overline{leader}(u_i)) \end{array}$$

In $P_i\langle u_i\rangle$, u_i is the unique identifier of the process. The process has c_{i-1} as its input channel and c_i as its output channel. It is composed of two parallel processes: a sender, $S_i\langle u_i\rangle$, and a receiver, $R_i\langle u_i\rangle$. The function of the sender process is to emit the node's identifier along channel c_i, whereas, the receiver is continuously listening along the input port of channel c_{i-1} for messages. On a receipt of a message, it discards it in case it is smaller than u_i, it forwards it in a clockwise direction in case it is larger (as expressed by the clause $\overline{c_i}(x)$), and it declares itself the leader along the common channel *leader*, if the received identifier is equal to its own.

The network is represented by as the parallel composition of the n nodes

$$LCR = (P_1\langle u_1\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L$$

where $L = \{c_1, \ldots, c_n\}$ contains all channels whose use is restricted within the system. The intended behavior of the algorithm is that the node with the maximum identifier is elected as the leader. In process-calculus terminology, we prove the following correctness result:

Theorem 2. $LCR \approx \overline{leader}(u_{max})$, where $u_{max} = \max(u_1, \ldots, u_n)$.

The proof is carried out in two steps. First, we show that LCR is capable of producing the required leader output and terminate. Then, we establish that it is confluent. The required result then follows easily from properties of confluence.

Lemma 6. $LCR \Longrightarrow \overset{\overline{leader}(u_{max})}{\longrightarrow} \Longrightarrow \approx \mathbf{0}$

Proof: Without loss of generality, we assume that node P_1 is the owner of u_{max}. Let us write $LCR_1 = (R_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel (P_n\langle u_n\rangle \parallel \overline{c_n}(u_1)))\backslash L$. From the definitions, it can be seen that $LCR \Longrightarrow LCR_1$ in $n-1$ transitions where, in the i^{th} transition, processes P_i and P_{i+1} communicate on channel c_i forwarding u_1 from the former to the latter. Clearly, these communications are enabled due to the fact that $u_1 > u_i$ for all $i \neq 1$. Consequently, we may derive:

$$\begin{array}{rcl} LCR_1 & \xrightarrow{\tau} & (Q_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \\ & \xrightarrow{\overline{leader}(u_1)} & (R_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \end{array}$$

where $Q_1\langle u_1\rangle = \overline{leader}(u_1) \parallel R_1\langle u_1\rangle$. Let us now consider the remaining processes. For P_2, we have that u_2 can be forwarded for at most $n-2$ steps. That is, there is a transition

$$(R_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \Longrightarrow (R_1\langle u_1\rangle \parallel R_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L$$

and, similarly, all u_i, $i > 2$, can proceed up to at most node P_1, and then become blocked:

$$(R_1\langle u_1\rangle \parallel R_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \Longrightarrow LCR_2 = (R_1\langle u_1\rangle \parallel \ldots \parallel R_n\langle u_n\rangle)\backslash L$$

By observing the definitions of the processes, we conclude that, since all processes are only willing to receive, but none is ready to send, the resulting process is a deadlocked process, that is, a process bisimilar to **0**. This completes the proof. □

We now have our key observation:

Lemma 7. *LCR* is confluent.

PROOF: This follows from Theorem 1 of the previous section. We check its hypotheses: Each P_i is clearly a composition of two confluent components. Thus, by Lemma 4(4), it is confluent. Since, in addition, (1) by Lemma 5 all processes are i-confluent, (2) exactly one process takes input from each channel c_i and (3) each $c_i \in L$, by Theorem 1, we may conclude that *LCR* is a confluent process. □

From the two previous results, and since confluence implies τ-inertness, we have that $LCR \approx LCR_1$, $LCR_1 \approx \overline{leader}(u_{max}).\, LCR_2$ and $LCR_2 \approx \mathbf{0}$. Consequently, $LCR \approx \overline{leader}(u_{max})$, as required.

4.2 The HS Algorithm

The second algorithm that we study is the HS algorithm of Hirschberg and Sinclair [3]. It is distinguished from the LCR algorithm in that here communication between the ring nodes is bidirectional, and the time complexity is $O(n \lg n)$.

In the HS algorithm, execution at a node proceeds in phases. In phase k, a node forwards its identifier in both directions in the ring. The intention is that the identifier will travel a distance of 2^k, assuming that it does not encounter a node with a greater identifier and, then, it will follow the opposite direction returning to its origin node. If both tokens return to their origin, the node enters phase $k+1$ and continues its execution. If the identifier completes a cycle and reaches its origin in the outbound direction, then, the node declares itself as the leader. For the correct execution of the algorithm, messages transmitted in the ring are triples of the form $\langle id, dir, dist\rangle$, where id is a node identifier, $dir \in \{in, out\}$ is the direction of the message (*in* represents the inward direction and *out* the outward direction) and *dist* is the distance the node has still to travel.

Let us describe the precise behavior of an HS-node in CCS_v. The architecture considered is shown in Fig. 2. We assume a set of channels $\{c_{i,i-1}, c_{i,i+1} \mid 1 \leq i \leq n\}$, where channel $c_{i,j}$ connects process P_i to process P_j. (Note that we employ the same interpretation of the summation and substraction operators as described in Sect. 4.1.)

A node-process is then modeled as follows, where u_i is the identifier of the process and ϕ_i the phase of the process, initially set to 0.

$$P_i\langle u_i, \phi_i\rangle \stackrel{\text{def}}{=} (S_i\langle u_i, \phi_i\rangle \parallel R_i\langle u_i\rangle \parallel E_i\langle u_i\rangle)\backslash\{cph, elect\}$$

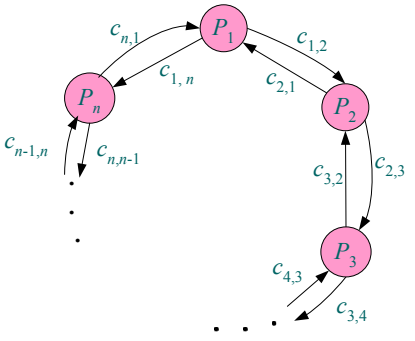


Fig. 2. The HS-algorithm architecture

Thus P_i is the parallel composition of three components responsible for sending and receiving messages and for electing a leader:

$$
\begin{aligned}
E_i\langle u_i\rangle &\stackrel{\text{def}}{=} elect.\,\overline{leader}(u_i)\\
S_i\langle u_i,\phi_i\rangle &\stackrel{\text{def}}{=} \overline{c_{i,i+1}}(u_i, out, 2^{\phi_i})\\
&\quad \parallel \overline{c_{i,i-1}}(u_i, out, 2^{\phi_i})\\
&\quad \parallel cph.\,cph.\,S_i\langle u_i, \phi_i + 1\rangle\\
R_i\langle u_i\rangle &\stackrel{\text{def}}{=} !\,c_{i-1,i}(x,d,h).\\
&\qquad \mathsf{cond}((x < u_i) \quad \triangleright \quad \mathbf{0},\\
&\qquad\quad (x > u_i \wedge d = out \wedge h \neq 1) \quad \triangleright \quad \overline{c_{i,i+1}}(x, out, h-1),\\
&\qquad\quad (x > u_i \wedge d = out \wedge h = 1) \quad \triangleright \quad \overline{c_{i,i-1}}(x, in, 1),\\
&\qquad\quad (x \neq u_i \wedge d = in) \quad \triangleright \quad \overline{c_{i,i+1}}(x, in, 1),\\
&\qquad\quad (x = u_i \wedge d = in) \quad \triangleright \quad \overline{cph},\\
&\qquad\quad \mathsf{true} \quad \triangleright \quad \overline{elect}\,)\\
&\quad \parallel !\,c_{i+1,i}(x,d,h).\\
&\qquad \mathsf{cond}((x < u_i) \quad \triangleright \quad \mathbf{0},\\
&\qquad\quad (x > u_i \wedge d = out \wedge h \neq 1) \quad \triangleright \quad \overline{c_{i,i-1}}(x, out, h-1),\\
&\qquad\quad (x > u_i \wedge d = out \wedge h = 1) \quad \triangleright \quad \overline{c_{i,i+1}}(x, in, 1),\\
&\qquad\quad (x \neq u_i \wedge d = in) \quad \triangleright \quad \overline{c_{i,i-1}}(x, in, 1),\\
&\qquad\quad (x = u_i \wedge d = in) \quad \triangleright \quad \overline{cph},\\
&\qquad\quad \mathsf{true} \quad \triangleright \quad \overline{elect}\,)
\end{aligned}
$$

Thus, E_i awaits a notification (triggered by process R_i) that a leader-event should be produced. S_i emits the message $(u_i, out, 2^{\phi_i})$ in both directions along the ring. Concurrently, it waits the receipt of two confirmations via channel cph (emitted by process R_i) that the token has successfully travelled in both directions through the ring, in which case it increases the phase by 1. On the other hand, process R_i is listening on ports $c_{i,i-1}$ and $c_{i,i+1}$. The first summand of the process deals with the former channel, and the second with the latter. We consider the first summand, the second one is symmetric. If a message (x, d, h) is received on channel $c_{i-1,i}$, six cases exist:

1. If $x < u_i$, the message is ignored.
2. If the message is travelling in the outbound direction and $h > 1$, it is forwarded to node $i + 1$ with h decreased by 1.
3. If the message is travelling in the outbound direction and $h = 1$, it is sent back to node $i - 1$ to begin its inward journey.
4. If $x \neq u_i$ and the message is travelling its inward journey, it is forwarded towards its origin.
5. If the node's own identifier is received by the process while travelling its inward journey, process R_i emits a notification along channel *cph*.
6. Finally, if none of the above holds, implying that $x = u_i$ and $d = out$ (that is the identifier has survived performing a cycle around the ring), the node produces a notification (to be received by process E_i) that the node should be declared ring leader.

The network is represented by the parallel composition of the n nodes

$$HS = (P_1\langle u_i, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L$$

where all channels in $L = \{c_{i,i-1}, c_{i,i+1} \mid 1 \leq i \leq n\}$ are restricted within the system. The correctness criterion is expressed, again, as the following equivalence between the algorithm and its specification:

Theorem 3. $HS \approx \overline{leader}(u_{max})$, where $u_{max} = \max(u_1, \ldots, u_n)$.

As before, the proof is carried out in two steps: We show that HS is capable of producing the required leader output and terminate, and that it is confluent.

Lemma 8. $HS \Longrightarrow \overset{\overline{leader}(u_{max})}{\longrightarrow} \Longrightarrow \approx \mathbf{0}$

Proof: Without loss of generality, we assume that node P_1 is the owner of u_{max}. Let us write $HS_i = (P_1\langle u_1, i\rangle \parallel P_2\langle u_2, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L$. From the definitions, it can be seen that

$$HS \Longrightarrow HS_1 \Longrightarrow HS_2 \Longrightarrow \ldots \Longrightarrow HS_k$$

where $k = \lceil \lg n \rceil$. Specifically, the transition $HS_i \Longrightarrow HS_{i+1}$ consists of $2 \cdot 2 \cdot 2^i$ internal actions pertaining to the outward and inward journey of distance 2^i, of identifier u_1 in the clockwise and anticlockwise direction within the ring. Clearly, these communications are enabled due to the fact that $u_1 > u_i$ for all $i \neq 1$. Then, we may derive:

$$\begin{aligned} HS_k \ (\overset{\tau}{\longrightarrow})^{n+1} \ & HS_{k+1} = (P_{1,1}\langle u_1, k\rangle \parallel P_2\langle u_2, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L \\ \overset{\overline{leader}(u_1)}{\longrightarrow} \ & HS_{k+2} = (P_{1,2}\langle u_1, k\rangle \parallel P_2\langle u_2, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L \end{aligned}$$

where $P_{1,2}\langle u_1, k\rangle = (\overline{c_{i,i-1}}(u_i, out, 2^k) \parallel cph.\, cph.\, S\langle u_1, k\rangle \parallel R_1\langle u_1\rangle)\backslash\{cph, elect\}$ and $P_{1,1}\langle u_1, k\rangle = P_{1,2}\langle u_1, k\rangle \parallel \overline{leader}(u_1)$. These $n + 1$ internal steps correspond to a cycle of u_1 in a clockwise direction, and its return to its origin node while in

the outbound direction. This triggers R_1 to produce the message *elect*, received by process E_i which in turn proceeds to declare the node as the leader.

Let us now consider the remaining enabled communications. First note that the prefix $\overline{c_{i,i-1}}\ (u_1, out, 2^k)$ can trigger a cycle of u_1 in the anticlockwise direction, while all other processes will forward their identifiers in the clockwise and anticlockwise direction. It is easy to show that the journey of each of these identifiers will eventually become blocked, on reaching a node with a larger identifier, possibly P_1. This gives rise to the transition

$$HS_{k+2} \Longrightarrow HS_{k+3} = (P_{1,3}\langle u_1, k\rangle \parallel Q_2\langle u_2\rangle \parallel \ldots \parallel Q_n\langle u_n\rangle)\backslash L$$

where $P_{1,3}\langle u_1, k\rangle = (cph.\, cph.\, S\langle u_1, k\rangle \parallel R_1\langle u_1\rangle \| \overline{elect}\rangle)\backslash\{cph, elect\}$, and for $2 \leq i \leq n$, $Q_i\langle u_i\rangle = S'_i\langle u_i, \phi_i\rangle \parallel R_i\langle u_i\rangle \parallel E_i\langle u_i\rangle)\backslash\{cph, elect\}$ where $S'_i\langle u_i, \phi_i\rangle$ is one of the processes $cph.\, cph.\, S_i\langle u_i, \phi_i\rangle$ and $cph.\, S_i\langle u_i, \phi_i\rangle$. By observation, no communication is enabled in the resulting process, a fact that renders it bisimilar to **0**. This completes the proof. □

We now have our key observation:

Lemma 9. *HS* is confluent.

Proof: This follows from a multiple application of Theorem 1. To establish the confluence of a P_i we observe that, by Lemma 5, each of E_i, S_i and R_i is i-confluent. Lemma 4 and simple observation leads to the conclusion that the components are also confluent. Since the components share between them only the names *cph* and *elect*, which are hidden at the top level of the process, and no two components share an input name, by Theorem 1, each P_i is confluent.

Since, in addition, (1) all P_i are i-confluent, (2) exactly one process takes input from each channel $c_{i,j}$, and (3) each $c_{i,j} \in L$, by Theorem 1, we may conclude that *HS* is a confluent process. □

From the two previous results, and since confluence implies τ-inertness, we have that $HS \approx HS_{k+1}$, $HS_{k+1} \approx \overline{leader}(u_{max}).\, HS_{k+2}$ and $HS_{k+2} \approx \mathbf{0}$. Consequently, $HS \approx \overline{leader}(u_{max})$, as required.

5 Conclusions

We have considered an asynchronous process calculus and we have developed its associated theory of confluence. In doing this, our main objective has been the elaboration of concepts and techniques useful in proving the correctness of distributed algorithms. Specifically, we have given results for establishing the confluence of systems in a compositional manner and we have exploited the property of τ-inertness possessed by confluent systems for showing that systems are correct. Using these ideas, we have illustrated the correctness of two leader-election algorithms. As we have already mentioned, these two algorithms were also proved correct in [6] using the I/O-Automata framework. In our view, our proofs offer additional interesting insights in that the use of confluence aids towards the algorithms' understanding and simplifies their analysis.

Regarding the applicability of the proposed methodology, initially, it appears that it can be useful in a variety of contexts were confluent computations are running in parallel, e.g., parallel and distributed algorithms for function computation. In future work, we plan to further investigate the applicability of the methodology and, especially, the notion of F^i-confluence. (We are currently considering application of the results to verify distributed leader-election algorithms in networks of arbitrary topological structures). Further, we would like to extend the theory to a setting with mobility and location primitives. Naturally, the property of confluence is not satisfied in general by distributed algorithms, thus, a final research direction is the development of analogous results for algorithm verification which employ weaker partial-confluence properties.

References

1. G. Boudol. Asynchrony and the π-calculus. Technical Report RR-1702, INRIA-Sophia Antipolis, 1992.
2. J. F. Groote and M. P. A. Sellink. Confluence for process verification. In *Proceedings of CONCUR'95*, LNCS 962, pages 204–218, 1995.
3. D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations. *Communications of the ACM*, 23(11):627–628, 1980.
4. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, LNCS 512, pages 133–147, 1991.
5. X. Liu and D. Walker. Confluence of processes and systems of objects. In *Proceedings of TAPSOFT'95*, LNCS 915, pages 217–231, 1995.
6. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.
7. N. A. Lynch and M. R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, 1989.
8. R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.
9. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
10. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1 and 2. *Information and Computation*, 100:1–77, 1992.
11. U. Nestmann. *On Determinacy and Non-determinacy in Concurrent Programming*. PhD thesis, University of Erlangen, 1996.
12. D. Park. Concurrency and automata on infinite sequences. In *Proceedings of* 5^{th} *GI Conference*, LNCS 104, pages 167–183, 1981.
13. A. Philippou and D. Walker. On transformations of concurrent object programs. In *Proceedings of CONCUR'96*, LNCS 1119, pages 131–146, 1996.
14. A. Philippou and D. Walker. On confluence in the π-calculus. In *Proceedings of ICALP'97*, LNCS 1256, pages 314–324, 1997.
15. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
16. M. Sanderson. *Proof Techniques for CCS*. PhD thesis, University of Edinburgh, 1982.
17. D. Sangiorgi. A theory of bisimulation for the π-calculus. In *Proceedings of CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 1993.
18. C. Tofts. *Proof Methods and Pragmatics for Parallel Programming*. PhD thesis, University of Edinburgh, 1990.

Mobile Agent Algorithms Versus Message Passing Algorithms

J. Chalopin[1], E. Godard[2], Y. Métivier[1], and R. Ossamy[1]

[1] LaBRI UMR 5800
ENSEIRB - Université Bordeaux 1 351 Cours de la Libération 33405 - Talence France
{chalopin, metivier, ossamy}@labri.fr
[2] LIF UMR 6166 Université de Provence 39 rue Joliot-Curie 13453 Marseille France
egodard@cmi.univ-mrs.fr

Abstract. In this paper, we are interested in the computational power of a mobile agent system and, more particularly, in the comparison with a message passing system. First we give formal definitions. Then we explain how a mobile agent algorithm can be simulated by a message passing algorithm. We also prove that any message passing algorithm can be implemented by a mobile agent algorithm. As a consequence of this result, known characterisations of solvable tasks by message passing algorithms can be translated into characterisations of solvable tasks by mobile agent algorithms. We illustrate this result with the election problem.

1 Introduction

The mobile agent paradigm has been developed to solve problems in dynamic and heterogeneous environment [8]. The agent model of this paper is quite general. It is based on the concepts of agents, communication links and places. An agent is an entity which executes an algorithm: it can move from place to place (with some data and its algorithm) through communication links and it can make local computations on a place (a place provides tools for local computations: data, memories and process). Thus a mobile agent system is defined:

- by a network or equivalently by an undirected labelled graph (the vertices correspond to the places) with a port numbering function,
- by a set of agents, and
- by an initial placement of the agents on the graph.

As a particular case of our model the network and the mobile agents may be anonymous: identities are available neither for the vertices of the network nor for the agents. A mobile agent (based) algorithm is defined by a mobile agent system where each agent is endowed with its proper algorithm. Classical problems for mobile agents include: election, naming, locating agents, rendez-vous, stabilisation, termination detection of agents, exploring, topology recognition.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 187–201, 2006.

There exist a lot of results for these problems assuming different properties of the environment [2,4,5,6,10,11,12]. A message passing system is a set of processes and a communication subsystem. It corresponds to the standard models given in [1,16]. The communication model is a point-to-point communication network which is represented as a simple connected undirected graph where the vertices represent processes and two vertices are linked by an edge if the corresponding processes have a direct communication link. Processes communicate by message passing, and each process knows from which channel it receives a message or it sends a message. We consider the asynchronous message passing model: processes cannot access a global clock and a message sent from a process to a neighbor arrives within some finite but unpredictable time. A message passing algorithm is a collection of local algorithms, one for each process. The anonymous case corresponds to the case where all local algorithms are the same.

In this paper we are interested in the computational power of a mobile agent system and, more particularly, in the comparison with a message passing system. It is easy to verify that a message passing system can simulate a mobile agent system (Section 4); it has been already indicated in [3]. But, given a mobile agent system, can mobile agents be used to implement arbitrary message passing algorithms? We give a positive answer in Section 5; and we obtain a theorem stating the equivalency of the two models (Theorem 10). Our result establishes a useful bridge between message passing systems and mobile agent systems. In Section 6, we give an example of consequence of this result with a translation of known results in message passing computing for the election problem into the mobile agents setting. In [10], it is proved that a leader can be elected among k agents on a graph having n vertices if k and n are co-prime; this result becomes a consequence of the characterisation given in [17] and of Theorem 10. Theorem 2 and Theorem 3 of [5] become corollaries of results presented in [13]. The same method can be applied for other classical problems such as the naming, the topology recognition, the spanning tree construction, etc.

2 Graphs and Labelled Graphs

We consider finite undirected connected graphs. A graph $G = (V(G), E(G))$ (or $G = (V, E)$ for short) is defined by a set $V(G)$ of vertices and a set $E(G)$ of edges; in this paper graphs are without multiple edges or self-loop. Two vertices u and v are said to be adjacent or neighbors if $\{u, v\}$ is an edge of G (thus u and v are necessarily distinct since no self-loop is admitted) and $N_G(v)$ will stand for the set of neighbors of v; u and v are the endvertices of e. An edge e is incident to a vertex v if $v \in e$ and $I_G(v)$ will stand for the set of all the edges incident to v. A homomorphism between graphs G and H is a mapping $\gamma\colon V(G) \to V(H)$ such that if $\{u, v\} \in E(G)$ then $\{\gamma(u), \gamma(v)\} \in E(H)$. We say that γ is an isomorphism if γ is bijective and γ^{-1} is a homomorphism.

Throughout the rest of this paper we will consider graphs whose vertices are labelled with labels from a recursive set L. A graph labelled over L will be denoted by (G, λ), where G is a graph and $\lambda\colon V(G) \to L$ is the vertex labelling

function. The graph G is called the underlying graph and the mapping λ is a labelling of G. Labelled graphs will be designated by bold letters like $\mathbf{G}$, $\mathbf{H}, \ldots$ If $\mathbf{G}$ is a labelled graph, then G denotes the underlying graph.

A mapping $\gamma\colon V(G) \to V(G')$ is a homomorphism from (G, λ) to (G', λ') if γ is a graph homomorphism from G to G' which preserves the labelling, i.e., such that $\lambda'(\gamma(v)) = \lambda(v)$ holds for every $v \in V(G)$.

3 The Formal Models

3.1 The Message Passing Model

Definitions given in this subsection follow [16] (p. 45-47) or [1] (p. 10-12).

Message Passing System. A message passing system (P, C) consists of a collection P of processes and a communication subsystem C. It is described by a simple connected undirected graph $G = (V, E)$, where the vertices represent the processes and the edges represent the bidirectional channels. The system is asynchronous: no global time is available; messages can arrive at arbitrary times and processes can take steps at arbitrary speeds. Processes communicate by asynchronous message passing and each process knows from which channel it receives a message or it sends a message: an edge between two processes p_1 and p_2 (or vertices v_1 and v_2) represents a channel connecting a port i of p_1 (or v_1) to a port j of p_2 (or v_2). Let δ be the port numbering function, we assume that for each vertex u (or process p) and each adjacent vertex v (or process q), $\delta_u(v)$ (or $\delta_p(q)$) is a unique integer belonging to $[1, deg(u)]$. Finally, the communication subsystem is described by $C = (V, E, \delta)$. Each process has an initial state defined by a labelling function λ. Thus the message passing system is defined by (P, C, λ) or equivalently by (V, E, δ, λ).

Remark 1. The labelling λ of processes may encode anonymous network (all the vertices have the same label) or any initial process knowledge. Examples of such knowledge include: (a bound on) the number of processes, (a bound on) the diameter of the communication subsystem, the topology of the communication subsystem, identities or partial identities of processes, distinguished processes, sense of direction.

Message Passing Algorithm. To each process is associated a transition system which can interact with the communication subsystem. The events which are associated with a process are internal events, send events and receive events. In a send (resp. receive) event a message is produced (resp. consumed). This definition contains the particular case where every processor executes the same algorithm.

Remark 2. In general, names are not available to the processes themselves. Nevertheless, for ease of exposition, a message m in transit is denoted by (p, m, p') where p is the sending process and p' is the receiving process.

Let $\mathcal{M}$ be the set of possible messages. Let p be a process. The local algorithm of the process p, denoted by $\mathcal{D}_p$, is defined by:

- the (recursive) set Q of possible states of p,
- the subset I of Q of initial states,
- the initial state of p equals to $\lambda(p)$,
- a relation $\vdash_p$ of events (internal events, send events or receive events).

Let p be a process. Let M be the multiset of messages in transit (initially M is empty). The state associated to p is denoted by $\texttt{state}(p)$. The transition associated to the process p is denoted by:

$$(c, in, m) \vdash_p (d, out, m'),$$

(where c and d are states, in and out are integers, and m and m' are messages) means that:

- if $in = out = 0$ then $m = m' = \perp$ (m and m' are undefined): it is an internal event, the new state of the process p is d (it was c before);
- if $in \neq 0$ then $out = 0$ and $m' = \perp$ (m' is undefined): it is a receive event, the state of the process p was c, p has received the message m through the port in and its new state is d; an occurence of m (of the form (p', m, p) where $\delta_p(p') = in$) is removed from M;
- if $out \neq 0$ then $in = 0$ and $m = \perp$ (m is undefined): it is a send event, initially the state of the process p is equal to c; after the transition it is equal to d and the message m' is sent via the port out; an occurence of m' (of the form (p, m', p') where $\delta_p(p') = out$) is added to M.

A message passing algorithm $\mathcal{D}$ for the message passing system (P, C, λ) is a collection of local algorithms $\mathcal{D}_p$, one for each process $p \in P$. It is denoted by $\mathcal{D} = (\mathcal{D}_p)_{p \in P}$. An event of the message passing algorithm is defined by an event on a process.

Execution of a Message Passing Algorithm. An execution $\mathcal{E}$ of the message passing algorithm is defined by a sequence $(\texttt{state}_0, M_0)$, $(\texttt{state}_1, M_1), \ldots,$ $(\texttt{state}_i, M_i), \ldots$ such that:

- for each i, M_i is the multiset of messages in transit,
- $M_0 = \emptyset$,
- for each i and for each process p, $\texttt{state}_i(p)$ denotes the state of the process p,
- for each process p, $\texttt{state}_0(p) = \lambda(p) \in I$,
- for each i, there exists a unique process p such that:
  - if $p' \neq p$ then $\texttt{state}_{i+1}(p') = \texttt{state}_i(p')$,
  - $\texttt{state}_{i+1}(p)$ and M_{i+1} are obtained from $\texttt{state}_i(p)$ and M_i by an event on the process p.

By definition, $(\texttt{state}_i, M_i)$ is a configuration. The execution $\mathcal{E}$ of the message passing algorithm is defined by:

$$\mathcal{E} = (\texttt{state}_i, M_i)_{i \geq 0}.$$

A terminal configuration is a configuration for which no more event is applicable. We can note that in a terminal configuration the multiset M of messages in transit is empty. In this case the length of the execution is the length of the sequence.

Termination Detection. Definitions given in this section are in [16] (Chapter 8). A state q of a process p is active if an internal or send event of p is applicable in q, and passive otherwise. In a passive state q only receipts are applicable, or no event is applicable at all, in which case q is a terminal state of p. Process p is said to be active if it is in an active state, and process p is said to be passive otherwise.

As it is explained in [16] (p. 270), some assumptions are made in order to simplify the description (any process can be easily modified to satisfy these assumptions) : an active process becomes passive only in an internal event, a process always becomes active when a message is received, the internal events in which p becomes passive are the only internal events of p (internal events in which p moves from one active state to another active state are ignored).

A message passing algorithm has terminated when all processes are passive and no message is in transit. In this case, termination is said to be implicit if the processes are not aware that the algorithm has terminated.

Termination is said to be explicit if at least one process detects the termination of the message passing algorithm in the sense that all processes have computed their final values. It then can call an algorithm which floods a termination message to all processes. Only after explicit termination can the result of a computation be regarded as final and variables used in the computation discarded.

3.2 The Mobile Agent Model

Mobile Agent System. A mobile agent system consists of :

- a collection $\mathbb{P}$ of execution places (or places for short),
- a navigation subsystem $\mathbb{S}$,
- a collection $\mathbb{A}$ of mobile agents,
- an injection $\pi_0 : \mathbb{A} \longrightarrow \mathbb{P}$ describing the initial placement of the agents,
- an initial labelling λ of the places and the agents.

Remark 3. The labelling λ of the places and of the agents may encode anonymous places (all the places have the same label), anonymous agents (all the agents have the same label) or any initial agent knowledge. Examples of such knowledge include: (a bound on) the number of places, (a bound on) the number of agents, (a bound on) the diameter of the navigation subsystem, the topology of the navigation subsystem, the topology of the placement of the agents, identities or partial identities of places or agents, distinguished places or agents, sense of direction.

The navigation subsystem is described by a simple undirected connected graph $G = (V, E)$, where the vertices V represent execution places and the edges

represent bidirectionnal navigation channels operating between them. In the sequel, we identify the places and the corresponding vertices, and we identify the edges and the corresponding channels.

Each agent which migrates from a place to another place knows through which channel it migrates, that is, for each place port numbers are assigned to its ports : let u be a vertex, let δ_u be the port numbering function which assigns to each adjacent vertex v of u a unique integer $\delta_u(v)$ belonging to $[1, deg(u)]$. Thus the navigation subsystem is defined by $\mathbb{S} = (G, \delta)$.

The system is asynchronous: no global clock is accessible; a migration is asynchronous: an agent which migrates arrives within some finite but unpredictable time on a place.

Mobile Agent Algorithm. Given a mobile agent system, we define a mobile agent algorithm. To each mobile agent is associated a transition system that can interact with the execution places and the navigation subsystem.

Let $Q_{\mathbb{P}}$ be a (recursive) set of states associated to the execution places, and let $Q_{\mathbb{A}}$ be a (recursive) set of states associated to the mobile agents. The initial state of each mobile agent $\mathbf{a}$ is $\lambda(\mathbf{a})$ and the initial state of each execution place $\mathbf{p}$ is $\lambda(\mathbf{p})$.

Let $\mathbf{p}$ be a place. We denote by $\texttt{state}(\mathbf{p})$ the state associated to $\mathbf{p}$. Let $\mathbf{a}$ be an agent. We denote by $\texttt{state}(\mathbf{a})$ the state associated to $\mathbf{a}$.

The transition associated to the mobile agent $\mathbf{a}$ in the state s on the place $\mathbf{p}$ in the state q, transforms s into s', q into q' and either $\mathbf{a}$ does not move or it migrates on an adjacent place through the port *out*. We denote the transition by :

$$(s, q, in) \vdash_{\mathbf{p}}^{\mathbf{a}} (s', q', out),$$

it means that the mobile agent $\mathbf{a}$ has migrated on the place $\mathbf{p}$ through the port *in* or after the transition it leaves the place $\mathbf{p}$ through the port *out*, with the convention that if the agent was already on the place and it does not move after the transition then $in = 0$ and $out = 0$; furthermore *in* and *out* cannot be simultaneously different from 0.

Remark 4. In general, names are not available to the places themselves. Nevertheless, for ease of exposition, an agent $\mathbf{a}$ in transit is denoted by $(p, \mathbf{a}, p')$ where p is the place of departure and p' is the place of arrival.

A configuration of the mobile agent system consists of the state of each place, the state of each agent, the collection $\mathbb{M}$ of agents in transit (initially $\mathbb{M}$ is empty) and a mapping π describing the placement of the agents which are not in a channel (several agents can be on the same place).

An event in the mobile agent system is defined by a transition associated to an agent $\mathbf{a}$ on a place $\mathbf{p}$, it has the form :

$$(s, q, in) \vdash_{\mathbf{p}}^{\mathbf{a}} (s', q', out),$$

the state of each agent different from $\mathbf{a}$ is not affected, the state of each place different from $\mathbf{p}$ is not affected, the new state of $\mathbf{a}$ is s' (it was s before the event), the new state of $\mathbf{p}$ is q' (it was q before the event), and:

- if $in = 0$ and $out = 0$ then π and $\mathbb{M}$ are not affected by the event,
- if $in = 0$ and $out \neq 0$ then the set of agents in transit after the event is $\mathbb{M} \cup \{(\mathbf{p}, \mathbf{a}, \mathbf{p}')\}$ (where $\mathbf{p}'$ is the adjacent place of $\mathbf{p}$ corresponding to the port out,) and π is no more defined for $\mathbf{a}$ and unchanged for the other agents,
- if $in \neq 0$ and $out = 0$ then the set of agents in transit after the event is $\mathbb{M} \setminus \{(\mathbf{p}', \mathbf{a}, \mathbf{p})\}$ (where $\mathbf{p}'$ is the adjacent place of $\mathbf{p}$ corresponding to the port in), $\pi(\mathbf{a}) = \mathbf{p}$ and π is unchanged for the other agents.

Execution. An execution of the mobile agent algorithm is defined by a sequence $(\mathtt{state}_0, \mathbb{M}_0, \pi_0), (\mathtt{state}_1, \mathbb{M}_1, \pi_1), ..., (\mathtt{state}_i, \mathbb{M}_i, \pi_i), ...$ such that :

- $\mathbb{M}_0 = \emptyset$,
- for each agent $\mathbf{a}$, $\mathtt{state}_0(\mathbf{a}) = \lambda(\mathbf{a})$ is an initial state,
- for each place $\mathbf{p}$, $\mathtt{state}_0(\mathbf{p}) = \lambda(\mathbf{p})$ is an initial state,
- π_0 is the initial placement of agents,
- for each i there exists a unique place $\mathbf{p}$ and a unique agent $\mathbf{a}$ such that:
  - if $\mathbf{p}' \neq \mathbf{p}$ then $\mathtt{state}_{i+1}(\mathbf{p}') = \mathtt{state}_i(\mathbf{p}')$,
  - if $\mathbf{a}' \neq \mathbf{a}$ then $\mathtt{state}_{i+1}(\mathbf{a}') = \mathtt{state}_i(\mathbf{a}')$,
  - $(\mathtt{state}_{i+1}(\mathbf{a}), \mathtt{state}_{i+1}(\mathbf{p}), \mathbb{M}_{i+1}, \pi_{i+1})$ is obtained from $(\mathtt{state}_i(\mathbf{a}), \mathtt{state}_i(\mathbf{p}), \mathbb{M}_i, \pi_i)$ by an event of the form :

$$(s, q, in) \vdash_{\mathbf{p}}^{\mathbf{a}} (s', q', out).$$

A configuration is defined by $(\mathtt{state}_i, \mathbb{M}_i, \pi_i)$. A terminal configuration is a configuration for which no more event can appear; we can note that in this case the collection of agents in transit is empty. By definition, the length of the sequence is the length of the execution.

A place which is the initial place of an agent is called an homebase. The initial placement of the agents can be encoded in the state of the place (it can be the first action of each mobile agent) thus we assume that the state of a place enables to know whether it is a homebase or not. Nevertheless, in general, an agent cannot know whether a homebase is its own homebase.

Finally, the mobile agent system is defined by:

$$(\mathbb{A}, \mathbb{P}, \mathbb{S}, \pi_0, \lambda),$$

the mobile agent algorithm is defined by:

$$\mathcal{A} = (\vdash_{\mathbf{p}}^{\mathbf{a}})_{\mathbf{a} \in \mathbb{A}, \mathbf{p} \in \mathbb{P}},$$

and an execution $\mathcal{E}$ is defined by :

$$\mathcal{E} = (\mathtt{state}_i, \mathbb{M}_i, \pi_i)_{i \geq 0}.$$

Termination Detection. An agent $\mathbf{a}$ in state s on the place $\mathbf{p}$ in state q is said passive if no transition is associated to this configuration. In a terminal configuration all the agents are passive. Termination is said implicit if no agent is aware that the mobile agent algorithm has terminated. Termination is said explicit if at least one agent detects the termination of the mobile agent algorithm in the sense that all places have their final values. If at least one agent detects the termination then a termination annoucement algorithm using the agents which have detected termination can be activated.

3.3 Equivalent Executions

We consider mobile agent algorithms and message passing algorithms such that the graph corresponding to the navigation subsystem and the graph corresponding to the communication subsystem are equal.

Various kinds of equivalences between mobile agent algorithms and message passing algorithms can be defined. In this work, we consider algorithms which always terminate.

A mobile agent algorithm and a message passing algorithm are equivalent for the terminal configurations if the set of graphs corresponding to the navigation subsystem labelled by the final states of places and the set of graphs corresponding to the communication subsystem labelled by the final states of processes are equal and if the termination of the two algorithms is implicit or the termination of the two algorithms is explicit.

4 Simulating a Mobile Agent Algorithm Through a Message Passing Algorithm

The purpose of this section is to verify that, given a mobile agent algorithm, it is possible to implement the same algorithm through a message passing algorithm. To reach this goal, we intend to prove that all the agents basic computation steps can be effectively simulated in the asynchronous message passing system. The main idea of this proof appears in [3].

Let $(\mathbb{A}, \mathbb{P}, \mathbb{S}, \pi_0, \lambda)$ be a mobile agent system and let $\mathcal{A} = (\vdash_{\mathbf{p}}^{\mathbf{a}})_{\mathbf{a}\in\mathbb{A},\mathbf{p}\in\mathbb{P}}$ be a mobile agent algorithm implemented on this system. Let $\mathbb{S} = (V, E, \delta)$ be the corresponding navigation subsystem. We assume that the system contains k agents. We define an additional labelling χ_{π_0} of the vertices of G such that $\chi_{\pi_0}(v) = 1$ if there exists an agent $\mathbf{a}$ such that $\pi_0(\mathbf{a}) = v$, and $\chi_{\pi_0}(v) = 0$ otherwise. Starting from the mobile agent system we build up a message passing system $(P, C, \lambda') = (V, E, \delta, \lambda')$. On each vertex v which corresponds to an execution place $\mathbf{p}$ we install a process p. Let $\sharp$ be a new label. The labelling function λ' encodes on each process p the label of the corresponding place $\mathbf{p}$, whether the place is a homebase and, in the case of a homebase, the label of the corresponding mobile agent $\mathbf{a}$, i.e., if v corresponds to the homebase of $\mathbf{a}$ $\lambda'(v) = (\lambda(v), 1, \lambda(\mathbf{a}))$ and if not $\lambda'(v) = (\lambda(v), 0, \sharp))$.

Now we build a message passing algorithm $\mathcal{D}$ such that each execution $\mathcal{E}$ of $\mathcal{A}$ can be simulated by an execution $\mathcal{E}'$ of $\mathcal{D}$. A state of a process is defined by: - the state of the corresponding place, - the presence of a mobile agent is encoded by a token and the state of the corresponding mobile agent is encoded by the value of the token. Finally, the set of possible states of the processes is the set of possible states defined by the places, the presence of the token and if there is a token by the state of the corresponding agent.

The presence of an agent $\mathbf{a}$ at a given vertex u is represented by the token $t(\mathbf{a})$ located at u. Each token has a *homebase* that corresponds to the initial location of the corresponding mobile agent. To each token is associated a state: the current state of the token $t(\mathbf{a})$ is equal to the state of the agent $\mathbf{a}$.

The translation of the event:

$$(s, q, in) \vdash_{\mathbf{p}}^{\mathbf{a}} (s', q', out)$$

of the mobile agent algorithm into an event of the message passing algorithm is done according to the following rules.

The state of each token different from the token which is associated to $\mathbf{a}$ is not affected, the state of each process different from p is not affected, the new state of the token associated to $\mathbf{a}$, i.e., $t(\mathbf{a})$, is s' (it was s before the event), the new state of p is q' (it was q before the event), and

- if $in = 0$ and $out = 0$ then the token $t(\mathbf{a})$ does not move,
- if $in = 0$ and $out \neq 0$ then the token $t(\mathbf{a})$ is sent via the port out,
- if $in \neq 0$ and $out = 0$ then the token $t(\mathbf{a})$ is received by the process p via the port in.

Let $\mathcal{D}_p$ be the algorithm induced by this construction on the process p. Let $\mathcal{D} = (\mathcal{D}_p)_{p \in P}$. By an induction on the length of the executions, we prove that if the mobile agent algorithm has the termination property then the message passing algorithm defined above has also the termination property. The message passing algorithm $\mathcal{D}$ terminates explicitly if and only if $\mathcal{A}$ terminates explicitly. A graph corresponding to the navigation subsystem labelled by the final states of places is obtained with $\mathcal{A}$ if and only if it can be obtained as a graph corresponding to the communication subsystem labelled by the final states of processes with $\mathcal{D}$. Finally:

Proposition 5. *Let $(\mathbb{A}, \mathbb{P}, \mathbb{S}, \pi_0, \lambda)$ be a mobile agent system.*
Let $\mathcal{A} = (\vdash_{\mathbf{p}}^{\mathbf{a}})_{\mathbf{a} \in \mathbb{A}, \mathbf{p} \in \mathbb{P}}$ be a mobile agent algorithm implemented on this system. Let (P, C, λ') be the message passing system built above. Let $\mathcal{D} = (\mathcal{D}_p)_{p \in P}$ be the message passing algorithm defined above. Then the executions of $\mathcal{D}$ are equivalent to the executions of $\mathcal{A}$.

5 Simulating a Message Passing Algorithm Through a Mobile Agent Algorithm

As it is mentionned by Tel ([16] p. 46), the transition systems serve as a theoretical model and algorithms are not necessarily described by an enumeration of their states and events but by means of variables and a convenient pseudocode (see [16], Appendix A).

In this section we turn our attention to the presentation of a procedure that, given a message passing algorithm $\mathcal{D}$ over the message passing system (V, E, δ, λ) and a number $k \geq 1$, generates an equivalent mobile agent algorithm $\mathcal{A}$ with k agents over a mobile agent system.

Our procedure works as follows. The navigation subsystem corresponds to (V, E, δ). The state of the place $\mathbf{p}$ which corresponds to the process p and (which is identified to the vertex v) is defined by the state of the process p (with the same initialisation) and by the values of the variables defined below. The states and the algorithms associated to the mobile agents are defined in the sequel.

Procedure 1

Step 1: *On wake-up, an agent* $\mathbf{a}$ *constructs a tree* $T_{\mathbf{a}}$ *using a partial traversal of the graph. This leads to a spanning forest of* k *trees where each tree is constructed by exactly one agent.*

Step 2: *The agent* $\mathbf{a}$ *executes the algorithm* $\mathcal{D}$ *on the vertices of* $T_{\mathbf{a}}$*. This execution is performed in rounds, where, in each round,* $T_{\mathbf{a}}$ *is traversed and, if possible,* d $(d \geq 1)$ *computation steps of* $\mathcal{D}$ *are executed at each vertex of* $T_{\mathbf{a}}$*. Thus,* $\mathbf{a}$ *is responsible for the computation steps performed on the vertices that belong to its constructed tree.*

Due to the fact that the vertices and the agents are possibly anonymous and because of a lack of global orientation in the network, it may be difficult for an agent to find its way through the graph. To overcome this problem, the agents have to make use of the local edge labelling informations in order to keep track of their way. Therefore, we must keep in mind that each edge $e = \{u, v\}$ has two labels $\delta_u(v)$ and $\delta_v(u)$ that respectively correspond to the labels of e at u and v. Whenever an agent traverses the graph, it stores in its memory the ordered sequence of the labels of the traversed edges. In the rest of this section *ePath* (exploration path) will refer to this sequence. When the edge e is traversed by an agent $\mathbf{a}$ from u to v, then the label $\delta_v(u)$ is appended to the path associated to the agent $\mathbf{a}$. This enables $\mathbf{a}$ to return back to the previously visited vertex (i.e., u) whenever it wants to. When it does so, the label $\delta_v(u)$ is deleted from *ePath*. Thus, at any time during the computation, *ePath* contains the sequence of the labels of the edges that $\mathbf{a}$ has to traverse (in reverse order) to return to its homebase from the current vertex.

The Tree Computation by an Agent. Each agent $\mathbf{a}$ computes its tree $T_{\mathbf{a}}$ by executing the following. Starting from its homebase, $\mathbf{a}$ performs a partial traversal of the graph. During this traversal, it marks all the unmarked vertices that are visited for the first time. At each vertex w marked by $\mathbf{a}$, $\mathbf{a}$ arbitrarily chooses an unexplored link incident to w and traverses it. This technique has been employed in [10].

Whenever $\mathbf{a}$ traverses an edge e to reach an unmarked vertex v, it marks v as *visited* and marks e as a T edge. On the other hand, when it reaches a vertex u that is already visited (the vertex u is already marked), the edge leading to u is marked as an NT edge and the agent immediately backtracks to the previous vertex (marked by it) and tries the other unexplored edges incident to that vertex. When there is no more unexplored edges, the agent backtracks to the previous visited vertex (by taking the last link in the *ePath* sequence) and then tries to explore any unexplored link at that vertex. Finally, when the agent has returned to its homebase and there is no more unexplored link at the current vertex, it stops the tree computation.

Remark 6. A mark on an edge can be done with marks on the corresponding ports on the endvertices of the edge.

The tree computation procedure executed by each agent is given in Algorithm1.

```
   Mark the homebase;
   Set ePath to empty;
 3: while there is an unexplored edge e = (u, v) at the current vertex u, do
      traverse e to reach vertex v;
      if v is already marked or v contains an agent a_1, then
 6:     return back to u and mark the link e with the label NT;
      else
        mark the link e as T and mark v as explored;
 9:   end if
    end while
    if there are no more unexplored links at the current vertex, then
12:   if ePath is not empty, then
        remove the last link from ePath, traverse that link and go to line 3;
      else
15:     Stop the tree computation;
      end if
    end if
```

Algorithm 1. The tree construction by a mobile agent

Fact 1. *Every vertex in the graph is marked by exactly one agent. If two vertices u_1 and u_2 are marked by the same agent then there exists exactly one simple path of T edges joining them. If two vertices u_1 and u_2 are marked by two different agents, then each path joining them contains at least one NT edge. There is no cycle consisting of only T edges.*

Encoding Message Passing Actions. Among the computation steps inherent to the message passing system, the operations *send a message via the port j* and *receive a message from the port j* are surely the most important. To encode these operations, we require that at each place there is a variable called *in-buf*, where received messages are stored. In this framework, the first part of a message always contains the port from which it was received.

Send message m via port j. Let $e = \{u, v\}$ and let **a** be an agent located at u with $\delta_u(v) = j$. The execution of the operation *Send message m via port j* by the agent **a** consists in traversing the edge e, writing the composed message $\langle \delta_v(u), m \rangle$ in the *in-buf* of v and backtracking the edge e.

Receive message m from port j. Let $e = \{u, v\}$ and let **a** be an agent located at u with $\delta_u(v) = j$. The execution of the operation *Receive message m from port j* by the agent **a** consists in looking for the first message arrived from port j in the *in-buf* of u (if the initial algorithm needs the FIFO property of channels, i.e., it requires that messages are received in the same order as they have been sent; if it is not the case then we take any message in *in-buf*). Once this message is found, it is deleted from the *in-buf* and stored in a temporary variable for purpose of computation.

Internal events. It suffices to apply the transformation corresponding to the transformation of the state of the process to the state of the place.

Remark 7. The execution of the *send* and *receive* operations allows an agent **a** to write informations in the *in-buf* of vertices that do not necessary belong to $T_{\mathbf{a}}$. Moreover, the simulation executed by Procedure 1 can be viewed as a distributed computation in a network whose processes are decomposed in clusters, and where processes, of the same cluster, execute their computation steps in turn.

Remark 8. Suppose that at each round of Step 2 of Procedure 1, we ask that each time a computation step can be performed on a vertex belonging to the tree of an agent, it simulates it within a finite number of rounds. Then any global state of the message passing system obtained by the message passing algorithm can be obtained by the mobile agents simulation.

Transforming a Terminating Message Passing Algorithm into a Mobile Agent Algorithm That Terminates. We are now interested in showing that if $\mathcal{D}$ is a terminating algorithm, then the mobile agent algorithm $\mathcal{A}$ has also the termination property. For this reason, we have adapted the behavior of each agent to this new goal. In fact, we add on each place two variables *fatherLink* and *fState*. The *fatherLink* of a vertex u, contains the port number through which an agent **a**, located at u, can reach the *father* of u in the tree that contains u. Let u be the homebase of the agent **a**, the *fState* of u indicates to **a** that it has to perform one more computation round on the tree $T_{\mathbf{a}}$. The *fState* is either the token *Finished* or the token *NotFinished*. Initially the *fatherLink* of each homebase contains 0 and the *fState* of each homebase is set to *NotFinished*. The *fatherLink* of the other vertices is 0 and the *fState* of the other vertices is *Finished*. All these changes lead to the following adapted version of Procedure1.

Procedure 2

Step 1: *On wake-up, each agent* **a** *constructs a tree* $T_{\mathbf{a}}$ *using a partial traversal of the graph. This leads to a spanning forest of* k *trees where each tree is constructed by exactly one agent. During this construction, the fatherLinks of all the vertices, other than the homebase, that were marked by* **a** *are actualized.*

Step 2: *The agent* **a** *executes the events of* $\mathcal{D}$ *on the vertices of* $T_{\mathbf{a}}$. *This execution is performed in rounds. The agent* **a** *is allowed to execute round* r *if and only if at the beginning of the round* r *the fState of its homebase has the value NotFinished and it has simulated all the possible steps in the former round otherwise it falls asleep. In each round,* **a** *sets the fState of its homebase to Finished, afterward* $T_{\mathbf{a}}$ *is traversed and, if possible,* d $(d \geq 1)$ *computation steps of* $\mathcal{D}$ *are executed at each vertex of* $T_{\mathbf{a}}$. *If an agent mimics the fact of sending a message to a vertex* u, *the agent takes advantage of the fatherLink to find the homebase* w *of the agent that has constructed the tree containing* u. *Once it arrives at* w, *it sets the fState of* w *to NotFinished (if there is a sleeping agent, it wakes it up) and goes back to* u.

One has to take notice of the fact that if the algorithm $\mathcal{D}$ terminates, then there exists a time t_1 such that no more computation step is performed after the time

t_1 in $\mathcal{D}$. Let $S_{\mathcal{D}}^{t_1}$ be the state of the network at time t_1. Due to Lemma 9, there also exists a time t_2, during the execution of $\mathcal{A}$, such that $S_{\mathcal{D}}^{t_1} = S_{\mathcal{A}}^{t_2}$. It is then quite simple to see that there exists a time $t_3 \geq t_2$ such that at time t_3 the *fState* of any homebase is empty. Thus Procedure 2 and $\mathcal{A}$ stop. Furthermore, the algorithm $\mathcal{D}$ terminates explicitly if and only the mobile agent algorithm $\mathcal{A}$ constructed as defined by the Procedure 2 terminates explicitly. Finally:

Lemma 9. *The mobile agent algorithm $\mathcal{A}$, constructed as defined by Procedure 2, is equivalent to the algorithm $\mathcal{D}$.*

From these two kinds of equivalences, we can give our main theorem, stating the equivalency of the two models considered in this paper.

Theorem 10. *There exists a mobile agent algorithm $\mathcal{A}$ that solves a problem $\mathcal{P}$ on a mobile agent system (G, δ, λ) with an initial placement π_0 if and only if there exists a message passing algorithm $\mathcal{D}$ that solves the problem $\mathcal{P}$ on (G, δ, λ') (λ' is defined in Section 4).*

6 Applications

In this section, we will use our main theorem to give a characterisation of the mobile agent systems where we can solve two equivalent problems that are *election* and *rendez-vous*. The same method can be applied for other classical problems such as the topology recognition, the naming, the spanning tree construction, etc.

Election and Rendez-vous. The election problem is one of the paradigms of the theory of distributed computing. It was first posed by LeLann [14]. A message algorithm solves the election problem if it always terminates and in the final configuration exactly one process is marked as *elected* and all the other processes are *non-elected*. Moreover, it is supposed that once a process becomes *elected* or *non-elected* then it remains in such a state until the end of the algorithm. Yamashita and Kameda [17] characterise the graphs for which there exists an election algorithm in the message passing model (see also [7,9]). In the mobile agents setting, the aim of a mobile agent election algorithm is to elect one agent. The elected agent enters a final state *leader*, whereas all other agents enter a final state *follower*. Another important problem in this setting is the rendez-vous problem. The aim of a rendez-vous algorithm is to arrive in a configuration where all the mobile agents gather in a same vertex of the graph. Another interpretation of a rendez-vous algorithm $\mathcal{A}$ is that the aim of $\mathcal{A}$ is to elect a vertex of the network. These two problems are equivalent, since once an agent has been elected, all the agents can gather in the homebase of the elected agent. Reversely, once all the agents gather in some node, the first agent on this node becomes the leader, whereas all the others become followers. Agent election and rendez-vous have been studied in [5,3,10]. Consequently, from our main theorem, there exists an algorithm that solves the rendez-vous and the mobile agent election in a mobile agent system $(\mathbb{A}, \mathbb{P}, \mathbb{S}, \pi_0, \lambda)$ with $\mathbb{S} = (V, E, \delta)$

if and only if there exists an election algorithm in the message passing system (V, E, δ, λ') (λ' is defined in Section 4).

From Message Passing Computations to Mobile Agent Computations. The characterization of Yamashita and Kameda given in [17] is based on the notion of view. Given a labelled graph $\mathbf{G} = (V, E, \delta, \lambda)$ and an integer d, the d-view $\mathbf{T}^d_{\mathbf{G}}(v_0)$ of a node $v_0 \in V(G)$ is a tree of height d that can be defined recursively as follows.

- $\mathbf{T}^0_{\mathbf{G}}(v_0)$ is a single-vertex graph whose node is denoted x_0 and $\lambda'(x_0) = \lambda(v_0)$,
- To define $\mathbf{T}^{d+1}_{\mathbf{G}}(v_0)$, we take a copy of $\mathbf{T}^d_{\mathbf{G}}(v_i)$ for each neighbor v_i of v_0 in G. The root of the new tree is a vertex x_0 labelled by $\lambda(v_0)$ and there is an edge between x_0 and the root x_i of each tree $\mathbf{T}^d_{\mathbf{G}}(v_i)$, such that $\delta_{x_0}(x_i) = \delta_{v_0}(v_i)$ and $\delta_{x_i}(x_0) = \delta_{v_i}(v_0)$.

The view $\mathbf{T}_{\mathbf{G}}(v_0)$ of a node v_0 is an infinite rooted labelled tree that can be defined recursively in the same way. The root of the tree is a vertex x_0 that corresponds to v_0 and is labelled by $\lambda(v_0)$. For each neighbor v_i of v_0 in G, there is an edge between x_0 and the root x_i of the tree $\mathbf{T}_{\mathbf{G}}(v_i)$, such that $\delta_{x_0}(x_i) = \delta_{v_0}(v_i)$ and $\delta_{x_i}(x_0) = \delta_{v_i}(v_0)$. The view of a vertex v in a graph $\mathbf{G}$ can also be obtained by considering all labelled walks in $\mathbf{G}$ starting from v. Clearly, the d-view of a vertex v is its view truncated at distance d. In [17], the following theorem is given:

Theorem 11 ([17]). *There exists an election algorithm over a graph (G, δ, λ) if and only if $\forall v, v' \in V(G)$ $(v \neq v')$, the labelled trees $\mathbf{T}_{\mathbf{G}}(v)$ and $\mathbf{T}_{\mathbf{G}}(v')$ are not isomorphic.*

Norris shows in [15] that $\mathbf{T}_{\mathbf{G}}(v)$ and $\mathbf{T}_{\mathbf{G}}(v')$ are isomorphic if and only if $\mathbf{T}^n_{\mathbf{G}}(v)$ and $\mathbf{T}^n_{\mathbf{G}}(v')$ are isomorphic, where $n = |V(G)|$. Each vertex can compute its $2n$-view, and then it will know all the n-views of the other vertices. Once each vertex knows the views of all the other nodes, the vertex with the weaker view is elected. From our main result, we can therefore give the following corollary for the mobile agent election problem. Let λ' the labelling function defined at the beginning of Section 4.

Corollary 12. *There exists an agent election algorithm or a rendez-vous algorithm in a mobile agent system $(\mathbb{A}, \mathbb{P}, \mathbb{S}, \pi_0, \lambda)$ with $\mathbb{S} = (V, E, \delta)$ if and only if for all vertices $v, v' \in V(G)$, the labelled trees $\mathbf{T}^n_{\mathbf{G}'}(v)$ and $\mathbf{T}^n_{\mathbf{G}'}(v')$ are not isomorphic, where $\mathbf{G}' = (G, \delta, \lambda')$ and $n = |V(G)|$.*

In the same way, using the results of Flocchini et al. [13], we can obtain similar characterizations for these problems in the mobile agent system where there is a sense of direction, and the results of [5] become corollaries of these characterizations.

References

1. H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. McGraw-Hill, 1998.
2. B. Awerbuch, M. Betke, R. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot (extended abstract). In *Proc. of the eighth annual conference on Computational Learning Theory, COLT'95*, pages 321–328. ACM Press, 1995.
3. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Can we elect if we cannot compare? In *Proc. of the fifteenth annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'03*, pages 324–332. ACM Press, 2003.
4. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Election and rendezvous in fully anonymous systems with sense of direction. In *Proc. of the 10th International Colloquium on Structural Information Complexity, SIROCCO'03*, volume 17, pages 17–32. Carleton Scientific, 2003.
5. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Rendezvous and election of mobile agents: impact of sense of direction. *Theory of Computing Systems*, to appear.
6. M. Bender and D. Slonim. The power of team exploration: Two robots can learn unlabeled directed graphs. In *Proc. of the 35th annual Symposium on Foundations of Computer Science, FOCS'94*, pages 75–85, 1994.
7. P. Boldi, B. Codenotti, P. Gemmell, S. Shammah, J. Simon, and S. Vigna. Symmetry breaking in anonymous networks: Characterizations. In *Proc. 4th Israeli Symposium on Theory of Computing and Systems*, pages 16–26. IEEE Press, 1996.
8. P. Braun and W. Rossak. *Mobile agents: basic concepts, mobility models and the tracy toolkit*. Morgan Kaufman, 2005.
9. J. Chalopin and Y. Métivier. A bridge between the asynchronous message passing model and local computations in graphs (*extended abstract*). In *Proc. of Mathematical Foundations of computer science, MFCS'05*, volume 3618 of *LNCS*, pages 212–223, 2005.
10. S. Das, P. Flocchini, A.Nayak, and N. Santoro. Distributed exploration of an unknown graph. In *Proc. of the 12th international colloquium on Structural Information and Communication Complexity, SIROCCO'05*, volume 3499 of *LNCS*, pages 99–114, 2005.
11. X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment. i: the rectilinear case. *J. ACM*, 45(2):215–245, 1998.
12. A. Dessmark, P. Fraigniaud, and A. Pelc. Deterministic rendezvous in graphs. In *Proc. of the 11th annual European Symposium on Algorithms, ESA'03*, volume 2832 of *LNCS*, pages 184–195, 2003.
13. P. Flocchini, A. Roncato, and N. Santoro. Computing on anonymous networks with sense of direction. *Theoretical Computer Science*, 301:355–379, 2003.
14. G. LeLann. Distributed systems: Towards a formal approach. In B. Gilchrist, editor, *Information processing'77*, pages 155–160. North-Holland, 1977.
15. N. Norris. Universal covers of graphs: isomorphism to depth $n-1$ implies isomorphism to all depths. *Discrete Applied Math.*, 56:61–74, 1995.
16. G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000.
17. M. Yamashita and T. Kameda. Computing on anonymous networks: Part i - characterizing the solvable cases. *IEEE Transactions on parallel and distributed systems*, 7(1):69–89, 1996.

Incremental Construction of k-Dominating Sets in Wireless Sensor Networks

Mathieu Couture, Michel Barbeau, Prosenjit Bose, and Evangelos Kranakis

School of Computer Science, Carleton University, Herzberg Building
1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6 Canada
{mathieu, jit}@cg.scs.carleton.ca,
{barbeau, kranakis}@scs.carleton.ca
http://www.scs.carleton.ca
Fax: 1-613-520-4334

Abstract. Given a graph G, a k-dominating set of G is a subset S of its vertices with the property that every vertex of G is either in S or has at least k neighbors in S. We present a new incremental distributed algorithm to construct a k-dominating set. The algorithm constructs a monotone family of dominating sets $D_1 \subseteq D_2 \ldots \subseteq D_i \ldots \subseteq D_k$ such that each D_i is an i-dominating set. For unit disk graphs, the size of each of the resulting i-dominating sets is at most six times the optimal.

Keywords: unit disk graph, dominating set, maximal independent set, approximation algorithms, distributed algorithms, fault-tolerance.

1 Introduction

An *ad hoc network* is a special type of wireless network where no node has a priori knowledge about the other nodes. Constructing and maintaining a structure allowing nodes to communicate with each other is one of the main challenges of ad hoc networks. Sensor networks are a specific type of ad hoc networks dedicated to a specific task: sensing (light, temperature, humidity, etc.). The dominating set structure helps ad hoc and sensor networks to perform routing. In sensor networks, dominating sets also help the sensing task itself. Since nodes located close to each other sense similar values, only a dominating set of the nodes is needed to monitor an area. This helps prolonging the network's lifetime by turning off the nodes that are not in the dominating set, thereby extending the battery life of these nodes.

Sensor networks typically contain more nodes and each node has less memory than in general ad hoc networks. Therefore, it is important to design algorithms with low memory complexity. An algorithm is *distributed* if the information needed by a node to perform its computation only concerns its direct neighbors. The amount of memory needed by each node to execute a distributed algorithm only depends on the number of its neighbors and not on the network size.

Sensor nodes are more error-prone than nodes in general ad hoc networks. They have limited energy resources and need to be periodically redeployed by

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 202–214, 2006.

adding new nodes to the network. The fact that they are error-prone calls for *fault-tolerance* in the design of algorithms for such networks.

We address the problem of distributively constructing k-dominating sets on unit disk graphs. Unit disk graphs are the standard structure used to model ad hoc and sensor networks. A k-dominating set is a dominating set where each node is either in the dominating set or has at least k neighbors in the dominating set.

We generalize dominating set algorithms based on the idea of maximal independent sets [1, 9, 13] to obtain k-dominating sets. A subset S of the nodes of a graph G is said to be *independent* if it does not contain two adjacent nodes. It is *maximal* if it does not have a proper independent superset. It is straightforward to show that a maximal independent set is also a dominating set. Our algorithm is distributed and, on unit-disk graphs, has a deterministic performance ratio of six. The *performance ratio* of a k-dominating set algorithm is defined as the ratio of the size of the k-dominating set it produces over the size of an optimal (minimum) k-dominating set. It is not position-aware, which means that nodes do not need to know their coordinate in the plane. It also constructs the k-dominating set incrementally. More specifically, it constructs a monotone family of dominating sets $D_1 \subseteq D_2 \ldots \subseteq D_i \ldots \subseteq D_k$ such that each D_i is an i-dominating set. Incremental construction of k-dominating sets is helpful when redeploying sensor networks. When senor nodes in the k-dominating set run out of batteries or experiment failure for diverse reasons, the k-dominating set has to be reconstructed. With an incremental algorithm, reconstruction of a k-dominating set can be done by keeping the current dominators.

The k-dominating set problem has been addressed by Dai and Wu [3] and Kuhn *et al.* [8]. However, our algorithm is the only one which has a constant deterministic performance ratio. It is also the only one to provide an explicit incremental construction. The rest of this paper is organized as follows: in Section 2, we present our algorithm. In Section 3, we analyze its performance ratio. In Section 4, we present some simulation results. We draw conclusions in Section 5.

2 Algorithm

Alzoubi *et al.* [1] and Wan *et al.* [13] addressed construction of a *connected* dominating set. Their algorithm consists of two phases. The first phase constructs a maximal independent set. A maximal independent set is also a dominating set. In this section, we generalize that first phase of the algorithm presented in [1, 13] to obtain a k-dominating set. Our generalization augments a $(k-1)$-dominating set in order to obtain a k-dominating set. More specifically, we want to construct a monotone k-dominating family.

Definition 1. *A* k-dominating family *is a sequence* $D_1, D_2, \ldots, D_k$ *of subsets of vertices of the unit disk graph such that for all* $i = 1, 2, \ldots, k$, D_i *is an* i*-dominating set. A* monotone k-dominating family *is a* k*-dominating family with the additional property that the sequence of dominating sets is monotonically increasing under inclusion, i.e.* $D_1 \subseteq D_2 \subseteq \cdots \subseteq D_k$.

The key idea of our algorithm is that we first construct a 1-dominating set by constructing a maximal independent set. Then, we construct a maximal independent set of the nodes that are not 2-dominated, which gives a 2-dominating set. We repeat the procedure until we have a k-dominating set. The construction of each dominating set is similar to the approach in [1, 13].

We now present an overview of our algorithm. Every node has a unique identifier. In initialization phase, each node sends its identifier to its immediate neighbors. After initialization, two types of messages are used: JOIN(id, i) and GIVE-UP(id, i), where id is the identifier of the sending node and $i = 1 \ldots k$ identifies a round. These messages are only sent to immediate neighbors. The JOIN(id, i) message means that the sender joins the j-dominating sets for $j = i \ldots k$. Such a node is said to be *marked* in round i. The GIVE-UP(id, i) message means that

Algorithm 1. DOMINATING SET(id, N, k)

Input: id, the node identifier
N, the list of the neighbors identifiers
k, the required number of dominators for a non-dominating node
Output: *dominator*, a boolean indicating whether the node is a dominator
Local Variables: *round*, the current round
candidate, a lookup table indicating whether or not a node n is a candidate to be a dominator in round r (all initial values are **true**)

```
 1: dominator ← false
 2: round ← 1
 3: if id < min(N) then
 4:    dominator ← true
 5:    send JOIN(id, 1)
 6:    exit
 7: end if
 8: while round ≤ k do
 9:    receive message
10:    if message is JOIN(n, r) then
11:       send GIVE-UP(id, round)
12:       round ← round + 1
13:       for i = r to k do
14:          candidate[n, i] ← false
15:       end for
16:    end if
17:    if message is GIVE-UP(n, r) then
18:       candidate[n, r] ← false
19:    end if
20:    if id < min{n ∈ N|candidate[n, round]} then
21:       dominator ← true
22:       round ← k + 1
23:       send JOIN(id, round)
24:       exit
25:    end if
26: end while
```

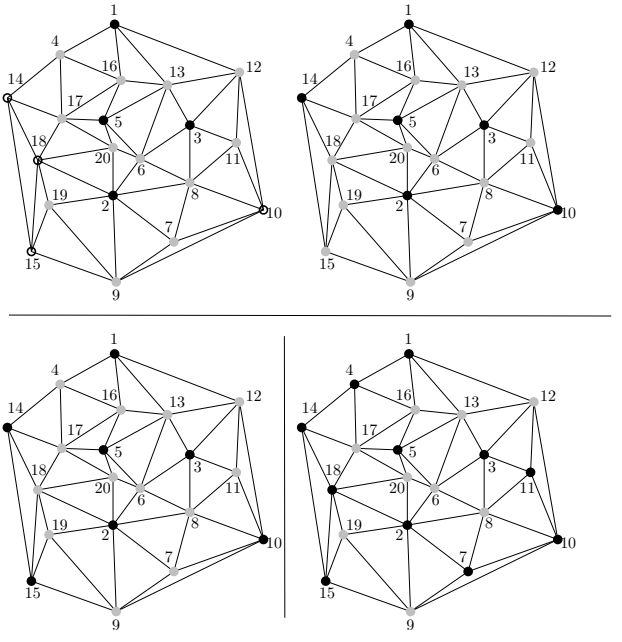


Fig. 1. Marking Process Example for $k = 1$ (above) and $k = 2$ and 3 (below)

the sender is excluded of the i-dominating set. After transmitting a JOIN message, the sender remains silent. A node is said to be a *candidate* for round i if it is not part of the $(i-1)$-dominating set and it has never sent the GIVE-UP(id, i) message. Following the completion of the initialization phase, every node that has an identifier lower than the ones of all its immediate neighbors sends the JOIN$(id, 1)$ message. The rest of the algorithm is message driven. Algorithm 1 specifies how each node should behave. Note that different nodes may execute simultaneously different rounds.

Figure 1, illustrates the marking process for $k = 1$, 2 and 3. Nodes in black are dominators. Nodes in grey are k-dominated. Nodes in white are not k-dominated. For $k = 1$, nodes 1, 2, 3 and 5 have the smallest identifier among their (0-dominated) neighbors and thus declare themselves dominators. Initially, node 10 can not declare itself a dominator because of nodes 7, 8 and 9. However, after node 2 has declared itself a dominator, nodes 7, 8 and 9 become 1-dominated. Node 10 is then allowed to declare itself a dominator. The same reasoning applies to nodes 1 and 14. The 1-dominating set is then $\{1, 2, 3, 5, 10, 14\}$. For $k = 2$, there is only one new dominator, node 15. For $k = 3$, the new dominators are nodes 4, 7, 11 and 18.

3 Theoretical Properties

In this section, we give an overview of the theoretical properties of our algorithm. We first show that our algorithm computes a valid k-dominating set and

a monotone k-dominating family. Then, we analyze the worst case performance ratio of our algorithm. In the latter part, we follow the general idea of Kuhn *et al.* [8]. More precisely, we first show that no unit disk can contain more than a given number of dominators (i.e. $5k$). Then, we use properties of k-dominating sets to show that this leads to a constant performance ratio.

Proposition 1. *Let S_i be the set of nodes that are marked in rounds $j = 0 \ldots i$ of Algorithm 1. Then S_i is an i-dominating set.*

Proof: We proceed by induction on i. To make the things simpler, we say that i ranges from 0 to k. The round zero chooses the empty set as a 0-dominating set. The base case is trivial, since all nodes of the graph have at least zero neighbors in the empty set. For the induction case, we have to show that if S_i is a valid i-dominating set, then S_{i+1} is also valid $(i+1)$-dominating set. In order to do this, we proceed by contradiction. Let n_1 be a node that is not $(i+1)$-dominated by S_{i+1}. This means that it has not sent a JOIN$(id, i+1)$ message. Consequently, it must have a neighbor n_2 with a lower identifier that is still a candidate for round $i+1$ (line 20), meaning that it is not $(i+1)$-dominated either (line 11 in n_2, and 17 in n_1). Since n_2 is not $(i+1)$-dominated, by the same reasoning, there must have a node n_3 that is not $(i+1)$-dominated and has a lower identifier than the one of n_2. This process allows to construct a path $n_1, n_2, \ldots, n_j, \ldots, n_k$ such that none of the n_j is $(i+1)$-dominated, $id(n_1) > id(n_2) > \ldots > id(n_j) > \ldots > id(n_k)$, and n_k does not have any neighbor with a lower identifier that is not $(i+1)$-dominated. Then, n_k should have elected himself as a dominator, contradicting the fact that it is not $(i+1)$-dominated. Therefore, every node is $(i+1)$-dominated, which completes the inductive case. □

Proposition 2. *For $i = 1 \ldots k$, let S_i be defined as above. Then for all $i = 0 \ldots k-1$, $S_i \subseteq S_{i+1}$.*

Proof: The monotonicity property claimed in the statement of the proposition is true by construction. That is, let $n \in S_i$. Then, it has been marked in some round $j \leq i < i+1$ and by definition of S_{i+1}, we have $n \in S_{i+1}$. □

Proposition 3. *In any given round, the nodes marked by Algorithm 1 form an independent set.*

Proof: Suppose that in the same round, two adjacent nodes n_1 and n_2 declare themselves dominators. Without loss of generality, suppose n_1 has a lower identifier than n_2. This means that as long as n_1 did not send a give-up message, n_2 can not elect itself a dominator. But since n_1 never sends such a message (no node sends both a give-up and a join message), n_2 can never declare itself a dominator. This means that no two adjacent nodes can declare themselves dominators. □

Proposition 4. *Let $G = (V, E)$ be a unit disk graph, C be a unit disk and $S \subseteq V$ be the set of nodes marked by Algorithm 1. Then $|S \cap C| \leq 5k$.*

Proof: By proposition 3, S is the union of k independent sets. Since no unit disk can contain more than 5 independent nodes [9], $S \cap C$ can not contain more than $5k$ nodes. □

Proposition 5. *Let $G = (V, E)$ be a graph, S a subset of V, t an integer and $OPT_k = \{v_1, \ldots, v_{|OPT_k|}\}$ an optimal k-dominating set of G. If $|S| > t|OPT_k|$, then there is at least one node $v \in OPT_k$ such that $|N(v) \cap S| > k(t-1)$, where $N(v)$ is the set formed by v and its neighbors.*

Proof: Let S' be $S \setminus OPT_k$. Since $|S'| \geq |S| - |OPT_k| > t|OPT_k| - |OPT_k|$, we have $|S'| > (t-1)|OPT_k|$. For each $v_i \in OPT_k$, define S_i as $N(v_i) \cap S'$. Since each node in S' is adjacent to at least k nodes in OPT_k, we have that

$$\sum_{i=1}^{opt_k} |S_i| \geq k|S'| > k(t-1)|OPT_k|$$

Therefore, by the pigeonhole principle, one of the S_i contains more than $k(t-1)$ nodes. The result follows from the fact that $S_i \subseteq N(v_i) \cap S$. □

Theorem 1. *Let $G = (V, E)$ be a unit disk graph, $S \subseteq V$ the set of nodes marked by Algorithm 1 and OPT_k an optimal k-dominating set. Then $|S| \leq 6|OPT_k|$. In other words, the performance ratio is not greater than six.*

Proof: Suppose $|S| > 6|OPT_k|$. By proposition 5, there is at least one node $v \in V$ such that $|N(v) \cap S| > 5k$. But this contradicts proposition 4, and therefore $|S| \leq 6|OPT_k|$. □

We now show that for any k, there exists graphs for which our algorithm has a performance ratio of five. It is an open question whether or not it is possible to close the gap between five and six. First, we need the following lemma:

Lemma 1. *Let $\triangle ABC$ be an isosceles triangle such that $\angle BAC = \angle ACB = \phi$, p be a point located on the line AB such that A is between p and B, and q be a point located on the line CB such that C is between q and B. Then $|pq| > |AC|$.*

Proof: If $|pB| = |qB|$, then $\triangle pBq$ is similar to $\triangle ABC$, and $|pB| > |AB|$ implies $|pq| > |AC|$. Suppose now that $|pB| < |qB|$, and let q' be the point located on the line CB such that C is between q' and B and $|q'B| = |pB|$. By the first case, $|pq'| > |AC|$. Now, since $\triangle ABC$ is isosceles, $\phi < \frac{\pi}{2}$, and since $\angle pq'q = \pi - \phi$, we have that $\angle pq'q > \frac{\pi}{2}$. Therefore, $\angle pq'q$ is the largest angle of $\triangle pq'q$, meaning that its opposite side, pq, is the largest side. In particular, we have $|pq| > |pq'| > |AC|$. The case where $|pB| > |qB|$ is identical, which completes the proof of the lemma. □

Proposition 6. *The worst case performance ratio of Algorithm 1 is at least five.*

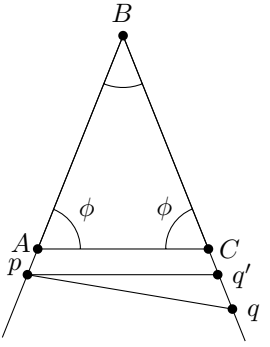


Fig. 2. Lemma 1

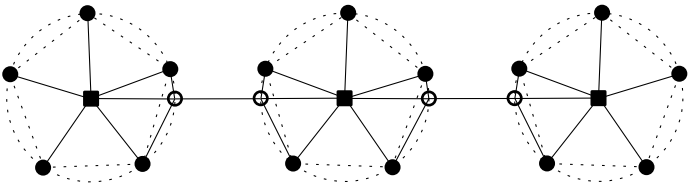


Fig. 3. Lower bound of five for Algorithm 1

Proof: For $k = 1$ and $n = 6$, place five nodes equally spaced on the boundary of a circle of radius 1, and place one other node in the center of that circle. Since the circle has radius 1, the center node shares an edge with all the other nodes. Also, since the distance between every pair of nodes on the circle is at least $2 \sin \frac{\pi}{5} > 1$, there is no other edge in the unit disk graph. In the remainder of the proof, this basic structure will be referred to as a *star*, the node placed in the center of the circle will be referred to as *the center* of the star and the five nodes on the boundary of the circle will be referred to as the *branches* of the star. The center of a star forms a dominating set of the whole star. However, if the center happens to be given a higher identifier than one of the branches, all branches would be marked as dominators, leading to a performance ratio of five.

Figure 3 depicts how to connect several stars to build cases with n as large as desired. More precisely, we show how to construct examples of size $14 + 8m$, for any given m (in Figure 3, $m = 1$). The construction goes as follows: place $m + 2$ stars on a horizontal line such that their centers are placed at x-coordinates $0, 3, 6, \ldots, 3(m+1)$ and no branch lies on the horizontal line. Since the circles in which the stars are inscribed are at distance at least 1 from each other, the only edges of the graph so far are the ones linking the branches of the stars to their centers. All that remains is to connect the graph. In order to do so, add nodes on the intersection of the inscribing circles with the horizontal line. These nodes will be referred to as *bridging nodes*. Since the centers of two consecutive stars are at distance 3 from each other, the two bridging nodes between them are at distance 1 from each other. Therefore, there is an edge between two bridging

nodes which are between the centers of two consecutive stars, making the whole network connected. To see how the performance ratio of five can be reached, notice that the star centers form a dominating set of size $m + 2$. However, since the set of all branches form an independent set, it could be that those nodes would be marked as dominators, leading to a dominating set of size $5(m + 2)$, which gives a performance ratio of at least five.

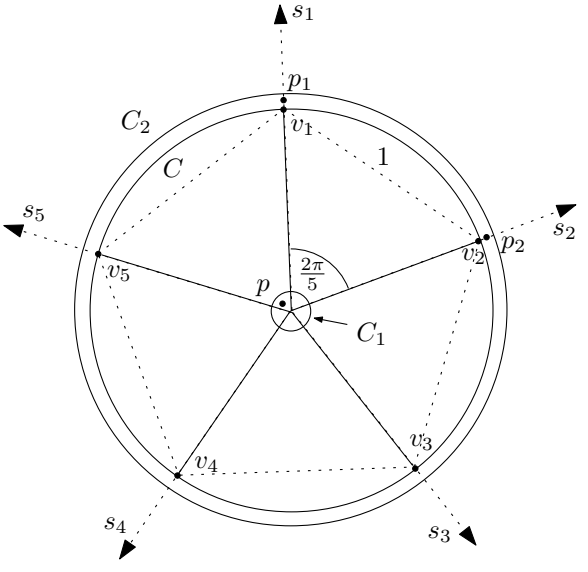


Fig. 4. Widget for $k > 1$

For $k > 1$, Figure 4 shows how to generalize the star structure. The goal is to map to each node of the star a set of k nodes such that:

1. nodes mapped to the center share an edge with every other node and
2. nodes mapped to the branches only share edges with nodes mapped to the center and nodes mapped to the same branch.

In order to achieve this, draw a regular pentagon having side length of 1. Let C be the inscribing circle of that pentagon and $r = \frac{1}{2\sin(\frac{\pi}{5})}$ be the radius of C. Now, let C_1 and C_2 respectively be the circles having the same center as C and radii $r_1 = \frac{1-r}{2}$ and $r_2 = r + r_1$. For each vertex v_i of the pentagon (i from 1 to 5), let s_i be the half-line from the center of C through v_i. Now, let p be a point located inside C_1, and p_1 and p_2 be two points located on some s_i and s_j ($i \neq j$), between C_2 and C. Then, Lemma 1 tells us that

$$|p_1, p_2| > |v_1, v_2| \geq 1$$

and from the triangle inequality, we have

$$|p, p_1| \leq r_1 + r_2 = 2(\frac{1-r}{2}) + r = 1.$$

Similarly, $|p, p_2| \leq 1$. The construction we need is then the following: place k points inside C_1 and k points on each of the s_i between C and C_2. We call the result of that construction a *generalized star*. The points located inside C_1 form a k-dominating set, but the algorithm may mark all nodes located on the s_i. Since there are $5k$ such points, the performance ratio is five in that case. To construct a lower bound example with $k > 1$ for large n, we link the generalized stars in a similar fashion as for the case $k = 1$. □

4 Simulation Results

We have generalized an existing independent set-based algorithm [1, 9, 13] in order to incrementally construct a k-dominating set. We have chosen to generalize this specific algorithm because it is distributed and has constant performance ratio. By simulation, we compare our algorithm with k-generalized versions of other available algorithms. Stojmenovic *et al.* [12] suggested the following heuristic to improve the independent set algorithm of [1, 9, 13]: instead of ordering the nodes according to their identifier, order them according to their degree first and then their identifier. Higher priority is granted to nodes having higher degree. The performance ratio is still at most five, but it has not been proven it is actually better than that. For the k-dominating set problem, it is not desirable to favor higher degree nodes. The reason is that nodes having degree less than k cannot have k dominating neighbors, so they must necessarily be in the k-dominating set.

Selecting nodes of higher degree is the same idea that is behind the greedy set-cover algorithm [4]. The greedy set-cover algorithm first favors nodes that dominate the largest number of nodes not yet dominated. Although this is a global selection criterion, it still has to be examined. At first sight, since it does not have constant performance ratio (its performance ratio is $H(\Delta)$, where Δ is the maximum degree of a node in the network and H is the harmonic function), one would believe that it would not perform as well as our algorithm. However, it turns out that in order to have $H(\Delta) > 5$, we need Δ to be at least 83, and to reach six, we need Δ to be at least 226. Since it is not likely to have nodes having that many neighbors in real situations, this algorithm still deserves attention.

In this section, we discuss simulation results comparing Algorithm 1 with k-generalized versions of both the algorithm presented in [12] and the greedy algorithm. We also compare it with the greedy construction of a maximal independent set. The k-generalized versions of those algorithms work the same way we generalized the maximal independent set algorithm: for $k = 1$, we run the standard algorithm on all nodes. For $k \geq 2$, we run the standard algorithm on nodes that are not yet k-dominated. We ran our simulations 200 times for networks of 200 nodes. We have chosen a communication range such that with high probability, the network is connected. According to Penrose [10, 11], for any integer $k \geq 0$ and real constant c, if the nodes have identical radius r given by the formula:

$$r = \sqrt{\frac{\ln n + k \ln \ln n + \ln(k!) + c}{n\pi}}$$

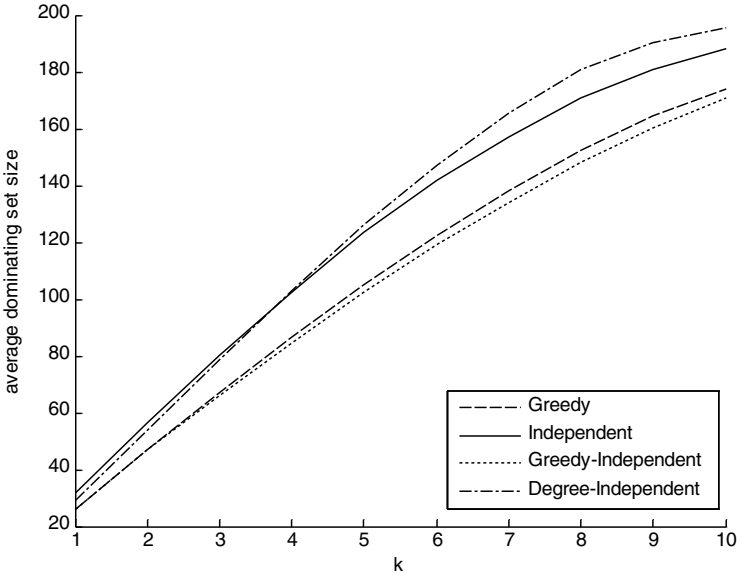


Fig. 5. Average dominating set size for 200 nodes

then the network is $(k+1)$-connected with probability $e^{-e^{-c}}$ as n goes to infinity. For $n = 200$, choosing $k = 1$ and $c = 5$, we then obtain that for a radius of $r \approx 0.138$, the network is 2-connected with probability 0.99.

Figure 5 shows the simulation results we have obtained. The algorithm which performed the best is the one in which we greedily constructed an independent set. Not far behind is the greedy algorithm. It is worth noting that even if those algorithms perform slightly better, neither of the two are distributed. This is because the greedy choice of the next node to be marked is based on global criteria. For the two distributed algorithms, it is interesting to note that the one using the ordered pair degree-id only performs better for small values of k (5 and less). After that, it is the one simply based on identifiers which performs better. With a 95% certainty, the expected values of the size of the dominating sets was at most ± 0.67 node.

Unfortunately, Figure 5 does not show the optimal solution. Since the dominating set problem is NP-complete, only exponential time algorithms are known to solve the problem. This is why only small instances of the problem can be addressed by simulation. Figure 6 compares the same algorithms with the optimal solution for a network of 35 nodes. We ran over 200 simulation cases. In that case, with a 95% certainty, the actual expected values of the dominating sets size was at most ± 0.28 nodes. Figure 7 shows the average performance ratio we obtained for each algorithm. For small values of k, the two global greedy algorithms are the best, followed by the distributed algorithm granting priority to high degree nodes. The algorithm simply based on identifiers performs the worst. However, as k grows, the results change completely. The algorithm simply

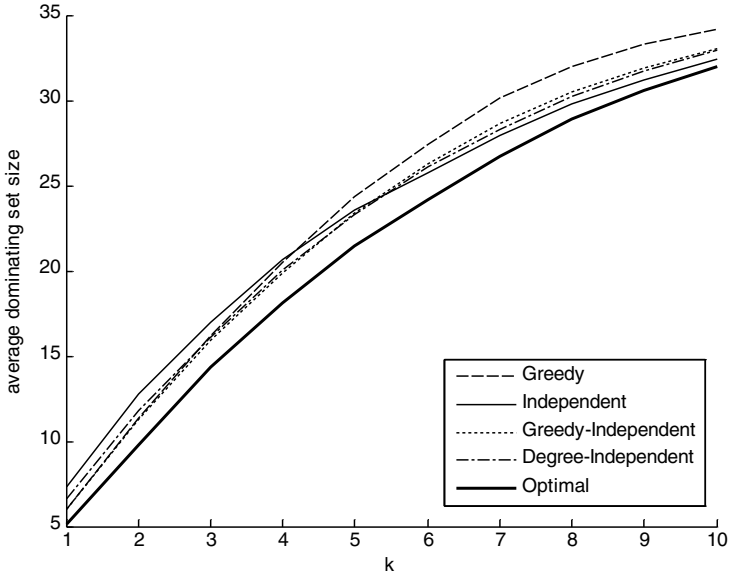


Fig. 6. Average dominating set size for 35 nodes

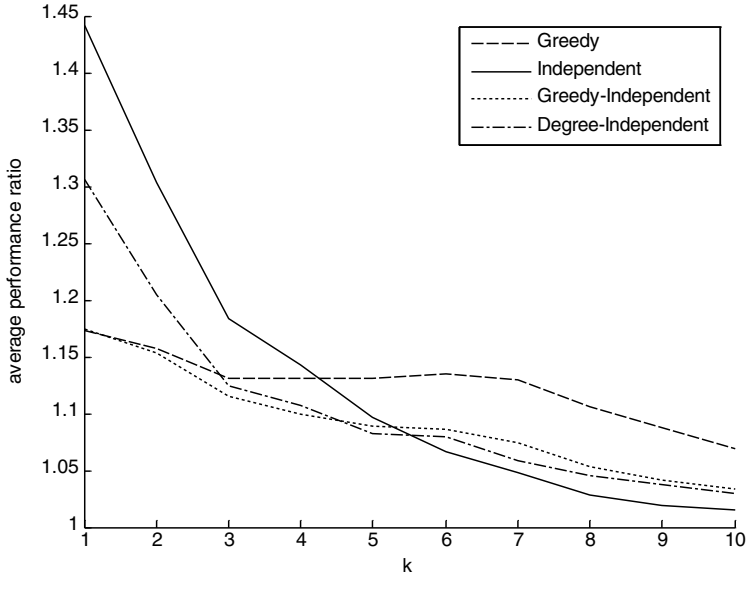


Fig. 7. Average performance ratio for 35 nodes

based on identifiers becomes the best, and the basic greedy algorithm becomes the worst. The algorithm constructing independent sets by granting priority to high degree nodes performs slightly better than the greedy construction of an independent set.

Exact simulation values can be found in the technical report version of this paper [2].

5 Conclusion

In this paper, we have introduced a new algorithm to address the k-dominating set problem. Our algorithm has a deterministic performance ratio of six. The previously best algorithm had an expected performance ratio of $O(k)$ for an unspecified constant [8]. We have shown that the size of the k-dominating set our algorithm produces may be five times bigger than the optimal one. However, it is an open issue whether or not the gap between five and six can be closed. The expected performance ratio is also unknown.

Simulation results have shown that in some cases, the k-generalized version of the greedy dominating set algorithm performs better than ours. Besides their worst-case performance ratio, an other important difference between the greedy dominating set algorithm and ours is that one is global while the other is distributed. We believe that differences between the performance of global, distributed and local algorithms is an interesting research avenue. Important work in that field has been done by Kuhn *et al.* [6, 7] and Kuhn [5].

Acknowledgment

The authors graciously acknowledge the financial support received from the following organizations: Natural Sciences and Engineering Research Council of Canada (NSERC) and Mathematics of Information Technology and Complex Systems (MITACS).

References

[1] K. M. Alzoubi, P.-J. Wan, and O. Frieder, Message-optimal connected dominating sets in mobile ad hoc networks. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pp. 157–164, ACM Press, New York, NY, USA, 2002.

[2] M. Couture, M. Barbeau, P. Bose, and E. Kranakis, Incremental construction of k-dominating sets in wireless sensor networks. Tech. Rep. TR-06-11, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, 2006.

[3] F. Dai and J. Wu, On constructing k-connected k-dominating set in wireless networks. In *IPDPS*, IEEE Computer Society, 2005.

[4] D. S. Johnson, Approximation algorithms for combinatorial problems. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pp. 38–49, ACM Press, New York, NY, USA, 1973.

[5] F. Kuhn, The Price of Locality: Exploring the Complexity of Distributed Coordination Primitives. In *PhD Thesis, ETH Zurich, Diss. ETH No. 16213*, 2005.

[6] F. Kuhn, T. Moscibroda, and R. Wattenhofer, What cannot be computed locally! In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pp. 300–309, ACM Press, New York, NY, USA, 2004.

[7] F. Kuhn, T. Moscibroda, and R. Wattenhofer, On the locality of bounded growth. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pp. 60–68, ACM Press, New York, NY, USA, 2005.

[8] F. Kuhn, T. Moscibroda, and R. Wattenhofer, Fault-Tolerant Clustering in Ad Hoc and Sensor Networks. In *26th International Conference on Distributed Computing Systems (ICDCS), Lisboa, Portugal*, 2006.

[9] M. Marathe, H. Breu, S. Ravi, and D. Rosenkrantz, Simple heuristics for unit disk graphs. *Networks*, **25**:59–68, 1995.

[10] M. D. Penrose, The longest edge of the random minimal spanning tree. *The Annals of Applied Probability*, **7(2)**:340–361, 1997.

[11] M. D. Penrose, On k-connectivity for a geometric random graph. *Random Struct. Algorithms*, **15(2)**:145–164, 1999.

[12] I. Stojmenovic, M. Seddigh, and J. Zunic, Dominating sets and neighbor elimination-based broadcasting algorithms in wireless networks. *IEEE Trans. Parallel Distrib. Syst.*, **13(1)**:14–25, 2002.

[13] P.-J. Wan, K. M. Alzoubi, and O. Frieder, Distributed construction of connected dominating set in wireless ad hoc networks. *Mob. Netw. Appl.*, **9(2)**:141–149, 2004.

Of Malicious Motes and Suspicious Sensors: On the Efficiency of Malicious Interference in Wireless Networks

Seth Gilbert[1], Rachid Guerraoui[2], and Calvin Newport[1]

[1] MIT CSAIL
{sethg, cnewport}@mit.edu
[2] EPFL IC
rachid.guerraoui@epfl.ch

Abstract. How efficiently can a malicious device disrupt communication in a wireless network? Imagine a basic game involving two honest players, Alice and Bob, who want to exchange information, and an adversary, Collin, who can disrupt communication using a limited budget of β broadcasts. How long can Collin delay Alice and Bob from communicating? In fact, the trials and tribulations of Alice and Bob capture the fundamental difficulty shared by several n-player problems, including reliable broadcast, leader election, static k-selection, and t-resilient consensus. We provide round complexity lower bounds—and (nearly) tight upper bounds—for each of those problems. These results imply bounds on adversarial efficiency, which we analyze in terms of *jamming gain* and *disruption-free complexity*.

1 Introduction

Ad hoc networks of wireless devices hold significant promise for the future of ubiquitous computing. Unfortunately, such networks are particularly vulnerable to adversarial interference due to their use of a shared, public communication medium and their deployment in unprotected environments. For example, a committed adversary can disrupt an ad hoc network by jamming the communication channel with noise. Continuous jamming, however, might be unwise for the adversary: it depletes the adversary's energy, allows the honest devices to detect its presence, and simplifies its localization—and subsequent destruction. The adversary, therefore, would rather be more efficient, disrupting the protocol using a minimal number of transmissions.

Jamming Gain. The efficiency of the adversary can be quantified, roughly speaking, by comparing the *duration* of the disruption to the adversary's *cost* for causing the disruption. In the systems literature, this metric has been informally referred to as *jamming gain* (e.g., [1]). In the context of round-based protocols (time-slotted wireless radio channels), the jamming gain can be defined as follows. Let $D_P(t)$ be the minimal number of broadcasts needed by the adversary to delay protocol P from terminating for t rounds, for some initial value. Then

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 215–229, 2006.

the *jamming gain* of protocol P is: $\mathsf{JG}(P) = \lim_{T\to\infty} \frac{T}{\max(D_P(T),1)}$. For example, if the adversary must broadcast in every round, the jamming gain is 1. By contrast, if the adversary need *never* broadcast to prevent termination, then the jamming gain is infinite.

Disruption-Free Complexity. A second metric, *disruption-free complexity*, measures how long the adversary can disrupt a protocol without performing *even one* broadcast. The uncertainty introduced by the possibility of adversarial broadcasts is sufficient to slow down many protocols. This is defined as: $\mathsf{DF}(P) = \max\{t : D_P(t) = 0\}$. If a protocol has large disruption-free complexity, then the adversary can significantly reduce the throughput of multiple consecutive executions, while avoiding the disadvantages of actually jamming.

This paper is the first theoretical examination of the efficiency of malicious disruption in a wireless ad hoc network. We begin by analyzing a 3-player game that captures many of the fundamental difficulties of wireless coordination. We then extend these results to several classical problems: reliable broadcast, leader election, static k-selection and consensus. For each problem, we present fundamental limits on the robustness to malicious interference, and we present algorithms that match these standards of robustness.

The 3-Player Game. The 3-player game consists of two honest players—Alice and Bob, and a third malicious player, Collin (the *Collider*). All three players share a time-slotted single-hop wireless radio channel. Alice and Bob each begin with a value to communicate. Colin is determined to prevent them from communicating, in either direction, for as long as possible. Collin can broadcast in any time slot (i.e., round), potentially destroying honest messages or overwhelming them with malicious data. In order to precisely measure the efficiency of a malicious adversary, we endow Collin with a budget of β messages, and analyze how long Alice and Bob can be disrupted. The size of β is not known *a priori* to Alice and Bob. (If it were, then Alice and Bob could communicate reliably by repeating each message $2\beta + 1$ times.)

3-Player Lower Bound. We show that Collin can delay Alice and Bob's communication for $2\beta + \lg|V|/2$ rounds, where V is the set of possible values that Alice and Bob may communicate. An immediate corollary is that no protocol for Alice and Bob can achieve a jamming gain better than 2. This result is surprising as it implies that every protocol has some semantic vulnerability that the adversary can exploit to gain extra efficiency. A second corollary is that the disruption-free complexity is $\Theta(\lg|V|)$. Therefore for large V, the passive presence of Collin can significantly reduce Alice and Bob's communication throughput. We prove these lower bounds (in Section 4) by exhibiting a strategy for Collin to delay Alice and Bob, exploiting the fact that they can never trust any message, since Collin could have *overwhelmed* it with a fake message.

3-Player Upper Bound. For our upper bound (Section 5), we consider a (harder) setting where Alice needs to transmit a value to Bob, who does not broadcast any

messages. We exhibit a protocol that allows Alice—using $\beta + \Delta$ broadcasts—to transmit her value to Bob in $2\beta + \max\{2\Delta 2^{\frac{\lg|V|}{\Delta}}, 4\lg|V|\}$ rounds. (Notice that if $\Delta < 1$, we show that Alice's task is impossible.) For $\Delta = \Omega(\lg|V|)$, the protocol matches our lower bound. For $\Delta < \lg|V|$, however, Collin can delay the communication more efficiently. For example, if $\Delta = 1$, the disruption-free complexity of our protocol increases to $|V|$. We show that a disruption-free complexity of $\max\{2\Delta 2^{\frac{\lg|V|}{\Delta}}, 4\lg|V|\}$ is unavoidable, highlighting an inherent tradeoff between Alice's message complexity and her throughput. Finally, we consider a variant of the 3-player game in which Alice and Bob do not start in the same round; Bob is activated asynchronously by the adversary. We present a protocol that solves this problem and still terminates within $2\beta + \Theta(\lg|V|)$ rounds (assuming Alice has an unrestricted message budget).

The n-Player Implications. The trials and tribulations of Alice and Bob capture something fundamental about how efficiently malicious devices can disrupt wireless coordination. In Section 6, we derive new lower bounds—via reduction to our 3-player game—for several classical n-player problems: reliable broadcast: $2\beta+\Theta(\lg|V|)$; leader election: $2\beta+\Omega(\log\frac{n}{k})$; static k-selection: $2\beta+\Omega(k\lg\frac{|V|}{k})$. For the latter two cases, k represents the number of participants contending to become leader and to transmit their initial value, respectively. As before, we draw immediate corollaries regarding the jamming gain and disruption-free complexity, resulting in a jamming gain of 2, and disruption-free complexity of $\Theta(\lg|V|)$, $\Omega(\log\frac{n}{k})$, and $\Omega(k\lg\frac{|V|}{k})$, respectively.

We next consider a more general framework that also includes crash failures: the malicious adversary can both broadcast β messages *and* also crash up to t honest devices. We study binary consensus as an archetypal problem in this framework, and derive a lower bound of $2\beta + \Theta(t)$ rounds. The $\Theta(t)$ factor is established by a technique that maintains the indistinguishability of two univalent configurations for t rounds. The 2β factor then follows from a (partial) reduction to consensus. This shows a jamming gain of 2, as before. By contrast, the disruption-free complexity, $\Theta(t)$, is significantly larger than for the crash-free models. (Notice that if the adversary is benign, then crash failures have no effect on the complexity.)

Finally, in Section 7, we present tight upper bounds for reliable broadcast and consensus and nearly tight bounds for leader election and static k-selection.

Assumptions and interpretations. Underlying our results on jamming gain and disruption-free complexity is an analysis of how long the adversary can disrupt communication given a limited broadcast budget. This interpretation is interesting in its own right: a limited broadcast budget models the (limited) energy available to a set of malicious devices.

Clearly, authentication—for example, using cryptographic keys—impacts our lower bounds. With authentication, the 3-player communication game completes in $\beta + 1$ rounds, resulting in a jamming gain and disruption-free complexity of 1. Intuitively, a jamming gain arises from semantic vulnerabilities in the protocol;

cryptographic techniques can eliminate this vulnerability. In general, however, deploying cryptographic solutions in wireless networks can be difficult. Public key authentication schemes are often expensive both in computation and, to some extent, communication. Symmetric key schemes (such as MACs) have been deployed in wireless networks (see, e.g., [2, 3]), yet the focus has generally been link-level security, rather than authenticated broadcast, and there remain issues with key distribution. For example, if only a single key is used, the system is easily compromised by a single corrupted node; if multiple keys are used, then keys must be exchanged and communication is complicated. One interpretation of our bound is that authentication should be deployed only if its cost is less than the cost of waiting an additional $\beta + \Theta(\lg |V|)$ rounds.

2 Related Work

This paper explores the damage that can be caused by a genuinely malicious (Byzantine) device that can reliably disrupt communication in a wireless ad hoc network. Koo [4], Bhandari and Vaidya [5], as well as Pelc and Peleg [6], study "t-locally bounded" Byzantine failures in wireless networks, in which the number of Byzantine nodes in a region is bounded. In these papers, the Byzantine devices are required to follow a strict TDMA schedule, thus preventing them from interfering with honest communication. Others have considered models with probabilistic message corruption [7, 8]. Wireless networks with crash failures (but not Byzantine failures) have also been studied extensively in both single hop (e.g., [9, 10]) and multihop (e.g., [11, 12]) contexts. By contrast, we consider a malicious adversary that can choose to send a message in any round, potentially destroying honest messages or overwhelming them with malicious data.

Simultaneous to this work, Koo, Bhandari, Katz, and Vaidya [13] have also considered a model where the adversary has a limited broadcast budget and can send a message in any round, overwhelming honest messages. A key difference, however, is that they assume that the adversary's budget is fixed *a priori* and known to all participants. By contrast, we do not assume that β is known in advance. (Thus it is no longer sufficient to repeat each message $2\beta + 1$ times.) Moreover, they focus primarily on feasibility, that is, determining the threshold density of dishonest players for which multihop broadcast is possible. By contrast, our paper focuses on the time complexity of the protocols and the efficiency of the adversary. Furthermore, we also consider the impact of combining crash failures with a malicious adversary.

Adversarial jamming of physical layer radio communication is a well studied problem in the electrical engineering community (see, e.g., [14]). In the context of wireless ad hoc networks, there has been recent interest in studying the jamming problem at the MAC layer and above. See, for example, [1, 15, 16, 17], which analyze specific MAC and network layer protocols, highlighting semantic vulnerabilities that can be leveraged to gain increased jamming efficiency.

3 Preliminaries

We assume a synchronous round-based Multiple Access Channel (MAC) model with collision detection (as in, e.g. [18, 19, 20]). We consider n honest devices, *the players*, named from the set $[1, n]$, and one additional malicious device incarnating *the adversary*. In each round, each device can decide to broadcast a message or listen. If there are no broadcasts in a round, then none of the players receive a message. If exactly one message is broadcast, then all players receive the message. If two or more messages are broadcast, then each player can either: (1) receive exactly one of the broadcast messages; or (2) detect noise on the channel, i.e., a collision. (This channel behavior represents the unpredictability of real networks, for example, shadowing effects [21].) Without loss of generality, we assume that the adversary determines for each honest player whether option 1 or 2 occurs; in case of option 1 the adversary's message is systematically received.

Throughout this paper, we endow the adversary with a budget of β broadcast messages, where β is *a priori* unknown to the players. Also, we assume no message authentication capabilities. That is, a player cannot necessarily distinguish a message sent from the adversary from a message sent by a fellow honest player.

The basic game we consider involves two honest players, Alice and Bob, and an adversary, Collin. Alice is initialized with value $v_a \in V$ and Bob with $v_b \in V$, where $|V| > 1$ and V is known to all. The players can $\mathsf{output}(v)$ for any $v \in V$ such that the following are satisfied. **Safety:** Bob only outputs v_a and Alice only outputs v_b; and **Liveness:** Eventually, either Alice or Bob outputs a value.

4 Lower Bound for the 3-Player Game

We prove in this section a lower bound on the round complexity of the 3-player communication game. Our lower bound holds even if Alice and Bob have an unlimited budget of messages.

To prove our lower bound, we describe a strategy for Collin to frugally use his β messages to prevent communication. Two assumptions are key to this strategy: (1) Collin's budget of messages β is unknown to Alice and Bob; (2) Alice and Bob cannot distinguish a message sent by Collin from an honest message. A *silent* round, on the other hand, cannot be faked: if Bob (for example) receives no message and no collision notification, then he can be certain that Alice did not broadcast a message. Therefore, in order to prevent Alice and Bob from communicating, it is sufficient, roughly speaking, for Collin to disturb silence.

Theorem 1. *Any* 3*-player communication protocol requires at least* $2\beta + \lg|V|/2$ *rounds to terminate.*

Assume, for contradiction, a protocol, A, that defies this worst-case performance. Consider any value $v \in V$ and denote by $\gamma(v)$ the $\lg|V|/2 - 1$ round (good) execution of A where Alice and Bob begin with initial value v, and Collin performs no broadcasts. If Alice and Bob both broadcast in the same round, assume both messages are lost. We begin with the following lemma:

	Alice		Bob		Collin	
	$\alpha(v)$	$\alpha(w)$	$\alpha(v)$	$\alpha(w)$	$\alpha(v)$	$\alpha(w)$
1	m	-	-	-	-	m
2	-	m	-	-	m	-
3	-	-	m	-	-	m
4	-	-	-	m	m	-
5	m	-	m'	-	-	m'
6	-	m	-	m'	m'	-
7	m	-	-	m'	-	m
8	-	m	m'	-	m	-

(a) α Rules

	Alice		Bob		Collin
	$\alpha(v)$	$\rho(w,v)$	$\alpha(w)$	$\rho(w,v)$	$\rho(w,v)$
1	m	m'	-	-	m
2	-	-	m'	m	m'
3	m	-	-	-	m
4	-	-	m	-	m
5	m	-	m'	-	m
6	-	m	m'	-	m'
7	m	m'	m''	-	m
8	m	-	m''	m'	m''

(b) $\rho(w,v)$ Rules

Fig. 1. Collin's behavioral rules for α and $\rho(w,v)$ executions

Lemma 1. *There exist two values $v, w \in V$ $(v \neq w)$, such that Alice (resp. Bob) broadcasts in round r of $\gamma(v)$ if and only if Alice (resp. Bob) broadcasts in round r of $\gamma(w)$.*

Proof. In each round, there are four possibilities: (1) Alice broadcasts alone, (2) Bob broadcasts alone, (3) Alice and Bob both broadcast, and (4) neither Alice nor Bob broadcasts. Accordingly, for a sequence of c rounds, there are 4^c possible patterns of broadcast behavior. Thus, there are at most $4^{\lg |V|/2-1} = \frac{|V|}{4}$ possible broadcast patterns that result from the $|V|$ possible γ executions. It follows by the pigeonhole principle that at least two such executions have the same pattern.

For the rest of this proof, we fix v and w to be the two values identified by Lemma 1. We define $\alpha(v)$ (resp. $\alpha(w)$) to be the execution of A in which Alice and Bob both begin with initial value v (resp. w) and Collin applies the α-rules described in Figure 1(a). In this table, "-" indicates silence and m and m' both represent a message broadcast. Each row matches a specific set of broadcast behaviors of Alice and Bob in two executions, with the corresponding broadcast behavior followed by Collin in these executions. Since Alice and Bob's algorithm is deterministic, Collin can predict their behavior in each round.

For example, Rule 1 from Figure 1(a) specifies that for any given round, if Alice broadcasts in exactly one α execution, and Bob is silent in both, then Collin replicates Alice's broadcast in the execution where she is silent. For any pattern of broadcast behavior not described in the table, assume that Collin performs no broadcasts. Also, assume that in any round where both Collin and Alice (resp. Bob) broadcast, only Collin's message is received by Bob (resp. Alice). We claim:

Lemma 2. *Neither Alice nor Bob can output during $\alpha(v)$ or $\alpha(w)$.*

Proof. We show that Bob cannot output in $\alpha(v)$ and Alice cannot output in $\alpha(w)$. The argument for Bob in $\alpha(w)$ and Alice in $\alpha(v)$ is symmetric.

We first define a third execution $\rho(w,v)$, of A, in which Alice starts with initial value w and Bob starts with initial value v. The behavior of Collin in execution ρ is defined by the rules in Figure 1(b). Notice that, in all three executions, we

assume that Collin has an unlimited broadcast budget. This is without loss of generality because Alice and Bob do not know the value of β, and we will later concern ourselves only with the prefixes of the α executions in which Bob has not yet broadcast more then β times.

We show, by induction on the round number, r, that $\rho(w, v)$ is indistinguishable from $\alpha(v)$ with respect to Bob, and that $\rho(w, v)$ is indistinguishable from $\alpha(w)$ with respect to Alice. The lemma follows immediately from this indistinguishability and the safety requirement of the communication game.

Since Bob begins with value v in both $\rho(w, v)$ and $\alpha(v)$, and Alice begins with value w in both $\rho(w, v)$ and $\alpha(w)$, the base case ($r = 0$) is immediate. We now consider the possible behaviors of Alice and Bob during round $r + 1$.

- **Case 1:** *Alice broadcasts in* $\alpha(w)$. By induction, this implies Alice also broadcasts in $\rho(w, v)$, therefore these two executions remain indistinguishable with respect to Alice (as broadcasters cannot listen). We turn our attention to Bob, bypassing the sub-case of Bob broadcasting in $\alpha(v)$ as this is Case 2. Two sub-cases: (1) Alice is silent in $\alpha(v)$. If Bob is also silent in $\alpha(w)$, then, by α-Rule 2, Collin broadcasts Alice's $\alpha(w)$ (and $\rho(w, v)$) message in $\alpha(v)$. If Bob broadcasts in $\alpha(w)$, then, by α-rule 6, Collin broadcasts Bob's message in $\alpha(v)$ and, by ρ-rule 6, Collin also broadcasts Bob's message in $\rho(w, v)$. (2) Alice broadcasts in $\alpha(v)$. By ρ-Rule 1 or 7 (depending on whether Bob broadcasts in $\alpha(w)$) Collin replicates Alice's $\alpha(v)$ message in the ρ execution. In all cases, Bob receives the same message in $\alpha(v)$ and $\rho(w, v)$.
- **Case 2:** *Bob broadcasts in* $\alpha(v)$. This argument is symmetric to Case 1.
- **Case 3:** *Alice does not broadcast in* $\alpha(w)$ *and Bob does not broadcast in* $\alpha(v)$. There are four sub-cases. (1) Alice and Bob don't broadcast in $\alpha(v)$ and $\alpha(w)$, respectively. Collin does nothing and the executions are clearly indistinguishable. (2) Alice broadcasts in $\alpha(v)$ and Bob is silent in $\alpha(w)$. By α-rule 1, Collin broadcasts Alice's message in $\alpha(w)$. By ρ-rule 3, Collin broadcasts Alice's message in $\rho(w, v)$. Therefore, Bob receives Alice's message in $\alpha(v)$ and $\rho(w, v)$, and Alice receives her message (from Collin) in $\alpha(w)$ and $\rho(w, v)$. (3) Alice is silent in $\alpha(v)$ and Bob broadcasts in $\alpha(w)$. By α-rule 4, Collin broadcasts Bob's message in $\alpha(v)$. By ρ-rule 4, Collin broadcasts Bob's message in $\rho(w, v)$. Therefore, Alice receives Bob's message in $\alpha(w)$ and $\rho(w, v)$, and Bob receive his message (from Collin) in $\alpha(v)$ and $\rho(w, v)$. (4) Alice broadcasts in $\alpha(v)$ and Bob broadcasts in $\alpha(w)$. By α-rule 7, Collin broadcasts Alice's message in $\alpha(w)$. By ρ-rule 5, Collin broadcasts Alice's message in $\rho(w, v)$. Therefore, Alice receives her message (from Collin) in $\alpha(w)$ and $\rho(w, v)$, and Bob receives Alice's message in $\alpha(v)$ and $\rho(w, v)$.

We now show that one of these two indistinguishable α executions requires only β broadcasts by Collin during the first $2\beta + \lg |V|/2 - 1$ rounds.

Proof (Theorem 1). By Lemma 2, Alice and Bob do not produce an output in either $\alpha(v)$ or $\alpha(w)$ as long as Collin continues to follow the α rules. It suffices to show that, in at least one of the two executions $\alpha(v)$ and $\alpha(w)$, Collin broadcasts in no more than β of the first $2\beta + \lg |V|/2 - 1$ rounds.

Algorithm 1: Bit Broadcast Sub-Protocol

```
1  bcast-Alice(b)
2    active ← true
3    while (active) do
4      if (b=1) then
5        bcast(vote)     ▷ Data round broadcast
6      m ← recv()        ▷ Data round receive
7      if (b=0) and (m ≠ ⊥) then
8        bcast(veto)     ▷ Veto round broadcast
9      m ← recv()        ▷ Veto round receive
10     if (m = ⊥) then
11       active ← false
12   return
```

```
1  recv-Bob()
2    active ← true
3    while (active) do
4      votes ← recv()    ▷ Data round receive
5      vetos ← recv()    ▷ Veto round receive
6      if (vetos = ⊥) then
7        active ← false
8    if (votes = ⊥) then
9      return 0
10   else
11     return 1
```

Algorithm 2: Sequence Broadcast Protocol

```
1  SEQ-Alice(s ∈ {0,1}^k, k)
2    count ← 1
3    while (count ≤ k) do
4      bcast-Alice(s[count])
5      count ← count + 1
```

```
1  SEQ-Bob(k)
2    count ← 1
3    while (count ≤ k) do
4      s[count] ← recv-Bob()
5      count ← count + 1
6    output(s)
```

We first consider rounds 1 through $\lg|V|/2 - 1$ of $\alpha(v)$ and $\alpha(w)$. We know by Lemma 1 that Alice and Bob broadcast on the same schedule for these initial rounds when they start both with v or both with w. Notice, however, that Collin broadcasts (according to the α rules) only in situations of asymmetric silence, i.e. when Alice and Bob are *not* on the same schedule. It follows that Collin does not broadcast in either $\alpha(v)$ or $\alpha(w)$ for the first $\lg|V|/2 - 1$ rounds.

Now we turn our attention to the 2β rounds that follow. For a given round, Collin only broadcasts in $\alpha(v)$ or $\alpha(w)$, but not both, since he only fills in asymmetric silent rounds. Therefore, by a simple counting argument, it is impossible for Collin to broadcast in more than half of the rounds in both executions. We therefore choose the execution in which Collin broadcasts in no more than half of the following 2β rounds. This delays both Alice and Bob from outputting for $2\beta + \lg|V|/2 - 1$ rounds. □

We conclude with an immediate corollary of Theorem 1:

Corollary 1. *Any 3-player communication protocol has a jamming gain of at least 2, and a disruption-free complexity of* $\Omega(\lg|V|)$. □

5 Upper Bounds for the 3-Player Game

We prove in this section that our (round complexity) lower bound is tight, by showing that there is a protocol that matches it. To strengthen our upper bound result, we consider the seemingly harder problem of Alice transmitting her value to Bob in a setting where Bob does not broadcast. Specifically, we give a protocol that, assuming Alice has a budget of $\beta + \Delta$ messages, transmits Alice's input value to Bob in $2\beta + \max\{2\Delta 2^{\frac{\lg|V|}{\Delta}}, 4\lg|V|\}$ rounds. For $\Delta = \Omega(\lg|V|)$, this protocol matches our lower bound. For $\Delta = o(\lg|V|)$ the round complexity grows. We show this to be unavoidable.

Our protocol broadcasts a sequence of bits (Algorithm 2), using a sub-protocol for each bit. Alice to Bob (Algorithm 1). The basic idea of Algorithm 1 is to alternate data rounds and veto rounds. In a data round, Alice transmits a message

if $b = 1$ and remains silent otherwise. If Collin interferes with the data round (i.e. by broadcasting in the case where $b = 0$), Alice indicates this interference by broadcasting in the veto round. At this point, Alice and Bob try again with a new pair of rounds. Of course, Collin can also interfere by broadcasting in a veto round. This too causes Alice and Bob to try again with a new pair of rounds. The sub-protocol continues until the first silent veto round.

Alice and Bob both know that Alice has a broadcast budget of $\beta+\Delta$. Typically, Alice would broadcast a binary encoding of her value, which might require $\lg|V|$ broadcasts. If $\Delta < \lg|V|$, we encode the value as bit strings of length k containing at most Δ 1's. We choose k to be the minimum value such that $\binom{k}{\Delta} \geq |V|$, that is, the smallest value that allows us to express all V values. Alice then transmits this encoding as described above. This is summarized in the following theorem, whose proof is in the full version:

Theorem 2. *There exists a protocol through which Alice transmits her initial value to Bob, within* $2\beta+\max\{2\Delta 2^{\frac{\lg|V|}{\Delta}}, 4\lg|V|\}$ *rounds, using a budget of* $\beta+\Delta$ *messages. This protocol thus has a jamming gain of 2, and a disruption-free complexity of* $\max\{2\Delta 2^{\frac{\lg|V|}{\Delta}}, 4\lg|V|\}$

Notice that Theorem 2 assumes Alice has a message budget that is strictly larger than Collin's budget (as indicated by the constraint $\Delta > 0$). This is in fact necessary, and it is impossible to communicate a value from Alice to Bob if Alice's budget is $\leq \beta$ since, in this case, Collin can successfully simulate Alice's behavior (see the full version). Notice that the round complexity grows significantly as Δ decreases below $\lg|V|$. We show this trade-off to be inherent:

Theorem 3. *Let* $k = \max\{\frac{\Delta 2^{\lg(|V|)/\Delta}}{e} - \Delta, \frac{lg|V|}{2}\}$. *If Alice has a budget of size* $\beta + \Delta$ *(*$\Delta > 0$*), then there exists no protocol through which Alice can transmit her initial value to Bob in less than* $2\beta + k$ *rounds. Thus every such protocol has a disruption-free running time of* $\Omega(k)$.

The Wake-Up Case. We have assumed that Alice and Bob begin in the same round. Consider the case where Bob is activated at an unknown point in the execution. Thus, Bob no longer has round numbers synchronized with Alice. This models the situation where Alice represents a base station that needs to transmit a value to intermittently awake tiny devices (i.e., Bob). There is (in the full version) a variant of our protocol that solves the problem in $2\beta+\Theta(\lg|V|)$ rounds after Bob awakes, asymptotically matching our lower bound from Section 4, despite the extra synchronization challenges. This variant requires Alice to never terminate, which is inevitable given that she can never distinguish between Bob and Collin pretending to be Bob (while Bob is still sleeping).

6 Lower Bounds for n-Player Problems

We generalize here our results to n-player coordination problems. We then consider the impact of combining malicious behavior with crash failures.

6.1 n-Player Reductions

We show here how Alice and Bob can together simulate an arbitrary n-player protocol. We then use this simulation to derive lower bounds, via reduction from the 3-player communication game, for several n-player problems. None of our round-complexity lower bounds restricts the message budget of honest players.

A simulation by Alice and Bob is defined by a 5-tuple: $\{A, n, S_A, S_B, I\}$, where: (1) A is the n-player protocol being simulated; (2) S_A and S_B partition the n players into two non-empty and non-overlapping sets; (3) I is a mapping of players to their respective initial values.

Alice simulates the players in S_A, initializing them according to I. (Alice is provided only the initial values for nodes in S_A, i.e., $I|S_A$.) In each round, if any of the players in S_A choose to broadcast, Alice arbitrarily chooses one of their messages to broadcast. She then delivers to each simulated player any messages or collision notifications from that round. Bob simulates the players in S_B in an equivalent manner. The following can be proved by straightforward induction:

Theorem 4. *Consider simulation $\{A, n, S_A, S_B, I\}$. For all r-round executions of the simulation, there exists an r-round execution α of A, initialized according to I, where the outputs of Alice and Bob are equivalent to the outputs in α, and Collin broadcasts the same number of messages in the simulation and α.*

Reliable Broadcast. In reliable broadcast, one player—the *source*—is provided with an input value $v_0 \in V$. Each player must receive this initial value. *Safety* requires that each player output only v_0, i.e., perform $\mathsf{output}(v)$ only if $v = v_0$. *Liveness* requires that all players eventually perform an output.

Theorem 5. *Any reliable broadcast protocol requires at least $2\beta + \lg|V|/2$ rounds to terminate.*

Proof. Assume by contradiction that A is a reliable broadcast protocol that terminates in $R < 2\beta + \lg|V|/2$ rounds for all initial values. We reduce 3-player communication, for value domain V, to A. Alice and Bob simulate A for n players, where: (1) S_A contains the source, S_B contains all other players, and (2) I maps the source to v_a, Alice's initial value. Bob outputs the first value output by a simulated player. By Theorem 4, Bob always outputs $v_0 = v_a$ by round R, contradicting Theorem 1.

Leader Election. In leader election, $k \leq n$ *participants* contend to become the leader. All n players should learn the leader, i.e., perform $\mathsf{output}(\ell)$, for some ℓ. *Safety* requires that the leader be a participant, and that there be only one leader. *Liveness* requires every player to perform an output.

Theorem 6. *Any leader election protocol requires at least $2\beta + \lg\lfloor\frac{n-1}{k}\rfloor/2$ rounds to terminate.*

Proof. Assume by contradiction that A is a leader election protocol that terminates in $R < 2\beta + \lg\lfloor\frac{n-1}{k}\rfloor/2$ rounds for all choices of k participants. We reduce

to leader election, the 3-player game defined over the value space V, where V contains every integer between 1 and $\lfloor \frac{n-1}{k} \rfloor$, to A.

Alice and Bob simulate A for n players where: (1) S_A contains players 1 through $n-1$, S_B contains player n, and (2) I activates player $i \in S_A$ if and only if $(i \mod \lfloor \frac{n-1}{k} \rfloor) + 1 = v_a$ and fewer than k nodes have been activated so far in I_A. Let i be the leader output by Bob's simulated player. Bob outputs $v_a = (i \mod \lfloor \frac{n-1}{k} \rfloor) + 1$, as required. By Theorem 4 Bob always outputs v_a within R rounds, contradicting Theorem 1, since $2\beta + \lg V/2 = 2\beta + \lg \lfloor \frac{n-1}{k} \rfloor /2$.

Static k-Selection. In static k-Selection, k *participants* are provided with values $v_i \in V$. Each player must receive all values. *Safety* requires that the first k outputs of a player equal the k values. *Liveness* requires that all players eventually perform at least k output actions. The protocol *terminates* when all players have performed at least k output actions. (The selection problem is well-studied in radio networks, e.g., [22, 23].)[1]

Theorem 7. *Any static k-selection protocol requires at least* $2\beta + \Omega(k \lg \frac{|V|}{k})$ *rounds to terminate.*

Proof. Assume by contradiction that A is a protocol that terminates in $R < 2\beta + o(k \lg |V|/k)$ rounds, for all initial values and choices of participants. We reduce to k-selection, the 3-player game for the value space V', where V' contains one entry for every multiset of k values drawn from V, to A.

Alice and Bob simulate A for n players where: (1) S_A contains players 1 through k, S_B contains the remaining players, and (2) I activates players 1 through k, and provides each a different value from the multiset described by $v_a \in V'$. Given k simulated outputs, Bob can reconstruct and output the unique multiset described by these values. By Theorem 4 Bob will always output v_a in R rounds, contradicting Theorem 1, since $2\beta + \lg |V'|/2 = 2\beta + \lg \frac{|V|^k}{k!}/2 = 2\beta + \Theta(k \lg \frac{|V|}{k})$ rounds.

Corollary 2. *Any protocol for reliable broadcast, leader election or static k-selection has a jamming gain of at least 2 and a disruption-free running time of* $\Omega(\log |V|)$, $\Omega(\lg \frac{n-1}{k})$, *and* $\Omega(k \lg \frac{|V|}{k})$, *respectively.* □

6.2 Combining Malicious and Crash Behavior

We now study the impact of combining malicious behavior with crash failures. We assume that the adversary, in addition to having a budget of β messages, can also crash up to t players. We consider the problem of *binary consensus.* The n honest players each propose a value. *Liveness* requires that all non-crashed players eventually decide a value. *Agreement* requires all players that decide to choose the same value. *Validity* requires that if all non-crashed players propose the same value, then all deciding players choose that value.

[1] Often k-selection is oblivious to initial values. We allow a dependence on the initial values, strengthening the lower bound.

By a simple indistinguishability argument, it is easy to see that consensus is impossible if $n \leq 2t$: it is impossible to distinguish a correct player from a crashed player that is simulated by the adversary; thus no player can decide in an execution in which t players propose '0' and t propose '1'.

We therefore assume that $n = 2t+1$, and establish a lower bound of $2\beta+\Theta(t)$ on the round complexity of consensus. Our bound reveals the interesting fact that the possibility of crashed honest devices increases the power of the malicious adversary. This is perhaps surprising as, if there is no malicious adversary, crash-failures have no effect on termination (in a synchronous broadcast network).

As before, we use a simulation by Alice and Bob of the (t-resilient) n-player consensus protocol. The simulation, however, is more challenging than those used for the n-player problems studied previously as we must compensate for the crash failures. We do not start the simulation from the initial configuration, but instead from one of two univalent configurations arising after t rounds. These configurations are constructed in Lemma 3, which is interesting in its own right as it exhibits executions in which information (about initial values) is transmitted at most one bit per round. By combining it with valency arguments, we show how the 3-player game can aid the construction of involved lower bounds.

Theorem 8. *Any t-resilient binary consensus protocol requires at least $2\beta + t$ rounds to terminate.*

We fix the environment such that if multiple messages are sent in a round, and the adversary does not broadcast, then the message sent by the player with the smallest id is received by everyone. An execution (or prefix) is *failure-free* if it includes no crashes or broadcasts by the adversary.

Given these assumptions, it is clear that each initial configuration results in a deterministic failure-free execution. We represent all of these possible failure-free executions as a tree T. Every execution begins at the root, and a node at depth r represents the execution at the beginning of round r. Each node at depth r contains one outgoing edge for every possible message m that may be received in round r, and one outgoing edge for a silent round (labeled $\perp$). Thus, every failure-free execution of A is represented by a single path in T. Accordingly, for each initial configuration c, we say that a node $x \in T$ is *reachable* from c—with respect to A—if the path associated with c's failure-free execution includes node x. We define the tree $T(A)$ to be T pruned to contain only reachable nodes. That is, if $x \in T(A)$, then there exists some initial configuration c for which x is reachable. Notice that if a depth r node x is reachable for two initial configurations c and c', and some player i has the same initial value in c and c', then at the beginning of round r, player i cannot distinguish c from c'. If c is 0-valent, and c' is 1-valent, then i cannot decide prior to round r.

Lemma 3. *There exists a path of length t in $T(A)$, ending at node R_t, where R_t is reachable from two initial configurations, c_0 and c_1, such that some player p_t has the same initial value in c_0 and c_1, and every crash-free extension of c_0 is 0-valent and every crash-free extension of c_1 is 1-valent, with respect to A.*

Proof. Starting at the root of $T(A)$, given an initial configuration c_0, construct a path of length t by applying the following: (1) If there exists ≥ 1 outgoing message edges, choose the message from the player with the smallest id. (2) Otherwise, follow the $\perp$ edge. Let R_t be the node reached after t iterations.

Configuration c_0 contains either a majority of '0's or a majority of '1's. Notice that a majority contains at least $t+1$ players, since $n = 2t+1$. Assume without loss of generality that a majority of players (i.e., at least $t+1$) propose '0' in c_0. This implies that any crash-free extension of c_0 must decide '0', since any such execution is indistinguishable from one in which all players propose '0', and those $\leq t$ players proposing '1' are crashed nodes emulated by the adversary—in which case a decision of '1' violates validity.

We now construct an initial configuration c_1. Denote by P the set of players that broadcast messages which were received along the path to R_t. Note that P contains $\leq t$ players. Choose c_1 such that the players in P propose the same initial value as in c_0, and the remaining players (at least $t+1$) all propose '1'. Choose some $p_t \in P$. (If $|P| = 0$, then arbitrarily choose one player p_t to have the same initial value in c_0 and c_1.) It is clear, by the same reasoning applied to c_0, that all crash-free extensions of c_1 must decide '1'. It follows that R_t is reachable from c_1, by a straightforward induction argument.

Proof (Theorem 8). Let α_0 (resp. α_1) denote the failure-free execution prefix starting from c_0 (resp. c_1) and ending at R_t. Executions α_0 and α_1 are indistinguishable with respect to p_t; hence p_t has not decided prior to round t. To this point, the adversary has used zero broadcasts. To achieve a further 2β delay, we defer to Alice and Bob, who can solve the binary communication game by performing a *crash-free* simulation of the n-player protocol, in which Alice begins in the final state of α_0 or α_1, and Bob simulates p_t. This simulation cannot terminate in fewer than 2β round, implying the desired bound. □

Corollary 3. *Any t-resilient binary consensus protocol has a jamming gain of at least 2 and a disruption-free complexity of $\Omega(t)$.* □

7 Upper Bounds for the n-Player Problems

We now present protocols for reliable broadcast, leader election, static k-selection, and binary consensus. Our reliable broadcast and consensus protocols match the lower bounds. Those for leader election and k-selection leave a gap.

Reliable Broadcast. An algorithm for reliable broadcast follows from the algorithm in Section 5. The source runs Alice's protocol, and all other players run Bob's protocol, resulting in a running time of $2\beta + O(\lg |V|)$, matching the lower bound. This protocol requires the source to have a budget of $\beta + \lg |V|$.

Binary Consensus. Assuming t crashes, consensus can be achieved using reliable broadcast: each of $2t+1$ players transmits their initial value sequentially. (Notice that a crashed player, if there is no malicious interference, transmits a '0', according to the protocol.) Everyone decides the majority value. The running time is $2\beta + \Theta(t)$. Each player needs a budget of $\beta + 1$ broadcasts.

Leader Election. In order to elect a leader, we use a *tournament tree*, a binary tree with n leaves, each labeled with a player's id. Assume $c \geq 1$ is an integer parameter. The protocol begins at the root, and at each step descends to a child or ascends to the parent. At each step, the protocol determines whether there are any participants in the left or right subtrees. First, each participant in the left subtree broadcasts up to c times. If all of these rounds are non-silent, the protocol descends to the left subtree. Otherwise, the first time a silent round occurs, it skips the remaining rounds and checks the right subtree: each participant in the right subtree broadcasts up to c times. If all of these rounds are non-silent, the protocol descends to the right subtree. Otherwise, on the first silent round, the protocol ascends to the parent. On reaching a leaf, the identified node uses reliable broadcast to transmit a '1' if it is participating and a '0' otherwise. In the latter case, the protocol ascends to the parent and continues. Each participant needs a budget of $2c \lg n + \beta + 1$ broadcasts.

Theorem 9. *The leader election protocol terminates after* $2\beta\frac{c+1}{c} + 2c \lg n + 2$ *rounds, for all* $c \geq 1$. □

k-Selection. A protocol for static k-selection can be obtained by repeating the leader election protocol k times, each time using reliable broadcast to transmit the initial value. The protocol completes when leader election finds no further contenders. Each participant needs a budget of $2c \lg n + \beta + \log |V|$ broadcasts.

Theorem 10. *The k-selection protocol terminates in* $2\beta\frac{c+1}{c} + 2kc \lg n + k \lg V + 2k + 2$, *which equals* $2\beta\frac{c+1}{c} + O(ck \lg |V|)$ *if* $\lg n = O(\lg |V|)$, *for all* $c \geq 1$. □

8 Concluding Remarks

Interestingly, our lower bounds hold for weaker games. Lemmas 1 and 2 imply that calculating equality, *bitwise-and* or *bitwise-or* have the same round complexity as the 3-player game. We also conjecture that even for a randomized algorithm, $2\beta + \Theta(\lg |V|)$ rounds are needed. A future research direction is to extend our results to multihop environments.

Ackowledgments

We are grateful to H. Attiya and G. Chockler for their comments and discussions.

References

1. Brown, T.X., James, J.E., Sethi, A.: Jamming and sensing of encrypted wireless ad hoc networks. Technical Report CU-CS-1005-06, UC Boulder (2006)
2. Perrig, A., Szewczyk, R., Tygar, J.D., Wen, V., Culler, D.E.: Spins: Security protocols for sensor networks. Wireless Networks **8**(5) (2002) 521–534

3. Karlof, C., Sastry, N., Wagner, D.: Tinysec: A link layer security architecture for wireless sensor networks. In: Embedded Networked Sensor Systems. (2004)
4. Koo, C.Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: Principles of Distributed Computing. (2004) 275–282
5. Bhandari, V., Vaidya, N.H.: On reliable broadcast in a radio network. In: Principles of Distributed Computing. (2005) 138–147
6. Pelc, A., Peleg, D.: Broadcasting with locally bounded byzantine faults. Information Processing Letters **93**(3) (2005) 109–115
7. Drabkin, V., Friedman, R., Segal, M.: Efficient byzantine broadcast in wireless ad hoc networks. In: Dependable Systems and Networks. (2005) 160–169
8. Pelc, A., Peleg, D.: Feasibility and complexity of broadcasting with random transmission failures. In: Principles of Distributed Computing. (2005) 334–341
9. Clementi, A., Monti, A., Silvestri, R.: Optimal f-reliable protocols for the do-all problem on single-hop wireless networks. In: Algorithms and Computation. (2002) 320–331
10. Chlebus, B.S., Kowalski, D.R., Lingas, A.: The do-all problem in broadcast networks. In: Principles of Distributed Computing. (2001) 117–127
11. Kranakis, E., Krizanc, D., Pelc, A.: Fault-tolerant broadcasting in radio networks. In: European Symposium on Algorithms. (1998) 283–294
12. Clementi, A., Monti, A., Silvestri, R.: Round robin is optimal for fault-tolerant broadcasting on wireless networks. J. Parallel Distributed Computing **64**(1) (2004) 89–96
13. Koo, C.Y., Bhandari, V., Katz, J., Vaidya, N.H.: Relibable broadcast in radio networks: The bounded collision case. In: Principles of Distributed Computing. (2006)
14. Stahlberg, M.: Radio jamming attacks against two popular mobile networks. In: Helsinki University of Technology Seminar on Network Security. (2000)
15. Negi, R., Perrig, A.: Jamming analysis of mac protocols. Technical report, Carnegie Mellon University (2003)
16. Hu, Y., Perrig, A.: A survey of secure wireless ad hoc routing. IEEE Security and Privacy Magazine **02**(3) (2004) 28–39
17. Gupta, V., Krishnamurthy, S., Faloutsos, S.: Denial of service attacks at the mac layer in wireless ad hoc networks. In: Military Communications Conference. (2002)
18. Abramson, N.: The aloha system - another approach for computer communications. Proceedings of Fall Joint Computer Conference, AFIPS **37** (1970) 281–285
19. Metcalf, R.M., Boggs, D.R.: Ethernet: Distributed packet switching for local computer networks. Communications of the ACM **19**(7) (1976) 395–404
20. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. Journal of Computer and System Sciences **45**(1) (1992) 104–126
21. Woo, A., Whitehouse, K., Jiang, F., Polastre, J., Culler, D.: Exploiting the capture effect for collision detection and recovery. In: Workshop on Embedded Networked Sensors. (2005) 45–52
22. Clementi, A., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: Symposium on Discrete algorithms, Philadelphia, PA, USA (2001) 709–718
23. Kowalski, D.R.: On selection problem in radio networks. In: Principles of Distributed Computing, New York, NY, USA, ACM Press (2005) 158–166

EMPIRE OF COLONIES

Self-stabilizing and Self-organizing Distributed Algorithms*

(Extended Abstract)

Shlomi Dolev and Nir Tzachar

Department of Computer Science, Ben-Gurion University of the Negev, Israel
dolev@cs.bgu.ac.il, tzachar@cs.bgu.ac.il

Abstract. Self-stabilization ensures automatic recovery from an arbitrary state; we define *self-organization* as a property of algorithms which display local attributes. More precisely, we say that an algorithm is self-organizing if (1) it converges in sublinear time and (2) reacts "fast" to topology changes. If $s(n)$ is an upper bound on the convergence time and $d(n)$ is an upper bound on the convergence time following a topology change, then $s(n) \in o(n)$ and $d(n) \in o(s(n))$. The self-organization property can then be used for gaining, in sublinear time, global properties and reaction to changes. We present self-stabilizing and self-organizing algorithms for many distributed algorithms, including distributed snapshot and leader election.

We present a new randomized self-stabilizing distributed algorithm for cluster definition in communication graphs of bounded degree processors. These graphs reflect sensor networks deployment. The algorithm converges in $O(\log n)$ expected number of rounds, handles dynamic changes locally and is, therefore, *self-organizing*. Applying the clustering algorithm to specific classes of communication graphs, in $O(\log n)$ levels, using an overlay network abstraction, results in a self-stabilizing and self-organizing distributed algorithm for hierarchy definition.

Given the obtained hierarchy definition, we present an algorithm for hierarchical distributed snapshot. The algorithms are based on a new basic snap-stabilizing snapshot algorithm, designed for message passing systems in which a distributed spanning tree is defined and in which processors communicate using bounded links capacity. The combination of the self-stabilizing and self-organizing distributed hierarchy construction and the snapshot algorithm form an efficient self-stabilizer transformer. Given a distributed algorithm for a specific task, we are able to convert the algorithm into a self-stabilizing algorithm for the same task with an expected convergence time of $O(\log^2 n)$ rounds.

1 Introduction

The availability and robustness, as well as the possibility for on-demand reconfiguration of large systems, are in many cases vital; be it clusters of servers that support commercial activity, a grid of computers that participate in a complicated computation or a

* Partially supported by IBM, Israeli ministry of science, Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences, Israeli Grid Consortium and the Lynn and William Frankel Center for Computer Sciences.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 230–243, 2006.

dynamic sensor network. In particular, an important aspect for large on-going systems is the ability to automatically recover from an inconsistent state, namely to be *self-stabilizing* ([11]) or in other words, to have a system that can be started in an arbitrary state.

To capture the need of the industry in autonomic and self-* systems, we propose combining self-stabilization (in fact SuperStabilization [12]) with *self-organization*. While self-stabilization is well defined, the self-organization property has no widely agreed upon definition. We propose to define self-organization as satisfying two main properties: locality and dynamicity. Namely, we require that (1) the algorithm stabilizes in sublinear time with regards to the number of processors and that (2) the addition and removal of processors influences a small number of other processors' states. In other words, if $s(n)$ represents the stabilization time and $d(n)$ represents an upper bound on the stabilization time (and number of state changes) following a dynamic topology change, then: $s(n) \in o(n)$ and $d(n) \in o(s(n))$.

In this work, we enable algorithms to define (on the fly) and then use *hyper communication links*, which are overlay links that are constructed of communication links along a path. We regard the time that a message travels over such a link as one time unit, as (almost) no processing is involved in forwarding messages over these links (e.g., [13, 26], MPLS [6]).

Main Contribution. We define the self-organization property to capture locality and dynamicity. We present a clustering algorithm (in fact, a distributed maximal independent set algorithm) which is both self-stabilizing and self-organizing. To realize the clustering algorithm in an asynchronous system we present a scheme of local synchronization, achieved by using a local snapshot protocol. We employ the aforementioned clustering algorithm to define a graph hierarchy which can be used to convert any distributed task to be self-stabilizing incurring only a sublinear time overhead.

• *Self-Stabilizing and Self-Organizing hierarchy definition.* The hierarchy of subsystems is defined by partitioning the communication graph into small clusters, after which clusters are merged to form bigger clusters and so on. The partition can be done according to a designer's input, using an automatic off-line clustering algorithm or even an on-line clustering algorithm that reflects the system's current behavior. In particular, we suggest a randomized self-stabilizing and self-organizing partition that is based on periodical collection of local topology (up to a certain distance). The collected local topology supports a randomized local leader election, in which a non leader processor that does not identify a leader within a certain distance x tries to convert itself to a leader. Leaders within distance x from each other are eliminated, until there are no leaders that are within distance x or less from each other. Higher level partitions, using larger distances and overlay network abstraction between leaders, are constructed in a similar way.

In asynchronous systems, our clustering algorithm uses (for each processor) a (local) self-stabilizing snapshot algorithm for obtaining local synchronization of actions.

• *Self-Stabilizing snapshots.* We present a snap-stabilizing (e.g., [7]) snapshot algorithm for distributed systems, that uses message passing with *bounded link capacity*, in which a spanning tree is distributively defined. Our snapshot algorithm is designed for

a message passing system in which any initial state of link contents is considered and in which the possibility of messages overflow (due to sending a message through a full link) is incorporated into the model.

Our snapshot algorithm can also be applied to systems with a general communication graph in which a rooted spanning tree is distributively defined by another self-stabilizing algorithm. The spanning tree may be an output of a self-stabilizing (BFS) rooted tree construction algorithm. In this case, however, we obtain only *on-demand stabilization* rather than snap-stabilization. On-demand stabilization ensures that regardless of the number of new requests (for snapshots), the system reaches a state, such that eventually any new request results in a correct output (snapshot). In other words, stabilization does not rely on repeated invocations of new (snapshot) requests. Our on-demand self-stabilizing snapshot algorithm serves us as a basic building block in order to obtain our hierarchical snapshot schemes.

• *Overlay network based snapshot.* We suggest an approach for hierarchical snapshot based on an (fifo preserving) overlay networks abstraction. We enable each subsystem to perform an independent snapshot, and further enable each level of the hierarchy to perform a local snapshot. We suggest the use of overlay communication links which "directly" connect leaders of clusters. It is worthwhile noting that an (fifo) overlay network link may be in fact a path of physical links. It is also evident that the communication over an overlay link is much faster than the sum of the single hop communication links that implement the overly link[1].

Leaders of subsystems are defined, and the communication between processors in different subsystems traverses the overlay communication links between the leaders of the subsystems. Thus, there is no need for recording the messages over physical links between subsystems unless they are part of an overlay communication link. When a snapshot is invoked by a leader of a subsystem (possibly due to a request forwarded to the leader by another processor), the leader uses the overlay network to notify (send snapshot markers to) the leaders of the subsystems that belong to its subsystem. These leaders, in turn, are responsible for performing a snapshot in their subsystem in the same manner.

Related Work

• *Self-organization.* In recent years, the concept of self-organization has been widely mentioned in the scope of distributed computing and peer to peer networks. Many works have claimed being self-organizing, but a mere fraction of these works also tries to give a specific definition of what self-organization really is. In [2] a framework for self-organization is proposed, including formal definitions of the self-organization concept and complementary proof techniques which can be used to prove that algorithms are indeed self-organizing.

Each algorithm is required to have an associated evaluation criterion, which operates on the immediate neighborhood of a process. This evaluation criterion does not take into account the influence of other local neighbors, say those that are within a constant distance.

[1] In some cases, preassigned frequencies or/and supporting switching hardware can be used. e.g.,MPLS–[6].

• *Fault containment.* Fault containment, using persistent bits, voting on replicated bits (usually for non reactive systems) is another way of addressing locality (e.g., [20, 16, 1, 4]). The idea is to repair transient faults starting from a safe global system configuration. In such a case, it is possible (unlike in the case of topology changes) to change the state of the affected processors back to the state prior to the fault. In this context, our algorithm is self-stabilizing and when started in a safe configuration can handle k transient faults *as well as topology changes* occurring approximately at the same time, in expected $O(\log k)$ rounds. Moreover, our scheme is the first to support many core distributed tasks, such as self-stabilizing leader election algorithm and snapshots algorithms in $O(\log^2 n)$ expected rounds.
• *Cluster and hierarchy construction.* Self-stabilizing and self-healing constructions of hierarchies, in the domain of sensor networks, appear in [28]. The authors divide the plane into hexagonal cells. In each cell a *head* that corresponds with a cluster leader is elected. The existence of a unique processor, the *big node*, which acts as an initiator is assumed. The big node determines the center of the first hexagon, fixating the location of its own cluster. The big node elects heads in adjacent hexagonal cells which will subsequently elect heads in their adjacent cells. The time complexity of this algorithm is obviously proportional to the diameter of the communication graph. Our algorithm does not assume a leader and converges within $O(\log n)$ expected number of rounds and reacts to dynamic changes locally. A *constant* time clustering algorithm is presented in [8]. The algorithm assumes that processors can measure time and therefore does not fit asynchronous systems.

Our clustering algorithm is in fact a maximal independent set algorithm. A classical maximal independent set algorithm is presented in [24]. The algorithm is designed for a synchronous system and converges (from a pre-defined initial state) within $O(\log n)$ expected convergence time. Our algorithm is designed for asynchronous systems, is self-stabilizing and self-organizing and converges within expected $O(\log n)$ rounds for constant degree graphs.

A recent work by Wattenhofer and Moscibroda [25] presents an algorithm for computing a maximal independent set in radio networks. The system model is fundamentally different from the one presented here: Processors can broadcast their messages asynchronously, but no collusion detection mechanism is provided. The algorithm presented converges in (expected) polylogarithmic time, and processors which join the algorithm are promised to be covered in (expected) polylogarithmic time.

In [21], the authors present lower bounds on distributed approximation algorithms for the minimum vertex cover problem. Their bounds can also be applied to the maximum independent set problem. We do not seek a maximum independent set, and our algorithm defines a maximal independent set.

Applications of hierarchy in the self-stabilization domain are described in [15]. The authors argue that the hierarchical construction can be used to shorten the convergence time of various self-stabilizing distributed algorithms. As an example, the authors present an application to spanning tree construction. However, the authors do not present an algorithm for defining the hierarchy but assume it is defined beforehand.
• *Snap-stabilization.* Snap-stabilizing algorithms were first introduced in [3]. A protocol is said to be snap-stabilizing if, upon the first invocation of the protocol by one

processor, the protocol behaves according to its specifications. The snapshot algorithm we present is snap-stabilizing, provided specific preconditions are met; Namely, a tree structure is defined and a leader is present beforehand. When the leader invokes a snapshot, the snapshot terminates with a correct answer.

• *Dynamic graph algorithms.* Extensive research on distributed dynamic algorithms appeared in the literature (e.g., [13] and the references therein). Still, our algorithm is the first self-stabilizing and self-organizing distributed (graph) algorithm. Another related aspect of our work is related to dynamic (graph) data structures (e.g., [17] and the reference therein). We achieve a committing time (logarithmic and polylogarithmic) in (fault tolerance) distributed settings for an important class of graphs.

Paper organization. In Section 2 we present the system model and in Section 3 the basic on-demand snapshot algorithm. Hierarchy construction schemes are described in section 4. Conclusions appear in Section 5. More details and most of the proofs are omitted from this extended abstract and can be found in [14].

2 System Model

The *system* consists of n *processors*, denoted by $p_1, p_2, \ldots, p_n$. The processors are connected by *communication links*. Each *processor* is modeled by a state machine that can send and receive *frames* (or low level messages) to/from a subset of the processors. We use a uni-directed communication graph $G = (V, E)$ to represent the system, where each processor p_i is represented by a vertex $v_i \in V$ and each communication link used for transferring frames from p_i to p_j is represented by an edge $(i, j) \in E$. We further assume that the existence of the edge $(i, j) \in E$ implies the existence of an opposite directed edge $(j, i) \in E$ and that the number of edges attached to a processor is bounded by a constant. We define the $dist$ of two processors p and q, $dist(p, q)$, as the length of the shortest path between p and q in the graph. For a processor p and a constant x, we denote $f_p(x)$ as the number of processor q such that $dist(p, q) \leq x$. We further define $f_G(x)$ (or just $f(x)$ where G is clear from the context) as the maximal $f_p(x)$ over all processors p in the graph.

Overlay edges. We use the term overlay edge to denote a path of edges that connects two processors in the system. When the path is predefined and fixed, it acts as a virtual link in which almost no processing is required by intermediate processors in order to forward the message from source to destination. We allow processors to define and use, on the fly, overlay edges to other processors, when the underlying path is known. We regard the time it takes a message to traverse such an overlay link as the time for traversing a link that directly connects two neighboring processors. The definition is motivated by (e.g., telephony) systems, where switches along a path are configured for a session and the path is essentially a wire. In such a case, messages are buffered only at the endpoints, resulting in an overly link of the same capacity as the original links.

We assume class of graphs for which a correlation exists between the number of edges along a shortest path and the geographical distance of the path's end-points.

The system is asynchronous, meaning that there is no correlation between the non constant rate of steps taken by the processors. We assume that the capacity of the communication channels (equivalently the number of items in the fifo queues that represent

the links) is bounded, by the constant lc. Whenever a processor p_i sends a frame to a neighbor p_j, when the link (i, j) already contains lc frames, we assume that one of the frames (not necessarily the new one) is lost while the fifo order of the rest of the frames is preserved. In fact, since frames can always be lost, we restrict the pattern of frame loss steps to be such that if frames are sent infinitely often, frames are also received infinitely often.

We further abstract the activity of communication links by assuming an underline snap-stabilizing ARQ data link algorithm that transfers frames in order to ensure that high level *messages* transfer respects the following: (1) messages sent from p_i to p_j are received by p_j in a finite (but yet unbounded) time (2) and message delivery respects the exactly once delivery and fifo ordering policies. We note that the ARQ algorithm performed on one link of a processor p_i does not block the receive operations (and corresponding steps) from the links attached to p_i. We assume that eventually when p_i sends a message m to p_j (and p_i does not send further messages), p_i receives acknowledgment for m after p_j received m.

A configuration c of the system is a tuple $c = (S, L)$; S is a vector of states, $\langle s_1, s_2, \cdots s_n \rangle$, where the state s_i is a state of processor p_i; L is a vector of *link states* $\langle l_{i,j}, \cdots \rangle$ for each $(i, j) \in E$. A *link* $l_{i,j}$ is modeled by a fifo queue of frames that are waiting to be received by p_j and the contents of the queue is the state of the link. Whenever p_i sends a frame f to p_j, f is enqueued in $l_{i,j}$. Also, whenever p_j receives a frame f from p_i, f is dequeued from $l_{i,j}$. A processor changes its state according to its transition function (or program). A transition of processor p_i from a state s_j to state s_k is called an *atomic step* (or simply a step) and is denoted by a. A step a consists of local computation and of either a single send or a single receive operation.

We model our system using the interleaving model. An *execution* is a sequence of global configurations and *steps*, $\mathcal{E} = \{c_0, a_0, c_1, a_1, \ldots\}$, so that the configuration c_i is reached from c_{i-1} by a step a_i of one processor p_j. The states changed in c_i, due to a_i, are the one of p_j (which is changed according to the transition function of p_j) and possibly that of a link attached to p_j. The content of a link state is changed when p_j sends or receives a frame during a_i. An execution $\mathcal{E}$ is *fair* if every processor executes a step infinitely often in $\mathcal{E}$ and each link respects the bounded capacity loss pattern. In the scope of self-stabilization we consider executions that are started in an arbitrary initial configuration.

A *task* is defined by a set of executions called *legal executions* and denoted LE. A configuration c is a *safe configuration* for a system and a task LE if every fair execution that starts in c is in LE. A system is self-stabilizing for a task LE if every infinite execution reaches a safe configuration with relation to LE. We sometimes use the term "the algorithm stabilizes" to note that the algorithm has reached a safe configuration with regards to the legal execution of the corresponding task.

The *snapshot task* $\mathcal{S}$ for a system is defined by a set of executions $\mathcal{E}_\mathcal{S}$ started in an arbitrary configuration, so that if a snapshot starts in an atomic step a_r, there is a configuration c_s, that follows a_r, in which a processor receives a global snapshot gs. Moreover, assuming r is minimal, there exists an execution $\mathcal{E}'$ that starts immediately before a_r, reaches gs and then continues to the configuration c_s. $\mathcal{E}'$ may be different from the execution which actually took place.

We use the notion of *asynchronous rounds* to measure the time complexity of an algorithm. The first *asynchronous round* in execution $\mathcal{E}$ is the shortest prefix of $\mathcal{E}$ in which each processor (or process) communicates with all of its neighbors (either through a directly connecting communication link or through an overlay edge). The second asynchronous round in $\mathcal{E}$ is the first asynchronous round of the suffix of $\mathcal{E}$ that immediately follows the first asynchronous round in $\mathcal{E}$. The time complexity of an algorithm is the number of asynchronous rounds (or simply rounds) that are required to achieve the task of the algorithm.

3 On-Demand (Snap-) Stabilizing Message Passing (Tree-) Snapshot Algorithm

Our starting point is the unbounded snapshot algorithm presented in [18] and the snap-stabilizing algorithm presented in [7] which we modify to a bounded message passing snap-stabilizing algorithm. Namely, we ensure that any new request for a snapshot will result in a correct snapshot. This requirement differs from the one presented in [18] where snapshots must be continuously and infinitely often invoked. In our case, the algorithm is ready for future requests even when no snapshot requests are made.

The snapshot algorithm uses, as a building block, a snap-stabilizing data link algorithm which is specifically designed for bounded capacity links (the data link algorithm appears in [14]. The algorithm uses three variables to control the data flow on the link. $current[q]$ holds the current value which is sent to q. $next[q]$ holds the next value to be sent, which is suspended until an acknowledgment on $current[q]$ arrives. $last[q]$ holds the last acknowledged message. The snapshot algorithm then uses $next[q]$ to send messages to a neighbor q, and waits for an acknowledgment in $last[q]$. To ensure self-stabilization, a sequence number is attached to each message sent, and is incremented by one modulo two times the link capacity plus one for each new message. To ensure snap-stabilization, each message is sent repeatedly two times the link capacity plus one – a step that ensures that the message had arrived and that the acknowledgment is valid; if lc is the link capacity, then there can be at most $2 \cdot lc$ messages in transit on the link. By using $2 \cdot lc + 1$ labels, one label is guaranteed to be a new label, which does not exist in the link. This, in turn, ensures that a correct acknowledgment is received.

The algorithm is designed for a system in which a rooted spanning tree is distributively defined. It is based on performing two consecutive tree-PIFs (propagation of information with feedback using a spanning tree) and then employing the original snapshot algorithm of [5]. Each PIF uses the rooted tree in order to propagate a command (*initialize* and then *prepare*) and receive feedback on the completion of the propagation (of the initialize and prepare commands, respectively). A processor that receives a command from its parent, propagates it to its children and also "cleans" the non-tree edges attached to it. Once a processor p receives an acknowledgment from all its children that their subtree received the command and once p finishes cleaning the attached non-tree links, p sends an acknowledgment to its parent regarding the completion of the command propagation. Both tree-PIFs are completed within $O(d)$ rounds (assuming a BFS tree is used). When the first (initialize) tree-PIF is completed, no marker of previous incarnations of the snapshot algorithm is present in the system and processors

disregard all incoming snapshot markers. After the second (prepare) tree-PIF is completed, processors do not ignore markers and the root may then initiate the original snapshot algorithm of [5].

A detailed description of the algorithm, as well as the correctness proof and complexity analysis, appears in [14]. We mention here the main properties of the snapshot algorithm:

Theorem 1. *Once the root of the tree initiates a snapshot cycle, a correct snapshot will be obtained in $O(h)$ rounds, where h is the height of the tree.*

4 Hierarchical Construction Schemes

A hierarchical system is represented by a communication graph, $G = (V, E)$ and a hierarchy tree $\mathcal{HT} = (V_h, E_h)$. Each node in $\mathcal{HT}$, l_i, represents a set of nodes in V, called a *subsystem*, so that if l_i and l_j are at the same level of $\mathcal{HT}$, then $l_i \cap l_j = \emptyset$. Furthermore, if K is a set of nodes at level i of $\mathcal{HT}$, then $\cup_{j \in K} l_j = V$. The nodes of the graph are processors and the edges are their communication channels. We require that each subsystem is a connected component of G.

Next we present a self-stabilizing and self-organizing algorithm for constructing clusters. In general, the clustering algorithm builds clusters of size smaller than a fixed parameter. Furthermore, each cluster is defined by a "native" leader.

• *Cluster construction.* The clustering algorithm we present is a maximal independent set algorithm, where each dominator is denoted as a *leader* and each dominatee joins the closest dominator (ties are broken by, say, leaders identifiers).

Each processor p uses several key variables: $leader_p, candidate_p, id_p$ and rtp_p. $leader_p$ denotes whether p is currently a leader. $candidate_p$ is set to true if p is trying to become a leader. id_p is the identifier each processor has, and rtp_p is a random temporary identifier used to break the symmetry between processors.

Predicates:
leader(C_p) :=
$\exists q \in C_p | q \neq p \wedge leader(q)$

1 $(leader_p \oplus leader(C_p)) = true$:
/* do nothing (stable). */

2 $leader_p = false \wedge leader(C_p) = false$:
3 $rtp_p \leftarrow random()$
4 $candidate_p \leftarrow true$
5 $C'_p \leftarrow$ new snapshot
6 **if** $leader(C'_p) = true$ **then**
7 $candidate_p \leftarrow false$
8 $leader_p \leftarrow false$
9 **else if** $\forall q \in C'_p \ candidate_q! = true \vee$
10 $(\langle rtp_q, id_q \rangle < \langle rtp_p, id_p \rangle)$ **then**
11 $leader_p \leftarrow true$
12 **else**
13 $candidate_p \leftarrow false$
14 $leader_p \leftarrow false$
15 **end**

16 $(leader_p = true \wedge leader(C_p) = true)$:
17 $candidate_p \leftarrow false$
18 $leader_p \leftarrow false$

Fig. 1. Asynchronous Leader Election Algorithm for Processor p

One may try using the processors' identifiers in order to break symmetry. However, occasionally an unfortunate order of id's may lead to a convergence time which is proportional to the diameter of the graph. We use randomness to break ties in order to overcome such a scenario.

The construction algorithm is composed of several parts. All processors participate in an (asynchronous) update algorithm up to distance x. The update algorithm is designed for an asynchronous system. Each processor p holds a table of tuples, each of the form $\langle id_q, dist_q, parent_q \rangle$. Each tuple represents a processor q in the communication graph. id_q is the unique identification of q, $dist_q$ is the minimal distance between p and q and $parent_q$ is the id of a neighboring processor of p, which is the first on a shortest path from p to q. Repeatedly, p combines all the tables of its neighbors and for each of the conflicting tuples (in which the id is the same), p chooses the tuple with the minimal $dist$ (further ties are broken using the $parent$ value). Next, p chooses only entries with $dist = k \leq x$, such that there exist entries with $dist = j$ for all $j < k$. All other entries are deleted. The removal of entries ensures fast stabilization [12], as an entry which is j hops away from p must be connected to p by a path, so there must exists entries which are $1, 2, \ldots, j-1$ hops from p. Afterwards, p adds 1 to the distance field of every tuple and finally adds the tuple $\langle id_p, 0, nil \rangle$ to form the new table.

We adapt the aforementioned update algorithm to our system in several manners. First, each tuple will hold two extra values, $leader_p$, rtp_p. Next, each processor p continuously sends its table to all neighboring processors. In addition, p maintains an internal array which consists of the most recent topology tables p received from each neighboring processor. The computation of p's topology table is done on the basis of this array. Furthermore, in the validation phase we also delete entries with $dist > x$. Consequently, p's table will reflect its neighborhood up to distance x from p. The correctness of the revised update algorithm is trivially preserved, and the convergence time is $O(x)$ rounds.

Based on the update tables, each processor p constructs a tree rooted at p and of depth not exceeding x. Using the tree, each processor invokes the snapshot algorithm to collect the state of its neighborhood. We use the snapshot algorithm to perform a PIF algorithm, and by adding information to the markers used in the snapshot process we achieve the desired PIF effect. The number of trees and snapshot protocols each processor must participate in can be calculated from the topology collected earlier.

Constantly (this is to say that the time frame is not important), each processor p will take a snapshot of the surrounding neighborhood (up to distance x). After the snapshot is collected, the algorithm in Figure 1 is invoked. Since the snapshot algorithm is guaranteed to be finished in each invocation (although the result might be incorrect, since the rooted tree has not stabilized yet), we are guaranteed that future invocations of the snapshot algorithm will take place. For a snapshot obtained at p, C_p, we denote $leader(C_p) = true$ if there exists a processor $q \neq p$ in C_p, such that $leader_q = true$.

Let us assume that a complete snapshot C_p is obtained at p. The four combinations of $leader_p$ and $leader(C_p)$ determine the course of actions p must follow. First, consider the most simple cases where $leader_p = false \wedge leader(C_p) = true$ or $leader_p = true \wedge leader(C_p) = false$. In these cases, p should avoid taking any action, since, as far as p can tell, the situation is correct. The complex cases are when there are no leaders

in p's vicinity and p is not a leader itself or when p is a leader and can see another leader within a distance of x from itself. In case $leader_p = false \wedge leader(C_p) = false$, p will first choose a random number (from a predetermined range) and store it in rtp_p. Then, p will assign $true$ to $candidate_p$ (Figure 1 lines3-4). The next operation is propagating the information that p wishes to become the leader of its neighborhood. This is achieved through the use of the snapshot protocol which results in a new snapshot at p, C'_p (line 5); such propagation can be achieved by piggy-backing information on the markers of the snapshot. Now, if C'_p does not contain information about a leader or another candidate, p can safely place itself as a leader and set $leader_p = true$. However, if $leader(C'_p) = true$ holds, p should set $candidate_p$ to $false$, since there is now a leader in p's neighborhood. Last, if there are other candidates in C'_p, p will become a leader if (and only if) the tuple $\langle rtp_p, id_p \rangle$ is larger than all other candidate's tuples in C'_p (line 10).

The last case is when $leader_p = true \wedge leader(C_p) = true$ (line 16). Upon detecting such a condition, p will immediately assign $leader_p$ and $candidate_p$ with $false$ and will start a new cycle of the algorithm..

The correctness proof, as well as the time complexity analysis, appears in [14]. We only mention the following corollary.

Corollary 1. *In every fair execution, each processor has a positive probability of becoming stable in every $O(x)$ rounds and it holds by [19] that within $O(\log n)$ expected number of rounds, the algorithm converges to a stable state.*

• *Hierarchy construction* Constructing the hierarchy is achieved by a repeated application of the clustering algorithm. We suggest using the clustering algorithm on the original graph G, constructing clusters with $x > 1$ (in essence, a minimal x-dominating set). We then propose to dynamically define an overlay network between the leaders of each cluster and apply the same scheme to the resulting graph. The process is completed after a single cluster, composed of the entire graph G, is finally defined. The resulting hierarchy is of $O(\log n)$ levels, and in each level i (level 0 is the original graph, G) there exist at most $\frac{n}{2^i}$ processors. This bound arises from the fact that each leader p has at least one processor directly connected to p, which is not directly connected to any other leader. Since there exist $O(\log n)$ levels in the hierarchy and since communication on overlay edges is considered non expensive, the hierarchy construction algorithm stabilizes within $O(\log^2 n)$ expected rounds ($O(\log n)$ for each level, times $O(\log n)$ levels), assuming the degree of each of the hierarchy levels is bounded.

Next, we describe the construction of the overlay network and present a graph class in which the degree of each hierarchy level is bounded.

• *Overlay network construction.* Let $G = G_0 = (V_0, E_0)$ be the original graph, to which we apply our clustering algorithm. We define $G_i = (V_i, E_i)$ so that $V_i = \{p \in V_0 | \, p \text{ is a leader in } V_{i-1}\}$ and $(p, q) \in E_i$ iff the length of the shortest path between p and q in G_0 is at most $2 \cdot x^i + x^{i-1}$ (where x is the parameter of the clustering algorithm). This construction can be easily achieved by each leader p by extending the update algorithm to include processors up to distance $x+1$ (instead of x) and adding the list of leaders at distance x to each processor p to p's tuple. We then apply the clustering algorithm on G_i, so that leaders will dominate processors up to distance x^{i+1} in G_0.

Note that the criteria for distance among leaders is expressed in terms of G_0 and the original x, namely; x^{i+1} for level i of the hierarchy.

Lemma 1. *Each resulting graph G_i is a connected graph.*

Next, we describe the *geographically affined* class of graphs such that the clustering algorithm and the overlay construction, applied on these graphs, produces an overlay graph of bounded degree. This class is implied by a typical deployment of sensor networks.

• *Geographically affined graphs.* In this class of graphs we wish to explore the relation between the Euclidean distance between processors and the length of the shortest path between them. This definition is similar to the embedding schemes presented in [23]. We first define the *geographically affined* class of graphs.

Definition 4.1. Let $G = (V, E)$ be a graph embedded in the Euclidean plane. For $p, q \in V$, define $\|(p, q)\|_2$ as the Euclidean distance between p and q, and $dist(p, q)$ as the number of hops in a shortest path from p to q in G. G is *Geographically affined* iff there exist a constant $c \leq 1$ such that $\forall p, q \in V : c \cdot dist(p, q) \leq \|(p, q)\|_2 \leq dist(p, q)$.
In [14], we show that each geographically affined graph has a bounded degree. Furthermore, we also show that the hierarchy construction algorithm presented above produces a bounded degree graph in each level of the hierarchy. The proof of the next Lemma appears in [14].

Lemma 2. *Let $G_0 = (V_0, E_0)$ be an Euclidean graph, such that G_0 is geographically affined. Each graph in the series $\{G_i\}_{i=0}^{\log n}$, resulting from the consecutive application of the clustering algorithm with parameter x^{i+1}, has a degree at most $\frac{16}{c^2} \cdot (2 \cdot x + 1)^2$.*

Self-organization properties. Next, we prove that our algorithms are self-organizing. Firstly, for the clustering algorithm, it is worthwhile noting that locality holds since the algorithm stabilizes within expected $O(\log n)$ rounds. Thus, we focus our discussion on *dynamic* changes of the communication graph — namely, on addition and removal of communication links. We wish to draw the readers' attention to the fact that addition (or removal) of processors can be modeled by the addition (or removal) of their communication links (which is a bounded number of operations). When we discuss addition of processors, we consider addition of processors in a predefined state or in an arbitrary state. We only consider topology changes after the algorithm has stabilized (otherwise, the global stabilization time applies).

Lemma 3. *Starting in a safe configuration of the clustering algorithm, if the update table of processor p has changed due to a channel (respectively, processor) addition or removal in configuration c_i and the channel (respectively, processor) is attached (a neighbor) to p, then within expected $O(x + \log f(x)) = O(1)$ rounds, a safe configuration is reached. Furthermore, for each processor q, such that $dist(p, q) > 2 \cdot x$, q will remain stable.*

We now consider the effects that channel additions have on the clustering algorithm. Let us assume that a new (bi-directional) channel, (p, q), is added between processors p and q. We argue that any stable processor distanced more than $2 \cdot x$ from either p or q will remain stable. Furthermore, within an expected constant number of rounds, the

algorithm will stabilize. This clearly follows from Lemma 3. Let us now assume that a channel (p, q) is removed. Let NL be the set of all processors, so that the removal of (p, q) leaves them leaderless or unstable. We argue that the constant number of processor in NL are at most at distance x from either p or q and that stable processors which are distanced farther than x will remain stable. Processor removal is easily reduced to the removal of all channels attached to this processor from the communication graph.

We also discuss additions and removals of processors. We argue that stable processors which are farther than $2 \cdot x$ from the removed/added processor will remain stable. This also clearly follows from Lemma 3.

Thus, our clustering algorithm is self-organizing, since the expected convergence time is $O(\log n) \in o(n)$ and the number of processors which change state due to a dynamic topology change is constant. In fact, when k changes occur approximately at the same time, the expected convergence time is $O(\log k)$ following the last change occurrence.

• *Application to hierarchy.* Let us examine a dynamic change at G_0. There are two processors, p and q, which are involved in the change ((p, q) was either added or removed). We first concentrate on p. From Lemma 3 we infer that only processors within a distance of $2 \cdot x + 1$ hops from p can be affected in G_0. The dynamic change can influence the state of leaders within this range, which can be regarded as a new dynamic change in G_1. The radius of the corresponding influenced region from p in G_1 is therefore $(2 \cdot x^2 + 2 \cdot x + 1) + (2 \cdot x + 1)$ around p in G_0. In a similar way, the radius of the influenced region from p in G_i is $2 \cdot x^i + 2 \cdot x^{i-1} + x^{i-2}+$ (the radius of influence in G_{i-1}). Overall, the area of effect around p in G_0 is less than $4 \cdot x^{i+2}$. Since G_0 is geographically affined, the Euclidean radius of such a circle is smaller than $4 \cdot x^{i+2}$. The minimal distance in G_0 between processor in G_i is at least x^i (when counting real edges, not virtual ones), since they are leaders in G_{i-1}. Again, since G_0 is geographically affined, the Euclidean distance between leaders is at least $c \cdot x^i$. Using simple geometric arguments (see [14]) it is evident that the number of processors affected at G_i because of p is at most $\frac{16 \cdot (4 \cdot x^{i+2})^2}{(x^i)^2} = 256 \cdot x^4 = O(1)$. Since we have to consider q as well, we double the total number of changes to have a total of $O(1)$ changes in each level.

To conclude, the hierarchy construction algorithm is self-organizing, since the expected stabilization time is $O(\log^2 n) \in o(n)$ and dynamic topology changes affect only $O(\log n) \in o(\log^2 n)$ processors. Similarly, when k changes occur approximately at the same time, the expected convergence time is $(O(\log^2 k)$ rounds following the last occurring change.

5 Conclusions

We have given a simple and intuitive definition of self-organization. Furthermore, we have displayed the relevance of self-stabilization with regards to self-organization. Our self-stabilizing and self-organizing snapshot algorithm implies sublinear time algorithms in the overlay network model for many core distributed tasks.

Self-stabilizing and self-organizing leader election. The hierarchy construction algorithm which is, by itself, a self-stabilizing and self-organizing algorithm, naturally

defines a leader for each subsystem. Thus, the topmost subsystem (which contains the entire system) also has a leader, which we define to be the output of the leader election algorithm. Hence, the output of the hierarchy construction algorithm can be used to define a self-stabilizing leader election algorithm which converges in $O(\log^2 n)$ expected number of rounds and handles topology changes gracefully in $O(\log n)$ rounds.

Self-stabilizing and self-organizing snapshots. Building on top of the hierarchy construction algorithm, we have presented in [14] a self-stabilizing snapshot scheme, where a global snapshot can be collected in $O(\log^2 n)$ rounds (in fact, if the hierarchy was previously defined, only $O(\log n)$ rounds are necessary).

Self-stabilizing converter. Our self-stabilizing and self-organizing snapshot algorithm implies a new efficient tool for converting distributed (reactive, or fixed output) algorithms to self-stabilizing algorithms in sublinear time; the leader of the system can take repeated snapshots and verify each snapshot for correctness. When a snapshot indicates an illegal state, a global reset procedure may be initiated, using the infrastructure created by the hierarchy definition algorithm, to reach a predefined (and safe) state.

Acknowledgments. Many thanks to Noga Alon and Boaz Patt-Shamir for helpful discussions.

References

[1] Afek, Y., and Dolev, S., "Local Stabilizer," *Journal of Parallel and Distributed Computing*, special issue on self-stabilizing distributed systems, Vol. 62, No. 5, pp. 745-765 (May 2002). Also in *Proc. of the 5th Israeli Symposium on Theory of Computing and Systems*, (ISTCS 1997), pp. 74-84, 1997.

[2] Anceaume, E., Defago, X., Gradinariu, M., and Roy, M., "Towards a theory of self-organization" *9th International Conference on Principels of Distributed Systems, OPODIS*, pp. 146-156, 2005.

[3] Bui A., Datta A.K., Petit F., and Villain V., " State-optimal snap-stabilizing PIF in tree networks ". *Proceedings of the Third Workshop on Self-Stabilizing Systems* 1999, pages 78-85.

[4] Burman, J., Kutten, S., Herman, T., Patt-Shamir, B. "Asynchronous and Fully Self-Stabilizing Time-Adaptive Majority Consensus" *9th International Conference on Principels of Distributed Systems, OPODIS*, 2005.

[5] Chandy, M., and Lamport, L., "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computing Systems*, 3(1):63-75, 1985.

[6] Tanenbaum, A., " Computer Networking, 4th edition", Prentice Hall, 2002.

[7] Cournier, A., Datta, A., Petit, F., and Villain, V,. "Enabling snap-stabilization," *Proc. of the 23rd International Conference on Distributed Computing Systems*, pages 12-19, 2003.

[8] Demirbas, M., Arora, A., Mittal, V., "FLOC: A fast local clustering service for wireless sensor networks". *Workshop on Dependability Issues in Wireless Ad Hoc Networks and Sensor Networks (DIWNAS/DSN)*, 2004.

[9] Dijkstra, E., W., "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, 17(11):643-644, 1974.

[10] Dolev, S., "Optimal Time Self-Stabilization in Uniform Dynamic Systems," *Parallel Processing Letters,* Vol. 8 No. 1, pages. 7-18, 1998.

[11] Dolev, S., *Self-stabilization*, MIT Press, 2000.

[12] Dolev, S., and Herman, T., "SuperStabilizing Protocols for Dynamic Distributed Systems". *Proc. of the 2nd Workshop on Self-Stabilizing Systems* May 1995. *Chicago Journal of Theoretical Computer Science*, 3(4) special issue on self-stabilization, 1997.

[13] Dolev, S., Kranakis, E., Krizanc, D., and Peleg, D., "Bubbles: Adaptive Routing Scheme for High-Speed Dynamic Networks," *SIAM Journal on Computing,* Vol. 29 No. 3, pages. 804-833, 1999. Also in *Proc. of the 27th ACM Symposium on Theory of Computing*, (STOC 1995) pages. 528-537, 1995.

[14] Dolev S., Tzachar N., "Colonies: Self-Stabilizing and Self-Organizing Distributed Algorithms", Technical report #06–01, Ben Gurion University of the Negev, October 2005, `http://www.cs.bgu.ac.il/~tzachar/papers/tech0601.pdf`.

[15] Felix C. Gärtner, Henning Pagnia, "Time-Efficient Self-Stabilizing Algorithms through Hierarchical Structures", in *Proc. to the Sixth Symposium on Self-Stabilizing Systems* Pages. 154-168, 2003

[16] Ghosh, S., Gupta, A., Herman, T., and Pemmaraju, S. "Fault-Containing Self-Stabilizing Algorithms" *PODC* 1996, pages 45–54.

[17] Henzinger, M., King, V., "Randomized Fully Dynamic Graph Algorithms with Polylogrithmic Time per Operation", *Journal of the ACM*, Vol. 46, No. 4, pp. 502-516, July 1999.

[18] Katz, S., and Perry, K., "Self-stabilizing extensions for message-passing systems" *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 91-101, 1990.

[19] Kirschenhofer P., Prodinger H., "A Result in Order Statistics Related to Probabilistic Counting", *Computing*, vol. 46 pages 15-27.

[20] Kutten, S., Peleg, D., "Tight Fault Locality", *Annual Symposium on Foundations of Computer Science (FOCS)* 1995.

[21] Kuhn, F., Moscibroda, T., Wattenhofer, R., "What cannot be computed locally!" *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, 2004, pages 300–309.

[22] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, 12(7):558-565, 1978.

[23] Linial N., London E., and Rabinovich Y. "The Geometry of Graphs and Some of its Algorithmic Applications". In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 577–591, October 1994.

[24] Luby M.,"a Simple Parallel Algorithm for the Maximal Independent Set Proble", *SIAM journal of Computing* vol 15(4), 1986, pages. 1036–1053

[25] Moscibroda, T., Wattenhofer, R., "Efficient Computation of Maximal Independent Sets in Unstructured Multi-Hop Radio Networks", *The 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, Fort Lauderdale, Florida, 2004.

[26] Plaxton, C. G., Rajaraman, R., and Richa, A. W. "Accessing nearby copies of replicated objects in a distributed environment". In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* 1997. ACM Press, New York, NY, pages. 311-320.

[27] Varghese, G., "Self-stabilization by counter flushing," *SIAM Journal on Computing*, 30(2):486-510, 2000, also in, *Symposium on Principles of Distributed Computing*, pages 244-253, 1994.

[28] Zhang, H., Arora, A., "GS^3: Scalable Self-configuration and Self-healing in Wireless Networks", *Symposium on Principles of Distributed Computing*, pages 58-67, 2002.

GLANCE: A Lightweight Querying Service for Wireless Sensor Networks

Murat Demirbas[1], Anish Arora[2], and Vinod Kulathumani[2]

[1] Computer Science & Engineering
University at Buffalo, SUNY Buffalo, NY, 14260
demirbas@cse.buffalo.edu
[2] Computer Science & Engineering
The Ohio State University Columbus, OH, 43210
{anish, vinodkri}@cse.ohio-state.edu

Abstract. Distance-sensitivity guarantee in querying is a highly desirable property in wireless sensor networks as it limits the cost of executing a "query" operation to be within a constant factor of the distance to the nearest node that contains an answer. However, such a tight guarantee may require building an infrastructure for efficient resolution of queries, and the cost of maintaining this infrastructure may be prohibitive. Here we show that it is possible to implement distance-sensitive querying in an efficient way by exploiting the geometry of the network. Our querying service *Glance* ensures that a "query" operation invoked within d distance of an event intercepts the event's "advertise" operation within $d * s$ distance, where s is a "stretch-factor" tunable by the user.

1 Introduction

A major application area for wireless sensor networks (WSNs) is environmental monitoring [1, 2, 3, 4, 5]. The grand vision for these applications is to scatter thousands of wireless sensor nodes across an area of interest upon which the nodes self-organize into a network and enable monitoring and querying of events in the area. An example application is a disaster evacuation scenario where the rescue workers query the network to learn about fire or chemical threats in the area.

There are two main modes of operation in most WSN monitoring applications. The first mode is "centralized monitoring and logging". For monitoring and logging purposes it is important to gather information about events in the network [6, 7]. This can be easily satisfied by enforcing events to exfiltrate data to a basestation that could forward the data to a monitoring and control center. In our disaster evacuation scenario, the control and command center needs to get data about events for logistical purposes, such as deciding how many rescue workers to send to each region and coordinating the rescue efforts. These data are also valuable for keeping logs and statistics of events.

The second mode of operation is "in-network querying" or "location-dependent querying". In the context of the evacuation scenario, the rescue workers in each region would need to query the network for nearby events, such as fire/chemical

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 244–259, 2006.

threats, and vital statistics from victims. It is inefficient and unscalable, for most cases, to force the queriers to learn about events only from the basestation, since it would compel a querier that is very close to an event to communicate all the way back to a basestation to learn about that event. The inefficiency of the scenario is amplified if the querier needs to get a stream of data from the event. Using long routes for forwarding data not only increases the latency but also depletes the battery power of the relaying nodes in the network quickly. Using the basestation for every query also leads to a communication bottleneck for the network. For these reasons it is important to be able to discover short (local) paths from queriers to nearby events.

The desirability of quick resolution of in-network queries via shortest possible paths is formalized by the distance-sensitivity property. Distance-sensitivity limits the cost of executing a query operation to be within a constant factor (we call this as the *stretch-factor*) of the distance to the nearest node that contains an answer. However, such a tight guarantee may require building an "in-network advertisement infrastructure" (such as a hierarchical partitioning of the network [8] or a network-wide advertisement tree [9, 10]) for efficient resolution of queries, and the cost of maintaining this infrastructure may be prohibitive. In fact many work on in-network querying [11,12,13] choose to avoid such a guarantee in favor of best-effort resolution of the queries.

Contributions of the paper. Here we show that it is possible to implement distance-sensitive querying in an efficient way–using minimal infrastructure– by exploiting the geometry of the WSN. Our main insight is to combine both modes of operation in WSN monitoring applications in a synergistic manner. As part of the data exfiltration mode, any interesting event detection is sent toward the basestation node, and so the basestation can act as a last resort for resolving an in-network query. As part of in-network querying mode, queries are also sent toward the direction of the basestation with the intention that the in-network advertisements of nearby events (if any) will intercept the query and answer it in a distance-sensitive manner, or else the query is answered at the basestation by default. It is at this point that using the geometry of the network comes handy. By using geometry, we determine the minimum area required for in-network advertisement for satisfying the distance-sensitivity requirement. More specifically, we observe that the local advertisements of events can safely ignore a majority of directions/regions while advertising and still satisfy a given distance-sensitivity requirement tightly.

As a result, we present a simple (using minimal infrastructure) and lightweight (cost efficient) distance-sensitive querying service, *Glance*. The distance-sensitivity of Glance, is easily tunable. Glance ensures that a query operation invoked within d distance of an event intercepts the event's advertisement information within $d * s$ distance, where s is a "stretch-factor" tunable by the user. By selecting appropriate values for s, the user can trade-off between query execution cost and advertisement cost.

Overview of Glance. Let C be a distinguished basestation node in the network. Let d_q be the Euclidean distance between a querying node q and C,

d_e the distance between an event e and C, and finally d the distance between q and e. We observe that for the cost of query operations there are two possible cases with respect to the angle z formed by locations of q, C, and e. Figure 1 illustrates these two cases, with respect to querying nodes q' and q''.

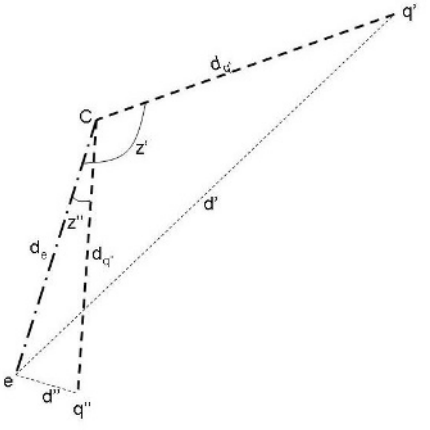


Fig. 1. Two cases with respect to z

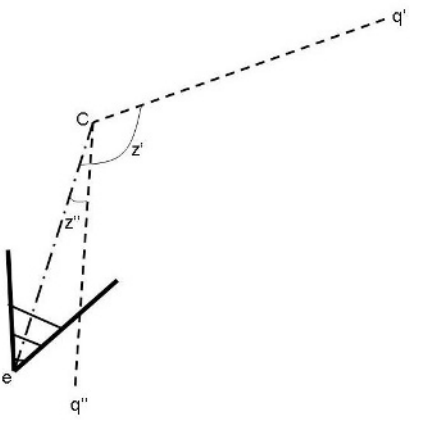


Fig. 2. Advertise operation for s=2

1. z is larger than a threshold: A large z implies that d is large relative to d_q and d_e. Thus, it is acceptable for the query to go to C to learn about the event, since the stretch-factor s can still be satisfied this way. For example, in Figure 1, z' is larger than the threshold angle and hence q' can still satisfy s by learning about e at C since $d_{q'} \leq d' * s$.
2. z is smaller than the threshold: A small z implies that d is small relative to d_q and d_e. Thus, it is unacceptable for the query to go to C, since this violates the stretch-factor property. For example, in Figure 1, z'' is smaller than the threshold angle and hence q'' cannot satisfy s by going to C since $d_{q''} > d'' * s$.

Our advertise operation in Glance (see Figure 2) seeks to optimize for the above two cases by combining both modes of operation in WSN monitoring applications:

- Data exfiltration to C proves useful in answering some in-network queries at C since that would still satisfy the stretch-factor for potential queriers with a large angle z' as in case 1 above.
- The advertise operation advertises the event in the network only along a cone boundary for some distance. The angle x for the advertisement cone is calculated based on the the stretch-factor s as $arcsin(1/s)$ (as described in Section 3.2). This cone-advertisement accounts for potential queriers q with a small angle z'', whose $d_q > d'' * s$.

The query operation is simply a *glance* to the direction of the basestation; it progresses as a straight path from the querying node toward C. Once an in-network advertisement for a matching event is found, the query operation stops forwarding the query any further and informs the querying node about the match by sending a reply. (In the worst case, the query reaches C and C sends a reply back to the querying node.) The querying node can then use the event location information included in the reply to learn more about the event.

By involving the basestation partially in answering of in-network queries and by exploiting geometry of the network, we observe that the advertise operation can constrain itself to a small region as in Figure 2 and still satisfy a given distance-sensitivity requirement tightly.

Outline of the paper. Next, we discuss related work. In Section 3 we present Glance. In Section 4, we analyze the performance of Glance. We conclude the paper with a discussion of future work in Section 5.

2 Related Work

Recently there has been much research on in-network querying. Early work includes adaptation of publish-subscribe tree structures from the Internet domain to the wireless ad hoc networks [14]. Although the basic ideas in publish-subscribe services may still be applicable for in-network querying problem in WSN, certain assumptions in the publish-subscribe model does not apply in WSN. For example, in contrast to the subscriptions that are long-lived, short-lived ad hoc queries is an important class of querying in WSN. These ad hoc queries may appear sporadically at any node in the network, as in our fire evacuation scenario. The event sources may be equally unpredictable in WSN, so it is unclear as to which nodes the subscription trees should be rooted at. Also typical network sizes considered in WSN are much larger than that of ad hoc network deployments and battery constraints are more severe in WSN, and hence scalability and inefficiency are a more critical concern for WSN querying services.

Rumor routing [11] provides a novel and tunable in-network querying mechanism without any need for localization information. In this approach, an event employs a set of advertising agents to do a random walk of the network creating paths that lead to the event. Querying node also sends out query agents which randomly traverse the network. Whenever a query agent encounters a path set up by an advertising agent with a matching interest, a route is established between the query and the event. The scheme is tunable in that for guaranteeing higher reliability it is possible to increase the number of agents sent from each event and query, however, rumor routing does not provide any distance-sensitivity guarantees or any deterministic guarantees for querying. Glance improves over rumor routing by providing a more structured approach to advertising and querying. Since both the advertise and query operations now target a common node, C, their meeting distance is shortened greatly compared to a random walk strategy. In addition, using the stretch-factor idea and the cone-advertisement, the meeting distance of the advertise and query are optimized. Glance also avoids

wasting energy by not advertising the event in the regions where meeting at C is already an acceptable solution for the query and event.

Combs and needles algorithm [9] maintains an advertisement infrastructure over the network for efficient resolution of in-network queries. More specifically, the event advertisement builds a network-wide routing structure that resembles a comb, and the query operation searches for an event using a structure resembling a needle. The paper shows that due to the shapes of these structures the event and query are guaranteed to intersect. By arranging the distance between the teeth of the comb structure, Combs&Needles tunes the minimum length for the needle structure to guarantee that query operation intersects the advertise operation. Combs&Needles protocol forces the user to fix the cost of querying to be a constant cost in advance, and compels the advertise operation to do as much work as necessary to guarantee the fixed cost for querying. In contrast, in Glance, the cost of querying is designed to be within a constant factor of the distance to the nearest event, not within a fixed constant cost per se. By allowing the cost of querying to increase linearly when there is no event nearby (of course within the constraints of distance-sensitivity), Glance reduces the cost for advertise operation significantly. Also by using a common node C to focus the dissemination of information and forwarding of the queries, Glance is able to construct a very lightweight structure for advertising.

A simple and lightweight solution to in-network querying problem is to use Geographic Hash Tables (GHT) [12], which store and retrieve information by using a geographic hash function on the type of the information. However, the basic GHT protocol is not distance-sensitive since it can hash the event information far away from the nearby event-query pair and thus violates the stretch-factor. In contrast to GHT protocol, Glance provides distance-sensitivity guarantees and also tunability of stretch-factors. The distance-sensitivity problem of GHT can be alleviated by using hierarchies: either by a structured replication at different levels of a hierarchical partitioning [12], or by using geographically bounded hash functions at increasingly higher levels of a hierarchical partitioning as employed in DIFS protocol [15]. Hierarchical clustering of the network (via a quadtree) is also employed by another in-network querying protocol, Geographic Location System (GLS) [16]. Hierarchical GHT and GLS protocols still cannot achieve distance-sensitivity for all query-event pairs due to the multi-level partitioning problem: In a hierarchical partitioning it is possible that a query-event pair nearby in the network might be arbitrarily far away in the hierarchy due to multi-level partitioning between them.

Stalk [17], a WSN tracking protocol for mobile objects, also uses a hierarchical partitioning, but to account for the multi-level partitioning effects a querying node performs lateral searches to neighboring clusterheads (in addition to its own clusterhead) at increasingly higher levels of the hierarchy to reach the event information in a distance-sensitive manner. Recently, Distance Sensitive Information Brokerage (DSIB) protocol [8] achieved distance-sensitivity in a hierarchically partitioned network by using a similar technique for querying of static events. Instead of adapting a pull-based approach and using lateral searches to

neighbors as in Stalk, DSIB adapts a push-based approach: an event advertises to neighboring clusterheads as well as its clusterhead at every level of the hierarchy. Accordingly, the responsibility of the query is decreased: querying node contacts immediate clusterheads at increasingly higher levels until it hits the event information.

3 Glance Algorithm

3.1 Model

We assume a dense, connected, and multihop WSN and the availability of localization information at the nodes. We use $dist(j, k)$ to denote the Euclidean distance between two nodes j and k. We assume an underlying geographic routing protocol, such as greedy perimeter stateless routing (GPSR) [18] or crossing link detection protocol (CLDP) [19], that achieves O(d) cost for communication over d distance.

We assume a distinguished basestation node C in the network. We denote the distance between a querying node q and C as d_q, and event e and C as d_e. We use d to refer to the distance between q and e. $z_{q,e}$ denotes the angle formed by location of q, C, and the location of e. We use a calculational proof notation [20] where a proof of $K \equiv M$ can be expressed as:

$$\begin{array}{ll} & K \\ \equiv & \{ \text{ reason why } K \equiv L \} \\ & L \\ \equiv & \{ \text{ reason why } L \equiv M \} \\ & M \end{array}$$

3.2 Details of the Glance Algorithm

Here we explain the cone-advertisement needed for the advertise operation in detail, and discuss how the advertise operation ensures distance-sensitivity for a given stretch-factor, s.

Areas where stretch-factor is readily satisfied. We mentioned in the Introduction that there are two possible cases for the cost of a query operation invoked at a node q, for an event e, with respect to the angle $z_{q,e}$. To account for the case where $z_{q,e}$ is less than the threshold angle x, the advertise operation needs to advertise on a cone boundary. For $z_{q,e}$ greater than x no advertising is required as the stretch-factor is readily satisfied even when q contacts C directly, incurring a d_q cost. In order to be able to determine the boundaries of the advertisement cone precisely, we first need to calculate the threshold angle x for a given stretch-factor s. Here we show how we calculate x by determining the areas for which stretch-factor is readily satisfied.

As a simple example, let's take $s = 1$. We calculate the region where the stretch-factor is readily satisfied by taking successively larger circles centered at e and C and intersecting them. Figure 3 illustrates this method. There the

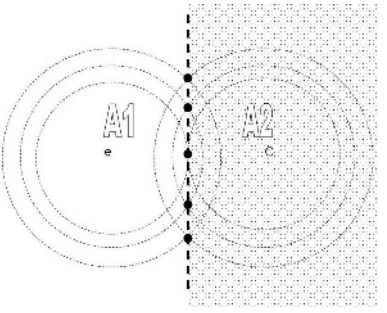


Fig. 3. Area where s=1 is readily satisfied

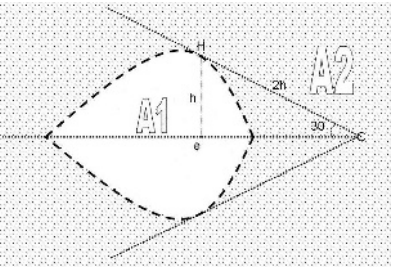


Fig. 4. Areas where s=2 is readily satisfied

dashed line consists of points obtained from intersecting circles with equal radii, r, $r \geq d_e/2$, centered at e and C. Thus, any point on the dashed line is equidistant to e and C. It follows that, any point in $A2$ is closer to C than it is to e, and hence, for any querying node in $A2$ stretch-factor $s = 1$ is readily satisfied by contacting C directly.

For $s > 1$, the same method is used for calculating the areas where stretch-factor is readily satisfied: we let a circle with radius r centered at e intersect with a circle with radius $s * r$ centered at C. Figure 4 shows an example for $s = 2$. Note that a circle centered at e with radius r intersects with the circle centered at C with radius $2r$ for $d_e/3 \leq r \leq d_e$. This is because for $r < d_e/3$ there is a gap of $d_e - 3r$ between the two circles, and for $r > d_e$ all the circles centered at e are subsumed by circles with radius $2 * r$ centered at C. Thus, the dashed line closes on itself and forms a bounded area $A1$. From our construction it follows that for $s = 2$ any point on the dashed line is twice as far away from C than it is from e. Also, for any point in $A2$ the distance to C is always less than twice as that to e. Hence, the stretch-factor $s = 2$ is readily satisfied for area $A2$. On the other hand, for area $A1$ stretch-factor may be violated, and cone-advertisement should account for the querying nodes in this region. From Figure 4 we observe that event e can safely ignore a majority of directions/regions advertising and still satisfy the given distance-sensitivity requirement tightly.

In Figure 4 consider a point H on the dashed line such that $\widehat{HeC}$ forms a right angle. Since any point on the dashed line is twice as far from C than it is from e, $|CH| = 2 * |eH|$, and hence $\widehat{eCH}$ is calculated as 30° from the HeC right triangle. Since $\widehat{HeC}$ is 90°, H determines the maximum angle between any intersection point and e with respect to C, so the threshold angle $x$for $s = 2$ is set as 30°. In general x is calculated as $arcsin(1/s)$, since $x = arcsin(|eH|/|CH|)$. For $s > 1$ (which we consider in this paper), we always have $90° > x > 0$. For $s = 1$ there is no feasible solution since $x = 90°$. For $s = 2$, $x = 30°$, and $s = 4$, $x = 14.5°$. So, as s increases the threshold angle decreases rapidly. The area $A1$ that the advertise operation has to account for is extremely small for $s = 4$.

The algorithm for query. Before we give the details of the advertise operation and prove its correctness, we present the query operation briefly. The query operation progresses as a straight path from the querying node toward C. For routing of the query toward C, GPSR is employed with the destination of the query packet set as C. During relaying of this query message hop-by-hop, if a node j with the advertisement information of a matching event is reached, j stops forwarding the query any further. (At worst the query will be answered at C, hence $j = C$ in that case.) To inform the querying node, whose address is included in the query message, about this matching event, j sends a reply to the querying node using the GPSR service. This reply contains advertised metadata about the event, such as its location, type, and time. By using the location information in the reply, the querying node can then contact the node that detected the event directly to learn more about the event.

For the cost of a query operation, we only include the communication cost of forwarding the query until it reaches a node j that has an answer to the query. We do not include the cost of j's reply to the query cost for the Glance protocol as well as for the other in-network querying protocols we consider in the analysis section (Section 4).

The algorithm for advertise. The advertise operation for $s = 2$ is depicted in Figure 5 with solid dark lines. Roughly speaking the advertisement is performed on the boundaries of a cone that is rooted at the event location and that widens toward C. The progress of the cone stops when the threshold angle is reached with respect to a straight line between the event and C, as there is no need to advertise outside the threshold angle. The idea behind this cone advertisement is to intercept any query that may be originating at $A1$ in a distance-sensitive manner. Note that without this cone advertisement the queries originating in $A1$ would go all the way to C violating the distance-sensitivity requirement. The angle for the cone advertisement for both the left-hand side cone boundary and the right-hand side cone boundary is selected to be equal to the threshold angle $x = arcsin(1/s)$. The reason behind this selection is to accommodate for varying threshold angles (those close to 90° as well as those close to 0°) using a uniform strategy for the advertisement. The user may want to define different stretch-factor requirements (which lead to varying threshold angles) with respect to the type (i.e., severity) of events. As we prove in Lemma1, selecting the angle for cone advertisement to be equal to the threshold angle satisfies the distance-sensitivity requirement for any stretch factor greater than 1.

Beside the advertisement on the cone boundary, there is a need for some lateral advertisements within the boundaries of the cone. These lateral advertisements are needed for intercepting any query originating within the cone boundaries in area $A1$. (Recall that those queries that are originating in $A1$ and outside the cone boundaries are intercepted by the advertisement on the cone boundary.) Consider a query within unit distance of e and that falls between e and C. By drawing the first lateral link at distance s from e, the stretch-factor is satisfied for this query. Moreover this first lateral link suffices for intercepting all queries within distance s of e inside the cone boundaries in a distance-sensitive manner.

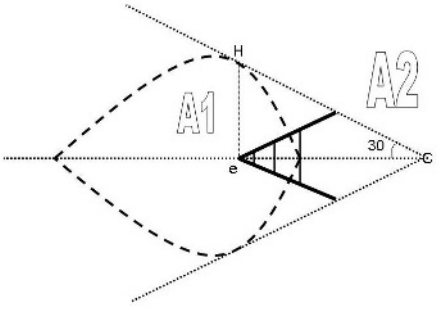


Fig. 5. Local advertisement for $s=2$

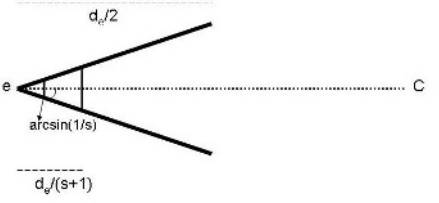


Fig. 6. Advertise operation

The second lateral link is drawn at distance s^2 and it handles queries within distance s–s^2 of e inside the cone boundaries. Proceeding in the same fashion other lateral links are drawn within area $A1$ inside the cone boundaries. The final lateral link is drawn at the boundary of $A1$, which is $d_e/(s+1)$ distance away from e. Figure 6 recaps the above discussion and shows the advertise operation for a given stretch-factor s.

For the implementation of the cone advertisement we exploit GPSR again. The event uses the angle of cone advertisement (defined as $\arcsin(1/s)$), its distance d_e from C, and its own coordinates to calculate the coordinates of the two endpoints of the cone and sends a "cone-boundary advertisement" message destined to each endpoint. While this message is being relayed hop-by-hop, each node it visits stores the metadata advertisement included in the message. Lateral link advertisement is performed similarly. The event calculates starting and ending points of lateral link advertisements, and sends a pair of "start lateral link" message to the calculated starting points on the eC line (i.e., 1, s, s^2, ...,$d_e/s+1$ away from e). The lateral link start points, when they receive these messages, repackages them into "lateral-link advertisement" messages, and send them to the endpoints precalculated by e. Each node relaying a lateral-link advertisement message stores the metadata about e included in the message.

Lemma 1. A query operation invoked in $A1$ within d distance of an event intercepts the event's advertise information within $d * s$ distance.

Proof: There are three cases. In the first case, the querying node q in $A1$ falls inside the boundaries of cone advertisement, whereas in the remaining two cases q is outside the cone advertisement boundaries. In case 2, the angle $\widehat{EQC}$ between the evader location, location of q, and that of C is less than 90° as depicted in Figure 7. And, in case 3 $\widehat{EQC}$ is greater than or equal to 90° as in Figure 8.

Case 1: In this case the querying node q falls within the cone boundaries. The lateral links advertisement within the cone satisfy the stretch-factor for these queries as discussed above.

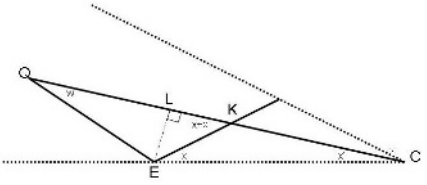


Fig. 7. Advertisement, case 2

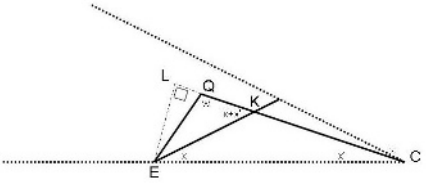


Fig. 8. Advertisement, case 3

Case 2 (see Figure 7): In order to prove that distance-sensitivity is satisfied for a query originating at Q, we need to show that the query is intercepted on its path from Q to C by the cone advertisement before the query travels more than $s * d$ distance. In other words, we need to prove that $|QK| < s|QE|$ in Figure 7.

$|QK| < s|QE|$
$\equiv$ { $|QK| = |QL| + |LK|$ and $|LK| = |LE|\cot(x + x')$}
$|QL| + |LE|\cot(x + x') < s|QE|$
$\equiv$ { $|QL| = |QE|\cos(w)$ and $|LE| = |QE|\sin(w)$}
$|QE|\cos(w) + |QE|\sin(w)\cot(x + x') < s|QE|$
$\equiv$ { $\sin(x) = 1/s$ (multiply both sides with $\sin(x)$) also eliminate $|QE|$ }
$\sin(x)\cos(w) + \sin(x)\sin(w)\cot(x + x') < 1$
$\equiv$ { Definition of $\cot(\alpha)$ }
$\sin(x)\cos(w) + \sin(x)\sin(w)\cos(x+x')/\sin(x+x') < 1$
$\equiv$ { Definition of $\cos(\alpha + \beta)$ and $\sin(\alpha + \beta)$}
$\sin(x)\cos(w) + \sin(x)\sin(w)(\cos(x)\cos(x') - \sin(x)\sin(x'))$
$/(\sin(x)\cos(x') + \cos(x)\sin(x')) < 1$
$\equiv$ { Arithmetic }
$\sin^2(x)\cos(w)\cos(x') + \sin(x)\cos(x)\cos(w)\sin(x') + \sin(x)\cos(x)\sin(w)\cos(x')$
$-\sin^2(x)\sin(w)\sin(x') < \sin(x + x')$
$\equiv$ { Arithmetic }
$\sin^2(x)(\cos(w)\cos(x') - \sin(w)\sin(x'))$
$+ \sin(x)\cos(x)(\sin(w)\cos(x') + \cos(w)\sin(x')) < \sin(x + x')$
$\equiv$ { Definition of $\cos(\alpha + \beta)$ and $\sin(\alpha + \beta)$}
$\sin^2(x).\cos(w + x') + \sin(x)\cos(x)\sin(w + x') < \sin(x + x')$
$\equiv$ { Arithmetic, definition of $\sin(\alpha + \beta)$}
$\sin(x)\sin(x + w + x') < \sin(x + x')$
$\equiv$ { Arithmetic }
$\sin(x + w + x') < \sin(x + x')/\sin(x)$

Note that $0 \leq x' \leq x \leq 90°$, also $x + x' + w < 180°$ as they are in a triangle. There are two cases.

Case A ($x + x' \leq 90°$): Then, $\sin(x + x')/\sin(x) > 1$ is satisfied due to property of *sine* for angles between $0° - 90°$. Since $\sin(\alpha) \leq 1$, for any α, we have $\sin(x + w + x') < \sin(x + x')/\sin(x)$.

Case B ($90° \leq x + x' < 180°$): Note that, $\sin(x + x')/\sin(x) > \sin(x + x')$, since $\sin(x) \leq 1$, for any x. Also, $\sin(x + x') > \sin(x + w + x')$, since $90° < x + w + x' < 180°$ and as angle increases sine decreases in that interval.

Case 3 (see Figure 8): Similar to Case 2, in order to prove that distance-sensitivity is satisfied, we need to prove here that $|QK| < s|QE|$ in Figure 8.

$|QK| = |LK| - |QL|$. Note that $|QL| = |QE| \cos(180 - w) = -\cos(w)|QE|$.

$|QK| < s|QE|$
$\equiv$ { $|QK| = |LK| - |QL|$}
$|LK| - |QL| < s|QE|$
$\equiv$ { $|QL| = |QE| \cos(180 - w) = -\cos(w)|QE|$}
$|LK| + |QE| \cos(w) < s|QE|$
$\equiv$ { $|LK| = |LE| \cot(x + x')$ and $|LE| = |QE| \sin(180 - w) = |QE| \sin(w)$}
$|QE| \sin(w) \cot(x + x') + |QE| \cos(w) < s|QE|$
$\equiv$ { Same inequality as in Case 1}
$\sin(x + w + x') < \sin(x + x')/\sin(x)$

Since $0 \leq x' \leq x \leq 90$ both subcases in Case 2 apply without modification. ◇

Theorem 1. A query operation invoked within d distance of an event intercepts the event's advertise information within $\mathbf{min}(d{*}s, d_q)$ distance, where d_q is the distance between the querying node and C.

Proof: There are two cases. If querying node is in $A1$, due to Lemma 1, the cost of querying is given as $d{*}s$. From the construction of $A1$, we have $d{*}s \leq d_q$ for any point q in $A1$, hence the querying cost = $\min(d{*}s, d_q)$ for this first case. If the querying node is in $A2$, then from construction $d_q < d{*}s$, and querying cost = $\min(d{*}s, d_q)$ is readily satisfied even in the worst case (when query goes d_q distance to C). ◇

4 Performance Evaluation

We analyze the cost of advertise and query operations as well as tradeoffs involved in these costs in Section 4.1. Then, in Section 4.2, we compare the performance of Glance with other in-network querying protocols in the literature.

4.1 Cost of Advertise and Query

From Figure 6, we calculate the cost of advertise operation as follows. Since we choose the angle for cone advertisement to be equal to the threshold angle $x = \arcsin(1/s)$, the cone meets the threshold angle halfway through d_e, and the length of the cone boundary is calculated as $(d_e/2)/\cos(x)$. Thus, the two cone boundaries induce $2 * (d_e/2)/\cos(x) = d_e/\cos(\arcsin(1/s))$ cost. We also need to account for the cost of lateral explorations inside the cone boundaries. Recall from Section 3.2 that lateral explorations are performed with exponentially increasing intervals between subsequent explorations and for up to distance $d_e/(s+1)$ away from the event. The height of a lateral link is obtained by multiplying the distance between the lateral link and the event with $\tan(x)$, and doubling the result. Thus, the cost for the lateral explorations is calculated as $2 * \sum_{i=0}^{\log_s(d_e/(s+1))} s^i * \tan(x)$. Hence, the overall cost for advertisement comes up to $d_e/\cos(\arcsin(1/s)) + 2 * \sum_{i=0}^{\log_s(d_e/(s+1))} s^i * \tan(\arcsin(1/s))$.

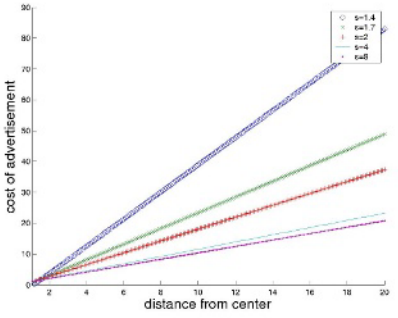


Fig. 9. Advertisement cost vs. d_e

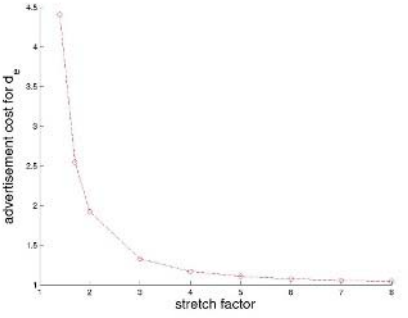


Fig. 10. Advertisement cost vs. s

We can simplify this term by using the formula for sum of geometric series as: $\frac{d_e}{\cos(\arcsin(1/s))} + \frac{2*\tan(\arcsin(1/s))*(\frac{s*d_e}{s+1}-1)}{(s-1)}$.

As seen in Figure 9, the advertisement cost for an event e increases linearly with respect to the distance d_e of the event from the basestation C. The slope of this linear increase is determined by s. For s very close to 1, that is, for x close to 90°, the cost of advertisement can get high as $\cos(x)$ decreases, $(s-1)$ gets close to 0, and $\tan(x)$ increases. For example, for $s=1.15$ (i.e., $x=60°$), the cost of advertise is around $14 * d_e$. However, as s increases, the cost of advertise drops significantly fast. For example, $s=1.4$ (i.e., $x=45°$), the cost of advertise is less than $4.3 * d_e$, and for $s=2$ (i.e., $x=30°$) the cost is less than $1.92 * d_e$, and for $s=4$ (i.e., $x=14.5°$) the cost is around $1.16 * d_e$. Figure 10 illustrates the relation between s and the cost of advertisement.

As proved in Theorem 1, the worst case cost of querying is $\min(d*s, d_q)$.

Analysis of tradeoffs in stretch-factor selection. In Glance, by tuning the stretch-factor s, the user specifies the level of distance sensitivity desired for answering queries. The cost of querying is directly proportional to s: by selecting small values for s, the cost is reduced. On the other hand, Figure 10 shows that the cost of advertise operation is inversely proportional to s: by selecting larger values for s, the cost of advertising is reduced. Thus, by tuning the value of s appropriately, the user can achieve tradeoffs between the cost of querying and advertising.

The user can define different stretch-factor requirements with respect to the type (i.e., importance) of events. One way to approach this tradeoff issue is to take a query-centric view. The user can first decide the highest tolerable stretch-factor in the application (e.g., based on real-time requirements of the query), and use this for the value of s. However, if there are no query-centric hard deadlines for the stretch-factor or the constraints for energy and communication efficiency dominates the design decisions, then it is possible to take an advertisement-centric approach. Here the user can first decide on the desired communication cost for advertising an event and then reverse engineer s using this cost. For

example after deciding that $1.92 * d_e$ is suitable for advertising cost, the user may select $s = 2$ and $x = 30°$ accordingly.

Analysis of scalability with respect to multiple events and queries. In the presence of multiple events and queries, Glance can be easily extended to use geographic hashing [12] and multiple basestations to improve load-balancing among basestations and achieve scalability with respect to the number of events and queries. The idea here is to partition events to multiple basestations based on the types of events so that network contention and bottlenecks are avoided at a basestation. Moreover, the user can define different stretch-factor requirements with respect to the type of events.

4.2 Performance Comparison

In our comparisons we add to the cost of our advertise operation in Glance an extra d_e cost: the cost of data exfiltration to C which is in fact a part of the centralized monitoring mode operations. We do this so as not to put the other protocols at a disadvantage.

Comparison with GHT and hierarchical GHT. GHT hashes an event and a query for the event to a common broker. For comparing GHT and Glance, we assume that this broker is C located in the middle of the network. The cost of storing an event at C corresponds to the cost of exfiltration of information to C in Glance. Hence, the cone advertisement in Glance remains as an extra cost over that of the advertise operation in GHT. For example, for $s = 2$, Glance pays an extra $1.92 * d_e$ cost for cone advertisement. The query operation in GHT, on the other hand, is more costly than that of Glance, since GHT does not satisfy distance-sensitivity. For a square network with diameter D, the average cost of querying (averaged over distance d_q of all querying nodes to C) in GHT is calculated as $D/3$. Note that this corresponds to the cost of going to C for the resolution of all queries. However, since Glance is distance-sensitive, queries are resolved in $\min(d*s, d_q)$ distance, where d is the distance to the nearest event, and a typical value for s is 2. Hence, the average cost of querying in Glance is lower than that of GHT. Especially, for a setup where the number of queriers are more than that of events, Glance would be more energy efficient than GHT, because the queries are answered locally. Also, Glance is preferable to GHT when there is a hard deadline (such as a real-time requirement) for the query operation.

Comparison with Stalk and DSIB. In Stalk, the advertisement cost of an event is calculated as $2 * d_e$. With this cost for the advertise operation, it is possible to achieve a stretch-factor of 4 for the querying cost in Glance. In contrast, the stretch factor in Stalk is given as $4 * w$, where w is the number of neighbors at any level of the hierarchy and ranges between 6 and 12. Thus, the cost of querying in Stalk is several times more than that calculated for Glance. However, we note that Stalk can achieve distance-sensitive tracking of mobile objects, whereas Glance does not address the mobility of events.

In DSIB, to achieve distance-sensitivity an event advertises to w, $6 \leq w \leq 12$ neighboring clusterheads as well as its clusterhead at every level of the hierarchy [8]. The cost of this advertisement is calculated as $2 * w * D$, where D is the

diameter of the network.[1] In turn DSIB proves a stretch factor of 4 for the query operation. For $s = 4$ the advertisement cost in Glance corresponds to $2.16 * d_e$, including the cost of data exfiltration to C. Since d_e is the distance between the event and C, it is guaranteed to be less than D. Hence, Glance is able to achieve the same cost for querying as DSIB with around 1/9th of the cost required for advertisement in DSIB. On the other hand, an advantage of DSIB is that it can be implemented using the discrete centric hierarchy method [21] in the absence of localization information.

5 Concluding Remarks

In this paper we showed that it is possible to devise a simple and lightweight solution for distance-sensitive in-network querying in WSN by exploiting basic geometry concepts. Our main insight was to use the basestation node in an opportunistic manner for answering of some in-network queries. The knowledge that all queries target the basestation by default, combined with the geometry of the network, was useful in determining the minimum area required for in-network advertisements to satisfy a given distance-sensitivity requirement. We observed that in-network advertisements can safely ignore a majority of directions/regions and focus their advertisement to a small cone to be able to satisfy a given distance-sensitivity requirement.

As a result, we presented a simple and lightweight querying service *Glance*, that ensures that a query invoked within d distance of an event intercepts the event's advertisement within $d * s$ distance, where s is a "stretch-factor" tunable by the user. The user may define different stretch-factor requirements (which lead to varying angles for cone advertisement) with respect to the type (i.e., severity) of events. By selecting appropriate values for s it is possible to achieve trade-offs between query execution cost and advertisement cost. Glance is also robust with respect to node failures and holes in the network.

It is possible to avoid the need for localization in the Glance protocol. The idea here is to use an approximation for the direction to the basestation node C. In this scheme, in the initialization phase C starts a one-time flood that annotates each node in the network with its hopcount from C and creates a spanning tree rooted at C. To send the query as a straight line to C, it is enough to route the message to the parent node along a branch in this tree. Since it is infeasible to draw cone borders as in Figure 5 in the absence of localization, our scheme approximates that with occasional lateral exploration inside the cone by visiting the nodes with same hopcount at predefined distances from the event. Due to reasons of space we relegate the details of this discussion to our technical report [22].

As a broader research direction, we will further investigate the adaptation of geometric ideas and techniques for devising distributed network algorithms. We note from our previous experience [17, 23, 24] that when the problem domain

[1] This cost is equal to the sum of $2^0 w + 2^1 w + \ldots + 2^{\log(D)} w$, as the number of levels in the hierarchy is $\log D$.

is constrained to geometric networks it is possible to devise simpler and more efficient algorithms than those designed for arbitrary graph topologies. With the recent advances in directional antenna technology and the availability of directional communication in WSN, we believe that the application of geometric ideas to the distributed WSN domain may yield new research opportunities.

References

1. Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., Anderson, J.: Wireless sensor networks for habitat monitoring. In: WSNA. (2002)
2. Szewczyk, R., Mainwaring, A., Polastre, J., Culler, D.: An analysis of a large scale habitat monitoring application. In: Sensys. (2004)
3. Tolle, G., Polastre, J., Szewczyk, R., Turner, N., Tu, K., Buonadonna, P., Burgess, S., Gay, D., Hong, W., Dawson, T., Culler, D.: A macroscope in the redwoods. In: SenSys. (2005)
4. Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., Mittal, V., Cao, H., Demirbas, M., Gouda, M., Choi, Y.R., Herman, T., Kulkarni, S.S., Arumugam, U., Nesterenko, M., Vora, A., Miyashita, M.: A line in the sand: A wireless sensor network for target detection, classification, and tracking. Computer Networks (Elsevier) **46**(5) (2004) 605–634
5. Arora, A., et. al.: Exscal: Elements of an extreme scale wireless sensor network. Int. Conf. on Embedded and Real-Time Computing Systems and Applications (2005)
6. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: Tinydb: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. **30**(1) (2005) 122–173
7. Yao, Y., Gehrke, J.E.: Query processing in sensor networks. Conference on Innovative Data Systems Research (CIDR) (2003)
8. Funke, S., Guibas, L.J., Nguyen, A., Wang, Y.: Distance-sensitive routing and information brokerage in sensor networks. In: DCOSS. (2006)
9. Liu, X., Huang, Q., Zhang, Y.: Combs, needles, haystacks: balancing push and pull for discovery in large-scale sensor networks. In: SenSys. (2004) 122–133
10. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Trans. Netw. **11**(1) (2003) 2–16
11. Braginsky, D., Estrin, D.: Rumor routing algorthim for sensor networks. In: WSNA. (2002) 22–31
12. Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., Shenker, S.: Ght: a geographic hash table for data-centric storage. In: WSNA. (2002) 78–87
13. Shenker, S., Ratnasamy, S., Karp, B., Govindan, R., Estrin, D.: Data-centric storage in sensornets. SIGCOMM Comput. Commun. Rev. **33**(1) (2003) 137–142
14. Huang, Y., Garcia-Molina, H.: Publish/subscribe tree construction in wireless ad-hoc networks. In: Proceedings of the 4th International Conference on Mobile Data Management. (2003) 122–140
15. Greenstein, B., Estrin, D., Govindan, R., Ratnasamy, S., Shenker, S.: Difs: A distributed index for features in sensor networks. Int. Workshop on Sensor Network Protocols and Applications (2003)
16. Li, J., Jannotti, J., Couto, D.S.J.D., Karger, D.R., Morris, R.: A scalable location service for geographic ad hoc routing. In: MobiCom. (2000) 120–130

17. Demirbas, M., Arora, A., Nolte, T., Lynch, N.: A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. 8th International Conference on Principles of Distributed Systems (OPODIS) (2004) 299–315
18. Karp, B., Kung, H.T.: GPSR: greedy perimeter stateless routing for wireless networks. In: MobiCom. (2000) 243–254
19. Kim, Y.J., Govindan, R., Karp, B., Shenker, S.: Geographic routing made practical. Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (2005)
20. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
21. Gao, J., Guibas, L.J., Nguyen, A.: Deformable spanners and applications. In: Symposium on Computational Geometry (SCG). (2004) 190–199
22. Demirbas, M., Arora, A.: Glance: A lightweight querying service for wireless sensor networks. Technical Report 2005-24, University at Buffalo, SUNY (2005)
23. Kulathumani, V., Arora, A., Demirbas, M., Sridharan, M.: Trail: A distance sensitive network protocol for distributed object tracking. Technical Report OSU-CISRC-7/06-TR67, The Ohio State University (2006)
24. Demirbas, M., Ferhatosmanoglu, H.: Peer-to-peer spatial queries in sensor networks. The Third IEEE Int. Conf. on Peer-to-Peer Computing (2003) 32–39

On Many-to-Many Communication in Packet Radio Networks

Bogdan S. Chlebus[1,⋆], Dariusz R. Kowalski[2], and Tomasz Radzik[3]

[1] Department of Computer Science and Engineering, University of Colorado at Denver and Health Sciences Center, Denver, CO 80217, USA
Bogdan.Chlebus@cudenver.edu
[2] Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
darek@csc.liv.ac.uk
[3] Department of Computer Science, King's College London, London WC2R 2LS, UK
Tomasz.Radzik@kcl.ac.uk

Abstract. Radio networks model wireless data communication when bandwidth is limited to one wave frequency. The key restriction of such networks is mutual interference of packets arriving simultaneously to a node. The many-to-many (m2m) communication primitive involves p participant nodes of a distance at most d between any pair of them, from among n nodes in the network, and the task is to have all participants get to know all input messages. We consider three cases of the m2m communication problem. In the *ad-hoc* case, each participant knows only its name and the values of n, p and d. In the *partially centralized* case, each participant knows the topology of the network and the values of p and d, but does not know the names of other participants. In the *centralized* case each participant knows the topology of the network and the names of all the participants. For the centralized m2m problem, we give deterministic protocols, for both undirected and directed networks, working in $O(d+p)$ time, which is provably optimal. For the partially centralized m2m problem, we give a randomized protocol for undirected networks working in $O((d+p+\log^2 n)\log p)$ time with high probability (whp), and we show that any deterministic protocol requires $\Omega(p\log_{n/p} n+d)$ time. For the ad-hoc m2m problem, we develop a randomized protocol for undirected networks that works in $O((d+\log p)\log^2 n+p\log p)$ time whp. We show two lower bounds for the ad-hoc m2m problem. One states that any m2m *deterministic* protocol requires $\Omega(n\log_{n/d+1} n)$ time when $n-p=\Omega(n)$ and $d>1$; $\Omega(n)$ time when $n-p=o(n)$; and $\Omega(p\log_{n/p} n)$ time when $d=1$. The other lower bound states that any m2m *randomized* protocol requires $\Omega(p+d\log(n/d+1)+\log^2 n)$ expected time.

1 Introduction

Radio networks model wireless communication when the bandwidth consists of one wave frequency. Packets arriving simultaneously to a node interfere with one another. To have the notion of simultaneity meaningful, radio networks are typically assumed to be fully synchronous, in that an execution of a communication protocol is structured as a sequence of global rounds. A packet transmitted to node v is *heard* by v if the

⋆ The work of this author is supported by NSF Grant 0310503.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 260–274, 2006.

transmission is received by v in its correct form rather than garbled from interference with other transmissions. Radio networks are characterized by the following properties:

(i) A node can transmit at most one packet at a round.
(ii) All the out-neighbors of a transmitting node receive the packet in the same round when it was transmitted.
(iii) The recipient of a packet can hear it when the packet is the only one delivered at the round.

A natural algorithmic goal for radio networks is to implement useful communication primitives. *Broadcast* is an algorithmic task to disseminate a message originated at some source node to all the remaining nodes. *Gossiping* is an algorithmic task in which each node v has its input message, called *rumor*, and the goal is to have all nodes get to know all rumors. Gossiping is a special case of the general *many-to-many communication (m2m)* primitive, which involves a subset of nodes as *participants* and is specified as follows: each participant has its input message (rumor) and the goal is to have all participants get to know all participant rumors. Participants are the only nodes initially active, while the remaining nodes may join in with the purpose to help in forwarding packets.

We use letter n to denote the number of nodes in a network, letter p to denote the number of participant nodes, and letter d to denote the maximum distance between a pair of participants. Nodes are assigned unique names from the set $[n] = \{1, \ldots, n\}$. Each node knows its own name and the values of n, p and d, in the sense that they can be used as a part of code run by the node.

Cases of many-to-many communication problem. We distinguish between "algorithms" and "protocols" in the following sense. The nodes of a network execute a *protocol* to perform a communication task at hand. A protocol is always executed by the nodes concurrently, and in this sense it is *a priori* distributed. The protocol is found earlier by a sequential *algorithm*. A protocol obtained as the output of algorithm $\mathcal{A}$ on an input network G is said to be *explicit* if the algorithm $\mathcal{A}$ runs in time that is polynomial in the size of G. All communication protocols that we develop are explicit.

When a complete specification of a network is given as an input to a sequential algorithm, then the communication problem is said to be *centralized*. A communication problem is *ad-hoc* when the code of a protocol run by a node is the same for all networks of the same size and may include the size of the network and the unique name of the node as a part of code run by the node. We consider three variants of the m2m problem, which are defined by the information that nodes initially have about the network and participants.

Centralized m2m problem: each participant knows the topology of the network and the names of all the participants.
Partially centralized m2m problem: each participant knows the topology of the network, numbers p and d and its own status of being among the participants, but it does *not* know the name of any other participant.
Ad-hoc m2m problem: all participants have only the minimum amount of knowledge we assume, that is, they know the values of n, p and d.

All participant nodes are initialized with (1) a sequential algorithm they are to perform in order to find the m2m protocol and (2) the round when the m2m protocol is to start. For a node v of the network that is not a participant, we assume that initially v knows only its name. Such a node v remains dormant until it receives a message from its in-neighbor, which brings sufficient information allowing v to join the execution.

To improve on the message complexity, it is advantageous to combine as many input messages as it is feasible into one transmitted packet. If there is no bound on the number of input messages that can be combined into one packet, then the model is of *combined* input messages. The opposite model, which requires a separate transmission to forward one input message, is of *separate* input messages. Many-to-many communication has either the flavor of routing, in the model of separate messages, or that of gossiping, in the model of combined messages. We work with the model of combined input messages. Both models have been studied in the literature. Bar-Yehuda, Israeli and Itai [3] considered multiple instances of point-to-point communication and broadcast in the model of separate messages. They developed randomized protocol for undirected networks. The problem of m2m communication was abstracted for radio networks by Gąsieniec, Kranakis, Pelc and Xin [13], who studied deterministic protocols for undirected networks in the model of combined messages, in the case that we call partially-centralized in this paper.

We say that an event *holds with high probability*, denoted whp, if it holds with probability of at least $1 - p^{-c}$, for some constant $c > 0$ and for sufficiently large p. We say that a randomized protocol is *correct*, if it completes the specified task with high probability.

Our results. We now overview our contributions in more detail.

I. Centralized m2m problem: We give deterministic protocols, for both undirected and directed networks, working in $O(d + p)$ time, which is provably optimal.

Such protocols are generalizations of gossiping in $O(n)$ time; a centralized $O(n)$ time broadcast protocol for undirected networks given in [6] can be interpreted as a gossiping protocol in the model of combined messages. Our protocols have transmission structure resembling that of the protocols used in [14] as subroutines in an $O(n \operatorname{polylog} n)$-time gossiping protocol for undirected networks.

II. Partially centralized m2m problem: We give a randomized protocol for undirected networks working in $O((d+p+\log^2 n)\log p)$ time whp. We show that any randomized protocol requires $\Omega(p+d)$ expected time, and that any deterministic protocol requires $\Omega(p \log_{n/p} n + d)$ time.

The problem of m2m communication in the partially-centralized case was considered by Gąsieniec, Kranakis, Pelc, and Xin [13], who developed an explicit *deterministic* protocol for undirected networks working in $O(d \log^2 n + p \log^3 n)$ time.

III. Ad-hoc m2m problem: We develop a randomized protocol for undirected networks that works in $O((d+\log p)\log^2 n + p \log p)$ time whp. We show two lower bounds. One states that any *deterministic* protocol requires $\Omega(n \log_{n/d+1} n)$ time, for any p such that $n - p = \Omega(n)$ and $d > 1$; when $n - p = o(n)$, then the time is $\Omega(n)$, and when $d = 1$, then the time is $\Omega(p \log_{n/p} n)$. It follows that randomization

helps in ad-hoc m2m communication in undirected networks, when both $1 < d = o(n/(\log n \log\log n))$ and $p = o(n/\log n)$ hold. The other lower bound states that any *randomized* protocol requires $\Omega(p + d\log(n/d+1) + \log^2 n)$ time, for sufficiently large d.

Previous work. The model of radio communication was introduced by Chlamtac and Kutten [4]. Subsequent work considered various communication problems and kinds of protocols. The most popular categorizations distinguish (1) centralized problems from ad-hoc ones, (2) deterministic protocols from randomized ones, and (3) explicit protocols from existential ones. Let the maximum distance between any pair of nodes in a (strongly) connected graph be denoted by D, while Δ is the maximum in-degree in a network.

We start a review of the most relevant previous work with centralized problems. Chlamtac and Kutten [4] showed that the problem to find an optimal broadcast schedule for a given network is NP-complete. A centralized explicit broadcast protocol for *directed* networks working in $O(D\log^2(n/D))$ time was given by Chlamtac and Weinstein [5]. Clementi *et al.* [8] developed an explicit protocol working in $O(D\log\Delta\log(n/D))$ time, and recently Kowalski and Pelc [23] have shown an explicit protocol working in $O(D+\log^2 n)$ time. This last protocol is asymptotically optimal because Alon, Bar-Noy, Linial and Peleg [1] proved that nondeterministic broadcast in networks with $D = \Theta(1)$ requires $\Omega(\log^2 n)$ time. Centralized gossiping with combined messages in undirected networks was considered by Gąsieniec, Potapov and Xin [17], and next by Gąsieniec, Peleg and Xin [15] who gave a deterministic explicit protocol of time performance $O(D+\Delta\log n)$. Centralized gossiping with separate messages was studied by Gąsieniec and Potapov [16]. The primitive of many-to-many communication for radio networks was first abstracted by Gąsieniec, Kranakis, Pelc, and Xin [13], who considered it only in the scenario that we call the partially-centralized m2m problem in this paper. They developed an explicit protocol for a given *undirected* network in the model of combined messages that terminates in $O(d\log^2 n + p\log^3 n)$ time.

Next we review the work that has been done on ad-hoc communication problems. The unique names of nodes are often assumed to be from a range larger than the size of the network. If this is the case, then a leader is not automatically identifiable, and neither the round-robin paradigm to learn the neighbors nor the token-traversal approach are immediately applicable in undirected networks.

The first ad-hoc randomized broadcast protocols of sub-quadratic expected time performance were given by Bar-Yehuda, Goldreich and Itai [2] and Bar-Yehuda, Israeli and Itai [3]. The first ad-hoc deterministic explicit broadcast protocols with sub-quadratic time performance were given by Chlebus *et al.* [6]. They were improved to time performance $O(n^{3/2})$ by Chlebus, Gąsieniec, Östlin and Robson [7], and then by Indyk [20] to $O(n^{1+o(1)})$. See [2,10,22,24] for discussion of the impact of knowledge and randomness on efficiency of ad-hoc radio broadcast. Bar-Yehuda, Israeli and Itai [3] considered point-to-point communication and broadcasts in the model of separate messages. They developed an ad-hoc randomized Las Vegas protocol, which is preceded by preprocessing performed in $O((n+D\log n)\log\Delta)$ expected time; then a set of k point-to-point transmissions can be performed in $O((k+D)\log\Delta)$ expected time, while k broadcasts can be achieved in $O((k+D)\log\Delta\log n)$ expected time.

The fastest explicit deterministic ad-hoc broadcast protocol for *undirected* networks of time performance $O(n \log n)$ was given by Kowalski and Pelc [22]. The fastest existential deterministic ad-hoc broadcast protocol running in $O(n \log^2 D)$ time was given by Czumaj and Rytter in [10]. The lower bound $\Omega(n \log n / \log(n/D))$ on time of deterministic broadcast protocols for *undirected* networks was proved by Kowalski and Pelc [22]. The fastest known ad-hoc deterministic broadcast for *directed* networks of time performance $O(n \log^2 D)$ was given by Czumaj and Rytter [10]; this protocol is not explicit. The lower bound $\Omega(n \log D)$ for deterministic protocols in *directed* networks was proved by Clementi, Monti and Silvestri [9].

The fastest known ad-hoc randomized broadcast for both directed and undirected networks are of the same expected time complexity. The best randomized protocols working in $O(D \log(n/D) + \log^2 n)$ expected time were developed independently by Czumaj and Rytter [10] and by Kowalski and Pelc [22]. The lower bound $\Omega(D \log(n/D))$ on the expected time of broadcast protocols was shown by Kushilevitz and Mansour in [25]. This bound combined with lower bound $\Omega(\log^2 n)$ from [1] shows that $D \log(n/D) + \log^2 n$ is the asymptotically optimal time complexity of a randomized broadcast.

The fastest ad-hoc deterministic gossiping for directed networks of time complexity $O(n^{4/3} \text{polylog } n)$ was given by Gąsieniec, Radzik and Xin [18]. The fastest ad-hoc randomized gossiping for directed networks of time performance $O(n \log^2 n)$ was developed in [10]. For undirected networks, an $O(n)$-time ad-hoc deterministic gossiping was shown in [6].

In this paper we are interested in protocols whose complexity bounds are given in terms of asymptotic notation. An alternative approach, as exemplified by the work of Elkin and Kortsarz [11,12], is to develop algorithms producing explicit protocols optimized for time measured in terms of approximation ratios.

Model of radio networks. A network is modeled as a graph $G = (V, E)$, which is directed in general, but may be restricted to be symmetric, that is, undirected. Nodes in V represent processing units. A directed arc $x \to y$ in E represents a possibility for node x to send a packet directly to y. For any such an arc, node x is an in-neighbor of y and node y is an out-neighbor of x. Graphs are assumed to be sufficiently connected. For the problem of broadcasting, we assume that every node is reachable from the source. For the gossiping and m2m problems, a graph is assumed to be connected, if it is undirected, and strongly connected, if it is directed. The *distance from node x to node y* is the length of the shortest path from x to y; these paths are directed when the graph under consideration is directed.

When more than one packets are received at a round at a node v, then they interfere with each other and none can be heard. Such interference is called a *collision*. There is no mechanism to notify a sender if collisions occurred at the receiving nodes. When a recipient of packets can detect a collision, in the sense of distinguishing it from the situation when no packet was received, then we say that the network is *with collision detection*. We work with the weaker model without a collision-detection mechanism.

A direct multicast to neighbors performed by a node of a radio network is called a *transmission*. When a number of nodes perform a multicast in a round, then all of them are also called a *transmission*. Phrases "node performs a transmission" and "node belongs to a transmission" are considered equivalent.

A sequence of transmissions is called a *schedule*. A sequential algorithm solving an instance of a centralized communication task returns a set of transmissions. They form a schedule by being sequenced in the same order in which they were returned, unless stipulated otherwise.

Logarithms are to the base 2, unless stated otherwise.

2 A Template of Many-to-Many Protocols for Undirected Networks

We give in Figure 1 a generic m2m protocol for undirected networks, which consists of 4 stages. The protocol resorts to other generic protocols, which are called ELECTION, FIND-BFS, CONVERGECAST-BFS and BROADCAST. All our m2m protocols for undirected networks are instances of this generic m2m protocol. We give the specification of the subroutines, and then show that the generic m2m protocol is correct in that it completes the m2m task whp.

ELECTION : after an execution of this protocol, exactly one participant has the status of a leader.

FIND-BFS(v) : there is a BFS tree rooted at the node v such that after an execution of this routine each node w of distance at most d maintains the path from w to v along edges of this tree.

CONVERGECAST-BFS(v) : there is a BFS tree rooted at the node v such that when this routine is called, then each node w of distance at most d maintains the path from w to v along edges of this tree; during an execution of this routine the node v receives the rumors of all participant nodes.

BROADCAST(v) : node v broadcasts to all the nodes of distance at most d from v.

The notations $T_E(n,d,p)$, $T_{BFS}(n,d,p)$, $T_C(n,d,p)$, and $T_B(n,d,p)$ denote the number of rounds that protocols ELECTION, FIND-BFS, CONVERGECAST-BFS, and BROADCAST take, respectively. Algorithms producing these protocols need to know the

---

`stage 1:` each participant node runs ELECTION routine, during exactly $T_E(n,d,p)$ rounds

`stage 2:` each leader v initiates FIND-BFS(v) routine to broadcast its name and paths in some BFS tree to all the nodes of distance at most d from v, during exactly $T_{BFS}(n,d,p)$ rounds

`stage 3:` each participant node w which received only one name v in stage 2 transfers its name, rumor and flag v to node v by executing CONVERGECAST-BFS(v) protocol with the received leader v as the sink, during exactly $T_C(n,d,p)$ rounds

`stage 4:` each leader v which received all p rumors of participant nodes with flag v during process CONVERGECAST-BFS(v) in stage 3 broadcasts its name and all the collected rumors of participant nodes using BROADCAST(v) protocol to all the nodes of distance at most d from v; it is done during exactly $T_B(n,d,p)$ rounds

---

Fig. 1. Generic m2m protocol M2M-GENERIC for undirected network

lengths of executions of these protocols, that is, the numbers $T_E(n,d,p)$, $T_{BFS}(n,d,p)$, $T_C(n,d,p)$, and $T_B(n,d,p)$. The nodes executing (an instance of) protocol M2M-GENERIC will therefore know when one stage ends and the next one begins. Protocol M2M-GENERIC works in $T_E(n,d,p)+T_{BFS}(n,d,p)+T_C(n,d,p)+T_B(n,d,p)$ time. Its correctness is clear when all the subroutines are deterministic: exactly one participant v is elected in stage 1 as a leader, v receives the input messages from the other participants in stage 3, and all input messages are broadcast from v to the other participants in stage 4.

Theorem 1. *If protocols* ELECTION, FIND-BFS, CONVERGECAST-BFS, *and* BROADCAST *used in protocol* M2M-GENERIC *complete their tasks whp, then protocol* M2M-GENERIC *completes the m2m task whp.*

Observe that the broadcasting in stage 4 of protocol M2M-GENERIC is executed only in the case when there is exactly one leader v, and it has received all input messages during stage 3. In this case all the participants receive the broadcast message from v in stage 4 by arranging broadcasting along the edges of the BFS tree used by CONVERGECAST-BFS(v) in stage 3. With this modification, the participant nodes can keep iterating protocol M2M-GENERIC until they receive a message in stage 4. The obtained protocol would always complete the m2m task and with high probability would work in $O(T_E(n,d,p)+T_{BFS}(n,d,p)+T_C(n,d,p)+T_B(n,d,p))$ time.

Our instances of protocol M2M-GENERIC for the centralized, partially centralized, and ad-hoc m2m problems will be named M2M-C, M2M-PC, and M2M-AH, respectively. For the names of instances of protocols ELECTION, FIND-BFS, CONVERGECAST-BFS, and BROADCAST, we will use prefixes C, PC, and AH to indicate which case is considered (for example, AH-CONVERGECAST-BFS).

3 Centralized Protocols

In this section we consider the centralized m2m problem. The participants know initially the network and the names of all participants. All routines, and therefore the resulting protocols as well, are deterministic. We consider first the undirected networks.

Routines C-ELECTION *and* C-FIND-BFS(v). The first routine always selects the participant with the lowest name as the leader, and the second routine selects the lexicographically first BFS tree rooted at v. Both routines work in $O(1)$ time since they do not require any transmissions.

Protocol C-CONVERGECAST-BFS(v). The algorithm computing protocol C-CONVERGECAST-BFS(v) is given in Figure 2. It returns this protocol as a sequence of transmissions P, where each transmission is a set of one or more pairs of nodes (z,x). If transmission i in P contains pairs $(z_1,x_1),\ldots,(z_q,x_q)$, then in round i of the execution of the protocol all the nodes $z_1,\ldots,z_q$ transmit together. The second node in a pair (z,x) included in a transmission is a node which correctly receives the message from z, and is used only to simplify description and analysis. The algorithm takes a minimal BFS tree T rooted at v and containing all participants, and considers the nodes in T level by level, starting from the last level $h \leq d$. Let L_k denote the set of nodes in

```
set L̄_h = L_h;
for k = h − 1 downto 0 do
    { L̄_{k+1} is the set of nodes in L_{k+1} which should transmit to L_k }
    set L̄_k to the set of participants which belong to L_k;
    for each x ∈ L_k with at least 2 neighbors in L̄_{k+1} do
        for each neighbor z of x in L̄_{k+1} do
            append {(z,x)} to P as the next transmission; remove z from L̄_{k+1};
        add x to L̄_k;
    let z_1, z_2, ..., z_q be all the nodes still in L̄_{k+1};
    let x_i be an arbitrary neighbor of z_i in L_k;
    append {(z_1,x_1), (z_2,x_2), ..., (z_q,x_q)} to P as the next transmission;
    add x_1, x_2, ..., x_q to L̄_k;
return P.
```

Fig. 2. Algorithm for finding protocol C-CONVERGECAST-BFS(v)

T at distance k from v, where $L_0 = \{v\}$. In iteration k, for $k = h-1, h-2, \ldots, 0$, the algorithm schedules transmissions to pass messages from L_{k+1} to L_k.

The iteration consists of two parts. First the algorithm finds a node x in L_k that has at least two neighbors in the set $\bar{L}_{k+1}$ of the nodes in L_{k+1} that should transmit but have not had a transmission scheduled yet. For each neighbor z of x in $\bar{L}_{k+1}$, a transmission $\{(z,x)\}$ is scheduled. In the second part of the iteration, for all the nodes $z_1, z_2, \ldots, z_q$ still remaining in $\bar{L}_{k+1}$, a single transmission $\{(z_1,x_1), (z_2,x_2), \ldots, (z_q,x_q)\}$ is scheduled, where x_i is an arbitrary neighbor of z_i in L_k. It can be shown that the network cannot have any "cross edges" $\{z_i, x_j\}$, $i \neq j$, so that messages from nodes z_i's can be correctly sent to nodes x_i's in a single round. Moreover, messages from nodes x_i can be sent to nodes z_i in a single round as well (this property will be used in the broadcasting protocol).

Lemma 1. *The algorithm in Figure 2 returns a correct $O(p+d)$-round protocol* C-CONVERGECAST-BFS(v).

Proof. (Sketch) The correctness of the computed protocol is based on the following invariant of the main "for" loop.

> For each $k = h-1, h-2, \ldots, 0$, at the beginning of iteration k, for each participant w which belongs to $\bigcup_{j=k+1}^{h} L_j$, either w belongs to $\bar{L}_{k+1}$ or the sequence P contains some transmissions $t_1 < t_2 < \cdots < t_q$ such that transmission t_i includes a pair (y_i, y_{i+1}), where $y_1 = w$, and $y_{q+1} \in \bar{L}_{k+1}$.

This invariant can be proven by induction. The other component of the proof of correctness is the property that for any two pairs (z',x') and (z'',x'') which are added to the same transmission in P, there are no cross edges in the network. This can be shown by analyzing one iteration and observing that at the end of the loop "for each $x \in L_k$", each node in L_k can have only one neighbor in $\bar{L}_{k+1}$.

To show the bound on the number of rounds in the computed protocol P, let H be the *transmission tree* of P. The edges of H are the pairs (z,x) contained in P, and H is

rooted at v. Tree H is a BFS tree rooted at v and the leaves are participant nodes, but H may not be the same as T. The number of transmissions in P that contain more than one pair (z,x) is at most $h \leq d$: at most one such transmission per one iteration of the main loop. Each transmission in P which contains only one pair (z,x) contributes 1 to the degree of node x in H, and this degree must be at least 2. The sum of the degrees of the nodes in H with degree at least 2 is at most twice the number of leaves in H. Thus the number of transmissions in P is $O(d+p)$. □

Protocol C-BROADCAST(v). It is sufficient to take the reverse of protocol C-CONVERGECAST-BFS(v), because of the property of this protocol that if two pairs of nodes (z',x') and (z'',x'') are included in one transmission, then the network does not contain any of the cross edges $\{z',x''\}$ and $\{z'',x'\}$.

For directed networks, protocols C-ELECTION, C-FIND-BFS, and C-CONVERGECAST-BFS are analogous, while protocol C-BROADCAST is obtained from protocol C-CONVERGECAST-BFS for the network with all edges reversed.

Theorem 2. *Both versions of protocol M2M-C, one for undirected networks and one for directed networks, are correct and terminate in $O(d+p)$ rounds.*

Lemma 2. *For any positive integers p and d, there is a specific undirected network $G(p,d)$ of p participants and with the maximum distance d between a pair of participants, such that any m2m protocol for undirected networks, possibly randomized, requires time $\Omega(p+d)$ when executed on $G(p,d)$.*

Proof. Let the network $G(p,d)$ be a path of length d connecting a star of $p-1$ participants at one end, with one participant at the other end of the path. The center of the star will learn all input messages eventually, because it lies on the unique path connecting any pair of distinct participants. It takes $p-1$ rounds for the rumors of the star-connected nodes to be learned by the center of the star, because each of them has to be heard separately. It takes $d-1$ rounds for the center of the star to hear the input message of the node at the other end of the long path, because of the distance. □

4 Ad-Hoc Protocol for Undirected Networks

In this section we show how to instantiate the generic protocol M2M-GENERIC for the ad-hoc case. It is sufficient to show how to implement the subroutines. We start with protocol AH-BROADCAST(v) since it will be used later as a sub-routine in protocols AH-ELECTION and AH-CONVERGECAST-BFS(v).

Randomized ad-hoc broadcast routine AH-BROADCAST(v). We use an explicit randomized broadcasting protocol given in [10,22].

Fact 1 ([10,22]). *There is an explicit randomized broadcasting protocol* AH-BROADCAST *which completes broadcast to all the nodes of distance at most d from the source in $T_B(n,d,p) = O(d\log(n/d)+\log^2 n)$ time with probability $1-1/n$, for sufficiently large n.*

Election routine AH-ELECTION. Protocol AH-ELECTION proceeds through $6\log p$ phases. Each phase takes exactly $T_B(n,d,p)$ rounds. The whole routine AH-ELECTION takes $T_E(n,d,p) = O(T_B(n,d,p)\log p)$ rounds. Every node has a certain node distinguished as its *preferred leader*, which can be modified in the course of the protocol. Eventually all these local variables stabilize to the same value with a large probability. In the very beginning, each participant has this value initialized to itself. Just before a phase starts, each participant decides if it is to be a *promoter* for this phase. A participant decides on 'yes' with probability of $1/p$, all these decisions are independent over participants and phases. A promoter v for a phase executes AH-BROADCAST(v) in the phase, with itself as the source. Nodes that are not promoters join in the execution of AH-BROADCAST(v) as soon as they hear a transmission in the execution of AH-BROADCAST(v). A node joins at most one instance of AH-BROADCAST(v) in a phase. A promoter broadcasts its preferred leader. A node that joins an instance of AH-BROADCAST(v) after hearing some message m, resets its preferred leader to the one propagated by m. When the protocol terminates, then each participant considers its current preferred leader to be the leader.

Lemma 3. *The probability that there is exactly one participant as the preferred leader among all participants, when protocol* AH-ELECTION *terminates, is at least* $1-1/p$. *The time complexity is*

$$T_E(n,d,p) = O(T_B(n,d,p)\log p) = O((d\log(n/d)+\log^2 n)\log p)\ .$$

Proof. There is exactly one promoter during a phase with probability of

$$p\cdot\frac{1}{p}\cdot\left(1-\frac{1}{p}\right)^{p-1} \geq \frac{p}{p-1}\cdot 4^{-1} \geq \frac{1}{4}\ .$$

The broadcast of this unique promoter is completed in $T_B(n,d,p)$ time with probability of at least $1-1/n \geq 1/2$ for $n \geq 2$. Therefore with probability of at least $1/4\cdot 1/2 = 1/8$ a phase ends with exactly one preferred leader. The probability that none among $6\log p$ phases achieves this is at most $(1-1/8)^{6\log p}$, which is smaller than $1/p$. The time complexity follows directly from the design of the protocol AH-ELECTION and bound $T_B(n,d,p) = O(d\log(n/d)+\log^2 n)$ for broadcast time in this model as in Fact 1. □

Building BFS tree: routine AH-FIND-BFS(v). Suppose there is a node v designated as a leader. Let $G(v,d)$ be the subgraph of G induced by the nodes of distance at most d from v. The problem is to determine some BFS tree T of $G(v,d)$ rooted at v and make each node of $G(v,d)$ know the path from the root v to itself in T. We develop a randomized protocol AH-FIND-BFS(v) to solve this problem. The protocol is related to a randomized broadcasting protocol given in [2], however that protocol does not construct a BFS tree.

Protocol AH-FIND-BFS proceeds through $d\log n$ phases. Phase i, for $1 \leq i \leq d\log n$, is given in Figure 3. A local instance of variable X stored by node w is denoted by X_w. At the start of protocol AH-FIND-BFS, the local instance of $\texttt{active}_v$ of root v is set to 0, while the remaining nodes have their local instances of this variable set to $-\infty$. A node w is said to be *active* when $\texttt{active}_w \neq -\infty$. All random decisions on transmissions

initialize $\texttt{path}_w$ to the empty string

execute $24\log n$ times

for $k = 0$ through $k = \log n$ do
- if $\texttt{active}_w = i-1$ then transmit $\texttt{path}_w$ with probability 2^{-k} else attempt to hear a message
- if active$= -\infty$ and transmission α heard from node x then
  - set $\texttt{active}_w = i$
  - set $\texttt{path}_w$ to α with x appended at the end

Fig. 3. Phase i of protocol AH-FIND-BFS(v) Code for node w in $G(v,d)$

in phases are independent. A broadcasted packet carries the sequence of nodes visited by its predecessors. If a node w_2 is activated by a packet carrying such a path α sent from some node w_1, then w_2 appends w_1 to path α and includes the obtained extended path in packets it transmits.

Lemma 4. *Protocol* AH-FIND-BFS *makes every node of $G(v,d)$ active after at most $O(d\log^2 n)$ rounds with probability of at least $1-1/n$.*

Proof. During phase i each node that has at least one neighbor activated in phase $i-1$ hears a packet with probability of at least

$$1-(1-1/8)^{24\log n} \geq 1-1/n^3\,,$$

since $(7/8)^6 \leq 1/2$. It follows that each node of distance i from the root receives a packet successfully in phase i, but not before phase i, which follows directly from the specification of the protocol, with probability of at least $1-n_i\cdot i\cdot 1/n^3$, where n_i denotes the number of nodes of distance i to the root. Consequently, each node of distance at most d from the root receives a packet successfully in a phase whose number is equal to its distance, with probability of at least

$$1-\sum_{i\leq d} n_i i/n^3 \geq 1-1/n\,.$$

Note that if such an event holds, then each node receives a shortest path from the root when activated. Additionally, this path is an extension of the path from the root to the node from which it receives a first (activation) message, and which is also a predecessor of this node in the path from the root. It follows that these shortest paths constitute a BFS tree on nodes at distance at most d from the root. □

Convergecast along BFS tree: routine AH-CONVERGECAST-BFS(v). We need to solve the following problem. There is a BFS tree of a size at least p and of a depth at most d with $p-1$ distinguished nodes (participants) holding rumors. The tree is rooted at the leader v of the network, who is the remaining participant. Actually any node in the network of distance at most d from the leader is in the tree as a node. The goal is to have the leader get to know all these rumors. The BFS tree is not assumed

to be known by the nodes, but each node in the tree knows its path to the root. Denote the path from node w to the leader, as known by w, by $s_w = \langle s_w(0), s_w(1), \ldots, s_w(\delta_w)\rangle$, where δ_w is the distance from w to the leader v, $s_w(0) = w$ and $s_w(\delta_w)$ is the leader.

Protocol AH-CONVERGECAST-BFS(v) consists of $24p\log p + d$ steps. Each participant w other than the leader, for each $i = 1, 2, \ldots, 8p\log p$, launches a packet for the leader in step $(3i-2)+(d-\delta_w)$ with probability $1/(p-1)$. The packet contains the rumor of w and sequence s_w. If the packet launched by node w in step t is in node $s_w(j)$ at the beginning of step $t+j$, then node $s_w(j)$ transmits it in step $t+j$ to the next node $s_w(j+1)$. This packet either reaches the leader in step $t+\delta_w-1$, or is lost somewhere on the way due to a collision.

Lemma 5. *Protocol* AH-CONVERGECAST-BFS *solves the convergecast problem in* $O(d+p\log p)$ *time with probability of at least* $1-p^{-1}$.

Proof. A packet P launched by a participant w in step $(3i-2)+(d-\delta_w)$ reaches the leader if no other participant u launches its packet in step $(3i-2)+(d-\delta_u)$. The distance between packet P and a packet P' launched by u in step $(3k-2)+(d-\delta_u)$ for $k \neq i$, is always at least 3, so these two packets cannot collide. Thus the probability that a participant w launches a packet in step $(3i-2)+(d-\delta_w)$ and this packet reaches the leader is at least

$$\frac{1}{p-1}\cdot\left(1-\frac{1}{p-1}\right)^{p-2} \geq \frac{1}{4p}\,.$$

Hence the probability that there is a participant which has not succeeded sending any of its packets to the leader is at most

$$p\cdot\left(1-\frac{1}{4p}\right)^{8p\log p} \leq \frac{1}{p}.$$

□

Combining the bounds on the running time from Fact 1 and Lemmas 3, 4, 5 with Theorem 1 yields the following result.

Theorem 3. *Protocol* M2M-AH *works in* $O((d+\log p)\log^2 n + p\log p)$ *time whp.*

To show the efficiency of our algorithm, we prove lower bounds for randomized and deterministic protocols.

Theorem 4. *Any m2m randomized protocol requires* $\Omega(p + d\log(n/d+1) + \log^2 n)$ *expected time, for sufficiently large d.*

Proof. (Sketch) Part $\Omega(p)$ follows directly from Lemma 2. For part $\Omega(\log^2 n)$ we use the lower bound of $\Omega(\log^2 n)$ for broadcasting time given in [1]. The analysis given in [1] implies that there is a network H_n with n nodes and radius 2 such that for any randomized broadcasting protocol running on this network, there is a node which requires time $\Omega(\log^2 n)$ to receive a source message with constant probability. Call this node *long-waiting*. Assume that the number of participants p is at most $n/2$. Take two copies H' and H'' of $H_{(n-p)/2}$, and attach half of the participants to the source node in H' and the other half to the source node in H''. Networks H' and H'' are joined by one edge. Consider now any randomized m2m protocol in the obtained network. This protocol

defines (partial) broadcasting protocols in H' and H'', and gives a long-waiting node w' in H' and a long-waiting node w'' in H''. If H' and H'' are joined by edge (w', w''), then with constant probability the randomized m2m protocol runs in $\Omega(\log^2 n)$ time.

To show the $\Omega(d \log(n/d+1))$ term in the lower bound claimed in the lemma, we use the $\Omega(D \log(n/D))$ lower bound shown in [25] on expected running time of ad-hoc randomized broadcasting protocols for networks with D layers (there are D layers and each node is connected with all the nodes in the preceding and the succeeding layers). This lower bound holds for any randomized protocol, where we are interested in the expected time of the first successful transmission to the last layer of the layered network. We can generalized this lower bound to m2m randomized protocols in a similar way as above by taking two suitable layered networks. □

Theorem 5. *Any m2m ad-hoc deterministic protocol requires $\Omega(n \log_{n/d+1} n)$ time, for any parameter p such that $n - p = \Omega(n)$ and $d > 1$; when $n - p = o(n)$, then the lower bound is $\Omega(n)$, and when $d = 1$, then the lower bound is $\Omega(p \log_{n/p} n)$.*

Proof. First consider $n - p = \Omega(n)$ and $d > 1$. The lower bound $\Omega(n \log_{n/d+1} n)$ on time of deterministic m2m protocols with $p \geq 2$ can be shown by adapting the technique given in [22] in case of broadcasting protocols. More precisely, for any ad-hoc deterministic broadcasting protocol there exists an undirected layered-like graph of diameter D, where consecutive layers are organized as a line such that if the source node in the first layer broadcasts then $\Omega(n \log_{n/D+1} n)$ rounds are needed to propagate the message to any node in the last layer. Similarly as in the proof of the lower bound in Theorem 4, we take two such networks and join them together.

If $n - p = o(n)$, then Lemma 2 implies the $\Omega(n)$ lower bound on any m2m protocol. For the case $d = 1$ the lower bound is $\Omega(p \log_{n/p} n)$ [19]. □

5 Partially Centralized Protocol for Undirected Networks

We show how to instantiate the generic protocol M2M-GENERIC in the model when the participants know the topology but initially do not know other participants. As in the previous section, we show how to implement the routines.

Broadcast routine PC-BROADCAST(v). We use a centralized deterministic broadcast protocol given in [23]. In this protocol node v computes the broadcast schedule to all the nodes of distance at most d from him, and then initiates this schedule. Every message sent during the broadcast contains a schedule pre-computed by v.

Fact 2 ([23]). *There is an explicit centralized deterministic broadcasting protocol* PC-BROADCAST(v) *which completes broadcast to all the nodes of distance at most d from the source v in $T_B(n,d,p) = O(d + \log^2 n)$ time.*

Routine PC-ELECTION. It is sufficient to consider the procedure AH-ELECTION and change the AH-BROADCAST(v) into PC-BROADCAST(v). Since $T_E(n,d,p) = O(T_B(n,d,p) \log p)$ as in Lemma 3, we have the following property.

Lemma 6. *Routine* PC-ELECTION *is correct and takes $O((d + \log^2 n) \log p)$ time whp.*

Routine PC-FIND-BFS(v). Since the nodes know the topology, the participants compute locally, in a single round, the same BFS tree (for example, the lexicographically first BFS tree).

Lemma 7. *Routine* PC-FIND-BFS(v) *is correct and takes* $O(1)$ *time.*

Routine PC-CONVERGECAST-BFS(v) is routine AH-CONVERGECAST-BFS(v). Combining Theorem 1 with Fact 2 and Lemmas 5, 6 and 7, we obtain the following result.

Theorem 6. *Protocol* M2M-PC *works in* $O((d+p+\log^2 n)\log p)$ *time whp.*

Finally, the following lower bounds follow from Lemma 2 and from the lower bound of $\Omega(p\log_{n/p} n)$ on deterministic protocols for the *p-selection problem* proven in [19].

Theorem 7. *Consider the partially-centralized model. Any m2m randomized protocol performs the expected* $\Omega(p+d)$ *number of rounds, and any m2m deterministic protocol performs* $\Omega(p\log_{n/p} n+d)$ *rounds.*

References

1. N. Alon, A. Bar-Noy, N. Linial, and D. Peleg, A lower bound for radio broadcast, *Journal of Computer and System Sciences*, 43 (1991) 290 - 298.
2. R. Bar-Yehuda, O. Goldreich, and A. Itai, On the time complexity of broadcast in radio networks: an exponential gap between determinism and randomization, *Journal of Computer and System Sciences*, 45 (1992) 104 - 126.
3. R. Bar-Yehuda, A. Israeli, and A. Itai, Multiple communication in multihop radio networks, *SIAM Journal on Computing*, 22 (1993) 875 - 887.
4. I. Chlamtac, and S. Kutten, On broadcasting in radio networks - problem analysis and protocol design, *IEEE Transactions on Communications*, 33 (1985) 1240 - 1246.
5. I. Chlamtac, and O. Weinstein, The wave expansion approach to broadcasting in multihop radio networks, *IEEE Transactions on Communications*, 39 (1991) 426 - 433.
6. B.S. Chlebus, L. Gąsieniec, A. Gibbons, A. Pelc, and W. Rytter, Deterministic broadcasting in unknown radio networks, *Distributed Computing*, 15 (2002) 27 - 38.
7. B.S. Chlebus, L. Gąsieniec, A. Östlin, and J.M. Robson, Deterministic radio broadcasting, in *Proc., 27th Colloquium on Automata, Languages and Programming (ICALP),* 2000, LNCS 1853, pp. 717-728.
8. A.E.F. Clementi, P. Crescenzi, A. Monti, P. Penna, and R. Silvestri, On computing ad-hoc selective families, in *Proc., 5th International Workshop on Randomization and Approximation Techniques in Computer Science (APPROX-RANDOM)*, 2001, LNCS 2129, pp. 211 - 222.
9. A.E.F. Clementi, A. Monti, and R. Silvestri, Distributed broadcast in radio networks of unknown topology, *Theoretical Computer Science* 302 (2003) 337 - 364.
10. A. Czumaj, and W. Rytter, Broadcasting algorithms in radio networks with unknown topology, in *Proc., 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2003, pp. 492 - 501.
11. M. Elkin, and G. Kortsarz, A logarithmic lower bound for radio broadcast, *Journal of Algorithms,* 52 (2004) 8 - 25.
12. M. Elkin, and G. Kortsarz, Polylogarithmic additive inapproximability of the radio broadcast problem, in *Proc., 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX-RANDOM)*, 2004, LNCS 3122, pp. 105 - 116.

13. L. Gąsieniec, E. Kranakis, A. Pelc, and Q. Xin, Deterministic M2M multicast in radio networks, in *Proc.,* 31*th Colloquium on Automata, Languages and Programming (ICALP),* 2004, LNCS 3142, pp. 670 - 682.
14. L. Gąsieniec, A. Pagourtzis, and I. Potapov, Deterministic communication in radio networks with large labels, in *Proc.,* 10*th European Symposium on Algorithms, (ESA)*, 2002, LNCS 2461, pp. 512 - 524.
15. L. Gąsieniec, D. Peleg, and Q. Xin, Faster communication in known topology radio networks, in *Proc.,* 24*th ACM Symposium on Principles of Distributed Computing (PODC)*, 2005, pp. 129 - 137.
16. L. Gąsieniec, and I. Potapov, Gossiping with unit messages in known radio networks, in *Proc.,* 2*nd IFIP International Conference on Theoretical Computer Science (TCS)*, 2002, pp. 193 - 205.
17. L. Gąsieniec, I. Potapov, and Q. Xin, Time efficient gossiping in known radio networks, in *Proc.,* 11*th Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2004, LNCS 3104, pp. 173 - 184.
18. L. Gąsieniec, T. Radzik, and Q. Xin, Faster deterministic gossiping in directed ad-hoc radio networks, in *Proc.,* 9*th Scandinavian Workshop on Algorithm Theory (SWAT)*, 2004, LNCS 3111, pp. 397 - 407.
19. A.G. Greenberg, and S. Winograd, A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels, *Journal of the ACM,* 32 (1985) 589 - 596.
20. P. Indyk, Explicit constructions of selectors and related combinatorial structures, with applications, in *Proc.,* 13*th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002, pp. 697 - 704.
21. J. Komlós, and A.G. Greenberg, An asymptotically nonadaptive algorithm for conflict resolution in multiple-access channels, *IEEE Transactions on Information Theory,* 31 (1985) 303 - 306.
22. D.R. Kowalski, and A. Pelc, Broadcasting in undirected ad hoc radio networks, *Distributed Computing,* 18(1) (2005) 43 - 57.
23. D.R. Kowalski, and A. Pelc, Optimal deterministic broadcasting in known topology radio networks, *Distributed Computing,* to appear.
24. D.R. Kowalski, and A. Pelc, Time of deterministic broadcasting in radio networks with local knowledge, *SIAM Journal on Computing*, 33 (2004) 870 - 891.
25. E. Kushilevitz, and Y. Mansour, An $\Omega(D\log(N/D))$ lower bound for broadcast in radio networks, *SIAM Journal on Computing*, 27 (1998) 702 - 712.

Robust Random Number Generation for Peer-to-Peer Systems

Baruch Awerbuch[1,⋆] and Christian Scheideler[2]

[1] Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA
baruch@cs.jhu.edu.
[2] Institute for Computer Science, TU Munich, 85748 Garching, Germany
scheideler@in.tum.de

Abstract. We consider the problem of designing an efficient and robust distributed random number generator for peer-to-peer systems that is easy to implement and works even if all communication channels are public. A robust random number generator is crucial for avoiding adversarial join-leave attacks on peer-to-peer overlay networks. We show that our new generator together with a light-weight rule recently proposed in [4] for keeping peers well-distributed can keep various structured overlay networks in a robust state even under a constant fraction of adversarial peers.

1 Introduction

Due to their many applications, peer-to-peer systems have recently received a lot of attention both inside and outside of the research community. Most of the structured peer-to-peer systems are based on two influential papers: a paper by Plaxton et al. on locality-preserving data management in distributed environments [20] and a paper by Karger et al. on consistent hashing and web caching [13]. The consistent hashing approach is a very simple and elegant approach that assigns to each peer a (pseudo-)random point in the $[0, 1)$-interval. Based on this approach, various local-control rules have been proposed to decide how to interconnect the peers so that they form a well-connected network with good routing properties that is easy to maintain (see, e.g., [18] for a general framework).

In open peer-to-peer systems, the presence of adversarial peers cannot be avoided. Hence, not only scalability but also robustness against adversarial behavior is an important issue. The key to scalability and robustness for peer-to-peer networks based on the consistent hashing approach is to keep the honest and adversarial peers well-distributed in the $[0, 1)$-interval. However, just assigning a random or pseudo-random point to each new peer (by using some random number generator or cryptographic hash function) does not suffice to keep the honest and adversarial peers well-spread [2]. People in the peer-to-peer community are aware of this problem [8,9] and various solutions have been proposed that may help alleviating it in practice [6,7,19,26,27,29] but until recently no

⋆ Partially supported by NSF grants CCF 0515080, ANIR-0240551, CCR-0311795, and CNS-0617883.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 275–289, 2006.

mechanism was known that can *provably* keep the peers in a well-distributed state without sacrificing the openness of the system.

Various light-weight perturbation rules that can keep the honest and adversarial peers well-distributed have recently been proposed in [4,10,25]. These rules do not need to be able to distinguish between the honest and adversarial peers, but a crucial prerequisite for them to work is a robust distributed random number generator. This random number generator has to work correctly in a system without mutual trust relationships and must be robust against arbitrary adversarial behavior to be applicable to peer-to-peer systems. Certainly, designing such a random number generator is not an easy task.

1.1 Robust Distributed Random Number Generation

How can we generate random numbers in a peer-to-peer system with adversarial presence? The most naive approach is to let every peer generate its own random numbers. This approach is problematic since in a dynamic peer-to-peer system it is impossible to collect sufficient statistical evidence to accuse a particular peer of generating non-random numbers. Yet, somewhat surprisingly, it is still possible to use this approach to maintain a robust peer-to-peer network, but at the cost of losing scalability [3]. So a different approach is needed.

A more reasonable approach is the following. Suppose that we need a random number generator that generates a number by selecting a binary string uniformly at random out of $\{0,1\}^s$ for some s. Consider the situation that a group P of the peers wants to generate a random number. Each (honest) peer p in P may then select a random number $x_p \in \{0,1\}^s$ and commit to it to all other peers in P using a bit commitment scheme (a particularly secure one-way hash function h for which $h(x)$ does not reveal anything about x) [12,17]. Once all commitments have been made, the peers will reveal their random numbers, and if they all do, every peer computes $x = \bigoplus_{p \in P} x_p$, where $\oplus$ is the bit-wise XOR operation. The XOR operation has the nice property that as long as at least one x_p is chosen uniformly at random and the other numbers are independent of it, x is distributed uniformly at random in $\{0,1\}^s$. Hence, *if* the scheme succeeds and at least one honest peer participates in it, a random number x will be generated. But the adversarial peers can easily let the scheme fail, and this not only in an oblivious manner but also in an adaptive manner (by just waiting for enough numbers x_p to be revealed before revealing their own numbers). Thus, in order to avoid a significant bias on the successfully generated random numbers, the fraction of adversarial peers in the system would have to be so small that no adversarial peer will be present in most of the groups P that are used for the random number generation. Such an approach was pursued in [2].

To avoid the problems above, we recently proposed a distributed random number generator that is based on verifiable secret sharing [4]. This random number generator can still fail if the peer initiating it does not behave correctly, but it has the advantage that if the peer initiating it is honest, then the random number generation is guaranteed to succeed, and whenever the random number generation succeeds, the number generated will be random.

Yet, using this scheme is not completely satisfying. First of all, an adversary can let it fail in an adaptive manner (i.e., it can let it fail after knowing the final key), which is sufficient to create a significant bias, even though the adversary cannot undermine the randomness of the generated key. It just has to run sufficiently many attempts until a key is generated that falls into a desired range. Furthermore, the scheme is not easy to implement and private channels are needed between the peers. So the question that led to this paper was:

Is it possible to design an elementary and sufficiently unbiased distributed random number generator that even works for public channels and a constant fraction of adversarial peers?

Remarkably, this paper shows that this is possible.

1.2 Related Work on Random Number Generation

Surprisingly little has been published about robust random number generators for distributed systems. Random number generators have mostly been studied in the context of pseudo-random number generators (PRNGs) with small seed or cryptographically secure random number generators (CSRNG). The main difference between a PRNG and a CSPRNG is that a CSPRNG should be indistinguishable from random on any examination, whereas a PRNG is normally only required to look random to standard statistical tests. For foundations and surveys on random number generators see, e.g., [11,16,23,31].

There are many protocols for distributed systems with adversarial presence that need random numbers for atomic broadcasting, leader election and almost-everywhere agreement (e.g., [14,21] for recent results), but in these it is sufficient that every peer chooses its own random numbers.

Unbiased random numbers can be computed via verifiable secret sharing or secure multiparty computation schemes (e.g., [5,28]), but these are not easy to implement (since they need error correction techniques), and they require private channels.

1.3 Details of Our Random Number Generator

The basic idea behind our random number generator is the insight that generating a *single* random number is difficult with public channels but generating a *batch* of random numbers is doable. An *m-random number generator* (or *m*-RNG) is a random number generator that generates a batch of up to m random numbers. We assume that every random number is represented as a binary string in $\{0,1\}^s$ for some fixed s. Given an m-RNG G and any subset $S \subseteq \{0,1\}^s$, let $\mathrm{E}_G(S)$ be the expected number of keys y generated by G with $y \in S$. Ideally, G should satisfy $\mathrm{E}_G(S) = m \cdot |S|/2^s$ for all $S \subseteq \{0,1\}^s$. Let $\mathrm{E}(S) = m \cdot |S|/2^s$. Then we define the *bias* $\beta(G)$ of G as

$$\beta(G) = \max_{S \subseteq \{0,1\}^s} \max \left\{ \frac{\mathrm{E}_G(S)}{\mathrm{E}(S)} , \frac{\mathrm{E}(S)}{\mathrm{E}_G(S)} \right\}$$

The m-RNG that we present in this paper is called *round-robin random number generator* (or short round-robin RNG). Let P be the group of m peers this protocol is applied to. The basic ideas of the protocol can be summarized as follows:

- When correctly initiated, every peer in P will supervise the generation of one random number in $\{0,1\}^s$. A peer whose random number generation fails can send an accusation to the peers in P in which it can accuse exactly one other peer. Honest peers will run the random number generation one after the other (using a proper timing scheme) so as to maximize the effect of the accusations and thereby minimize the number of times an adversarial peer can cause the failure of a random number generation supervised by an honest peer.
  dealer and the others being a group of players. Both the players and the dealer commit to a key. However, as we will see, the dealer key is a special master key that is committed first and revealed last. In this way, the dealer is the only one that can *adaptively* decide whether to let the random number generation fail or not. However, this is the only way in which the dealer can bias the random number generation. It cannot make its probability distribution non-uniform if at least one honest player is participating in it.

More details are given in Section 2. For this protocol, the following theorem is shown.

Theorem 1. *Suppose that $|P| = m$ and there are $t < m/6$ adversarial peers in P. Then the round-robin RNG generates random keys $y_1, y_2, \ldots, y_k \in \{0,1\}^s$ with $m - 2t \leq k \leq m$ and the property that for all subsets $S \subseteq \{0,1\}^s$ with $\sigma = |S|/2^s$,*

$$\mathrm{E}[|\{i \mid y_i \in S\}|] \in [(m-2t)\sigma,\ m \cdot \sigma]$$

The worst-case message complexity of the protocol is $O(m^2)$.

Hence, the bias of our m-RNG is just $1 + \frac{2t}{m-2t}$, which is a constant. It turns out that this bias is small enough in order to maintain a scalable and robust peer-to-peer network.

1.4 Application to Robust Peer-to-Peer Networks

In the area of peer-to-peer systems, work on robustness in the context of overlay network maintenance has mostly focused on how to handle a large fraction of faulty peers (e.g., [1,24,30]) or churn, that is, peers frequently enter and leave the system (e.g., [15,22]). However, none of these approaches can protect a peer-to-peer network against adaptive join-leave attacks. In an adaptive join-leave attack, adversarial peers repeatedly join and leave a network in order to occupy certain areas of the network. To prevent them from doing this, proper join and leave protocols have to be found so that the honest and adversarial peers are kept well-spread in the $[0,1)$-interval. More precisely, what we would like to aim for

is that at any time point with n peers in the system the following two conditions can be met for every interval $I \subseteq [0,1)$ of size at least $(c \log n)/n$ for a constant $c > 0$:

- *Balancing condition:* I contains $\Theta(|I| \cdot n)$ peers.
- *Majority condition:* the honest peers in I are in the majority.

If this is the case, then proper region-based overlay networks and routing rules can be defined to guarantee connectivity and correct routing (e.g., [4]). However, maintaining the two conditions under adaptive adversarial join-leave attacks turns out to be quite tricky. Just assigning a random or pseudo-random point to each new peer (by using some random number generator or cryptographic hash function) does not suffice to preserve the balancing and majority conditions [2]. Fortunately, just recently we found a join operation, called cuckoo rule, that can solve this problem [4].

1.5 The Cuckoo Rule

In the following, a *region* is an interval of size $1/2^r$ in $[0,1)$ for some integer r that starts at an integer multiple of $1/2^r$. Hence, there are exactly 2^r regions of size $1/2^r$. A *k-region* is a region of size (closest from above to) k/n, and for any point $x \in [0,1)$, the k-region $R_k(x)$ is the unique k-region containing x.

Cuckoo rule: If a new node v wants to join the system, pick a random $x \in [0,1)$. Place v into x and move all nodes in $R_k(x)$ to points in $[0,1)$ chosen uniformly and independently at random (without replacing any further nodes).

Suppose that we have n honest peers and ϵn adversarial peers in the system for some $\epsilon < 1$. For the situation that the adversary adaptively rejoins the system with its peers in a one-by-one fashion, it was shown [4] that as long as $\epsilon < 1 - 1/k$, the k-cuckoo rule satisfies the balancing and majority conditions for a polynomial number of rejoin operations, with high probability. However, for the cuckoo rule to be implementable in a distributed system, a robust distributed random number generator is needed. Furthermore, the cuckoo rule may need up to $O(\log^2 n)$ random bits in the worst case (for $O(\log n)$ peers that need to be replaced).

1.6 The Round-Robin Cuckoo Rule

The problem with $O(\log^2 n)$ bits is solved by proposing a slight adaptation of the cuckoo rule that we call the *de Bruijn cuckoo rule*. The new rule has the benefit that only $O(\log n)$ random bits are needed in the worst case (for two random points in $[0,1)$).

In order to solve the problem with the random number generator, we combine the round-robin RNG with the de Bruijn cuckoo rule to the so-called *round-robin cuckoo rule*. It works in a way that for every successful random number generation in the round-robin RNG, the de Bruijn cuckoo rule is used. The protocol has the following performance.

Consider adversarial join-leave attacks in a system with n honest peers and ϵn adversarial peers. Let β be the bias of the round-robin RNG. Then it holds:

Theorem 2. *For any constants ϵ, k and β with $\epsilon < 1/\beta - 1/k$, the round-robin cuckoo rule satisfies the balancing and majority conditions for a polynomial number of rounds, with high probability, for any adversarial strategy within our model.*

Hence, Theorem 2 is a natural extension of the result in [4].

1.7 Structure of the Paper

In Section 2, we present the round-robin random number generator, and in Section 3 we show how to use it to counter join-leave attacks in peer-to-peer networks. The paper ends with conclusions.

2 Robust Random Number Generation

In this section we consider the situation that we have a set P of m players denoted $p_1, \ldots, p_m$. We distinguish between honest and adversarial players. The honest players follow the protocol in a correct and timely manner, whereas the adversarial players may behave in an arbitrary way, including arbitrary collusion among the adversarial players. Our goal is to find *elementary* protocols that construct random numbers with a uniform distribution in $\{0,1\}^s$ for some given s, even under adversarial presence.

First, we state some assumptions, and then we present the round-robin random number generator. After its analysis, we discuss some extensions for peer-to-peer systems.

2.1 Assumptions

We assume that only point-to-point communication is available and that all information sent out by a player can be seen by the adversary. Thus, no broadcasting primitive and no private channels are given, which is often the case in other robust distributed protocols like verifiable secret sharing. We just need a mechanism that allows the players to verify the sender of a message. For this, we assume the existence of a non-repudiable signature scheme. A message m signed by player p will be denoted as $(m)_p$.

Honest players are supposed to act not only in a correct but also a timely manner (which is important to maintain dynamic systems such as peer-to-peer networks). We assume that any message sent from one honest player to another honest player needs at most δ time steps to be received and processed by the recipient for some fixed δ, and we assume that the clock speeds of the honest players are roughly the same. But the clocks do not have to be synchronized (i.e., show the same time) nor do we require the protocols to run in a synchronous mode (i.e., all players must send their messages at exactly the same time). The

latter assumption makes it hard to generate unbiased random keys even though there is a notion of time because the adversarial players can always choose to be the last to send out messages, thereby maximizing the control they have on the generation of the random number.

For the random number generation, we need a bit commitment scheme h, i.e., a scheme where $h(x)$ does not reveal anything about x. In practice, a cryptographic hash function might be sufficient for h so that the protocols below can be easily implemented. Furthermore, we assume that all honest players have a perfect random number generator. In practice, pseudo-random number generators that pass a certain collection of statistical tests (such as the diehard tests) might be sufficient here.

2.2 Round-Robin Random Number Generator

Suppose that we have a set P of m players, $p_1, \ldots, p_m$, that mutually know each other and the indexing, with any t of them being adversarial for some $t < m/6$. The round-robin random number generator works as follows for some player $p^* \in P$ initiating it.

1. p^* sends a signed request to initiate the random number generation to all players in P.
2. Once player $p_i \in P$ receives p^*s signed initiation request for the first time (from anywhere), it forwards it to all other players in P. Afterwards, it sets $P_i := P \setminus \{p_i\}$ and waits for $i \cdot 8\delta$ time steps. Each time it receives an accusation $(p_k)_{p_j}$ from a player $p_j \in P$ it has not received an accusation from yet, it sets $P_i := P_i \setminus \{p_k\}$. Once the $i \cdot 8\delta$ steps are over, p_i initiates step (3). p_i terminates after $(m+1)8\delta$ steps.
3. If $|P_i| \geq 2m/3$, then p_i chooses a random $x_i \in \{0,1\}^s$ and sends $(h(x_i), P_i)_{p_i}$ to all players in P_i. Otherwise, p_i aborts the protocol (which will not happen if $t < m/6$).
4. Each player $p_j \in P_i$ receiving a message $(h(x_i), P_i)_{p_i}$ for the first time from p_i with $P_i \geq 2m/3$ chooses a random $x_j \in \{0,1\}^s$ and sends the message $(p_i, h(x_j), P_i)_{p_j}$ to p_i. Otherwise, it does nothing.
5. If all players in P_i reply within 2δ time steps, then p_i sends $(\{(p_i, h(x_j), P_i)_{p_j} \mid p_j \in P_i\})_{p_i}$ to all players in P_i. Otherwise, p_i sends an accusation $(p_j)_{p_i}$ for any $p_j \in P_i$ that did not reply correctly or in time to all players in P and stops its attempt of generating a random number.
6. Once $p_j \in P_i$ receives $(\{(p_i, h(x_k), P_i)_{p_k} \mid p_k \in P_i\})_{p_i}$ from p_i, p_j sends $(x_j)_{p_j}$ to p_i.
7. If p_i gets a correct reply back from all players in P_i within 2δ time steps, then it sends $(x_i, \{(x_j)_{p_j} \mid p_j \in P_i\})_{p_i}$ to all players in P_i and computes $y_i = x_i \oplus \bigoplus_{p_j \in P_i} x_j$ where $\oplus$ is the bit-wise XOR operation. Otherwise, p_i sends an accusation $(p_j)_{p_i}$ to all players in P for any $p_j \in P_i$ that did not reply correctly or in time and stops.
8. Once $p_j \in P_i$ receives $(x_i, \{(x_k)_{p_k} \mid p_k \in P_i\})_{p_i}$, p_j verifies that all keys are correct. Then p_j computes $y_j^{(i)} = x_i \oplus \bigoplus_{q_k \in P_i} x_k$ and sends the message $(y_j^{(i)})_{p_j}$ to p_i.

9. If p_i receives y_i from at least $2m/3$ players in P within 2δ time steps, it accepts the computation and otherwise sends an accusation $(p_j)_{p_i}$ to all players in P for any $p_j \in P_i$ that did not reply correctly or in time.

We define the random number generation of p_i to be *successful* if p_i receives the same key from at least $2m/3$ many players in step (9). This is important for p_i since it will need the support of at least $2m/3$ other players for further operations that we will discuss in the next section.

2.3 Analysis of the Round-Robin RNG

The round-robin RNG has the following performance.

Theorem 3. *Suppose that $|P| = m$ and there are $t < m/6$ adversarial players in P. Then the round-robin RNG generates random keys $y_1, y_2, \ldots, y_k \in \{0,1\}^s$ with $m - 2t \leq k \leq m$ and the property that for all subsets $S \subseteq \{0,1\}^s$ with $\sigma = |S|/2^s$,*

$$\mathrm{E}[|\{i \mid y_i \in S\}|] \in [(m-2t)\sigma, m \cdot \sigma] \ .$$

The worst-case message complexity of the protocol is $O(m^2)$.

In order to prove the theorem, we start with some simple claims.

Some basic facts. Because of the flooding strategy in step (2) and the definition of δ it holds:

Claim. No matter whether p^* is adversarial or not, all honest players start the protocol within δ steps.

Since each honest player p_i needs at most 7δ time steps to complete the protocol from step (3) to (9) and starts after waiting for $i \cdot 8\delta$ steps, the claim above implies the following claim.

Claim. No two honest players execute their random number generation scheme (steps (3) to (9)) at the same time.

Hence, honest player p_i can make use of the accusations of all honest players p_j with $j < i$ in order to keep its own problems with the random number generation as small as possible.

Next, we bound the size of any P_i for an honest player p_i. Recall that honest players are supposed to work in a correct and timely manner. Hence, honest players will never accuse other honest players of any wrongdoing but only adversarial players. Since every adversarial player can issue at most one accusation, there will be at least $m-2t$ honest players left in every set P_i of an honest player p_i throughout the protocol. Hence, we get:

Lemma 1. *If $t < m/6$ then $|P_i| \geq 2m/3$ throughout the protocol for every honest player p_i.*

Moreover, every player p_i can only be successful for one key. This is because all players in P_i have to see commitments to the same P_i for all players in P_i and $|P_i| \geq 2m/3$ before revealing their random keys in step (6). Since $t < m/6$, this means that there must be more than $m/2$ honest players in P_i, which can only be possible for at most one P_i. Hence, we get.

Lemma 2. *If $t < m/6$ then every player can be successful for at most one key.*

Analysis of steps (3) to (9). Next, we focus on the execution of steps (3) to (9) by some fixed peer p_i. First, we consider the case that p_i is honest, and then we consider the case that p_i is adversarial.

Lemma 3. *If p_i is honest and $|P_i| \geq 2m/3$, then no matter how many adversarial players there are in P_i, if the protocol terminates successfully, then the key y_i generated by p_i is distributed uniformly at random in $\{0,1\}^s$ and all honest players in P_i compute the same key as p_i.*

Proof. p_i will not reveal x_i before the keys in P_i have all been revealed. Hence, the probability distribution on $z = \bigoplus_{p_j \in P_i} x_j$ must be independent of x_i. But for any probability distribution on $z = \bigoplus_{p_j \in P_i} x_j$ that is independent of x_i it holds that if x_i is chosen uniformly at random in $\{0,1\}^s$, then also $y_i = x_i \oplus z$ is distributed uniformly at random in $\{0,1\}^s$. Moreover, also the decision of the adversarial players to let the random number generation fail must be independent of x_i and can only be a function of z because x_i will not be revealed before. Hence, it holds for any adversarial strategy and any $y^* \in \{0,1\}^s$ that

$$\Pr[y_i = y^* \mid \text{generation of } y_i \text{ successful}] = \Pr[y_i = y^*] = \frac{1}{2^s}$$

If p_i succeeds with computing y_i, then it informed all players in P_i about the revealed keys, and all honest among them will accept these keys since they match the message sent out by p_i in step (5). Hence, all honest players in P_i compute the same key as p_i. □

Notice that if the adversarial players knew about x_i *before* deciding to let the random number generation fail, they can create a significant bias, even if the other keys were chosen independent of x_i. A simple example for this would be:

Focus on any fixed $y^* \in \{0,1\}^s$. If $y_i = y^*$, then let the attempt fail, and otherwise let it be successful.

It is easy to see that this would make it very unlikely for the round-robin RNG to generate y^* (since it would have to be generated more than t times to be successful at least one). Hence, it is crucial that x_i is only revealed *after* all the other keys have been revealed. Next, we consider the case that p_i is adversarial.

Lemma 4. *If p_i is adversarial, then no matter what p_i and the other adversarial players in P_i do, whenever an honest player p_j reveals its key x_j, $y_i^{(j)}$ has a uniform distribution on $\{0,1\}^s$.*

Proof. An honest player p_j will only reveal x_j once it receives $(\{(p_i, h(x_k), P_i)_{p_k} \mid p_k \in P_i\})_{p_i}$ from p_i and $p_j \in P_i$ (so that $y_j^{(i)}$ is well-defined). In this case, x_j is a random number that is independent of $z = x_i \oplus \bigoplus_{p_k \in P_i \setminus \{p_j\}} x_k$, and since x_j is independent of z and chosen uniformly at random, $y_j^{(i)} = x_j \oplus z$ has a uniform distribution. □

Notice, however, that p_i can commit to different sets P_i to different honest players without being detected, so the keys $y_j^{(i)}$ can differ among the honest players. Nevertheless, if p_i wants to be successful (i.e., collect commitments to the same key from at least $2m/3$ many players), it must let more than $m/2$ honest players p_j succeed with computing the same $y_j^{(i)}$, which has a uniform distribution.

Still, the adversarial players can create a bias on the *successfully* computed keys since after knowing y_i, an adversarial player p_i still has the option to let the key generation be successful or not. Fortunately, this bias cannot be too large, as shown in the following lemma.

Analysis of the entire protocol

Lemma 5. *If $t < m/6$ then at least $m - 2t$ of the $m - t$ random number generations initiated by the honest players are successful, irrespective of whether p^* is adversarial or not. Furthermore, it holds for all subsets $S \subseteq \{0,1\}^s$ with $\sigma = |S|/2^s$ that $\mathrm{E}[|\{i \mid y_i \in S \text{ for a successful } y_i\}|] \in [(m-2t)\sigma,\ m \cdot \sigma]$*

Proof. According to Lemma 4, every key y that an honest player p commits to must be distributed uniformly at random in $\{0,1\}^s$. However, whereas the adversarial players can adaptively abort the random number generation initiated by adversarial players, it follows from Lemma 3 that they can only do this in an oblivious way for the honest players. We know that the adversarial players can only sabotage the random number generation of at most t honest players. Hence, at least $m-2t$ random number generations of honest players p_i will be successful, and their success does not depend on their values. Thus, the probability for any of these players p_i that $y_i \in S$ is equal to σ and, therefore, the expected number of successful p_i's with $y_i \in S$ is at least $(m-2t)\sigma$.

On the other hand, at most m key generations can be successful, and since every successfully generated key y_i is distributed uniformly at random in $\{0,1\}^s$, the probability for any y_i to be in S is equal to σ. Hence, the expected number of successful p_i's with $y_i \in S$ is at most $m \cdot \sigma$. □

The next lemma follows immediately from the protocol.

Lemma 6. *The message complexity of the round robin-random RNG is $O(m^2)$.*

2.4 Extensions

In our random number generator we assumed that the players in P know each other and the indexing. This assumption can be problematic in peer-to-peer systems since there might be disagreement among the honest players about the

set of adversarial players in P. However, if at least all the honest players know each other and there are, for example, less than $m/8$ adversarial players in P, this can easily be fixed. Suppose that every honest player p_i uses a threshold of $3m_i/4$ instead of $2m/3$, where m_i is the number of players that p_i knows in P and $m = |P|$. Then all results above still hold since $3m_i/4 \geq (3/4) \cdot (7m/8) > m/2$.

Another problem is how to fix the indexing issue. When there is disagreement about P, it will not be possible for the honest players to agree on a common indexing scheme. Instead, they can use the following simple trick. Each player p_i picks a random slot out of $c \cdot m_i$ many slots for generating a random number, where c is a fixed constant. Then it is easy to calculate that the number of slots occupied by the honest players is at least $(1 - 1/(2c))m_h$, where m_h is the number of honest players. Hence, the adversarial players could manage now to let up to $t + m_h/(2c)$ random number generations of honest players fail instead of just t, which is still acceptable if c is sufficiently large.

3 Application to Robust Peer-to-Peer Networks

In this section we show how to use the round-robin random number generator above to satisfy the balancing and majority conditions for any adversarial join-leave strategy for a polynomial number of rejoin operations, with high probability. We start with a formal model. Then we present the de Bruijn cuckoo rule, and afterwards we combine it with the round-robin RNG to obtain the round-robin cuckoo rule.

3.1 Model

Recall that we want to associate all peers with points in $[0, 1)$. These points can be encoded as binary strings from $\{0, 1\}^s$ (in a sense that $b = (b_1, \ldots, b_s)$ represents $x_b = \sum_{i \geq 1} b_i/2^i$) for a sufficiently large s (in SHA-1, which is used by the Chord system, for example, $s = 160$).

There are n blue (or honest) nodes and ϵn red (or adversarial) nodes for some fixed constant $\epsilon < 1$. There is a rejoin operation that, when applied to node v, lets v first leave the system and then join it again from scratch. The leaving is done by simply removing v from the system and the joining is done with the help of a join operation to be specified by the system. We assume that the sequence of rejoin requests is controlled by an adversary. The adversary can only issue rejoin requests for the red nodes, but it can do this in an arbitrary adaptive manner. That is, at any time it can inspect the entire system and select whatever red node it likes to rejoin the system. The goal is to find an *oblivious* join operation, i.e., an operation that does not distinguish between the blue and red nodes, so that for *any* adversarial strategy above the balancing and majority conditions can be kept for any polynomial number of rejoin requests.

3.2 The de Bruijn Cuckoo Rule

Recall the original cuckoo rule in Section 1.5. We present a slight but crucial modification to this rule, called the *de Bruijn cuckoo rule*, which only needs two

random numbers in $\{0,1\}^s$, irrespective of k. The prefix *de Bruijn* was chosen because the rule can be easily implemented in dynamic de Bruijn graphs (e.g., [18]).

de Bruijn cuckoo rule: If a new peer v wants to join the system, pick random $x, y \in [0,1)$. Place v into x and replace all peers in $R_k(x)$ in the following way. If $|R_k(x)| = 0$, we are done, and if $|R_k(x)| = 1$, then the peer in $R_k(x)$ is moved to position y. Otherwise, let $b = \lceil \log |R_k(x)| \rceil$. Given that y is represented by a binary string $(y_1, \ldots, y_s) \in \{0,1\}^s$, peer $i \geq 0$ in $R_k(x)$ is moved to position $((y_{s-b+1}, \ldots, y_s) \oplus (i)_2) \circ (y_1, \ldots, y_{s-b})$ where $(i)_2$ represents the binary representation of i and $\circ$ the concatenation.

For example, suppose that $y = 0100110$ and $|R_k(x)| = 3$. Then the new positions of the three peers are $(10 \oplus 00) \circ 01001 = 1001001$ for peer 0, $(10 \oplus 01) \circ 01001 = 1101001$ for peer 1, and $(10 \oplus 10) \circ 01001 = 0001001$ for peer 2. This rule of mapping peers to new points has the following property:

Lemma 7. *Every replaced peer is moved to a position that is distributed uniformly at random in* $\{0,1\}^s$.

Proof. Consider peer i in $R_k(x)$ for any fixed i and suppose that y is distributed uniformly at random in $\{0,1\}^s$. Then $(y_{s-b+1}, \ldots, y_s) \oplus (i)_2$ is distributed uniformly at random in $\{0,1\}^b$ and $(y_1, \ldots, y_{s-b})$ is distributed uniformly at random in $\{0,1\}^{s-b}$, resulting in the lemma. □

Moreover, any two peers in a region $R_k(x)$ with p peers have a distance of at least $(1/2)^{\log p - 1} \geq 1/(2p)$ of each other. Hence, when looking at the analysis in [4], it turns out that all results still hold when using a perfect random number generator (though in Lemma 2.6 and Lemma 2.10 the independence property of the new node positions has to be replaced by negative correlation, but the negative correlation is so small that it is negligible).

Theorem 4. *For any constants* ϵ *and* k *with* $\epsilon < 1 - 1/k$*, the de Bruijn cuckoo rule with parameter* k *satisfies the balancing and majority conditions for a polynomial number of rounds, with high probability, for any adversarial strategy within our model. The inequality* $\epsilon < 1 - 1/k$ *is sharp as counterexamples can be constructed otherwise.*

3.3 The Round-Robin Cuckoo Rule

Finally, we show how to combine the de Bruijn cuckoo rule and the round-robin random number generator into a simple and efficient join protocol called *round-robin cuckoo rule* that achieves a result similar to Theorem 4.

Recall the definition of a region in Section 1.5. Given a node $v \in [0,1)$, we define its *quorum region* R_v as the unique region of size closest from above to $(\gamma \log n)/n$, for a fixed constant $\gamma > 1$, that contains v.

We demand that whenever a new node u wants to join the system, it has to do so via a node v already in the system. v then initiates the following protocol:

1. v initiates the round-robin RNG in R_v (i.e., v acts as p^*).
2. For each successful node $v_i \in R_v$, v_i initiates the de Bruijn cuckoo rule by sending a message $(y_i, \{(y_j^{(i)})_{p_j} \mid v_j \in P_i\})_{p_i}$ with $1 + 2m/3$ signed keys to all nodes in R_v.
3. Once node $w \in R_v$ receives a correctly signed $(y_i, \{(y_j^{(i)})_{p_j} \mid v_j \in P_i\})_{p_i}$ containing more than $2m/3$ keys, it forwards it to all other nodes in R_v and initiates the de Bruijn cuckoo rule.

In the de Bruijn cuckoo rule, majority decision is done to execute the proper actions (see [4] for more details). Since step (3) ensures the "all or nothing" principle concerning the honest nodes, the de Bruijn cuckoo rule can be guaranteed to be executed in a correct and timely manner. The new node u can choose to assume any one of the new positions of a successfully executed de Bruijn cuckoo rule. It just needs to commit to one to R_v. If the node v just wants to rejoin the system (like in the adversarial strategies considered here), then we identify v with u.

3.4 Perturbation with Biased Randomness

Recall that we consider adversarial join-leave attacks in a system with n honest nodes and ϵn adversarial nodes. Let β be the bias of the round-robin RNG. Then it holds:

Theorem 5. *For any constants ϵ, k and β with $\epsilon < 1/\beta - 1/k$, the round-robin cuckoo rule with the round-robin RNG with bias β satisfies the balancing and majority conditions for a polynomial number of rounds, with high probability, for any adversarial strategy within our model.*

Proof. (Sketch) Recall the Lemmas in Section 2 of [4]. Let $\delta > 0$ be a small constant. Lemma 2.4 holds as before. In Lemma 2.5, the age range of a region R consisting of $c \log n$ k-regions has to be adjusted to $[(1-\delta)(c\log n)(n/k), (1+\delta)\beta(c\log n)(n/k)]$, where the age of R is the sum of the ages of its k-regions and the age of a k-region is defined as the number of RNG attempts back in time till that region was last hit by a new node. Lemma 2.6 still holds. In Lemma 2.9, the range for the number of evicted honest nodes in a time interval of size T has to be adjusted to $[((1-\delta)/\beta)T \cdot k, (1+\delta)T \cdot k]$, and the range for the adversarial nodes has to be adjusted to $[((1-\delta)/\beta)T \cdot \epsilon k, (1+\delta)T \cdot \epsilon k]$. With these bounds, we obtain a worst-case ratio between honest and adversarial nodes if R has an age of $(1-\delta)(c\log n)(n/k)$. In this case, there are at least $((1-\delta)^2/\beta)(c\log n)k$ honest nodes and at most $(1-\delta)(1+\delta)(c\log n)\epsilon k + c\log n$ adversarial nodes in R, w.h.p. In order to satisfy the majority condition, it must hold that $\epsilon < 1/\beta - 1/k$. □

4 Conclusions

In this paper, we presented a simple and robust random number generator sufficient for keeping honest and adversarial peers well-distributed in $[0, 1)$. We only

proved our results assuming a sequential execution of rejoin operations (see our model) though we expect that as long as not too many rejoin operations are executed concurrently, there should be only insignificant side effects (see also the comments in [25]).

Interesting problems for future work are, how to extend our results to general β-biased m-RNGs and how to extend our rejoin operation so that we can even make a peer-to-peer network robust against adaptive join-leave behavior by both honest and adversarial peers.

References

1. J. Aspnes and G. Shah. Skip graphs. In *Proc. of the 14th ACM Symp. on Discrete Algorithms (SODA)*, pages 384–393, 2003.
2. B. Awerbuch and C. Scheideler. Group Spreading: A protocol for provably secure distributed name service. In *Proc. of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.
3. B. Awerbuch and C. Scheideler. Robust distributed name service. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.
4. B. Awerbuch and C. Scheideler. Towards a scalable and robust DHT. In *Proc. of the 18th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 2006. See also http://www14.in.tum.de/personen/scheideler.
5. M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 183–192, 1994.
6. M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for structured peer-to-peer overlay networks. In *Proc. of the 5th Usenix Symp. on Operating Systems Design and Implementation (OSDI)*, 2002.
7. M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the 2nd Usenix Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.
8. S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Usenix Security*, 2003.
9. J. R. Douceur. The sybil attack. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.
10. A. Fiat, J. Saia, and M. Young. Making Chord robust to Byzantine attacks. In *Proc. of the European Symposium on Algorithms (ESA)*, 2005.
11. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*, volume 17 of *Algorithms and Combinatorics*. Springer-Verlag, 1998.
12. S. Halevi and S. Micali. Practical and provably-secure commitment schemes from collision-free hashing. In *CRYPTO 96*, pages 201–215, 1996.
13. D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahi. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *29th ACM Symp. on Theory of Computing (STOC)*, pages 654–663, 1997.
14. V. King, J. Saia, V. Sanwalani, and E. Vee. Towards a secure and scalable computation in peer-to-peer networks. In *Proc. of the 47th IEEE Symp. on Foundations of Computer Science (FOCS)*, 2006.
15. F. Kuhn, S. Schmid, and R. Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

16. M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996.
17. M. Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, 1991.
18. M. Naor and U. Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *Proc. of the 15th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 2003.
19. S. Nielson, S. Crosby, and D. Wallach. Kill the messenger: A taxonomy of rational attacks. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.
20. G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the 9th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, 1997.
21. H.V. Ramasamy and C. Cachin. Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast. In *9th Conference on Principles of Distributed Systems (OPODIS)*, pages 88–102, 2005.
22. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Technical Conference*, 2004.
23. T. Ritter. RNG implementations: A literature survey. http://www.ciphersbyritter.com/RES/RNGENS.HTM.
24. J. Saia, A. Fiat, S. Gribble, A. Karlin, and S. Saroiu. Dynamically fault-tolerant content addressable networks. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.
25. C. Scheideler. How to spread adversarial nodes? Rotate! In *Proc. of the 37th ACM Symp. on Theory of Computing (STOC)*, pages 704–713, 2005.
26. A. Singh, M. Castro, A. Rowstron, and P. Druschel. Defending against Eclipse attacks on overlay networks. In *Proc. of the 11th ACM SIGOPS European Workshop*, 2004.
27. E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.
28. K. Srinathan and C.P. Rangan. Efficient asynchronous secure multiparty distributed computation. In *Proc. of the 1st Int. Conference on Progress in Cryptology*, pages 117–129, 2000.
29. M. Srivatsa and L. Liu. Vulnerabilities and security threats in structured overlay networks: A quantitative analysis. In *Proc. of the 20th IEEE Computer Security Applications Conference (ACSAC)*, 2004.
30. I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM '01*, 2001. See also `http://www.pdos.lcs.mit.edu/chord/`.
31. J. Viega. Practical random number generation in software. In *Proc. of the 19th Annual Computer Security Applications Conference*, 2003.

About the Lifespan of Peer to Peer Networks*,**

Rudi Cilibrasi[1], Zvi Lotker[1], Alfredo Navarra[2], Stephane Perennes[3], and Paul Vitanyi[1]

[1] CWI - Kruislaan 413, NL-1098 SJ Amsterdam, Netherlands.
{Rudi.Cilibrasi, Z.Lotker, Paul.Vitanyi}@cwi.nl
[2] LaBRI - Université de Bordeaux 1, 351 cours de la Liberation, 33405 Talence, France
Alfredo.Navarra@labri.fr
[3] MASCOTTE project, I3S-CNRS/INRIA/Université de Nice Sophia Antipolis, France
Stephane.Perennes@sophia.inria.fr

Abstract. In this paper we analyze the ability of peer to peer networks to deliver a complete file among the peers. Early on we motivate a broad generalization of network behavior organizing it into one of two successive phases. According to this view the network has two main states: first centralized - few sources (roots) hold the complete file, and next distributed - peers hold some parts (chunks) of the file such that the entire network has the whole file, but no individual has it. In the distributed state we study two scenarios, first, when the peers are "patient", i.e, do not leave the system until they obtain the complete file; second, peers are "impatient" and almost always leave the network before obtaining the complete file.

We first analyze the transition from a centralized system to a distributed one. We describe the necessary and sufficient conditions that allow this vital transition. The second scenario occurs when the network is already in the distributed state. We provide an estimate for the survival time of the network in this state, i.e., the time in which the network is able to provide all the chunks composing the file. We first assume that peers are patient and we show that if the number of chunks is much less than e^n, where n is the number of peers in the network, then the expected survival time of the network is exponential in the number of peers. Moreover we show that if the number of chunks is greater than $\frac{\log n}{n+1}e^{n+1}$, the network's survival time is constant. This surprisingly suggests that peer to peer networks are able to sustain only a limited amount of information. We also analyze the scenario where peers are impatient and almost always leave the network before obtaining the complete file. We calculate the steady state of the network under this condition. Finally a simple model for evaluating peer to peer networks is presented.

Keywords: P2P, file sharing, chunk, downloading rate, survivability.

* The research was partially funded by the European projects COST Action 293, "Graphs and Algorithms in Communication Networks" (GRAAL) and COST Action 295, "Dynamic Communication Networks" (DYNAMO).

** Special thanks go to Jean-Claude Bermond. The research was done while the authors were meeting at the MASCOTTE project site in Sophia Antipolis.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 290–304, 2006.

1 Introduction

Over recent years, peer to peer (P2P) networks have emerged as the most popular method for sharing and streaming data (see for instance [1]). There has been popular adoption and widespread success due to the high efficiencies that these networks obtain for broadcast data. Apart from personal usage, many companies, like for instance **red**hat®, provide links in order to download their free distributions in a P2P fashion. In doing so, companies avoid the problem of too many clients connected to their server. This solves bottleneck or high concentrated transmission cost on a single node with a significant chance of failure at peak loads. On the other hand, companies are not the only benefactor. Indeed, it makes things much faster from the point of view of the user even though at the expense of "being used" by other users.

Another very interesting application where such networks are highly successful concerns the distribution of data for storage purposes. The idea, in fact, to collect data among users spread over all the world is increasing more and more. Instead of having (for each one) a full copy of everything, a community can share resources hence obtaining a distributed storage device. This permits them to collectively maintain more and more data and it increases also the reliability. It ensures, in fact, that data will not disappear due to the malfunction of a small number of devices.

The main differences among the two applications we have just outlined are a more collaborative environment and a lower percentage of disappearing pieces of data, with the second being the more reliable in this sense. For downloading purposes, in fact, the aim is sometimes to download the required data as fast as possible and then leave the network. This implies also a higher frequency in peers disappearing. On the other hand, in an ideal world where people collaborate for a common final purpose, we like to imagine both the applications are quite equivalent.

We consider the following processes. A file is divided into k chunks. The network contains a large number of nodes. We distinguish among *peers*, i.e., nodes with a number of chunks less than k and *roots*, i.e., nodes owning all the chunks. We assign to peers a probability α for which they may disappear. This means that peers live on average $\frac{1}{\alpha}$ rounds. For the roots we chose a probability $\alpha_R \geq \alpha$. We consider closed network, i.e., every time a peer or a root leaves the network a new peer will join with no chunks. For the sake of simplicity we consider a synchronous model. During a round each node can receive or send one chunk; and at the end of each round any node disappears with respect to the related probability. We study the following three scenarios:

Spreading or Centralized Scenario: Peers contain no chunks and there are R roots. We wonder if the *file is spread into the network*. This happens if all the chunks are sent from the roots inside the network where they multiply themselves. This can give us a measure for the file length with respect to the life of the last surviving root. After the last root leaves the network, in fact, either the

file has been spread or it is not possible to build it back. We say in this case, the network (or the file) is "dead".

Distributed Scenario: The chunks are widely spread in the network. There is not a fixed amount of roots, R can be also zero, we only require that the whole chunks composing the file are there. We wonder if the network life is long or short. As we are going to see, the network life is long when some almost steady state is reached. This can give us a measure about the conditions ensuring a long life for the network, and, when this happens, how often a full download is completed.

Survivability Scenario: In this scenario we are interested in studying the network behavior under extreme conditions. We consider, in fact, the case in which a file is almost never downloaded since peers have a very high volatility. They almost never stay in the network long enough to perform a full download. However, it is still very interesting to note that the network is able to survive. We recall that a network is said to survives whenever it still contains all the k chunks composing the original file, no matter where they reside.

Given some parameter settings, our aim is to answer the question of how long we can expect a network to continue producing new completed downloads. For all the previous three scenarios we provide a stochastic formulation. We show how parameters should be set up in order to obtain the desired results concerning network survivability and file downloading rate. It turns out that the eventual fate of the network is mainly dependent on the number of nodes n and the number of chunks k in which the file is split. We show how the network may pass through the previous three scenarios before eventually dying.

1.1 Related Work

A lot of work has been devoted to the area of file sharing in P2P networks. Many experimental papers provide practical strategies and preliminary results concerning the behavior of these kind of networks. In [2] for instance, the authors essentially describe properties like liveness and downloading rate by means of extended experiments and simulations under several assumptions.

Concerning analytical models it is very difficult to capture suitable features in order to describe what happens and why protocols like BitTorrent [3] are so powerful in practice. Suitable models are hard to find that describe what sometimes is easily observable by simulations. One of the main assumption made in the literature in order to describe the behavior of such networks at a top level concern Flow models [4,5], Queueing theory [6], Network Coding [7] and Coupon Collector aspects [8]. This latter paper mainly focuses on systems in which peers owning some chunks (usually one at random) appear in the network with some probability and disappear as soon as they complete their download. Recently in [9], the distribution of k chunks on a network with diameter d and maximum degree D has been proved to require at most $O(D(k+d))$ rounds of concurrent downloads with high probability. This is tight within a factor of D. They also

specialized to the networks used by BitTorrent improving the bound to $O(k \ln\ n)$ rounds where n is the number of nodes.

1.2 A First Thought

Such results are quite interesting from a theoretical point of view but sometimes not truly representative of the real life. The main assumption that collides with practical aspects is that the number of peers participating in the protocol is assumed to be huge, hence obtaining asymptotically optimal results in terms of network survivability and spreading speed of the desired file. Moreover, different from our model many of the previously cited papers do not assume the possibility for a peer to leave the network before it completes the whole file. This aspect is indeed introduced also in [5]. On the other hand we should immediately point out that if there is at least one root that stays indefinitely, then the file will always be available in the network, if a peer is willing to wait. We call such a scenario "*trivial*" since there is no question about the behavior of the network. In contrast, if all the original R roots disappear (at some time t) then there are many possibilities. We say a chunk is present in the network if at least one peer has that chunk. If there exists a chunk that is not present on the network, then no more full downloads are possible. Therefore, the most interesting case is when no roots are persistent ($t < \infty$).

When using BitTorrent [3] or similar programs in order to download desired files, usually such networks look quite different. The assumption for which a huge number of nodes is participating in the protocol given in [7,8,5] is indeed too strong. Moreover, in practice, the number of chunks is usually much bigger than the number of nodes composing the network. This is due to the fact that even if a file is spread among thousands of users, they do not participate concurrently in the protocol. At any given time the network is usually quite small if compared to the actual number of downloads.

Figure 1 shows a standard screenshot of the advanced BitTorrent window while downloading a file of size roughly $272Mb$. As it is described in the figure, there are 38 peers participating in the protocol with 15 roots and 23 peers while the number of chunks is 546 that is $512Kb$ per chunk[1]. It is worth noting that under those circumstances the success of these protocols has to reside in the adopted strategies, in contrast with that outlined in [8]. In such a setting, for instance, the "rarest pieces" distribution becomes quite important. This is the peculiarity in BitTorrent for which once an empty peer appears in the network it is provided with the least common chunk among the network. Also the "altruistic user behavior" is quite crucial from the point of view of the network survivability. It is based on the observable fact for which peers that terminate their download do not immediately disappear. In [2] for instance, it is pointed out how friendly users usually behave in the network. They do not disappear as soon as their download is finished thus ensuring that all the chunks are available. Indeed most

[1] Indeed in the BitTorrent specifications [3], the default size of a chunk is $256Kb$, hence obtaining 1092 chunks.

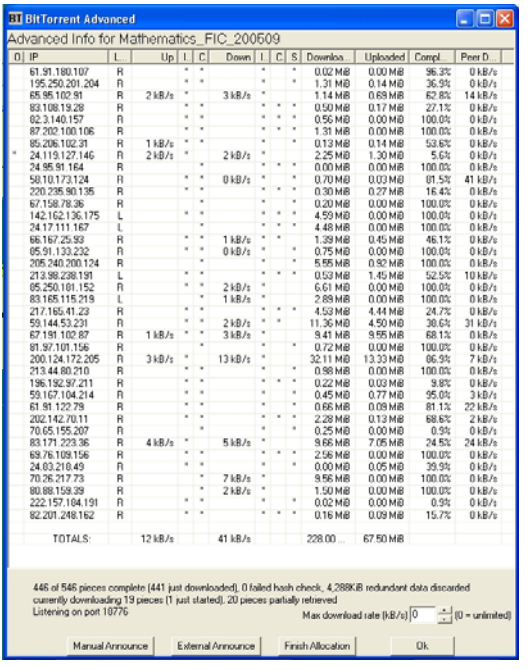


Fig. 1. Screenshot of the advanced properties of BitTorrent during a downloading phase

of the time this happens since whoever is downloading has just left the computer unattended but working.

1.3 Outline

The remainder of the paper is organized as follows. In the next section we give our first insight of P2P networks by introducing the so called Spreading Scenario. We show under which conditions a file is successfully spread over the network, hence remains alive. Section 3 describes the so called Distributed Scenario. We show under which conditions a file spread over the network can survive according to the number of peers composing the network and the number of chunks in which the file is split. Section 4 is devoted to the so called Survivability Scenario. In this case the behavior of the network is studied under critical circumstances like the high volatility of peers persistence. Section 5 provides a simple model for P2P networks in order to obtain numerical results about peers volatility, chunks distributions and downloading rate. Finally, Section 6 provides some conclusive remarks.

2 Spreading Scenario

In this section we study our first scenario in which a file must be distributed among the network by spreading its k chunks. Our model is synchronous. At

round t the network is composed by R_t roots that disappear with probability α_R. In this scenario we do not take into account the number of peers. Usually peers are much more than roots but for our analysis we just consider that a root can provide one chunk to one single peer at each round. This can be seen also by considering at round t a number of peers equal to R_t since we are just analyzing the number of rounds needed by the roots in order to spread the file. At a generic round t each peer asks a root for one random chunk. A root answers with a random chunk that has never been spread inside the network before. This implements the previously described "rarest pieces" issue. After each round, roots coordinate with each other in order to maintain the list of chunks that have been sent across the network.

The process is then similar to a coupon collector problem [8]. All the k chunks have to be collected, but a chunk can be collected several times during a round. Moreover the number of roots from which one can collect a coupon decreases exponentially. Let K_t be the number of chunks to be collected at time t, i.e., chunks that have not been distributed until round t.

For a given chunk x:

$$Pr(x \text{ is not collected}) = \left(1 - \frac{1}{K_t}\right)^{R_t} = \left(1 - \frac{1}{K_t}\right)^{K_t \frac{R_t}{K_t}} \approx e^{-\frac{R_t}{K_t}}$$

hence $E[K_{t+1}|K_t] \approx e^{-\frac{R_t}{K_t}} K_t$. Assuming for now that $K_t = E[K_t]$ with probability 1 we get $E[K_{t+1}] \approx e^{-\frac{R_t}{K_t}} E[K_t]$.

For R_t the situation is simpler since R_t is just the sum of R independent variables, (each one being described by the series $\sum Pr(R_{t+1} = i) z^i = (\alpha_R + z(1 - \alpha_R))^{R_t}$, and R_t is concentrated around its mean $(1 - \alpha_R)R_t$. So we have $E[R_{t+1}] = \alpha_R E[R_t]$.

Let $\rho_t = E[R_t]/E[K_t]$, we get

$$\rho_{t+1} = (1 - \alpha_R) e^{\rho_t} \rho_t.$$

From this, two situations can follow. If $(1 - \alpha_R)e^{\rho_0} \geq 1$, then ρ_t always increases, and this increase is faster and faster. This implies that the spreading will easily succeed since the number of chunks not spread decreases much faster than the number of roots that leave the network. Conversely, if $(1 - \alpha_R)e^{\rho_0} < 1$, ρ decreases and keep doing it faster and faster. This means that the process dies soon.

In the first situation we almost always collect all the chunks otherwise never. Of course, either at some point the chunks are all distributed or there are no roots that can provide the missing chunks. We can conclude from this first analysis that the file gets spread whenever $\alpha_R < 1 - e^{-\frac{R}{k}}$. Note that when $k >> R$ this actually means $\alpha_R < \frac{R}{k}$ and this is usually the case. For $k = R$ we get $\alpha_R \leq \frac{e-1}{e}$ but, in real world scenarios, this usually does not happen.

3 Distributed Scenario

In the previous section we gave a necessary and sufficient condition to describe the asymptotic behavior of the network. That is, the network must move from the initial state to the distributed state. In this section we study the behavior of P2P networks in the distributed state, that is, there are no roots, yet every chunk is available on the network after time t. In the following we refer to [10] for the applied probabilistic tools.

3.1 Upper Bound

Our next goal is to show that networks that are in the distributed state will not survive if the number of chunks is exponentially big in the number of peers. In order to show this we assume the following model. In each time step each peer asks a random chunk among all the chunks that the peer is missing. We assume that if the chunk is anywhere on the network then the peer will get this chunk in the next time step. Clearly this assumption is optimistic and will help the survival of the network. Importantly, this makes the network's gross behavior deterministic and thus we can say with certainty that every peer stays precisely k timesteps before leaving, since he gets exactly 1 chunk per timestep. When a new peer enters, it has no chunks and we may use the variable i, with $0 \leq i < n$, to indicate its ordering when all peers are sorted according to their number of chunks. This ordering is equivalent to the chronological ordering. After $\frac{k}{n}$ time steps, each peer is promoted to the next ordinal position. We point out that nothing changes in the network viability unless a peer leaves, and further the only peer that may leave is the last one, or the one with the most chunks; eventually the last peer will have k chunks when the file has been completely downloaded. We wish to bound the probability that a chunk will be missing. Therefore the number of chunks that each peer i has is $\frac{k(i-1)}{n}$ whenever a peer is leaving.

To be precise, we imagine the total network state at any time to be given by a binary vector of length kn; that is,

$$\Omega = \left(\{0,1\}^k\right)^n.$$

In Ω, the first k coordinates describe the chunks held by the first peer. In this first part, the first coordinate is 1 if and only if the first peer holds the first chunk, otherwise, it is 0. The next coordinate indicates the next chunk, and so on for all k chunks.

We will define an event G on Ω such that $\forall i = 1..n$, peer i has exactly $\frac{ik}{n}$ chunks. Let A_i^j be the event that peer i has chunk j. Let X_i^j be the indicator variable of A_i^j. Let $Y^j = \max\{X_1^j, X_2^j ..., X_n^j\}$, i.e., Y^j is the indicator variable of the event that chunk j is in the system. We define the random variable $Z = \min\{Y^1, ..., Y^k\}$ as follows: it is the indicator variable of the event that there is a missing chunk in the network. In other words, $Z = 0$ means the network has died, and $Z > 0$ means it continues to distribute the file.

Lemma 1. *Let n and k be the number of peers and chunks in the system respectively. For $n > 2$, the probability that there is a missing chunk can be bounded by* $\Pr[Z = 0] \leq kne^{-n}$.

Proof. by the Union bound over all k chunks,

$$\Pr[Z = 0] < k \Pr[Y^1 = 0] = k \prod_{i=1}^{n} \frac{i}{n} = k \frac{n!}{n^n}$$

and by Sterling's approximation,

$$k \frac{n!}{n^n} < kne^{-n}. \qquad \square$$

The next corollary shows that if the number of chunks is small the probability for the network to die approaches 0.

Corollary 1. *For all* $k < \frac{e^n}{n \log n}$,

$$\lim_{n \to \infty} \Pr[Z = 0] = 0.$$

The next corollary shows that if the number of chunks is small the system survives for a long time.

Corollary 2. *If the number of chunks is a polynomial* $\mathrm{Poly}(n)$ *then the expected survival time is at least* $\frac{e^n}{n\mathrm{Poly}(n)}$.

3.2 Lower Bound

The main problem in proving the lower bound is the dependence between the random variables Y^j and $Y^{j'}$. To remove this technical difficulty we use a different model, the binomial model. The idea is to make Y^j and $Y^{j'}$ i.i.d. variables. In order to prove a lower bound on the previous model we increase the expected number of chunks that peer i has at the time the last peer leaves the network. I.e., we relax the assumption that, at the time the last peer leaves the network, the number of chunks in the peer i is $\frac{k(i-1)}{n}$. Moreover we assume that the number of chunks is a binomial random variable. This assumption is legitimate since the binomial distribution is highly concentrated. The problem with this approach is that now we are no longer sure that each peer has enough chunks. The way we solve this problem is by strengthening the peer capabilities by increasing the chance that a peer has received chunk i. Since in a normal file sharing system chunks are correlated and peers have a smaller number of chunks, our lower bound also captures the behavior of these systems. This is justified since both assumptions (increasing the number of packets and the fact that packets are i.i.d) decrease the probability of failure. We do not offer a method to achieve this, but rather we use this approach to prove a lower bound with high probability. We posit this property (the binomial distribution) for the proof. We assume that the peers $i = 1, ..., n$ have $\frac{ki}{n+1}$ chunks on average. More precisely, let $\mathcal{X}_i^j$

be a Bernoulli random variable such that $E[\mathcal{X}_i^j] = \frac{i}{n+1}$. Let $\mathcal{G}_i = \sum_{j=1}^{k} \mathcal{X}_i^j$ be a random variable that counts the number of chunks that peer i has. Note that $E[\mathcal{G}_i] = \frac{ki}{n+1}$. The next lemma bounds the probability that peer i will have less than $\frac{(i-1)k}{n}$ chunks.

Lemma 2. *Let $\mathcal{G}_i$ be the number of different chunks belonging to peer i. For all $1 \leq i \leq n$, $\Pr[\mathcal{G}_i < \frac{(i-1)k}{n}] < e^{-\frac{k}{2n^3(n+1)}}$.*

Proof.

$$\Pr\left[\mathcal{G}_i < \frac{k(i-1)}{n}\right] = \Pr\left[\mathcal{G}_i < \left(1 - \frac{n+1-1}{in}\right)\frac{ki}{(n+1)}\right] <$$

$$< e^{-\left(\frac{n+1-i}{in}\right)^2 \frac{ki}{2(n+1)}} < e^{-\frac{k}{2n^3(n+1)}}$$

The first equality follows from algebra, and a Chernoff bound yields the next inequality. □

If $k >> n^4$ we get that the probability that the i-peer (Bernoulli process) has less than $\frac{(i-1)k}{n}$ chunks is exponentially small.

Let $Q = \bigcap_{i=0}^{n-1} \{\mathcal{G}_i \geq \frac{(i-1)k}{n}\}$. Note that Q is the event that all peers have more chunks than they are supposed to have, i.e., for all i, $\mathcal{G}_i \geq \frac{(i-1)k}{n}$.

Lemma 3. *For all $\log k > n$, $\Pr[Q] > 1 - ne^{-\frac{k}{2(n+1)n^3}}$.*

Proof.

$$\Pr[Q] = \prod_{i=1}^{n} \Pr\left[\mathcal{G}_i \geq \frac{k(i-1)}{n}\right] =$$

$$= \prod_{i=1}^{n} \left(1 - \Pr\left[\mathcal{G}_i < \frac{k(i-1)}{n}\right]\right) \geq \prod_{i=1}^{n} \left(1 - e^{-\frac{k}{2(n+1)n^3}}\right).$$

We apply Lemma 2 to derive the last inequality above. We choose the smallest term in the product and raise it to the n power for the bound:

$$\Pr[Q] \geq \left(1 - e^{-\frac{k}{2(n+1)n^3}}\right)^n > 1 - ne^{-\frac{k}{2(n+1)n^3}}.$$ □

Using the previous lemma it follows that the probability for which $\overline{Q}$ holds is exponentially small.

Corollary 3. *For all $\log k > n$, $\Pr[\overline{Q}] < ne^{-\frac{k}{2(n+1)n^3}}$.*

After bounding the probability that all the Bernoulli peers will have more chunks than the discrete peers, we analyze the probability that the Bernoulli peers will fail, i.e., some chunk is missing. Let $\mathcal{Y}^j = \max\{\mathcal{X}_1^j, \mathcal{X}_2^j ..., \mathcal{X}_n^j\}$, $\mathcal{Z} = \min\{\mathcal{Y}^1, ..., \mathcal{Y}^k\}$. Note that $\mathcal{Z} = 0$ is equivalent to say that the Bernoulli peers will fail.

The following lemmata lead to the last corollary that shows under which condition the network goes to miss some chunk, i.e., it is not able to deliver any further complete download.

Lemma 4. *The probability that the Bernoulli peers will fail is,*

$$\Pr[\mathcal{Y}^j = 0] > \frac{1}{e^{n+1}}.$$

Proof. The proof follows from the following computation.

$$\Pr[\mathcal{Y}^j = 0] = \prod_{i=1}^{n} \frac{i+1}{n+1} = \frac{(n+1)!}{(n+1)^n} > \frac{\frac{(n+1)^{n+1}}{e^{n+1}}}{(n+1)^{n+1}} = \frac{1}{e^{n+1}}. \qquad \square$$

Lemma 5.

$$\Pr[\mathcal{Z} > 0] < e^{-\frac{k(n+1)}{e^{n+1}}}.$$

Proof. In order to prove the claim we make use of Lemma 4, followed by the limit definition of e and then Sterling's approximation, hence obtaining

$$\Pr[\mathcal{Z} > 0] = \prod_{j=1}^{k} \left(1 - \Pr[\mathcal{Y}^j = 0]\right) = \left(1 - \frac{(n+1)!}{(n+1)^n}\right)^k \cong e^{-\frac{k(n+1)!}{(n+1)^n}} < e^{-\frac{k(n+1)}{e^{n+1}}}.$$

$\square$

Lemma 6. *For* $\log k > n$*,*

$$\Pr[Z = 0] \geq 1 - e^{-\frac{k(n+1)}{e^{n+1}}} - ne^{-\frac{k}{2(n+1)n^3}}.$$

Proof. From Bayes Law and the complementary events property,

$$\Pr[Z = 0] > \Pr[\mathcal{Z} = 0|Q] = \Pr[\mathcal{Z} = 0] - \Pr[\overline{Q}]\Pr[\mathcal{Z} = 0|\overline{Q}] >$$

$$> \Pr[\mathcal{Z} = 0] - \Pr[\overline{Q}] \geq 1 - e^{-\frac{k(n+1)}{e^{n+1}}} - ne^{-\frac{k}{2(n+1)n^3}}. \qquad \square$$

Corollary 4. *For all* $k \geq \frac{\log n}{n+1} e^{n+1}$*,*

$$\lim_{n\to\infty} \Pr[Z = 0] = 1.$$

From the previous corollary it follows that if the number of chunks k is bigger than or equal to $\frac{\log n}{n+1} e^{n+1}$, then the expected survival time is constant.

4 Survivability Scenario

In this section we study how likely the network is to survive in extreme conditions. With extreme conditions we mean that a file is almost never downloaded since peers have a very high volatility and almost never stay in the network long enough to perform a full download. However, it is still very interesting to note that the network is able to survive. We remind that a network survives whenever it still contains all the k chunks composing the original file, no matter where they reside. We will assume that peers leave the system with probability α while

roots or *experts* (peers that succeed at the full download) leave the network with probability α_R.

During the process chunks are duplicated and hence created while others disappear because of nodes leaving the system. Let us denote by N the number of nodes (peers plus experts, assuming roots as experts) in the network and by P_i the probability (percentage of peers) that a peer has i chunks. The number of chunks lost during one time step is then:

$$N\left(\alpha\sum_{i=0}^{k-1} iP_i + \alpha_R P_k\right).$$

The amount C of chunks created, depends on how many successful download are performed during a step. This strongly depends on the chosen protocol. In any case this amount cannot be more than the number of peers with one packet $N(1-P_0)$ multiplied by the probability to stay inside the network, i.e., $C \leq (1-\alpha)N(1-P_0)$.

Hence a necessary condition for the network survival is that

$$N(1-P_0)(1-\alpha) \geq N\left(\alpha\sum_{i=0} k-1iP_i + \alpha_e P_k\right) \geq N\alpha(1-P_0)$$

and this leads to have $1-\alpha \geq \alpha$, i.e., $\alpha \leq \frac{1}{2}$.

When $\alpha = \frac{1}{2}$ the stationary distribution is as follows $P_0 = P_1 = \frac{1}{2}$, after each communication step all the nodes have a chunk, but then half of them die (reset to zero chunks). Note that such a network dies quite quickly, from deviations. However, as soon as $\alpha > \frac{1}{2}$ the network lives almost forever.

Our process is very similar to a birth and death process, each node lives on average $\frac{1}{1-\alpha}$ rounds. And at each round it generates $1-\alpha$ chunks, hence its average number of children is $\frac{1-\alpha}{\alpha}$. It holds that when this number is strictly greater than 1 the network survives.

Let us assume the network to be in some random state with $P_0 = P_1 = \frac{1}{2}$, with k chunks regularly spread across the peers with one chunk each. Let $F_{i,t}$ be a random variable that denotes the percentage of peers with the i-th chunk at time t. $F_{i,0}$ is deterministic with value $\frac{N}{2k}$.

After the first step of the protocol, we have:

$$P[F_{i,1} = k] = \binom{2F_{0,t}}{k}/2^{2F_{i,0}}$$

So after the initial step, chunks are distributed as the sum of $\frac{N}{2k}$ random bits.

We study this phenomenon at the critical point in its most canonical form. At time t we have a number S_t of chunks alive, we double each chunk and randomly destroy half of the chunks.

First we consider the future of a single chunk. Let $F_t(z)$ denote the generating series at time t associated with the future of chunk z.

Considering one time step we have: with probability $\frac{1}{4}$ the process dies, with probability $\frac{1}{2}$ the process restarts with 1 chunk and $t-1$ time units remain, with probability $\frac{1}{4}$ we get two chunks and $t-1$ time units remain.

$$F_t(z) = \frac{1}{4} + \frac{1}{2}F_{t-1}(z) + \frac{1}{4}F_{t-1}^2(z) = \left(\frac{F_{t-1}(z)+1}{2}\right)^2$$

Note that by setting $\varepsilon_t(z)_t = F_t(z) - 1$, we obtain

$$\varepsilon_t(z) = \varepsilon_{t-1}(z) + \left(\frac{\varepsilon_{t-1}(z)}{2}\right)^2$$

and $\varepsilon_t(0)$ is the probability to stop before time t. So the probability to have the process alive at time t is about $\frac{t}{\ln t}$, and if one considers n bits one needs $n \ln n$ time units to kill all of them.

This scenario is quite optimistic since one assumes that the network exchanges $N(1-P_0)$ chunks during a round which corresponds to a *perfect* situation. This happens, in fact, by means of a *perfect matching* between the nodes that meet their necessities. Indeed, if we make use of a non optimal strategy, i.e., matching the node in a perfectly arbitrarily way, it is worth noting that this does not affect and degrade the process too much.

A typical way to marry the nodes is to choose randomly for each node a server, and to elect a node randomly for each server. Despite the simplicity of this process (nodes which have all the chunks still demand some, and nodes with no chunks are considered as able to provide some), it does not affect the network survivability as we are going to see.

Note that, by means of a simple matching algorithm, the number of edges is already of order $(1-\frac{1}{e})N$. Among those edges some are unable to duplicate chunks (if the server does not have any chunks that the client needs). A critical stage occurs when almost all the nodes with chunks have almost no chunks, so the probability for an edge to be useful is indeed exactly $(1-P_0)$. In such a situation the number of chunks replicated is

$$\left(1-\frac{1}{e}\right)(1-P_0)(1-\alpha)$$

while the number of chunks destroyed is at least $\alpha(1-P_0)$. This implies that the network survives when $\alpha \leq \frac{1-\frac{1}{e}}{2-\frac{1}{e}} \simeq .3873$.

5 Simple Evaluation Model for P2P Networks

To better understand the behavior of this kind of network we propose a simplified model and protocol. Such a model can be easily applied and modified in order to find preliminary results on P2P networks. We have also compared our easy model's results to the outputs of more sophisticated simulators. Even though the comparison is outside the scope of this paper, it is worth mentioning that the deviation from the simulations is negligible.

- At the beginning of each time slot, each peer chooses another peer randomly. An inquired peer randomly selects one of its customers. We call this customer *lucky*. Next the uploading peer delivers to its lucky customer a useful random chunk, i.e., a chunk that he has and that this customer has not. The unlucky customers do not get any chunk at this round.
- In order to get a stable situation, each peer that disappears (with probability α) is immediately replaced by a new empty peer (i.e., with no chunks).

The probability *luck* that a customer gets lucky is indeed equal to the proportion of customers served, which is the number of peers having at least one customer. Hence, $luck = 1 - \frac{1}{e}$.

We first compute the probability that a node with i chunks gets a new useful one. To do this we make a strong assumption that chunks remains almost identically distributed, i.e., a random node with i chunks contains a given chunk with probability $\frac{i}{k}$.

To get a chunk, a node needs first to be lucky, then the probability that it gets a chunk depends on the number of chunks of its uploading peer. Assume that a customer with i chunks contacts an uploading peer with j chunks, its probability to receive a chunk is

$$\Delta_{i,j} = 1 - \frac{\binom{i}{j}}{\binom{k}{j}}$$

Note that we use the standard convention $\binom{i}{j} = 0$ whenever $j \geq i$, but in the case $i = j = 0$ we have $\Delta_{i,j} = 0$, so we consider $\binom{0}{0} = 1$. It follows that a lucky node in state i that does not vanish (i.e., conditioned on all those events) moves to states $i + 1$ with probability

$$\sum_{j=0*}^{k} \Delta_{ij} P_j$$

From that it follows that a peer in state $i \neq 0$ moves to:

- State $i + 1$ with probability $T_{i,i+1} = (1 - \alpha) luck \sum_{j=0}^{k} \Delta_{i,j} P_j$
- State i with probability $T_{i,i} = (1 - \alpha) - T_{i,i+1}$
- State 0 with probability α

To summarize we have:

$$P_0 = \alpha \sum_{i=0}^{k} P_j + (1 - \alpha) \left(1 - luck \sum_{j=0}^{k} \Delta_{0,j} P_j \right),$$

$\forall i > 0$,

$$P_i = (1 - \alpha) \left(P_i \left(1 - luck \sum_{j=0}^{k} \Delta_{i,j} P_j \right) + P_{i-1} \left(luck \sum_{j=0}^{k} \Delta_{i-1,j} P_j \right) \right).$$

For the aim of preliminary and experimental results such a model is already enough in order to get an idea of the general behavior of this class of networks.

6 Conclusion

In this paper we have studied the behavior of P2P networks. We have considered three main scenarios. In the first one there are some peers owning all the chunks (roots) composing a file and the aim is to study the time required to ensure that every chunk is spread out on the network. This is very important to understand since, of course, it reflects the required behavior for peers that want to share their information. We have shown that the success of the spreading phase depends on two main parameters. Namely, the number of roots in the network and the number of chunks in which the file is divided. The probability α_R for which R roots can leave the network should be smaller than $1 - e^{-\frac{R}{k}}$ where k is the number of chunks. In the second scenario, we have started with a configuration in which many peers have subsets of the whole chunk set and the aim is to study the probability for the network to survive, i.e., every chunk must belong to some peer. This is also very important since it gives a measure of the behavior that peers should exhibit in order to maintain the viability of their download and archival capabilities. We have shown that if k is much less than e^n, with n being the number of peers in the network, the expected survival time of the network is exponential in n. Moreover, if the number of chunks is greater than $\frac{\log n}{n+1}e^n + 1$, the network survival time is constant. The third proposed scenario concerns the critical setting for the peers in terms of volatility. We have shown how under this setting the network is still able to survive. Namely, our estimated maximum value of the probability α for which a peer can leave the network while guaranteing its survivability is $\alpha \leq .3873$. From the point of view of experimental results, we have also proposed a simple way for analyzing and modeling P2P networks.

Our study has raised many open questions that might be investigated for further research. Many variations of our proposed models are possible and interesting. An important issue, for instance, concerns file sharing protocols that cope with security aspects. Deep analysis of tit-for-tat strategies for avoiding the so called free-riders problem is of primary interest to better understand the success of these protocols (see [11] for preliminary results). Free-riders are users that download files from the network but do not share their own chunks. In BitTorrent, those kind of users are allowed even though the performance of their downloads is much slower than for "friendly" users.

References

1. Sun, T., Tamai, M., Yasumoto, K., Shibata, N., Ito, M., Mori, M.: MTcast: Robust and Efficient P2P-based Video Delivery for Heterogeneous Requirements. In: Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS). Lecture Notes in Computer Science, Springer-Verlag (2005)
2. Izal, M., Urvoy-Keller, G., Biersack1, E.W., Felber, P., Hamra, A.A., Garces-Erice, L.: Dissecting BitTorrent: Five Months in a Torrent's Lifetime. In: Proceedings of the 5th International Workshop on Passive and Active Network Measurement (PAM). Volume 3015 of Lecture Notes in Computer Science., Springer-Verlag (2004) 1–11

3. Cohen, B.: Incentives Build Robustness in BitTorrent. In: Proceedings of the 1st Workshop on the Economics of Peer-to-Peer Systems (Econ P2P). (2003)
4. Clevenot, F., Nain, P.: A Simple Fluid Model for the Analysis of the Squirrel Peer-to-Peer Caching System. In: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), IEEE Computer Society (2004)
5. Qiu, D., Srikant, R.: Modeling and performance analysis of bittorrent-like peer-to-peer networks. In: Proceedings of Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM), ACM Press (2004) 367–378
6. Ge, Z., Figueiredo, D.R., Jaiswal, S., Kurose, J., Towsley, D.: Modeling peer-peer file sharing systems. In: Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), IEEE Computer Society (2003)
7. Jain, K., Lovasz, L., Chou, P.A.: Building scalable and robust peer-to-peer overlay networks for broadcasting using network coding. In: Proceedings of the 24th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC), ACM Press (2005) 51–59
8. Massouli, L., Vojnovi, M.: Coupon replication systems. ACM SIGMETRICS Performance Evaluation Review **33**(1) (2005) 2–13
9. Arthur, D., Panigrahy, R.: Analyzing bittorrent and related peer-to-peer networks. In: Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm (SODA), ACM Press (2006) 961–969
10. Williams, D.: Probability with Martingales. Cambridge University Press (1991)
11. Jun, S., Ahamad, M.: Incentives in bittorrent induce free riding. In: Proceeding of the 3rd Workshop on Economics of Peer-to-Peer Systems (Econ P2P), ACM Press (2005) 116–121

Incentive-Based Robust Reputation Mechanism for P2P Services

Emmanuelle Anceaume and Aina Ravoaja*

IRISA, Campus Universitaire de Beaulieu, Rennes, France
{anceaume, aravoaja}@irisa.fr

Abstract. In this paper, we address the problem of designing a robust reputation mechanism for peer-to-peer services. The mechanism we propose achieves high robustness against malicious peers (from individual or collusive ones) and provides incentive for participation. We show that the quality of the reputation value of trustworthy and participating peers is always better than the one of cheating and non participating ones. Finally we formally prove that, even when a high fraction of peers of the system exhibits a collusive behavior, a correct peer can still compute an accurate reputation mechanism towards a server, at the expense of a reasonable convergence time.

Keywords: Reputation, credibility, free-riding, collusion, peer-to-peer systems.

1 Introduction

With the emergence of e-commerce in open, large-scale distributed marketplaces, reputation systems are becoming attractive for encouraging trust among entities that usually do not know each other. A reputation system collects, distributes, and aggregates feedback about the past behavior of a given entity. The derived reputation score is used to help entities to decide whether a future interaction with that entity is conceivable or not. Without reputation systems, the temptation to act abusively for immediate gain can be stronger than the one of cooperating. In closed environments, reputation systems are controlled and managed by large centralized enforcement institutions. Designing reputation systems in P2P systems has to face the absence of such large and recognizable but costly organizations capable of assessing the trustworthiness of a service provider. The only viable alternative is to rely on informal social mechanisms for encouraging trustworthy behavior [9]. Proposed mechanisms often adopt the principle that "you trust the people that you know best", just like in the word-of-mouth system, and build transitivity trust structures in which credible peers are selected [19,20,21]. However such structures rely on the willingness of entities to propagate information. Facing free-riding and more generally under-participation is a well known problem experienced in most open infrastructures [2]. The efficiency and accuracy of a reputation system depends heavily on the amount of feedback it receives from participants. According to a recognized principle in economics, providing rewards is an effective way to improve feedback. However rewarding participation may also increase the incentive for providing false information.

* Aina Ravoaja is partially supported by a grant from Brittany region.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 305–319, 2006.

Thus there is a trade-off between collecting a sizable set of information and facing unreliable feedback [7]. An additional problem that needs to be faced with P2P systems, is that peers attempt to collectively subvert the system. Peers may collude either to discredit the reputation of a provider to lately benefit from it (bad mouthing), or to advertise the quality of service more than its real value to increase their reputation (ballot stuffing). Lot of proposed mechanisms break down if raters collude [8].

In this paper we address the robust reputation problem. Essentially this problem aims at motivating peers to send sufficiently honest feedback in P2P systems in which peers may free-ride or be dishonest. This work has been motivated by a previous one in which the proposed architecture is built on top of a supervising overlay made of trusted peers [3]. The mechanism we propose achieves high robustness to attacks (from individual peers or from collusive ones), and provides incentive for participation. This is accomplished by an aggregation technique in which a bounded number of peers randomly selected within the system report directly observed information to requesting peers. Observations are weighted by a credibility factor locally computed. Incentive for participation is implemented through a fair differential service mechanism. It relies on peer's level of participation, a measure of peers' contribution over a fixed period of time, and on the credibility factor, assessing the confidence one has in a peer.

Our results are promising: We prove that through sufficient and honest cooperation, peers increase the quality of their reputation mechanism. We show that the reputation estimation efficiently filters out malicious behaviors in an adaptive way. Presence of a high fraction of malicious peers does not prevent a correct peer from computing an accurate reputation value, at the expense of a reasonable convergence time. Furthermore, the trade-off between the sensitivity of the mechanism facing up malicious peers and the duration of the computation is tuned through a single input parameter. These properties, combined with the incentive scheme, makes our mechanism adapted to P2P networks. Finally, we provide a full theoretical evaluation of our solution. For space reasons proofs of correctness are given in the full version of the paper [4].

The rest of the paper is organized as follows. In Section 2 related work is reviewed. Section 3 presents the model of the environment, and the specification of the robust reputation problem. Section 4 presents the incentive-based mechanism. Section 5 analyses its asymptotic behavior, its resistance to undesirable behavior and its convergence time.

2 Related Work

There is a rapidly growing literature on the theory and applications of reputation systems, and several surveys offer a large analyze of the state of art in reputation systems [15,11,8]. According to the way ratings are propagated among entities and the extent of knowledge needed to perform the needed computations, reputation systems fall into two classes, namely centralized or distributed. An increasing number of online communities applications incorporating reputation mechanisms based on centralized databases has recently emerged. The eBay rating system used to find traders allows partners to rate each other after completion of an auction. Despite its primitive reputation system, ebay is the largest person-to-person online auction with more than 4 millions auctions open at a time [16]. Regardless of this success, centralized approaches (see for

example, [21,18]) often pay little attention to misbehaving entities by assuming that entities give honest feedback to the requesting entity. More importantly, they rarely address non-participation and collusive behaviors.

Regarding decentralized p2p architecture, several research studies on reputation-based P2P systems have emerged. Among the first ones, Aberer and Despotovic [1] propose a reputation mechanism in which trust information is stored in P-Grid, a distributed hash table-based (DHT) overlay. Their mechanism is made robust by guaranteeing that trust information is replicated at different peers, and thus can be accessed despite malicious entities. However, the efficiency of their approach relies on peers propensity to fully cooperate by forwarding requests to feed the P-Grid overlay. Additionally, as for most of the DHT-based approaches, peers have to store data they are not concerned with. Thus, malicious peers may discard it to save private resources, leading to a loss of information. Other systems relying on the trust transitivity approach face false ratings by assuming the presence of specific faithful and trustworthy peers (e.g. [13]), or by weighting second-hand ratings by senders' credibility [7,19,20]. Opposed to the aforementioned works, Havelaar reputation system [10], exploits long-lived peers by propagating reports between sets of well defined peers identified through hash functions. A report contains the observations made during the current round, the aggregated observations made by the predecessors during the previous round, and so on for the last r rounds. By relying on such an extensive aggregation, false reports hardly influence the overall outcome. Furthermore by using hash functions collusion is mostly prevented. The efficiency of their approach mainly relies on the readiness of peers to store and propagate large amount of data, and to remain in the system for relatively long periods of time. To motivate peers to participate, Jurca and Faltings [12] propose an incentive-compatible mechanism by introducing payment for reputation. A set of brokers, the R-agents, buy and sell feedback information. An entity can receive a payment only if the next entity reports the same result. Weakness of such an approach is the centralization of the whole information at R-agents, and its robustness against malicious R-agents. Finally, Awerbuch et al. [5,6] give lower bounds on the costs of the probes made by honest peers to find good objects in eBay-like systems, and propose algorithms that nearly attain these bounds.

In contrast to these works, we propose a fully distributed mechanism based on local knowledge that provides malicious and non-participating entities an incentive for participation and honest behavior.

3 Model

3.1 A P2P Service Model

We consider a P2P service system where *service providers* (or *servers*) repeatedly offer the same service to interested *peers*. We assume that the characteristics of a server (capabilities, willingness to offer resources, etc) are aggregated into a single parameter θ called type. This type influences the *effort* exerted by the server through a cost function c. The effort determines the Quality of the Service (QoS) provided by the server. We assume that the effort exerted by a server is the same for all the peers that solicit him and takes its value within the interval $[0, 1]$.

Definition 1 (Effort). *The effort of a service provider s is a value q_s^* that determines the quality of the service offered to the peers that interact with s.*

After each interaction with server s, each client (or peer) has an imperfect observation of the effort exerted by s. Peers may have different tastes about a server QOS. But basically these observations are closely distributed around s's effort. Thus, we reasonably assume that an observed quality of service takes its value within the interval $[0, 1]$ and follows a normal distribution of mean q_s^* and variance σ_s^*.

Definition 2 (Observed Quality of Service Level). *The Observed Quality of Service Level of a service provider s observed by peer p at time t is a value $obs_p^s(t)$ which is drawn from a normal distribution over $[0, 1]$ with mean q_s^*. The value 1 (resp. 0) characterizes the maximal (resp. minimal) satisfaction of p.*

Estimation of the expected behavior of a server is based on its recent past behavior, that is, its recent interactions with the peers of the system. Such a restriction is motivated by game theoretic results and empirical studies on ebay that show that only recent ratings are meaningful [8]. Thus, in the following, only interactions that occur within a sliding window of width D [1] are considered. This approach is also consistent with Shapiro's work [17] in which it is proven that in an environment where peers can change their effort over time the efficiency of a reputation mechanism is maximized by giving higher weights on recent ratings and discounting older ratings. Using a sliding time window is approximately equivalent to this model. Every time a peer p desires to interact with a server s, p asks for feedback from peers that may have directly interacted with s during the last D time units. We adopt the terminology *witness* to denote a peer solicited for providing its feedback. If $\mathcal{P}_p^s(t)$ represents the set of peers k whose feedback has been received by peer p by time t, then the reputation value of a server is defined as follows:

Definition 3 (Reputation Value). *The reputation value $r_p^s(t)$ of server s computed by peer p at time t is an estimation of the effort q_s^* exerted by s based on the feedbacks provided by the peers in $\mathcal{P}_p^s(t)$.*

3.2 Specification of Undesirable Behaviors

In practice, peers may not always reveal their real ratings about other peers. They can either exaggerate their ratings (by increasing or decreasing them), or they can simply reveal outright ratings to maximize their welfare. This behavior is usually called *malicious*, and can either be exhibited by a node independently from the behavior of other peers, or be emergent of the behavior of a whole group. By providing false ratings, malicious peers usually try to skew the reputation value of a server to a value which is different from its true effort. Let $\overline{q}$ be this value, and d be the distance between the true effort of the server and the false rating ($d = |q_s^* - \overline{q}|$). Then, we characterize the behavior of a peer by $w_q^s = 1 - d^\alpha$, with α a positive real value which represents the sensitivity of w_q^s to the distance between the effort and the expected observation given by q. A malicious peer tries to skew the reputation value to $\overline{q}$ by sending ratings that are distributed around $\overline{q}$.

[1] D can have any pre-defined length of time, i.e., a day, a week or a month. In the sequel, we suppose that D is insensitive to clock drift.

Definition 4 (Malicious). *A peer p is called malicious if it lies on the reputation of peer s or creates s's reputation out of thin air. Formally :*

$$\mathrm{E}(obs_p^s(t)) = \overline{q} \neq q_s^*, \forall t.$$

Definition 5 (Collusive Group). *A group of peers is called a collusive group if all the peers of this group behave maliciously towards a same goal. Formally, the set $\mathcal{C}$ is a colluding group if :*

$$\mathrm{E}(obs_k^s(t)) = \overline{q} \neq q_s^*, \forall t, k \in \mathcal{C}.$$

Another common behavior in P2P systems is peers non-participation. There are two main reasons why peers may not participate: either because they believe that their work is redundant with what the others in the group can do, and thus their participation can hardly influence the group's outcome, or because they believe that by not contributing they maximize their own welfare (note that information retention could be another pretense of not participating, however this is out of the scope of the paper). The latter behavior depicts what is typically called *free-riding*, while the first one is described in the Collective Effort Model (CEM) as *social loafing* [14]. Note that although effects of both behaviors are similar, i.e., "non-participation", their deep cause is different. Peers exhibiting one of these two behaviors are called in the following *non-participating* peers and are characterized as follows:

Definition 6 (Non Participating). *A peer is called* non participating *if it exerts less effort on a collective task than it does on a comparable individual task or consumes more than its fair share of common resources.*

A peer is called *correct* if during the time it is operational in the system it is neither malicious nor non-participating. Note that a malicious peer may not participate, on the other hand, a non participating one is not malicious.

3.3 Specification of the Robust Reputation Problem

Within this context, we address the problem of evaluating the reputation of a service provider in a dynamic environment in which peers are not necessary correct. This problem is referred in the sequel as the *robust reputation problem*. A solution to this problem should guarantee the following two properties. The first one states that eventually correct peers should be able to estimate the reputation value of a target server with a good precision. The second one says that with high probability, correct peers have a better estimation of the reputation value of a target server than non correct ones. Formally:

Property 1 (Reputation Value ϵ-Accuracy). Eventually, the reputation of server s, evaluated by any correct peer reflects s's behavior with precision ϵ. That is, let $\beta \in]0, 1[$ be some fixed real, called in the sequel confidence level, then:

$$\exists \overline{t} \; s.t. \; \forall t \geq \overline{t}, \; \mathrm{Prob}(|r_p^s(t) - q_s^*| \leq \epsilon) \geq 1 - \beta$$

Let $|\mathrm{E}(r_p^s(t)) - q_s^*|$ be the bias of the reputation value $r_p^s(t)$ estimated by peer p. Suppose that two peers p and q interact with the same target servers at the same time, solicit the same witnesses, and get the same feedbacks at the same time. That is from the point of view of their interaction p and q are indistinguishable. However p is correct while q is not. Then, we have:

Property 2 (Incentive-Compatibility). Eventually, the bias of the reputation value of server s estimated by p is greater than or equal to the one estimated by peer q. That is, for a given level of confidence β, we have:

$$\exists \overline{t}\ s.t.\ \forall t \geq \overline{t},\ \mathrm{Prob}(|\mathrm{E}(r_p^s(t)) - q_s^*| \geq |\mathrm{E}(r_q^s(t)) - q_s^*|) \geq 1 - \beta$$

4 The Reputation Mechanism

We propose a distributed reputation service which builds a social network among peers. Briefly, every peer records the opinion about the late experiences it has had with a target server. Peers provide their information on request from peers willing to interact with that server. Providing a feedback based on direct observations (also called first-hand observations) prevents the *rumors* phenomenon, i.e., the propagation of opinions about others, just because they have been heard from someone else [19], however is better adapted to applications with modest churn. Upon receipt of "enough" feedback, the requesting peer aggregates them with its own observations (if any) to estimate the reputation of the target server, and provides this estimation to its application. Information is aggregated according to the trust the requesting peer has in the received feedback. Pseudo-code of the reputation mechanism is presented in Algorithm 1. The efficiency of the reputation mechanism fully depends on *i)* the number of received feedbacks (i.e., aggregating few feedbacks is not meaningful and thus not helpful), and *ii)* the quality of each of them (i.e., the trustworthiness of the feedback). The contribution of this work is the design of a reputation mechanism that enjoys both properties. The analysis presented in Section 5 shows the importance of each factor on the convergence time and accuracy of the reputation mechanism.

The solution we propose is a reputation mechanism, and therefore independent of the rewarding strategy used by the application built on top of this mechanism. That is, the willingness of a peer to interact with a server results from the application strategy, not from the peer's one. Clearly, the strategy of the application is greatly influenced by the reputation value but other factors may also be taken into account.

4.1 Collecting Feedbacks

When a peer decides to evaluate the reputation value of a service provider, it asks first-hand feedback from a set of witnesses in the network. Finding the right set of witnesses is a challenging problem since the reputation value depends on their feedback. Our approach for collecting feedbacks follows the spirit of the solution proposed by Yu et al [20], in which feedbacks are collected by constructing chains of referrals through which peers help one another to find witnesses. We adopt the walking principle.

However, to minimize the ability of peers to collude, witnesses are randomly chosen within the system. We assume, in the following, that the network is regular. Specifically, our approach is based on a random walk (RW) sampling technique. We use the random walk technique as shown in Algorithm 1. Function `query` is invoked by the requesting peer that wishes to solicit x witnesses through r random walks bounded by ttl steps. The requesting peer starts the random walks at a subset of its neighbors, and runs them for ttl steps. Each peer p involved in the walk is designated as witness, and as such sends back to the requesting peer its feedback. When a peer q receives a request from p to rate server s, it checks whether during the last sliding window of length D, it has ever interacted with s. In the affirmative, p sets its feedback to $F_p^s(t) = \{(obs_p^s(t_0), t_0), \ldots, (obs_p^s(t_l), t_l)\}$ with $obs_p^s(t_i)$ the QoS of s observed at time t_i, where $t_i \in [\max(0, t - D), t]$. In case p has not recently interacted with s, p sends back to q a default feedback $F_p^s(t) = \{(obs_{max}, \perp)\}$. As will be shown later, this feedback prevents p from being tagged "non participant" by q.

Because of non-participation (volunteer or because of a crash), random walks may fail: it suffices that one of the peers in the walk refuses to participate or crashes to prevent the walk from successfully ending. If we assume that among its d neighbors, a fraction μ of them do not participate, then among the r initial peers that start a random walk, the expected number of peer that may "fail" their random walk is μr. Then during the next step, $\mu(1-\mu)r$ walks may "fail", and so on until the TTL value is reached. In consequence, only x feedbacks may be received, with $x = \sum_{t=1}^{ttl}(1-\mu)^t r$. By setting r to $\frac{x}{\sum_{t=1}^{ttl}(1-\mu)^t}$ the requesting peer is guaranteed to receive at least x feedbacks (see line 5 in Algorithm 1). In addition to its feedback, each peer sends to the requesting peer p (through the witness message, see lines 19 and 46) the identity of the next potential witness on the walk, i.e., the peer it has randomly chosen among its neighbors. Sending this piece of information allows p to know the identity of all potential witnesses. As will be shown in Section 4.4, this allows to detect non participants (if any) and so to motivate their participation through a "tit-for-tat" strategy. As for feedbacks, non participation may prevent the requesting peer from receiving witness messages. A similar analysis to the preceding one shows that if r random walks are initiated then $y = \sum_{t=0}^{ttl-1}(1-\mu)^t r$ witness messages will be received.

Note that a requesting peer can adapt its collect policy according to its knowledge of the target server, or of its neighborhood. Specifically, to get x witnesses, a peer can either increase ttl and restrict r, or increase r and lower ttl (assuming that $r > \mu d$ holds). Enlarging ttl would be more sensitive to colluding peers that bias the random walk. However, this technique would increase the set of crawled witnesses, and thus would afford new peers the opportunity to be known by other peers and consequently to increase both their participation and their credibility. Conversely, enlarging r would crawl only peers in the neighborhood of the requesting peer. However, this technique would increase the chance to find a path that does not contain colluding peers [2].

[2] Remark that selecting peers according to their credibility should be more efficient in the sense that only "highly" credible peers would be selected, however, newcomers may be penalized by this filtering. Furthermore, the resilience of the crawling technique to collusion highly relies on the way the graph of witnesses is constructed. Studying these issues is part of our future work.

4.2 Reputation of a Server

Estimation of the reputation value of a target server is based on the QoS directly observed at the server (if any) and on the feedbacks received during the collect phase. The accuracy of the estimation depends on the way these informations are aggregated. The aggregation function we propose answers the following qualitative and quantitative preoccupations: First, to minimize the negative influence of unreliable information, feedbacks are weighted by the credibility of their senders. Briefly, credibility is evaluated according to the past behavior of peers and reflects the confidence a peer has in the received feedback. Credibility computation is presented in the next subsection. Second, to prevent malicious nodes from flooding p with fake feedback and thus from largely impacting the accuracy of its estimation, p keeps only a subset of each received feedback. More precisely, among the set of observations sent by each witness over the last D time units, only the last f ones are kept, with f the size of the smallest non-empty set of non-default feedbacks received by p (i.e., $f = \min_{k \in \mathcal{P}_p^s(t)}(|F_k^s(t)|)$ with $t \in [\max(0, t-D), t]$). Finally, if among all the witnesses (including p) none has recently directly interacted with s (i.e., $f = 0$), then p affects a maximal value obs_{max} to s's reputation value. Affecting a maximal value reflects the key concept of the Dempster-Shafer theory of evidence which argues that "there is no causal relationship between a hypothesis and its negation, so lack of belief does not imply disbelief". In our context, applying this principle amounts in fixing an *an priori* high reputation to unknown servers, and then updating the judgment according to subsequent interactions and observations [20].

We can now integrate these principles within the aggregation function we propose. Let us first introduce some notations: Let $\mathcal{F}_k^s(t)$ be the union of the last f non-default feedbacks received from k during the last D time units ($t \in [\max(0, t-D), t]$); $\mathcal{P}_p^s(t)$ be the set of witnesses k for which $\mathcal{F}_k^s(t)$ is non empty; $\rho_k^s(t)$ represent the mean value of the observations drawn from $\mathcal{F}_k^s(t)$; and $c_{p,k}^s(t)$ the credibility formed by p at time t about k regarding s. Then, at time t, p estimates s's reputation value as follows:

$$r_p^s(t) = \begin{cases} \frac{1}{\sum_{k \in \mathcal{P}_p^s(t)} c_{p,k}^s(t)} \sum_{k \in \mathcal{P}_p^s(t)} c_{p,k}^s(t).\rho_k^s(t) & \text{if } f \neq \emptyset \\ obs_{max} & \text{otherwise} \end{cases} \quad (1)$$

with,

$$\rho_k^s(t) = \frac{1}{f} \sum_{(obs_k^s(t'), t') \in \mathcal{F}_k^s(t)} obs_k^s(t')$$

4.3 Trust in Witnesses

In this section, we tackle the issue of malicious peers. As remarked in the Introduction, malicious peers may alter the efficiency of the reputation mechanism by sending feedbacks that over-estimate or sub-estimate the observed QoS of a server to inflate or tarnish its reputation. This is all the more true in case of collusion. We tackle this issue by evaluating peers credibility. Credibility is a [0,1]-valued function which represents the confidence formed by peer p about the truthfulness of q's ratings. This function is local and is evaluated on the recent past behavior of both p and q peers. It is locally used to prevent a false credibility from being propagated within the network. Specifically, peer p estimates at time t how credible q is regarding server s as a decreasing

function of the distance between q's feedbacks on s's effort and p's direct observations on s's QoS. As for the reputation value computation, the distance is computed on the last f observations made by both p and q during the last D time units. Note that in case p has not recently observed s's QoS, then credibility of all its witnesses are set to a default value c_0. Indeed, p cannot evaluate the distance between its own observations and those observed by witnesses. Determining c_0 value needs to solve the following trade-off: by affecting a high value to the default credibility one increases the vulnerability of the system to the *whitewashing* phenomenon, that is, the fact that peers change their identity in order to reset their credibility to the default value. However, by setting this variable to a low value the mechanism tends to filter out new witnesses and thus, loses the benefit of the potential information a new peer can afford, which clearly decreases the usefulness of the reputation mechanism. In order to cope with that, we set c_0 to the value of a decreasing function of ϕ, with ϕ an estimation of the number of whitewashers in the network. By adopting the notations of Equation 1, $c^s_{p,q}(t)$ represents the credibility formed by p at time t about q regarding the target server s, and is given by:

$$c^s_{p,q}(t) = \begin{cases} 1 - |\rho^s_q(t) - \rho^s_p(t)|^\alpha & \text{if } f \neq \emptyset \\ c_0 & \text{otherwise} \end{cases} \quad (2)$$

where $|\rho^s_q(t) - \rho^s_p(t)|^\alpha$ represents the distance between q and p's observations. Note that α is the variable introduced in Section 3. Then we have the following lemma:

Lemma 1. ***(Credibility Accuracy)*** *Eventually, credibility of a peer q evaluated by any correct peer p reflects q's behavior with a precision ϵ. That is, let $\beta \in]0,1[$ be some fixed real, there exists $\overline{t}$ such that, for all $t \geq \overline{t}$,*

$$\text{Prob}(|c^s_{p,q}(t) - w^s_q(t)| \leq \epsilon) \geq 1 - \beta.$$

4.4 Incentive for Participation

Non participation may jeopardize the efficiency of the reputation mechanism. A certain amount of participation is required before reputation can induce a significant level of cooperation. Facing non-participation in the reputation problem is challenging and has deserved few attention [20]. To motivate peers to send their feedback we adopt a "tit-for-tat" strategy. We introduce the level of participation notion as the propensity of a peer for replying to a rating request. It is described by function $l^s_{p,q}$ such that $l^s_{p,q}(t)$ represents the percentage of times q provided its feedback to p's queries regarding server s's QoS over the last D time units, with $l^s_{p,q}(t=0) = l_0 = 1$. Its computation is performed after p's collect phase (see line 10 of the algorithm. Note that factor μ prevents correct peers from being penalized by walking breaks.).

We apply the tit-for-tat strategy during the collect phase. When a peer p receives a rating request for s from peer q, then with probability $l^s_{p,q}(t)$ p provides its feedback to q, otherwise it sends a default feedback $(\perp, \perp)$ to prevent p from being tagged as non-participant. By providing this default feedback, p lets q knows that its recent non-participation has been detected. Consequently, by not participating, requesting peers drive correct witnesses providing them worthless feedback, which clearly makes their reputation mechanism useless. Hence there is a clear incentive for non participating

peers to change their behavior. The following lemma proves that participation decreases the bias of the reputation value. As previously, let us consider two peers p and q such that both peers are indistinguishable from the point of view of the servers with which they interact, that is both p and q observe the same QoS from these servers at the very same time, solicit and are solicited by the same set of peers at the same time; However, p is correct while q is non participating. Then we claim that:

Lemma 2. *Participation decreases the bias of the reputation value. That is,*

$$|\mathrm{E}(r_p^s(t)) - q_s^*| \leq |\mathrm{E}(r_q^s(t)) - q_s^*|$$

4.5 Incentive for Truthful Feedbacks

We now address the problem of motivating peers to send truthful feedbacks. So far we have presented strategies aiming at improving the quality of the reputation value estimation by aggregating more feedbacks and by weighting feedback according to the credibility of their sender. We have shown that by using both strategies, utility of correct peers increases. However, none of these solutions have an impact on the effort devoted by a witness to send a truthful feedback. To tackle this issue we use the credibility as a way to differentiate honest peers from malicious ones. As for non-participating peers, when peer p receives a request to rate server s from a requesting peer q then p satisfies q's request with probability $c_{p,q}^s(t)$. By doing so, p satisfies q's request if it estimates that q is trustworthy, otherwise it notifies q of its recent faulty behavior by sending it the $(\perp, \perp)$ feedback. As previously, by cheating, a malicious peer penalizes itself by pushing correct witnesses to send meaningless feedbacks to it, leading to its effective isolation. We claim that this social exclusion-based strategy motivates q to reliably cooperate.

Lemma 3. *High credibility decreases the bias of the reputation value. That is,*

$$|\mathrm{E}(r_p^s(t)) - q_s^*| \leq |\mathrm{E}(r_q^s(t)) - q_s^*|$$

Finally, to elicit sufficient and honest participation, both strategies are combined, i.e., upon receipt of a rating request from peer q, with probability $\min(c_{p,q}^s(t), l_{p,q}^s(t))$ p provides its feedback, otherwise it sends the default feedback $(\perp, \perp)$ (see line 31 in Algorithm 1).

Theorem 1. *The reputation mechanism described in Algorithm 1 is Incentive-Compatible in the sense of Property 2.*

5 Analysis

Computing the reputation of a peer reduces to estimating, in the statistical sense, its effort. Our algorithm falls into the category of robust estimation algorithms. Indeed, robust estimation techniques consider populations where a non-negligible subset of data deliberately pollute the system. This analysis describes the asymptotic behavior of the reputation mechanism and its convergence time according to undesirable behaviors. In the following, we assume that a fraction γ of witnesses are malicious.

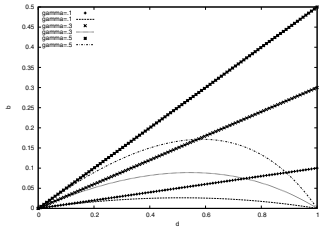


Fig. 1. Bias for $\alpha = 1$ and $n = 10$

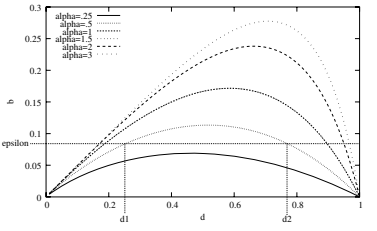


Fig. 2. Bias for $\gamma = .5$ and $n = 10$

5.1 Asymptotic Behavior

In this section we determine the accuracy of the reputation mechanism with respect to parameters α and ϵ. Thus for the purpose of this analysis, we assume that the number of aggregated feedbacks f is infinite. Recall that w_q denotes the characterization of q's behavior, with $w_q = 1$ if q is correct, and $w_q = 1 - d^\alpha$ otherwise. By Lemma 1, $c^s_{p,q}(t) = w_q$, when $t \to \infty$. Moreover, the expected number of correct witnesses is $(1-\gamma)n+1$ while the expected number of malicious ones is γn, with n the expected number of witnesses (Figures are plotted for $n = 10$). Thus, by replacing $c^s_{p,q}(t)$ with their asymptotic values in Equation 1, the expected reputation value of a server s estimated by a correct peer p when $t \to \infty$ is given by Equation 3:

$$r^s_p(t)_{t\to\infty} = \frac{1}{1-\gamma d^\alpha + \frac{1}{n}} \left(\left(1 - \gamma + \frac{1}{n} \right) q^*_s + \gamma \left(1 - d^\alpha \right) \overline{q} \right). \tag{3}$$

The bias of the reputation value, when $t \to \infty$ is given by the following equation:

$$|r^s_p(t)_{t\to\infty} - q^*_s| = \gamma d \frac{1 - d^\alpha}{1 - \gamma d^\alpha + \frac{1}{n}}. \tag{4}$$

Figures 1 and 2 show the bias of the reputation value with respect to d for increasing values of γ (resp. increasing values of α). Recall that $d = |q^*_s - \overline{q}|$ reflects colluders' behavior. Unlike mean-based reputation value estimation for which the bias linearly increases with d (as shown by the crossed curves in Figure 1), our algorithm bounds the power of colluders whatever their percentage (dotted curves). Indeed, witnesses' ratings are weighted by a decreasing function of d which filters out false ratings.

Figure 2 shows the impact of α on the bias of the reputation value. As can be observed, the bias decreases with decreasing values of α, reflecting the sensitivity of the reputation value to the distance between direct observations and received feedbacks. Thus, decreasing values of α makes the reputation mechanism very sensitive to false feedbacks.

Theorem 2. *The reputation value is ϵ-accurate, with $\epsilon > \gamma d \frac{1-d^\alpha}{1-\gamma d^\alpha + \frac{1}{n}}$.*

From the above, assuming an upper bound on γ, by setting the maximal bias to ϵ and solving the corresponding equation one can derive an upper bound on α under which the reputation value converges to the true effort with an accuracy level of ϵ. Hence, for

α with $\alpha \leq \overline{\alpha}$, the reputation value converges to the true effort with an accuracy level of ϵ, with $\overline{\alpha}$ given by:

$$\overline{\alpha} = \frac{\ln\left(\frac{n+1-\sqrt{(1-\gamma)n^2+(2-\gamma)n+1}}{\gamma n}\right)}{\ln(\epsilon)} \tag{5}$$

To conclude, one can always find a value of α such that eventually the reputation value is accurate. This parameter, however, significantly influences the convergence time of the algorithm. The next Section addresses this issue.

5.2 Convergence

In this section, we study the convergence time of the reputation mechanism. To do so, we assume that the ratings of a malicious peer are drawn from a normal distribution with mean $\overline{q}$ and variance $\overline{\sigma}$ over $[0, 1]$. This assumption includes a wide range of possible behaviors. Indeed, a small value of $\overline{\sigma}$ depicts peers that try to rapidly skew the reputation value to $\overline{q}$ by giving reports tightly distributed around $\overline{q}$. In contrast, a high value of $\overline{\sigma}$ depicts peers that try to hide their mischievous behavior to other peers by giving sparse reports. While the first behavior is easily detected, the second one hardly skew the reputation value to $\overline{q}$.

Recall that the reputation value is estimated by aggregating the last f interactions witnesses have had with the target server during a sliding time window of length D. Finding the optimal value of D is important. Indeed, it determines the resilience of the mechanism to effort changes and the confidence level in the estimation. The optimal value of D is the one for which the estimation is at most ϵ-far from the true effort with a given confidence threshold β. To determine such a value, let us first assume that f is known, and determine a lower bound on $\text{Prob}(|r_p^s(t) - q_s^*| \leq \epsilon)\ \forall t \in D$. Suppose that the credibility $c_k^s(t)$ is ϵ'-far from w_k^s for all the witnesses k. Then, because of Bayes' Theorem, we know that $\text{Prob}(|r_p^s(t) - q_s^*| \leq \epsilon) \geq \text{Prob}(|r_p^s(t) - q_s^*| \leq \epsilon|_{|c_k^s(t)-w_k^s(t)|\leq\epsilon',\forall k\in\mathcal{P}_p^s(t)}) \cdot \text{Prob}(|c_k^s(t) - w_k^s(t)| \leq \epsilon', \forall k \in \mathcal{P}_p^s(t))$. Remark that, assuming that the witnesses' credibility are ϵ'-far from w_k^s, the probability that $r_p^s(t)$ is ϵ-far from q_s^* is maximal under the following condition (C): credibility of correct witnesses is minimal, i.e., equal to $1 - \epsilon'$, and the one of malicious ones is maximal, i.e., equal to ϵ'. Then, we have:

$$\begin{aligned}\text{Prob}(|r_p^s(t) - q_s^*| \leq \epsilon) \geq \text{Prob}(|r_p^s(t) - q_s^*| \leq \epsilon|_{(C)}) \cdot \\ \text{Prob}(c_k^s(t) \geq 1 - \epsilon'|_{kcorrect})^{(1-\gamma)n} \cdot \text{Prob}(c_k^s(t) \leq \epsilon'|_{kmalicious})^{\gamma n}\end{aligned} \tag{6}$$

By Lemma 1, the probability that the witness's credibility is at most ϵ'-far from w_k^s converges to 1 when t, and thus f, increase. Thus, this bound approaches $\text{Prob}(|r_p^s(t) - q_s^*| \leq \epsilon)$ when f increases. Knowing the probability distribution of the reports, deriving a closed form of the lower bound can be done. Then, given a desired confidence threshold β for $\alpha \geq \overline{\alpha}$, solution of Equation 7 provides two threshold values of d ($d1$ and $d2$ on Figure 2) beyond which the false reports are eliminated:

$$\epsilon = \gamma d \frac{1 - d^\alpha}{1 - \gamma d^\alpha + \frac{1}{n}} \qquad \text{within } [0, 1]. \tag{7}$$

input : p: requesting peer; s: target server; x: expected number of feedbacks
output: r_p^s: estimation of s reputation value

$r \leftarrow \frac{x}{\sum_{l=1}^{ttl}(1-\mu)^l}$; $y \leftarrow \sum_{l=0}^{ttl-1}(1-\mu)^l r$;
$ttl \leftarrow$ default value; $t \leftarrow$ getTime(); let D be the the time interval [max(t-D,0),t];
`query` (*p,s,ttl,r,t*) ;
wait until *(($F_q^s(t)$ messages are received from x peers) and ($witness(s,w,t)$ messages are received from y peers))*;
$P^s(t) \leftarrow \bigcup \{q$ such that a feedback message is received from $q\}$;
$F^s(t) \leftarrow \bigcup \{F_q^s(t)$ for all $q \in P(t)\}$;
$W^s(t) \leftarrow \bigcup \{w$ for all $witness(s,w,t)$ that have been received$\}$;
foreach $k \in W(t)$ **do**
    $l_{p,k}^s(t) \leftarrow \min(\frac{\sum_{t\in D}(|\mathcal{P}_{|k}(t)|}{\sum_{t\in D}|W_{|k}(t)|}(1-\mu) + \mu.l_0, 1)$;
end
return $r_p^s(t)$ to application;
`query` (*p,s,ttl,r,t*) **begin**
    $New \leftarrow$ pick a random subset of r peers from p's neighbors;
    forall *($next \in New$)* **do**
        send a rw (p, s, $ttl-1$,t) message to $next$;
        send a witness (s,$next$,t) message to p;
        if *p has interacted with s at time $t_0,\ldots,t_l$ in the last D time units* **then**
            $F_p^s(t) \leftarrow \{(obs_p^s(t_0),t_0),\ldots,(obs_p^s(t_l),t_l),p\}$;
        else
            $F_p^s(t) \leftarrow \{(obs_{max},\perp),p\}$;
        end if
        send $F_p^s(t)$ to p;
    end
end
upon *(receipt of a* rw *(p,s,ttl,t) message at peer q)* **do**
    with *(probability* $\min(l_{p,q}^s(t), c_{p,q}^s(t))$ **do**
        if *q has interacted with s at time $t_0,\ldots,t_l$ in the last D time units* **then**
            $F_q^s(t) \leftarrow \{(obs_q^s(t_0),t_0),\ldots,(obs_q^s(t_l),t_l),q\}$;
        else
            $F_q^s(t) \leftarrow \{(obs_{max},\perp),q\}$;
        end if
    otherwise
        $F_q^s(t) \leftarrow \{(\perp,\perp),q\}$;
    end do
    send $F_q^s(t)$ to p;
    if *($ttl \neq 0$)* **then**
        $next \leftarrow$ pick one of q's neighbor with probability $\frac{1}{d}$, $d = q$'s neighbors #;
        send a rw (p,s,$ttl-1$,t) message to $next$;
        send a witness (s,$next$,t) to p;
    end if
end do

Algorithm 1. Estimation of the reputation value of server s by peer p

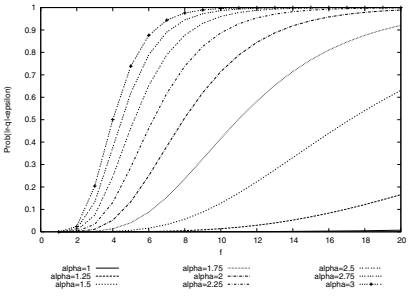


Fig. 3. Confidence level for $\gamma = .5$, $n = 10$, $\epsilon = .1$, $d = 1$, $\sigma_s^* = .2$, and $\overline{\sigma} = .2$.

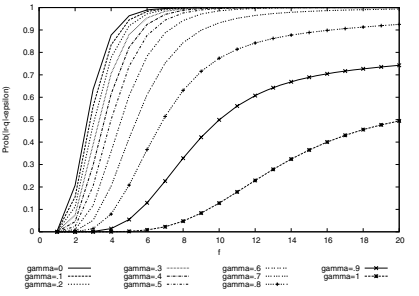


Fig. 4. Confidence level for $\alpha = 3$, $n = 10$, $\epsilon = .1$, $d = 1$, $\sigma_s^* = .2$, and $\overline{\sigma} = .2$.

Using Equation 6, we can see the effect of α on the convergence time of the reputation mechanism to reach the exact effort exerted by the server (see Figure 3). Decreasing values of α significantly increases the convergence time while it decreases the bias of the reputation mechanism as shown in Figure 2. Thus a trade-off exists between the robustness of the mechanism and its convergence time. By setting the value of D, the application designer may derive the corresponding minimal number of interactions f and finally tune the value of α such that the desired confidence level is achieved within f steps through Equation 6. The resulting reputation estimation is less sensitive to false reports, but still eliminates peers that try to skew the reputation of the server to a value that is far from the true effort, by filtering out extreme values of d (see Figure 2).

Finally, Figure 4 shows the impact of malicious peers on the convergence time. As can be seen, a relatively small percentage of malicious peers has a minor impact on the convergence time since the number of correct feedbacks is hardly influenced by false ones. On the other hand, whenever a requesting peer has to face a large proportion of malicious peers, it can only rely on its own feedback to estimate the effort exerted by the target server which clearly takes longer than when helped by correct witnesses. The same result applies for non-participating peers.

6 Conclusions

In this paper we have proposed a reputation mechanism that achieves high robustness to attacks and provides incentive for participation. This is achieved by an aggregation function in which a subset of the information provided by randomly chosen peers is kept and weighted by a confidence factor locally computed. We have proposed a simple and local incentive mechanism that guarantees a better quality of the reputation value estimation. Lessons learned from simulations are twofold: first, decreasing values of α guarantees a greater sensibility of the mechanism to false ratings. It however increases the number of required feedbacks as well, and thus the time to get an accurate estimation of the effort exerted by the target server. Second, the presence of a large number of malicious and non-participating peers does not prevent the mechanism from being accurate, however has an impact on its convergence time.

References

1. K. Aberer and Z. Despotovic. Managing trust in a peer-to-peer inforation system. In *Proc. of the Tenth Int'l ACM Conference on Information and Knowledge Management (CIKM)*, 2001.
2. E. Adar and B. Huberman. The impact of free-riding on gnutella. Online at http://www.firstmonday.org/issues/issue5_10/adar/index.html, 2000.
3. E. Anceaume, M. Gradinariu, and A. Ravoaja. Incentives for p2p fair resource sharing. In *Proc. of the Fifth IEEE Int'l Conf. on Peer-to-Peer Computing (P2P)*, 2005.
4. E. Anceaume and A. Ravoaja. Incentive-based robust reputation mechanism for p2p services. Technical Report 1816, IRISA, 2006.
5. B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Tuttle. Collaboration of untrusting peers with changing interests. In *Proc. of the Fifth ACM Conference on Electronic Commerce (EC)*, 2004.
6. B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Tuttle. Adaptive collaboration in peer-to-peer systems. In *Proc. of the Twenty-Fifth IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.
7. S. Buchegger and J.Y. Le Boudec. A robust reputation system for p2p and mobile ad-hoc networks. In *Proc. of the Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.
8. C. Dellarocas. Reputation mechanisms. In T. Hendershott, editor, *Handbook on Information Systems and Economics*. Elsevier Publishing, 2006.
9. Z. Despotovic. *Building trust-aware P2P systems : from trust and reputation management to decentralized e-commerce applications*. PhD thesis, EPFL, 2005.
10. D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer. Havelaar: A robust and efficient reputation system for active peer-to-peer systems. In *Proc. of the First Workshop on the Economics of Networked Systems (NetEcon)*, 2006.
11. A. Josang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 2006. Forthcoming. Accessible at http://sky.fit.qut.edu.au/ josang/publications.html.
12. R. Jurca and B. Faltings. An incentive compatible reputation mechanism. In *Proc. of the IEEE Conference on E-Commerce (CEC)*, 2003.
13. S. Kamvar, M. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proc. of the 12th Int'l Conference on World Wide Web (WWW)*, 2003.
14. S. Karau and K. Williams. Social loafing: A meta-analytic review and theoretical integration. *Journal of Personality and Social Psychology*, pages 681–706, 1993.
15. L. Mui, M. Mohtashemi, and A. Halberstadt. Notions of reputation in multiagent systems: a review. In *Proc. of the First ACM-SIGART Int'l joint conference on Autonomous agents and multiagent systems (AAMAS)*, 2002.
16. P. Resnick, R. Zeckhauser, J. Swanson, and K. Lockwood. The value of reputation on ebay: A controlled experiment. *Experimental Economics*, 2006.
17. C. Shapiro. Consumer information, product quality, and seller reputation. *Bell Journal of Economics*, 1981.
18. B. Yu and M. Singh. A social mechanism of reputation management in electronic communities. In *Proc. of the Seventh Int'l Conf. on Cooperative Information Agents*, 2000.
19. B. Yu and M. Singh. Detecting deception in reputation management. In *Proc. of the Second ACM-SIGART Int'l joint Conf. on Autonomous agents and multiagent systems (AAMAS)*, 2003.
20. B. Yu, M. P. Singh, and K. Sycara. Developing trust in large-scale peer-to-peer systems. In *Proc. of First IEEE Symposium on Multi-Agent Security and Survivability*, 2004.
21. G. Zacharia. Trust management through reputation mechanisms. In *Proc. of the Third Int'l Conf. on Autonomous Agents (Workshop on Deception, Fraud and Trust)*, 1999.

Searching for Black-Hole Faults in a Network Using Multiple Agents*

Colin Cooper[1], Ralf Klasing[2], and Tomasz Radzik[1]

[1] Department of Computer Science, King's College, London WC2R 2LS, UK
{Colin.Cooper, Tomasz.Radzik}@kcl.ac.uk
[2] LaBRI – Université Bordeaux 1 – CNRS, 351 cours de la Libération, 33405 Talence cedex, France
Ralf.Klasing@labri.fr

Abstract. We consider a fixed communication network where (software) agents can move freely from node to node along the edges. A *black hole* is a faulty or malicious node in the network such that if an agent enters this node, then it immediately "dies." We are interested in designing an efficient communication algorithm for the agents to identify all black holes. We assume that we have k agents starting from the same node s and knowing the topology of the whole network. The agents move through the network in synchronous steps and can communicate only when they meet in a node. At the end of the exploration of the network, at least one agent must survive and must know the exact locations of the black holes. If the network has n nodes and b black holes, then any exploration algorithm needs $\Omega(n/k + D_b)$ steps in the worst-case, where D_b is the worst case diameter of the network with at most b nodes deleted. We give a general algorithm which completes exploration in $O((n/k) \log n / \log \log n + bD_b)$ steps for arbitrary networks, if $b \leq k/2$. In the case when $b \leq k/2$, $bD_b = O(\sqrt{n})$ and $k = O(\sqrt{n})$, we give a refined algorithm which completes exploration in asymptotically optimal $O(n/k)$ steps.

Keywords: Graph exploration, mobile agent, black hole faults.

1 Introduction

The network search problem which we consider in this paper is motivated by the following scenario. Mobile (software) agents can move through a network of computers, but some host, called *black holes* terminate any agent visiting it. The problem of protecting mobile agents from such malicious hosts has been studied in [6,7,10,11]. We assume that agents are a limited resource, so they should first locate black holes to avoid entering them and dying.

Initially, the agents are at the same *start node* s and know the topology of the whole network, but do not know the number and the location of black holes.

* Research supported in part by Royal Society Grant ESEP 16244, COST Action TIST 293 (GRAAL), and a visiting fellowship from LaBRI/ENSEIRB for Colin Cooper.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 320–332, 2006.

Also, no information about black holes is available in the safe nodes of the network (the nodes which are not black holes). Thus, in order to locate a black hole, at least one agent must visit it. An agent entering a black hole disappears there, so later this agent cannot show up where expected by the other agents, or communicate with them in any other expected way. On this basis, the other agents may be able to deduce the exact location of a black hole. We want to design an efficient communication algorithm for the agents to identify all black holes reachable from the start node. (If black holes disconnect the network into separate components, then we can only hope to explore the component which contains the start node.)

The black hole search was studied in [3,4,5] under the scenario of totally asynchronous networks, that is, without assuming any bound on the ratio of the times of two different edge traversals. The authors considered the case of two agents and one black hole. To solve the problem in this setting, the network must be 2-connected. The complexity measure considered is the total number of moves performed by the agents. For arbitrary networks of n nodes, it is shown in [5] that $\Theta(n \log n)$ moves are necessary and sufficient.

We consider synchronous networks and assume that agents can communicate, and exchange their knowledge, only when they meet. They cannot leave any messages in the nodes of the network. In one synchronized step, each agent can either stay in its current host or move to a neighbouring one. An agent X may infer the location of a black hole, if it expects to meet another agent Y, but agent Y does not show up. This model, but with the added restriction that there are only two agents and at most one black hole, has been previously studied in [1,2,8,9]. Those papers give NP-hardness results and approximation algorithms for the problem of calculating an optimal (shortest) traversal schedules.

In this paper, we show two efficient algorithms for black hole search with multiple agents. That is, the number k of initially available agents is an independent parameter. At the beginning of the computation all agents are at a start node s. Let D_b be the maximum distance from s to a node reachable from s in a network obtained from the given network by deleting up to b nodes (the maximum over all possible deletions). If the network has $b \leq k-2$ black holes (see the next section for justification of considering this bound on the number of black holes), then a black hole search with k agents must have $\Omega((n/k)+D_b)$ steps in the worst case. If $b \leq k/2$, then our first algorithm takes $O((n/k)\log n/\log\log n + bD_b)$ steps in the worst case, while our second algorithms takes $O(n/k)$ steps, provided that additionally $k = O(\sqrt{n})$ and $bD_b = O(\sqrt{n})$. Our algorithms are the first algorithms for searching for black holes with multiple agents with non-trivial upper bounds on the number of steps. Note that k agents can trivially discover up to $k-1$ black holes in $O(n)$ steps by exploring the network using edges of an arbitrary spanning tree.

In the running time of a black hole search algorithm we count only the number of "traversal" steps, but do not count the computational time of deciding which traversals the agents should take in the current step. In our algorithms, this computational time is polynomial.

2 Preliminaries

The input to our black-hole search problem is an undirected connected graph $G = (V, E)$, a subset of nodes $S \subseteq V$ known to be safe, a *start node* $s \in S$, and a positive integer k. The objective of the problem is to identify black holes in $V \setminus S$ using k agents. The agents are numbered from 1 to k and they are initially at the start node s. The agents move in synchronized steps. In each step, each agent can either wait in its current position $v \in V$, or move to a node adjacent to v. Agents communicate, exchanging their whole knowledge, when they are at the same node at the same step.

For a subset of nodes $W \subseteq V$ and a node $v \in W$, $G[W]$ denotes the subgraph of G induced by W, and $G_v[W]$ denotes the connected component of $G_v[W]$ containing node v.

An *exploration algorithm* works correctly, if it terminates, and at the termination there is at least one surviving agent, all surviving agents know the set of *identified* black holes $B \subseteq V \setminus S$, and they all know that all nodes in $G_s[V \setminus B]$ are safe. Observe that a node can be identified as a black hole or as a safe node only if it can be reached from s along a path of safe nodes. Thus, it is not possible to find out anything about the nodes in other components of $G[V \setminus B]$. Note that in our notation in this paper (unlike in some previous papers on this topic) B stands for the set of *identified* black holes, which is not necessarily the whole set of black holes in the network.

We assume that $k \geq 3$. If there was initially only one agent, then no exploration would be possible since the agent would risk entering a black hole by attempting any movement. The black-hole search problem with two agents has been comprehensively studied before.

Our exploration algorithms work correctly, if there are at most $k - 2$ black holes in G, losing only one agent for each identified black hole. If there are $k - 1$ or more black holes in the network, then the algorithms (slightly modified to stop when only one agent remains) return a set B of $k - 1$ black holes, but the surviving agent may not know whether all nodes in $G_s[V \setminus B]$ are safe (some nodes in $G_s[V \setminus B]$ may be left unexplored, so can be additional black holes, and cannot be explored with one agent).

The running time of an exploration algorithm is the number of steps executed until its termination. We define D_b to be the maximum diameter of $G_s[V \setminus B]$ over all $B \subseteq V$, $|B| \leq b$. Any exploration algorithm runs in the *worst case* placement of the black holes in $\Omega(n/k)$ time , if $|V \setminus S| = \Omega(n)$, and in $\Omega(D_b)$ time, if b black holes are identified. The asymptotic bounds on the performance of our algorithms include this parameter D_b, but we do not consider in this paper the problem of estimating D_b.

For a graph H, $V[H]$ denotes the set of nodes in H. A *subtree* of a rooted tree T is a connected subgraph of T. The root of a subtree H of T is the node in H closest to the root of T. We say that J is the subtree of T spanned by a vertex set X, if J is the smallest subtree of T containing the vertex set X. For a node v in T, *the subtree of T rooted at v* contains v and all its descendants in T.

3 Procedures Used in the Main Algorithms

Let P be a path with node s as one end. Two agents can explore P in the following simple way. The agents move together along the path *probing* unexplored nodes (nodes not known to be safe). That is, if they are to move from a node u to an unexplored node w then one agent moves from u to w and back to u, while the other agent waits in u. If the agent which has moved to w comes back to u, then both agents learn that u is a safe node, so they move there together in the next step. If the probing agent does not come back to u from w, then the waiting agent learns that u is a black hole and goes back to s. The whole exploration takes at most $4d$ steps, where d is the length of P. Next, we describe a natural extension of this simple algorithm from paths to trees. Note that by exploring a path or a tree we mean here a partial exploration, as no nodes in the path/tree which are "behind" the black holes are reached (even if in the overall network there are other edges than the path/tree edges leading to those nodes).

Let T be a tree rooted in a start node s, and S be the set of known safe nodes in T ($s \in S$). Let h denote the height of T and l denote the number of leaves in T. Procedure ExploreTree(T, s, S) explores T in $O(h)$ steps using $2l$ agents starting from s. (If there are more than $2l$ agents available at the beginning of the computation of this procedure, then still only $2l$ agent are used and the remaining agents wait in s.) The agents are assigned to the leaves of T, two agents per each leaf. The agents move from s towards their leaves probing each unexplored node. More precisely, if a number of agents are to move from a node u to an unexplored node w on their way towards the leaves in the subtree of T rooted at w, then one of the agents moves from u to w and back to u, while the others wait in u. If the selected agent comes back to u, then the waiting agents know that w is safe, so they all move there together. If the selected agent does not come back to u, then the waiting nodes learn that w is a black hole and go back to s. When two agents reach the leaf assigned to them, then they go back to s.

Procedure ExploreTree(T, s, S) returns the set B of discovered black holes, runs in at most $4h$ steps, loses only $|B|$ agents, and establishes that all nodes in the subtree of T obtained by removing the subtrees rooted in B are safe. At the termination, all surviving agents are back at the start node s. We use procedure ExploreTree(T, s, S) as a subroutine in our exploration algorithms for general graphs.

Another procedure which we use is ExploreSubtrees$(T, s, S', \mathcal{H})$. Here T is a tree rooted in a start node s, S' is the set of initially known safe nodes in T ($s \in S'$), and $\mathcal{H}$ is a family of rooted subtrees of T with the following properties.

1. $\mathcal{H}$ covers all nodes in $V[T] \setminus S'$, that is, for each $v \in V[T] \setminus S'$, there is a subtree $H \in \mathcal{H}$ containing v.
2. If a node belongs to two subtrees in $\mathcal{H}$, then it is the root of one or both of them.
3. For each subtree $H \in \mathcal{H}$, each node on the path in T between s and the root of H, including the root of H, belongs to S' (that is, is initially known to be safe).

Procedure ExploreSubtrees$(T, s, S', \mathcal{H})$ explores T using $2|\mathcal{H}|$ agents in the following way. For each subtree in $\mathcal{H}$, two agents are assigned to explore it, and the exploration of all subtrees is conducted in parallel. The two agents assigned to a subtree $H \in \mathcal{H}$ first walk from s to the root of H (this path is safe by condition 3 above), and then walk along an Euler tour of H probing nodes not in S'. If they have found a black hole in H, then the surviving agent goes back to s. If they explore the whole subtree without finding a black hole, then they both go back to the root. Condition 2 and 3 implies that exploration of all subtrees in $\mathcal{H}$ can be done in parallel and *independently*, since a possible common node of two trees must be safe. Procedure ExploreSubtrees$(T, s, S', \mathcal{H})$ terminates in $O(h + \max_{H \in \mathcal{H}} |H|)$ steps, where h is the height of T, and returns the set B of found black holes and the set S'' of established safe nodes ($S' \subseteq S''$). Observe that for each $H \in \mathcal{H}$, this procedure finds a black hole in H (leaving some nodes in H unexplored) or establishes that all nodes in H are safe.

In our algorithms for exploration of arbitrary graphs, we consider only balanced families of subtrees. For T, $S' \subseteq V[T]$, $s \in S'$, and a positive integer x, an *x-balanced family* $\mathcal{H}$ of subtrees of tree T contains at most x subtrees, satisfies conditions 1 and 2 above, and for each $H \in \mathcal{H}$, $|V[H]| = O(|V[T]|/x)$ and $|V[H] \cap U| = O(|U|/x)$, where $U = V[T] \setminus S'$. An x-balanced family of subtrees is returned by procedure BalancedSubtrees(T, s, U, x). We present details of this procedure in Section 6.

4 Algorithm for Arbitrary Networks

We describe our algorithms using the following notation:

- $\bar{B}$ – the set of nodes already identified as black holes,
- $\bar{G} = G_s[V \setminus \bar{B}]$ – the current network,
- $\bar{S}$ – the set of nodes already known to be safe, $S \subseteq \bar{S} \subseteq V[\bar{G}]$,
- $\bar{k}$ - the number of live agents.

Initially, $\bar{B} = \emptyset$, $\bar{G} = G$, $\bar{S} = S$, and $\bar{k} = k$. The details of our first graph exploration algorithm ExploreGraph1 are given in Figure 1. The algorithm consists of a number of *rounds*. In each round, the algorithm tries to explore more nodes. Analysing how the number of unexplored nodes decreases in each round will be the basis for bounding the total running time.

Each round consists of two parts. The first part, the "repeat" loop in lines 3–9, establishes a shortest path tree T in $\bar{G}$ to all nodes in $V[\bar{G}] \setminus \bar{S}$ and a balanced family $\mathcal{H}$ of $\lfloor \bar{k}/2 \rfloor$ subtrees which meets also condition 3 given in Section 3. This is done by an iterative process of computing a shortest path tree T, taking the $\lfloor \bar{k}/2 \rfloor$-balanced family $\mathcal{H}$ of subtrees of T returned by the procedure call BalancedSubtrees$(T, s, V[\bar{G}] \setminus \bar{S}, \lfloor \bar{k}/2 \rfloor)$, and checking if this family satisfies condition 3. This checking is done by calling ExploreTree$(J, s, \bar{S} \cap V[J])$ for the subtree J of tree T containing only the tree paths from s to the roots of the subtrees in $\mathcal{H}$. If all nodes in subtree J turn out to be safe, then

```
ExploreGraph1(G, s, S, k):
1:  (Ḡ, S̄, k̄, B̄) ← (G, S, k, ∅);
    { the current graph, the current set of known safe nodes, the number of
      agents which are still alive, and the current set of known black holes; }
2:  while S̄ ≠ V[Ḡ] do
        { one round }
3:      repeat
            { all live agents are in the start node s }
4:          T ← a shortest path tree in Ḡ from s to all nodes in V[Ḡ] \ S̄;
5:          H ← BalancedSubtrees(T, s, V[Ḡ] \ S̄, ⌊k̄/2⌋);
            { H is a balanced family of at most ⌊k̄/2⌋ subtrees of T covering
              all nodes not in S̄; }
6:          J ← the subtree of T spanned by s and the roots of subtrees in H;
7:          B_J ← ExploreTree(J, s, S̄ ∩ V[J]);
8:          update Ḡ, S̄, k̄, and B̄;
9:      until V[J] ⊆ S̄;  {that is, all nodes in tree J are safe }
10:     (B_H, S_H) ← ExploreSubtrees(T, s, S̄, H);
11:     update Ḡ, S̄, k̄, and B̄;
12: end_while;
13: return B̄.
```

Fig. 1. ExploreGraph1– a graph exploration algorithm with $O(\log n/\log\log n)$ rounds

we have found required T and $\mathcal{H}$. If a black hole has been found in J, then the process continues by computing new T and $\mathcal{H}$.

For T and $\mathcal{H}$ established in the first part of a round, in the second part the subtrees in $\mathcal{H}$ are explored by calling procedure ExploreSubtrees$(T, s, \bar{S}, \mathcal{H})$ (line 10). The algorithm proceeds to the next round, if there are still nodes left in $V[\bar{G}] \setminus \bar{S}$ (unexplored nodes).

If the number of black holes in G is $b \leq k-2$, then algorithm ExploreGraph1 terminates correctly, and at the termination there are $k-b$ surviving agents and all nodes in $V[\bar{G}]$ have been identified as safe. The running time, however, can be as high as $\Theta(n)$, even if D_b is small. We show next a bound on the running time of ExploreGraph1, if $b \leq k/2$.

Theorem 1. *Let $G = (V, E)$ be an n-node connected graph, $S \subseteq V$ be the set of known safe nodes in G, $s \in S$, and an integer $k \geq 3$. If there are initially k agents at node s and the number of black holes in G is $b \leq k/2$, then the algorithm* ExploreGraph1(G, s, S, k) *completes the search for black holes in $O((n/k)\log n/\log\log n + bD_b)$ steps.*

Proof. We refer to the description of algorithm ExploreGraph1 given in Figure 1. At each point of the computation of algorithm ExploreGraph1, the number of live agents is $\bar{k} \geq k - b \geq k/2$. The agents walk through the network only when executing procedure ExploreTree in line 7 and procedure ExploreSubtrees in line 10. All other computation, including procedure BalancedSubtrees, is done locally from the knowledge of the network and already located black holes, and is therefore not counted in the time complexity.

One execution of procedure ExploreTree in line 7 takes $O(h)$ steps, where h is the height of tree J. Since J is a shortest path tree in $\bar{G}$ and the diameter of $\bar{G}$ is at most D_b, so $h \leq D_b$. If no black hole is found during the current execution of procedure ExploreTree, then the inner ("repeat") loop ends and procedure ExploreSubtrees is executed. If no black hole is found during this execution of ExploreSubtrees, then the black-hole search is completed and the whole algorithm terminates. Thus, if the current execution of procedure ExploreTree is not the last one, then between the beginning of the current execution of procedure ExploreTree and the beginning of the next execution of this procedure at least one black hole is found. Hence there are at most $b+1$ executions of procedure ExploreTree throughout the whole execution of algorithm ExploreGraph1.

Since procedure ExploreSubtrees is applied in line 10 to subtrees of sizes $O(n/\bar{k})$, then the general bound on the running time of this procedure implies that its execution in line 10 takes $O((n/\bar{k}) + D_b) = O((n/k) + D_b)$ steps. Procedure ExploreSubtrees is executed once in each round (each iteration of the outer "while" loop) of algorithm ExploreGraph1. Let q denote the number of rounds. At least one new black hole is found in each round other than the last one, so $q \leq b + 1$. We show that we also have $q = O(\log n / \log\log n)$.

For round i, let $\mathcal{H}_i$ denote the family of subtrees used in procedure ExploreSubtrees in line 10, and let b_i denote the number of black holes found by this procedure call. We have $\sum_{i=1}^{q} b_i \leq b$. Let $U_i = V[\bar{G}] \setminus \bar{S}$, $u_i = |U_i|$, and $k_i = \bar{k}$ before the last call to procedure BalancedSubtrees in this round (in line 5), which calculates $\mathcal{H}_i$. A node v belongs to U_{i+1}, only if v belongs to U_i, v belongs to a subtree in $\mathcal{H}_i$ containing a black hole, and v is not the root of this subtree. There are exactly b_i subtrees in $\mathcal{H}_i$ which have black holes, and each of these subtrees contains at most $6u_i/\lfloor k_i/2 \rfloor$ nodes other than the root which belong to U_i (see Lemma 1, part 3). Since $k_i \geq k/2$ and $k \geq 3$, then $\lfloor k_i/2 \rfloor \geq k/6$, so

$$u_{i+1} \;\leq\; b_i \frac{6u_i}{\lfloor k_i/2 \rfloor} \;\leq\; \frac{36 b_i}{k} u_i. \tag{1}$$

This implies

$$1 \;\leq\; u_q \;\leq\; \left(\frac{36}{k}\right)^{q-1} \left(\prod_{i=1}^{q-1} b_i\right) u_1 \leq \left(\frac{36b}{k(q-1)}\right)^{q-1} u_1 \leq \left(\frac{18}{q-1}\right)^{q-1} u_1, \tag{2}$$

using $\prod_{i=1}^{q-1} b_i \leq [(\sum_{i=1}^{q-1} b_i)/(q-1)]^{q-1}$. Inequalities (2) and $u_1 \leq n$ imply that $q = O(\log n / \log\log n)$.

Thus, all executions of procedure ExploreTree take together $O(bD_b)$ steps and all executions of procedure BalancedSubtrees take together $O((n/k + D_b) \min\{b, \log n/\log\log n\})$ steps. Hence the execution of algorithm ExploreGraph1 takes $O((n/k)\log n/\log\log n + bD_b)$ steps. □

5 An Algorithm for Networks with Bounded Diameter

The running time of algorithm ExploreGraph1 is $O((n/k)\log n/\log\log n)$, if there are at most $k/2$ black holes and D_b is small. Our second algorithm ExploreGraph2 has asymptotically optimal $O(n/k)$ running time, provided that some bounds on k and D_b are satisfied.

The details of algorithm ExploreGraph2 are given in Figure 2. This algorithm has only one round (lines 2–16), instead of $O(\log n/\log\log n)$ rounds of algorithm ExploreGraph1. The nodes which remain unexplored after this single round are then explored in the second part of the algorithm using procedure ExploreTree (lines 17–21). To guarantee that not too many unexplored nodes are left for the second part of the algorithm, in the first part of the algorithm we consider x-balanced families of subtrees for a parameter $x \geq k$. The number of subtrees in such a family $\mathcal{H}$ can be greater than $k/2$, so we cannot check if $\mathcal{H}$ satisfies condition 3 by only one call to procedure ExploreTree, and we cannot explore subtrees in $\mathcal{H}$ by only one call to procedure ExploreSubtrees, as we did in one round of algorithm ExploreGraph1. Instead the first task is accomplished by a sequence of calls to procedures ExploreTree (lines 6–10), and the second task is accomplished by a sequence of calls to procedure ExploreSubtrees (lines 12–16).

If the number of black holes in G is $b \leq k-2$, then algorithm ExploreGraph2 terminates correctly, and at the termination there are $k-b$ surviving agents and all nodes in $V[\bar{G}]$ have been identified as safe. We show next a bound on the running time of ExploreGraph2, if $b \leq k/2$.

Theorem 2. *Let $G = (V, E)$ be an n-node connected graph, $S \subseteq V$ be the set of known safe nodes in G, $s \in S$, and integers $x \geq k \geq 3$. If there are initially k agents at node s and the number of black holes in G is $b \leq k/2$, then algorithm* ExploreGraph2(G, s, S, k, x) *terminates in $O((n/k)+(bD_b/k)(x + (n/x)))$ steps.*

Proof. We refer to the description of algorithm ExploreGraph2 given in Figure 2. Throughout the computation of algorithm ExploreGraph2, the number of live agents is $\bar{k} \geq k - b \geq k/2$. We bound the total number of steps taken by procedures ExploreTree and ExploreSubtrees. All other computation is done locally and is not counted in the time complexity.

Each execution of procedure ExploreTree in lines 8 and 19 takes $O(D_b)$ steps. The outer "repeat" loop in lines 2–11 has at most $b+1$ iterations, since at least one black hole is found in each iteration other than the last one. In each iteration of this loop, there are at most $O(x/k)$ iterations of the inner "repeat"

```
ExploreGraph2(G, s, S, k, x):
1:  (Ḡ, S̄, k̄, B̄) ← (G, S, k, ∅);
2:  repeat
        { all live agents are in the start node s }
3:      T ← a shortest path tree in Ḡ from s to all nodes in V[Ḡ] \ S̄;
4:      H ← BalancedSubtrees(T, s, V[Ḡ] \ S̄, x);
5:      J ← the subtree of T spanned by s and the roots of the subtrees in H;
6:      repeat
7:          Q ← subtree of J spanned by s and next ⌊k̄/2⌋ leaves in J;
8:          B_Q ← ExploreTree(Q, s, S̄ ∩ V[Q]);
9:          update Ḡ, S̄, k̄, and B̄;
10:     until V[J] ⊆ S̄ or B_Q ≠ ∅;
11: until V[J] ⊆ S̄;  {that is, all nodes in tree J are safe }
12: repeat
13:     F ← next ⌊k̄/2⌋ subtrees in H;
14:     (B_F, S_F) ← ExploreSubtrees(T, s, S̄, F);
15:     update Ḡ, S̄, k̄, and B̄;
16: until all subtrees in H have been considered;
17: while S̄ ≠ V[Ḡ] do
18:     T ← a shortest path tree in Ḡ from s to ⌊k̄/2⌋ nodes in V[Ḡ] \ S̄;
19:     B_T ← ExploreTree(T, s, S̄ ∩ V[T]);
20:     update Ḡ, S̄, k̄, and B̄;
21: end_while;
22: return B̄.
```

Fig. 2. ExploreGraph2: a graph exploration algorithm with a single round followed by an additional "cleaning-up" process using algorithm ExploreTree

loop. Thus, all executions of procedure ExploreTree in line 8 take together $O(xbD_b/k)$ steps.

Since procedure ExploreSubtrees is applied in line 14 to subtrees of sizes $O(n/x)$, then the general bound on the running time of this procedure implies that its one execution takes $O((n/x) + D_b)$ steps. There are $O(x/k)$ iterations of the "repeat" loop in lines 12–16, so all executions of procedure ExploreSubtrees take together $O((n/k) + (x/k)D_b)$ steps.

At the termination of the loop in lines 12–16, each node in $V[\bar{G}] \setminus \bar{S}$ belongs to a subtree in $\mathcal{H}$ which has a black hole. There are at most b subtrees in $\mathcal{H}$ which have black holes, and each of these subtrees contains $O(n/x)$ nodes. Hence $|V[\bar{G}] \setminus \bar{S}| = O(bn/x)$ at the beginning of the "while" loop in lines 17–21. Each execution of

procedure ExploreTree in line 19 other than the last one decreases the number of nodes in $V[\bar{G}] \setminus \bar{S}$ by at least $\lfloor \bar{k}/2 \rfloor$ or finds at least one new black hole. Therefore there are $O((bn)/(xk)+b)$ executions of procedure ExploreTree in line 19, and all these executions take together $O(D_b((bn)/(xk)+b))$ steps.

We conclude that the number of steps required by algorithm ExploreGraph2 is

$$O\left(\frac{xbD_b}{k}\right) + O\left(\frac{n}{k} + \frac{xD_b}{k}\right) + O\left(\frac{bnD_b}{xk} + bD_b\right) \;=\; O\left(\frac{n}{k} + \frac{bD_b}{k}\left(x + \frac{n}{x}\right)\right).$$

□

Corollary 1. *Let $G = (V, E)$ be an n-node connected graph, $S \subseteq V$ be the set of known safe nodes in G, $s \in S$, and an integer k such that $3 \leq k = O(\sqrt{n})$. If there are initially k agents at node s, the number of black holes in G is $b \leq k/2$, and $bD_b = O(\sqrt{n})$, then for $x = \max\{k, \lfloor \sqrt{n} \rfloor\}$, algorithm* ExploreGraph2(G, s, S, k, x) *terminates in $O(n/k)$ steps.*

6 Computing a Family of Balanced Subtrees

For a tree T rooted at node s, a subset of nodes $U \subseteq V[T]$ and a positive integer x, procedure BalancedSubtrees(T, s, U, x) computes an x-balanced family $\mathcal{H}$ of subtrees of T. The computation proceeds in iterations. In each iteration, a subtree H is cut off from the tree and added to $\mathcal{H}$.

Let $\bar{T}$ denote the current tree, and for a node $v \in \bar{T}$, let *size*(v) and *weight*(v) denote the number of nodes in the subtree of $\bar{T}$ rooted at v and the number of nodes in this subtree which belong to U, respectively. Similarly, for a subtree H of tree T, let *size*(H) and *weight*(H) denote the number of nodes in H and the number of nodes in H which are in U, respectively. In one iteration of procedure BalancedSubtrees, a node v in $\bar{T}$ is selected, which is a lowest (furthest from the root) node in $\bar{T}$ such that *size*$(v) \geq 3n/x$ or *weight*$(v) \geq 3u/x$. Thus, for each child w of v, *size*$(w) < 3n/x$ and *weight*$(w) < 3u/x$. Order the children of node v in an arbitrary way. The subtree H selected in this iteration is obtained by taking node v as the root and adding the subtrees rooted in the first q children of node v which make *size*$(H) \geq 3n/x$ or *weight*$(H) \geq 3u/x$. This subtree H is added to $\mathcal{H}$ and all nodes in H other than the root node v are removed from $\bar{T}$. Node v is also removed from $\bar{T}$, if v becomes a leaf. The computation continues until the remaining tree $\bar{T}$ has less than $3n/x$ nodes and less than $3u/x$ nodes in U. If the final tree $\bar{T}$ is not empty, then it is added to $\mathcal{H}$.

Next, we show that the family $\mathcal{H}$ of subtrees of T computed by BalancedSubtrees(T, s, U, x) is indeed x-balanced.

Lemma 1. *For an n-node tree T rooted at a node s, a subset U of u nodes in T, and a positive integer $x < u$, procedure* BalancedSubtrees(T, s, U, x) *returns a family $\mathcal{H}$ of subtrees of T with the following properties.*

1. *The subtrees in $\mathcal{H}$ cover all nodes in U, that is, for each $v \in U$, there is a subtree $H \in \mathcal{H}$ containing v.*

BalancedSubtrees(T, s, U, x):

1: $(n, u) \leftarrow$ the number of nodes in T and U, respectively;
2: $\mathcal{H} \leftarrow$ empty family of subtrees of T;
3: $\bar{T} \leftarrow T$; { the remaining tree to be covered }
4: **while** $|V[\bar{T}]| \geq 3n/x$ or $|V[\bar{T}] \cap U| \geq 3u/x$ **do**
5: let $size(v)$ and $weight(v)$ be the number of nodes in the subtree of $\bar{T}$ rooted at v and the number of nodes in this subtree which are in U;
6: $v \leftarrow$ a lowest (furthest from the root) node in $\bar{T}$ such that $size(v) \geq 3n/x$ or $weight(v) \geq 3u/x$ (thus for each child w of v, $size(w) < 3n/x$ and $weight(w) < 3u/x$);
7: $H \leftarrow$ a subtree of $\bar{T}$ obtained by taking node v as the root and adding subtrees rooted at children of v until $size(H) \geq 3n/x$ or $weight(H) \geq 3u/x$;
{ thus $size(H) \leq (6n/x) + 1$ and $weight(H \setminus \{v\}) \leq 6u/x$; }
8: add H to $\mathcal{H}$;
9: update $\bar{T}$:
remove from $\bar{T}$ all nodes in H other than the root node v;
remove also v, if it becomes a leaf;
10: **end_while**;
11: **if** $\bar{T}$ is not empty **then** add $\bar{T}$ to $\mathcal{H}$;
12: **return** $\mathcal{H}$.

Fig. 3. Computing balanced subtrees of a tree T covering the nodes in a given set U

2. *If a node belongs to two subtrees in $\mathcal{H}$, then it is the root of one or both of them.*
3. *Each subtree in $\mathcal{H}$ has at most $(6n/x) + 1$ nodes and at most $6u/x$ nodes other than the root which belong to U.*
4. *There are at most x subtrees in $\mathcal{H}$.*

Proof. Parts 1, 2, and 3 of the lemma are easy consequences of the way procedure BalancedSubtrees constructs the subtrees in $\mathcal{H}$. We give a detailed argument only for part 4 of the lemma.

Let $\mathcal{H}_1$ be the family of subtrees in $\mathcal{H}$ with sizes at least $3n/x$, $\mathcal{H}_2$ be the family of subtrees in $\mathcal{H}$ with weights at least $3u/x$, $h_1 = |\mathcal{H}_1|$, and $h_2 = |\mathcal{H}_2|$. Since each subtree in $\mathcal{H}$, except possibly the last one added in line 11, has size at least $3n/x$ or weight at least $3u/x$, then $|\mathcal{H}| \leq h_1 + h_2 + 1$. We have

$$\frac{3n}{x} h_1 \;\leq\; \sum_{H \in \mathcal{H}_1} |H| \;\leq\; h_1 + |\bigcup \mathcal{H}'| \;\leq\; h_1 + n, \tag{3}$$

where the second inequality holds because each node can be a non-root node in at most one tree in $\mathcal{H}$. Similarly we have

$$\frac{3u}{x}h_2 \;\leq\; \sum_{H\in\mathcal{H}_2} weight(H) \;\leq\; h_2 + u. \tag{4}$$

Inequality (3) implies that

$$h_1 \leq \frac{x}{3 - x/n} < \frac{x}{2},$$

so $h_1 \leq (x-1)/2$, as x is an integer. Similarly (4) implies that $h_2 \leq (x-1)/2$. Therefore,

$$|\mathcal{H}| \;\leq\; h_1 + h_2 + 1 \;\leq\; x. \qquad \square$$

7 Conclusions

We showed two efficient algorithms for black hole search with multiple agents. If there are k agents and the network has n nodes and b black holes, then any exploration algorithm needs $\Omega(n/k + D_b)$ steps in the worst-case, where D_b is the worst case diameter of the network with at most b nodes deleted. We gave a general algorithm which completes exploration in $O((n/k)\log n/\log\log n + bD_b)$ steps for arbitrary networks, if $b \leq k/2$. In the case where $b \leq k/2$, $bD_b = O(\sqrt{n})$ and $k = O(\sqrt{n})$, we gave a refined algorithm which completes exploration in asymptotically optimal $O(n/k)$ steps. Our algorithms are the first algorithms for searching for black holes with multiple agents with non-trivial upper bounds on the number of steps.

Even though our second algorithm is asymptotically optimal for restricted values of k in networks with bounded diameter, it remains a question for further research whether an algorithm can be devised that is asymptotically optimal for general k and/or for arbitrary networks. Another important issue is to derive good bounds on the involved constants for practical implementations. Moreover, in this paper, we only considered the case when the topology of the whole network is known in advance. It would be interesting to consider the case when initially no, or only partial, information about the network is available.

Acknowledgements

The authors would like to thank the anonymous referees for their very helpful comments and suggestions.

References

1. J. Czyzowicz, D. Kowalski, E. Markou, and A. Pelc. Searching for a black hole in tree networks. In *Proceedings of 8th International Conference on Principles of Distributed Systems (OPODIS 2004)*, pages 34–35, 2004. Also: Springer LNCS vol. 3544, pages 67-80, also to appear as "Searching for a Black Hole in Synchronous Tree Networks" in Combinatorics, Probability and Computing.

2. J. Czyzowicz, D. Kowalski, E. Markou, and A. Pelc. Complexity of searching for a black hole. *Fundamenta Informaticae*, 71(2-3):229–242, 2006.
3. S. Dobrev, P. Flocchini, R. Kràlovic, G. Prencipe, P. Ruzicka, and N. Santoro. Black hole search in common interconnection networks. *Networks*, to appear. Preliminary version under the title "Black Hole Search by Mobile Agents in Hypercubes and Related Networks" in *Proc. 6th Int. Conf. on Principles of Distributed Systems (OPODIS 2002)* pages 169-180, 2002.
4. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile search for a black hole in an anonymous ring. *Algorithmica*, to appear. Preliminary version in *Proc. 15th Int. Symposium on Distributed Computing (DISC 2001)*, Springer LNCS vol. 2180, pages 166-179, 2001.
5. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Searching for a black hole in arbitrary networks: Optimal mobile agents protocols. *Distributed Computing*, to appear. Preliminary version in *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 153-161, 2002.
6. F. Hohl. Time limited black box security: Protecting mobile agents from malicious hosts. In G. Vigna, editor, *Proceedings of Conference on Mobile Agent and Security*, Springer LNCS vol. 1419, pages 90–111, 1998.
7. F. Hohl. A framework to protect mobile agents by using reference states. In *Proc. 20th Int. Conf. on Distributed Computing Systems (ICDCS '00)*, pages 410–417, 2000.
8. R. Klasing, E. Markou, T. Radzik, and F. Sarracco. Approximation bounds for black hole search problems. In *Proceedings of 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, Springer LNCS vol. 3974, to appear.
9. R. Klasing, E. Markou, T. Radzik, and F. Sarracco. Hardness and approximation results for black hole search in arbitrary graphs. In *Proc. 12th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO 2005)*, Springer LNCS vol. 3499, pages 200–215, 2005. Also: *Theoretical Computer Science*, to appear.
10. S. Ng and K. Cheung. Protecting mobile agents against malicious hosts by intention of spreading. In H. Arabnia, editor, *Proc. Int. Conf. on Parallel and Distributed Processing and Applications (PDPTA'99) Vol. II*, pages 725–729, 1999.
11. T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. In *Proc. Conf. on Mobile Agent Security*, Springer LNCS vol. 1419, pages 44–60, 1998.

Gathering Asynchronous Mobile Robots with Inaccurate Compasses*

Samia Souissi[1], Xavier Défago[1], and Masafumi Yamashita[2]

[1] School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{ssouissi, defago}@jaist.ac.jp
[2] Department of Computer Science and Communication Engineering,
Kyushu University, Fukuoka, Japan
mak@csce.kyushu-u.ac.jp

Abstract. This paper considers a system of asynchronous autonomous mobile robots that can move freely in a two-dimensional plane with no agreement on a common coordinate system. Starting from any initial configuration, the robots are required to eventually gather at a single point, not fixed in advance (gathering problem).

Prior work has shown that gathering oblivious (i.e., stateless) robots cannot be achieved deterministically without additional assumptions. In particular, if robots can detect multiplicity (i.e., count robots that share the same location) gathering is possible for *three or more* robots. Similarly, gathering of any number of robots is possible if they share a common direction, as given by compasses, with *no errors*.

Our work is motivated by the pragmatic standpoint that (1) compasses are error-prone devices in reality, and (2) multiplicity detection, while being easy to achieve, allows for gathering in situations with more than two robots. Consequently, this paper focusses on gathering two asynchronous mobile robots equipped with *inaccurate* compasses. In particular, we provide a self-stabilizing algorithm to gather, in a finite time, two oblivious robots equipped with compasses that can differ by as much as $\pi/4$.

1 Introduction

Background. The problem of reaching agreement among autonomous robots has attracted considerable attention within the last few years. One problem of particular interest is the gathering problem, where robots are required to meet at a single location not predetermined in advance, and without agreement on a common coordinate system. This problem has been studied extensively in the literature, under different models and various assumptions [3,4,9,17]. In fact, several factors render this problem difficult to solve. In particular, in all these studies, the problem has been solved only by making some additional assumptions regarding robots' capabilities.

* Work supported by MEXT Grant-in-Aid for Young Scientists (A) (Nr. 18680007).

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 333–349, 2006.

In this paper, we focus on solving the gathering problem in asynchronous models. In the asynchronous model CORDA [12], Prencipe [13] has shown that there exists no deterministic algorithm to solve the gathering problem in finite time with oblivious robots. Cieliebak et al. [4] have introduced multiplicity, and have shown that gathering is possible for three or more robots, when they are able to detect multiple robots at a single point.

Flocchini et al. [9] have solved the gathering problem for any number of robots when they share a common direction, as provided by a compass[1]. However, their result holds when compasses are perfectly consistent (i.e., with no errors). Yet, in practice sensors are error-prone and sensitive to magnetic interference. Consequently, in this paper, we concentrate on the gathering of two asynchronous mobile robots when their compasses are subject to errors.

This work is motivated by the facts that: (1) in practice, compasses are rather inaccurate sensors, and (2) with multiplicity detection, the gathering is solvable only for more than two robots. For example, the accuracy of compasses typically varies from 1 degree to over 10 degrees, depending on sensor quality (cost) and environment conditions. Therefore, our aim is to fill the gap of solving the gathering problem for two robots relying on oblivious computations, and to provide effective answers to the following two questions. First, is it possible to gather two asynchronous mobile robots when their compasses are inaccurate by some unknown angle? Second, what is the bound of that angle?

Contribution. The main contribution of this paper is to study the solvability of the gathering of two asynchronous mobile robots in the face of compass inaccuracies. In particular, we address the problem when robots are oblivious (or memoryless), meaning that they can not remember their previous states, their previous actions or the previous positions of the other robots. While this is a somewhat over-restrictive assumption, developing algorithms in this model is interesting because any algorithm that works correctly for oblivious robots is intrinsically self-stabilizing[2]. We thus provide an algorithm that gathers in a finite number of steps, two asynchronous oblivious mobile robots equipped with compasses that can differ by as much as $\pi/4$.

Difficulty of the problem. In the asynchronous model CORDA, where robots are equipped with inaccurate compasses, it is difficult to gather two robots or compare them in a consistent manner. This is mainly due to the issue of breaking the symmetry between these robots. Let us illustrate this point using a simple example. Assume that there exists a naive algorithm for comparing two asynchronous robots A and B in a consistent manner when their compasses are inaccurate. First, consider that A and B are equipped with accurate compasses, and place them at the two endpoints of a horizontal diameter of a unit circle. Then, a naive algorithm can be based on the comparison of the angles that A and B form respectively with some global North N (i.e., they share the same north) and the

[1] A compass does not only indicate the North direction, but also gives a unified clockwise orientation.

[2] Self-stabilization is the property of a system which, starting in an arbitrary state, always converges toward a desired behavior [7,14].

segment $\overline{AB}$ in clockwise direction. For instance, if the angle is less than or equal to $\pi/2$, the robot wins. Otherwise, if the angle is greater than $\pi/2$, the robot loses. Then, a robot, say A, wins. Then, we rotate the diameter to exchange the positions of A and B. Now B wins. We thus, color the perimeter of the circle by Win and Lose, where at any point which is colored Win or Lose, A wins or loses. Then, there is a point p that is a boundary between a Win and a Lose segment. By introducing error to their compasses, at p, we can derive a contradiction. That is, we can not decide which robot wins, and which one loses.[3]

Related Work. In their SYm model [17], referred to a semi-synchronous model, Suzuki and Yamashita proposed an algorithm to solve the gathering problem deterministically in the case where robots have unlimited visibility. For a system with two robots, they have proven that it is impossible to achieve the gathering of two *oblivious* mobile robots that have no common orientation under their semi-synchronous model, in a finite time. The difficulty of the problem is inherent in breaking the symmetry between the two robots.

Using the same model, Ando et al. [2] proposed an algorithm to address the gathering problem in systems wherein robots have limited visibility. Their algorithm *converges* toward a solution to the problem, but it does not solve it deterministically. The gathering problem also has been studied in the presence of faulty robots by Agmon and Peleg [1] in synchronous and asynchronous settings. In particular, they proposed an algorithm that tolerates one crash-faulty robot in a system of three or more robots. They also showed that in an asynchronous environment, it is impossible to perform a successful gathering in a 3-robot system with one Byzantine[4] failure. Later on, Défago et al. [6] strengthen the impossibility of gathering in systems with Byzantine robots by showing that it still holds in stronger models. They also show the existence of randomized solutions for systems with Byzantine-prone robots.

In some of our recent work [15], we introduced the notion of unreliable compasses for robots, and we studied the solvability of the gathering problem in the face of compass instabilities. In particular, we proposed a gathering algorithm that solves the problem in the semi-synchronous model SYm for many robots, with compasses that are eventually stabilizing.

Recently, Cohen and Peleg [5] addressed the issue of analyzing the effect of errors in solving gathering and convergence problems. In particular, they studied imperfections in robot measurements, calculations and movements. They showed that gathering cannot be guaranteed in environments with errors, and illustrated how certain existing geometric algorithms, including ones designed for fault-tolerance fail to guarantee even convergence in the presence of small errors. One of their main positive results is an algorithm for convergence under bounded measurement, movement and calculation errors. However, their work does not relate to compasses.

[3] The argument is similar to the bi-valent argument in the impossibility result of the consensus problem [8].

[4] A robot is said to be Byzantine if it executes arbitrary steps that are not in accordance with its local algorithm [18].

While preparing the print-ready version of this manuscript, it came to our attention that a similar result has been presented by Imasu et al. [10] at a domestic workshop in Japan.

Structure. The remainder of this paper is organized as follows. In Sect. 2, we describe the system model and the basic terminology. Sect. 3 describes the algorithm to gather two asynchronous oblivious mobile robots under compass inaccuracies, and Sect. 4 proves its correctness. Finally, Sect. 5 concludes the paper.

2 System Model and Definitions

2.1 System Model

In this paper, we consider the CORDA model of Prencipe [12,11], which is defined as follows. The system consists of a set of autonomous mobile robots $\mathcal{R} = \{r_1, \cdots, r_n\}$. A robot is modelled as a unit having computational capabilities, and which can move freely in the two-dimensional plane. In addition, robots are equipped with sensor capabilities to observe the positions of other robots, and form a local view of the world. The robots are modelled and viewed as points in the Euclidean plane.[5] The local view of each robot includes a unit of length, an origin and the directions and orientations of the two x and y coordinate axes as given by a compass.

The robots are completely *autonomous*. Moreover, they are *anonymous*, in the sense that they are a priori indistinguishable by appearance, and they do not have any kind of identifiers that can be used during their computations. Furthermore, there is no direct means of communication among them.

We further assume that the robots are *oblivious*, meaning that they keep information neither on previous observations nor on past computations.

The cycle of a robot consists of four states: Wait-Look-Compute-Move.

- *Wait.* In this state, a robot is idle. A robot cannot stay permanently idle (see Assumption 2) below. At the beginning all robots are in Wait state.
- *Look.* Here, a robot *observes* the world by activating its sensors, which will return a snapshot of the positions of all other robots with respect to its local coordinate system. Since each robot is viewed as a point, the positions in the plane are just the sets of robots' coordinates.
- *Compute.* In this state, a robot *performs* a local computation according to its deterministic, oblivious algorithm. The algorithm is the same for all robots, and the result of the *compute* state is a destination point.
- *Move.* The robot *moves* toward its computed destination. If the destination is its current location, then the robot is said to perform a *null movement*; otherwise, it is said to execute a *real movement.* The robot moves toward the computed destination, but the distance it moves is unmeasured; neither infinite, nor infinitesimally small (see Assumption 1). Hence, the robot can only go towards its goal, but the move can end anywhere before the destination.

[5] We assume that there are no obstacles to obstruct vision. Moreover, robots do not obstruct the view of other robots and can "see through" other robots.

The (global) time that passes between two successive states of the same robot is finite, but unpredictable. In addition, no time assumption within a state is made. This implies that the time that passes after the robot starts observing the positions of all others and before it starts moving is arbitrary, but finite. That is, the actual movement of a robot may be based on a situation that was observed arbitrarily far in the past, and therefore it may be totally different from the current situation.

In the model, there are two limiting assumptions related to the cycle of a robot.

Assumption 1. *It is assumed that the distance travelled by a robot r in a move is not infinite. Furthermore, it is not infinitesimally small: there exists a constant $\delta_r > 0$, such that, if the target point is closer than δ_r, r will reach it; otherwise, r will move towards it by at least δ_r.*

Assumption 2. *The time required by a robot r to complete a cycle (Wait-Look-Compute-Move) is not infinite. Furthermore, it is not infinitesimally small; there exists a constant $\epsilon_r > 0$, such that the cycle will require at least ϵ_r time.*

2.2 Definitions

Definition 1 (Absolute north). *An absolute north $\overrightarrow{\mathcal{N}}$ is a vector that indicates a fixed north direction. The absolute north is collocated with an absolute y positive axis.*

It is important to stress that the absolute north is not known to the robots, and is used only for the sake of explanation.

Definition 2 (Compass). *A compass is a function of robots and time. The function outputs a relative north direction $\overrightarrow{N_r}(t)$ for some robot r at time t.*

Definition 3 (γ^*-Inaccurate compasses). *Informally, compasses are* γ^*-Inaccurate *iff., for every robot r, the absolute difference between the north indicated by the compass of r and $\overrightarrow{\mathcal{N}}$ is at most γ^* at any time t (also referred to as* error *of the compasses). In addition, for every robot r, the error of its compass is consistent or invariant, i.e., the error of the compass does not fluctuate over time. In other words, a pair of γ^*-Inaccurate compasses can differ by as much as $2\gamma^*$ at any time t, and the difference is invariant. The special case when $\gamma^* = 0$ represents perfect compasses.*

Formally, compasses are γ^*-Inaccurate *iff., the following two properties are satisfied:*

1. *γ^*-Inaccuracy:* $\forall r \in \mathcal{R}, \forall t, |\measuredangle \overrightarrow{\mathcal{N}} \overrightarrow{N_r}(t)| \leq \gamma^*$,
2. *Invariance:* $\forall r, \forall t, t', \overrightarrow{N_r}(t) = \overrightarrow{N_r}(t')$.

2.3 Notations

Given some robot r, $r(t)$ is the position of r at a time t. Let A and B be two points, with $\overline{AB}$, we indicate the segment starting at A and terminating at B, and $\|\overline{AB}\|$ is the length of such a segment. Given three distinct points A, B, and C, we denote by $\triangle(A, B, C)$, the triangle having them as corners, and by $\widehat{BAC}$, the angle formed by A, B and C, and centered at A. Finally, given a region $X(t)$ at time t, we denote by $|X(t)|$, the number of robots in that region at time t. The parameter t is omitted whenever clear from the context.

3 Gathering with Inaccurate Compasses

The basic intuition behind the algorithm is to break the symmetry between two robots, that is, to forbid symmetric configurations of two robots. More precisely, with a perfect compass, it is easy to break the symmetry between two robots. For instance, by making one robot move and the other remain stationary. However, with inaccurate compasses, it is difficult to design an algorithm that breaks the symmetry between the two, as they can end up in a situation in which neither do move, which results in a deadlock situation or in situation inc which both move in such a way they cycle forever. In order to avoid such situations, it is first necessary to ensure that the two robots do not see each other on the same zone.

The main idea of our algorithm is to make each robot partition the plane into four different zones, so that two similar zones for two different robots should not overlap. Then, depending on the different possible configurations (resulting from the partitions) of the two robots, we design their movements such that a configuration is transformed to gathering, or to an intermediate configuration leading to the gathering, without introducing cycles between configurations or deadlock situations.

Before we describe the algorithm in more detail, we will first explain how robots divide the plane.

3.1 Partitions

First, a robot needs to partition the plane into four sectors that do not overlap, namely the *North*, *South*, *East* and *West* sectors. Let α_N, α_S, α_E and α_W be the respective angular measurements of these sectors. Also, by Λ_N, Λ_S, Λ_E and Λ_W, we denote the rays delimiting these sectors, respectively (refer to Fig. 1).

Now, let us assume there exits a constant $\gamma^* \geq 0$ that represents the maximum angle inaccuracy between the relative north $\overrightarrow{N_r}$ of some robot r and the absolute north $\overrightarrow{\mathcal{N}}$. Then, the following conditions must be satisfied in order to avoid a situation in which both robots see each other in the same sector because of compass inconsistencies.

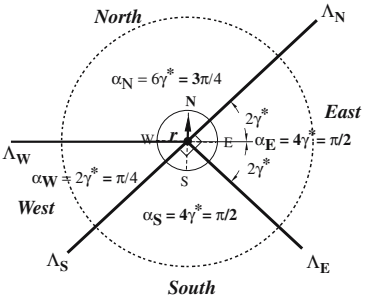


Fig. 1. The four sectors *North*, *South*, *East* and *West* for robot r

$$\alpha_N \leq \pi - 2\gamma^* \tag{1}$$
$$\alpha_S \leq \pi - 2\gamma^* \tag{2}$$
$$\alpha_E \leq \pi - 2\gamma^* \tag{3}$$
$$\alpha_W \leq \pi - 2\gamma^* \tag{4}$$

We further set the following conditions on the sectors. These conditions will help to avoid the occurrence of infinite executions, i.e., having robots looping in the same configuration.

$$\alpha_E + \alpha_S \leq \pi \tag{5}$$
$$\alpha_N + \alpha_W \leq \pi \tag{6}$$

By summation of Equation (1) and Equation (5), we get:

$$\begin{aligned} \alpha_N + \alpha_E + \alpha_S &\leq 2\pi - 2\gamma^* \quad \text{then,} \\ \alpha_N + \alpha_E + \alpha_S + \alpha_W &\leq 2\pi - 2\gamma^* + \alpha_W \\ 2\pi &\leq 2\pi - 2\gamma^* + \alpha_W \\ 2\gamma^* &\leq \alpha_W \end{aligned}$$

After finding the condition in the *West* sector, we choose the minimal value for α_W. That is, $\alpha_W = 2\gamma^*$. Then, by summation of Equation (1), and Equation (2), we get:

$$\begin{aligned} \alpha_N + \alpha_S &\leq 2\pi - 4\gamma^* \quad \text{then,} \\ \alpha_N + \alpha_S + \alpha_E &\leq 2\pi - 4\gamma^* + \alpha_E \end{aligned}$$

By hypothesis, $\alpha_N + \alpha_S + \alpha_E \leq 2\pi$ then, by subtraction, we get:

$$\begin{aligned} 0 &\leq -4\gamma^* + \alpha_E \quad \text{then,} \\ 4\gamma^* &\leq \alpha_E \end{aligned}$$

Thus, we choose $\alpha_E = 4\gamma^* = \alpha_S = \pi/2$ (From Equation (5)). This means that $\gamma^* = \pi/8$. It follows that, $\alpha_W = 2\gamma^* = \pi/4$. Finally, from Equation (1), and the fact that the sum of the four sectors is equal to 2π, we get, $\alpha_N = \pi - 2\gamma^* = 3\pi/4$.

We have derived the condition that $\gamma^* \leq \pi/8$. Thus, in the remainder of the paper, we consider the largest inaccuracy value of γ^*, i.e., $\gamma^* = \pi/8$.

We now describe in more detail the features of each sector, as follows:

- *East*(r) sector: it is centered at r, has the East direction (indicated by its compass) $\overrightarrow{E_r}$ as bisector, and its angular sector α_E is equal to $4\gamma^*$, which is $\pi/2$. Note that *East*(r) is delimited by $\Lambda_N(r)$ and $\Lambda_E(r)$. However, only $\Lambda_E(r)$ is a part of *East*(r).
- *South*(r) sector: it is adjacent to *East*(r) in clockwise direction, and its angular sector α_S is equal to α_E, which is equal to $4\gamma^*$ (i.e., $\pi/2$). Note that *South*(r) is delimited by $\Lambda_E(r)$ and $\Lambda_S(r)$. However, only $\Lambda_S(r)$ is included in *South*(r).
- *West*(r) sector: it is adjacent to *South*(r) in clockwise direction and its angular sector α_W is equal to $2\gamma^*$, that is $\pi/4$. Note that *West*(r) is delimited by $\Lambda_W(r)$ and $\Lambda_N(r)$. However, only $\Lambda_W(r)$ is a part of *West*(r) sector.
- *North*(r) sector: this is the remaining sector, and its angular sector α_N is equal to $6\gamma^*$, that is $3\pi/4$. Note that *North*(r) is delimited by $\Lambda_N(r)$ and $\Lambda_W(r)$. However, only $\Lambda_N(r)$ is included in *North*(r) sector.

In the following, we will describe the possible configurations of the two robots, given the above partitions.

3.2 Valid Configurations

We consider two robots r and r' that are equipped with compasses that can diverge by as much as $2\gamma^*$, that is $\pi/4$. Let r and r' divide the plane as described in Sect. 3.1. Then, r and r' can only be in one of the following valid configurations, or a symmetric configuration:

1. Configuration *North*/*South*: $r' \in$ *South*(r) (i.e., robot r sees r' on its *South* sector) and $r \in$ *North*(r'), or vice versa.
2. Configuration *North*/*East*: $r' \in$ *East*(r) and $r \in$ *North*(r'), or vice versa.
3. Configuration *North*/*West*: $r' \in$ *West*(r) and $r \in$ *North*(r'), or vice versa.
4. Configuration *East*/*West*: $r' \in$ *West*(r) and $r \in$ *East*(r'), or vice versa.
5. Configuration *East*/*South*: $r' \in$ *South*(r) and $r \in$ *East*(r'), or vice versa.

Based on the partitions described in Sect. 3.1, Table 1 summarizes possible and impossible configurations when robots's compasses are inaccurate by at most $\gamma^* = \pi/8$, with respect to some global north. By design, the partitions prevent the occurrence of some undesirable configurations, such as *North*/*North*, that could lead to a deadlock situation by using the algorithm[6](see Sect. 3.3).

[6] It is important to mention that when γ^* is equal to zero, i.e., when the compasses of r and r' are consistent or, the configurations *East*/*South* and *North*/*West* are impossible.

Algorithm 1. Gathering Two Robots with $\pi/8$-Inaccurate Compasses

1: Robot r divides the plane into four sectors: *North*, *South*, *East* and *West* (see Sect. 3.1);
2: r' := the other robot visible to r at some time t;
3: **if** (r sees only itself) **then** {*gathering terminated*}
4: Do_nothing();
5: **else**
6: **if** ($|South(r)| > 0$) **then** {r' is to the *South*: *direct move*}
7: Move(r');
8: **else if** ($|East(r)| > 0$) **then** {r' is to the *East*: *side move up*}
9: $\Psi_E(r)$:= the parallel axis to $\Lambda_E(r)$ passing through r';
10: $H := \Lambda_N(r) \cap \Psi_E(r)$ (see Fig. **??**);
11: $Goal := p \in \Lambda_N(r)$ such that $\|\overline{rGoal}\| > \|\overline{rH}\|$ and $r\widehat{Goal}r' \geq \widehat{rr'Goal}$;
12: Move($Goal$);
13: **else if** ($|West(r)| > 0$) **then** {r' is to the *West*: *side move down*}
14: $\Psi_W(r)$:= the parallel axis to $\Lambda_W(r)$ passing through r';
15: $H' := \Lambda_S(r) \cap \Psi_W(r)$ (see Fig. **??**);
16: $Goal := p \in \Lambda_S(r)$ such that $\|\overline{rGoal}\| > \|\overline{rH'}\|$ and $r\widehat{Goal}r' \geq \widehat{rr'Goal}$;
17: Move($Goal$);
18: **else** {r' is to the *North*: *no movement*.}
19: Do_nothing();
20: **end if**
21: **end if**

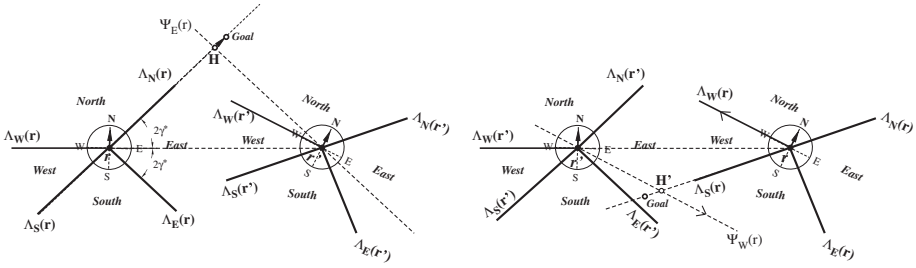
Table 1. Different configurations and movements of robot r and r' ($\gamma^* = \pi/8$)

	Robot r			
Robot r'	*North* (no movement)	*South* (direct move)	*East* (side move up)	*West* (side move down)
North (no movement)	no	◯	◯	◯
South (direct move)	◯	no	◯	no
East (side move up)	◯	◯	no	◯
West (side move down)	◯	no	◯	no

3.3 Movements

The algorithm is given in Algorithm 1, and Table 1 summarizes the different movements of robot r and r' (the table is symmetrical). Let us consider the movement of robot r. First, robot r creates the four sectors, and then it decides its movement based on the sector in which it has locates robot r', as follows:

- *No movement* (Algorithm1:line 18): If $r' \in North(r)$, then r does not move. That is, if r sees r' in its *North* sector, it remains stationary.
- *Direct move* (Algorithm1:line 6): If $r' \in South(r)$, then r moves directly in a linear movement to r'.
- *Side move up* (Algorithm1:line 8): If $r' \in East(r)$, then r performs a *side move up*. The need for such a move is explained as follows: given the valid



(a) Side move up on $\Lambda_N(r)$: $r' \in East(r)$, then r performs a *side move up* to *Goal*.

(b) Side move down on $\Lambda_S(r)$: $r' \in West(r)$, then r performs a *side move down* to *Goal*.

Fig. 2. Principle of the algorithm

configurations described in Sect. 3.2, if $r' \in East(r)$, then $r \in South(r')$ or $r \in North(r')$ or $r \in West(r')$. Robot r (also r') cannot figure out in which configuration they are, for instance the *East*/*South* or *North*/*East* configuration. Then, if we let robot r make a *direct move* toward r', then in the case when both robots are in the configuration *East*/*South*, they will swap their positions endlessly. Also, if we make robot r stay still, then, if both robots are in the configuration *North*/*East*, none of the robots will ever move and they will always remain in a deadlock situation. Therefore, the aim of this *side move up* is to bring both robots eventually into the configuration *North*/*South*, where one robot can move and the other remains stationary, which can lead to gathering by our algorithm.

A *side move up* is computed by robot r as follows: let H be the intersection of $\Lambda_N(r)$ and the axis $\Psi_E(r)$, with $\Psi_E(r)$ parallel to $\Lambda_E(r)$ passing through robot r'. Then, the destination *Goal* of robot r is any point that belongs to $\Lambda_N(r)$, such that the distance $\|\overline{rGoal}\| > \|\overline{rH}\|$, and the angle $\widehat{rGoalr'}$ is greater than or equal to the angle $\widehat{rr'Goal}$ (refer to Fig. 2(a)).

– *Side move down* (Algorithm1:line 13): If $r' \in West(r)$, then r performs a *side move down*. The aim of this move is similar to the *side move up*, and it is computed by robot r as follows: let H' be the intersection of $\Lambda_S(r)$ and the axis $\Psi_W(r)$, with $\Psi_W(r)$ parallel to $\Lambda_W(r)$ passing through robot r' (refer to Fig. 2(b)). Then, the destination *Goal* of robot r is any point that belongs to $\Lambda_S(r)$, such that the distance $\|\overline{rGoal}\| > \|\overline{rH'}\|$, and the angle $\widehat{rGoalr'}$ is greater than or equal to the angle $\widehat{rr'Goal}$ (refer to Fig. 2(b)).

4 Correctness

In this section, we will prove that our algorithm solves the problem of gathering two robots in a finite time, assuming $\pi/8$-Inaccurate compasses. Due to space limitations, we only give the complete proof of two lemmas that are central to

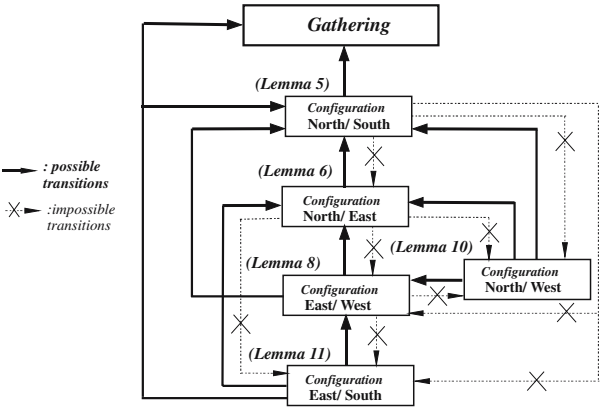


Fig. 3. Different configurations allowed by Algorithm 1, and their transformation to gathering

the paper. For all other lemmas, we give an outline of the idea behind the proof. All the complete proofs can be found in the technical report version [16]. We first state some lemmas, to illustrate that some incompatible configurations are ruled out by the algorithm. Second, we show how any possible configuration by the algorithm is transformed into gathering in a finite time. Fig. 3 summarizes the different possible configurations, and their transformation to gathering.

Under the partitions described in Sect. 3.1 and by considering $\gamma^* = \pi/8$, trivially, we derive the following two lemmas:

Lemma 1. *Under the partitions, and assuming $\pi/8$-Inaccurate compasses, the system can not be in the configuration North/North or East/East or South/South or West/West at any time t.*

Lemma 2. *Under the partitions, and assuming $\pi/8$-Inaccurate compasses, the system can not be in the configuration West/South at any time t.*

From the above two lemmas, we derive the following theorem:

Theorem 1. *By the algorithm, the possible configurations are North/South, North/East, North/West, East/West and East/South, and their symmetric ones (i,e. South/North, East/North, West/North, West/East and South/East).*

Lemma 3. *Given a robot r and its target point H with $r \neq H$, r reaches its target in a finite number of steps.*

Proof (Lemma 3). The proof derives from Assumption 1. In one cycle, r travels at least $\delta_r > 0$ of the desired distance. Besides, by Assumption 2, the cycle of a robot is finite. Thus, the number of steps required for robot r to reach its destination H is at most $\lceil \|\overline{rH}\|/\delta_r \rceil$, which is finite, and the lemma holds.

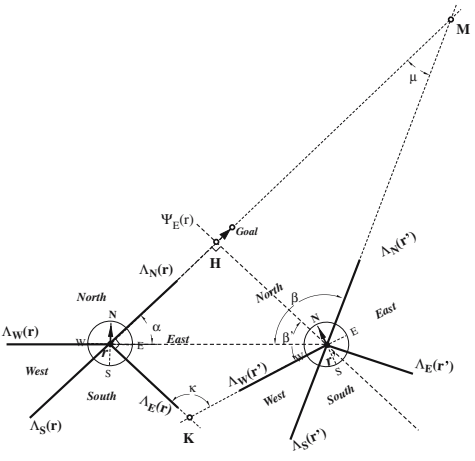


Fig. 4. Transformation of *North/East* configuration

Lemma 4. *Given two robots r and r' that are in the configuration North/East or East/West or East/South at some time t_0, with $r' \in East(r)$ and r is either in $North(r')$ or $West(r')$ or $South(r')$. Then, the destination Goal computed by robot r (resulting from its side move up) is on the $North(r')$.*

Proof (Lemma 4).

We will prove the *North/East* configuration only. The *East/West* and *East/South* configurations can be proved in a similar way.

Assume that $r' \in East(r)$ and $r \in North(r')$ at time t_0. First, observe that if $\Lambda_N(r) \cap \Lambda_N(r') = \emptyset$ (i.e., $\Lambda_N(r)$ and $\Lambda_N(r')$ are parallel or do not intersect), then $Goal \in North(r')$ because $r \in North(r')$, and $Goal \in \Lambda_N(r)$.

Now assume that, $\Lambda_N(r) \cap \Lambda_N(r') = M$. Let $H = \Psi_E(r) \cap \Lambda_N(r)$ (refer to Fig. 4). To show that $Goal \in North(r')$, we will show that, always, $Goal \in \triangle(r, r', M)$. In other words, we need to show that $H \in \triangle(r, r', M)$ and the distance $\|\overline{HM}\| \neq 0$.

Consider the triangle $\triangle(r, r', M)$. Let α, β, and μ denote the angles at r, r' and M that are within the triangle $\triangle(r, r', M)$, respectively. First, if all three angles α, β, and μ are acute, then obviously the foot H of the perpendicular starting from r' is inside $\triangle(r, r', M)$, and $\|\overline{HM}\| \neq 0$. Second, if the angle β at r' is obtuse, then again the foot H of the perpendicular starting from r' is inside $\triangle(r, r', M)$, and $\|\overline{HM}\| \neq 0$. Now consider the angle α at r. By hypothesis, α_E is equal to $\pi/2$. This means that α cannot be an obtuse angle, and it is at most $\pi/2$. In this later case where $\alpha = \pi/2$, we have the foot H of the perpendicular starting from r' equal to r (in this case $\Lambda_E(r)$ passes by r'), and the triangle $\triangle(r', r, M)$ has a right angle at r. Consequently, $\|\overline{rM}\| = \|\overline{HM}\| \neq 0$ and $Goal \in \triangle(r, r', M)$.

Now, we will prove that the angle μ at M can not be an obtuse angle (because if μ is an obtuse angle, H is outside $\triangle(r, r', M)$). Let $K = \Lambda_E(r) \cap \Lambda_W(r')$ and

κ be the angle at K. We also denote by β' the angle at r' formed by $\Psi_E(r)$ and $\Lambda_W(r')$. Consider the quadrilateral formed by r, H, r' and K. Then, we have: (1) $\kappa+\beta' = \pi$ since the respective angles at r and H are equal to $\pi/2$. Consider now the quadrilateral formed by r, K, r' and M. Then, we have: (2) $\kappa + \mu = 3\pi/4$ since $\alpha_E(r)$ is equal to $\pi/2$, and $\alpha_N(r')$ is equal to $3\pi/4$ by hypothesis. By subtraction of (1) from (2), we get: (3) $\beta' - \mu = \pi/4$. By assumption, $\beta' < 3\pi/4$ because $\Psi_E(r)$ can not be equal to $\Lambda_N(r')$ as $\Lambda_N(r')$ can not be perpendicular to $\Lambda_N(r)$ by the partitions described in Sect. 3.1. Consequently, the angle μ at M is less than $\pi/2$. Thus, μ can not be an obtuse angle. As a result, in all cases the foot H of the perpendicular starting from r' is inside the triangle $\triangle(r, r', M)$, and $\|\overline{HM}\| \neq 0$. Then, $\forall p \in \overline{HM}$, $p \in North(r')$. We have by the algorithm, $\widehat{rGoalr'} \geq \widehat{rr'Goal}$. Since μ is not an obtuse angle and $\widehat{rr'M}$ can be an obtuse angle, then the triangle $\triangle(r, r', Goal)$ is included in $\triangle(r, r', M)$. This proves that $Goal \in \triangle(r, r', M)$, and thus $Goal \in North(r')$. This completes the proof.

In the following, we will show the different possible transitions that each valid configuration can take, in order to reach gathering in a finite time. The impossible transitions can be derived implicitly, so we do not prove them explicitly.

4.1 Transition of *North/South* Configuration to Gathering

Lemma 5. *Let r and r' be two robots that are in the configuration North/South with $r' \in South(r)$ at some time t_0. Then, there is a time $\bar{t} > t_0$ when r and r' gather at the same point. Moreover, r and r' can not shift to any other configuration except gathering.*

Proof (Lemma 5). By the algorithm, r will perform a *direct move* toward r'. Also, during the movement of r, r' is unable to move. Consequently, by Lemma 3, r reaches r' in a finite time. This terminates the proof.

4.2 Transition of *North/East* Configuration to Gathering

Lemma 6. *Let r and r' be two robots that are in the configuration North/East with $r' \in East(r)$, and $r \in North(r')$ at some time t_0. Then, there is a finite time $\bar{t}$ at which this configuration is transformed into North/South configuration with $r' \in South(r)$. Moreover, r and r' can not shift to any other configuration except the North/South configuration.*

Proof (Lemma 6). The proof is a direct consequence from Lemma 4. Let *Goal* be the destination of r. Initially, $r \in North(r')$. Besides, by Lemma 4, $\forall p \in \overline{rGoal}$, $p \in North(r')$. Then, r' is unable to move during the movement of r to *Goal*. When r reaches its destination *Goal*, $\Lambda_E(r)$ is above r', thus $r' \in South(r)$. Consequently, r and r' enter the configuration *North/South* in a finite time.

From Lemma 5 and Lemma 6, we conclude that:

Theorem 2. *Any North/East configuration of two robots equipped with $\pi/8$-Inaccurate compasses is transformed after a finite time to gathering.*

4.3 Transition of *East/West* Configuration to Gathering

Lemma 7. *Given two robots r and r' at some time t_0, where r and r' are in the configuration East/West, with $r \in West(r')$ and $r' \in East(r)$, then the destination Goal$'$ of r' (resulting from its side move down) belongs to $East(r)$ or $South(r)$.*

Proof (Lemma 7). Let $H' = \Psi_W(r') \cap \Lambda_S(r')$. Consider the triangle $\triangle(r, r', Goal')$, and let α, α' and β be the angles at r, r' and $Goal'$, respectively. By hypothesis, $\alpha' \leq \alpha_W = \pi/4$. Then, $\alpha+\beta \leq 3\pi/4$. By the algorithm, $\alpha \leq \beta$. Thus, $\alpha \leq 3\pi/8 < \pi/2$. Let $M = \Lambda_E(r) \cap \Lambda_S(r')$. Then, the angle $\widehat{r'rM} \leq \pi/4$ since r and r' are in the configuration *East/West*. It follows that if $Goal' \in \overline{H'M}$, then $Goal' \in East(r)$. Otherwise, $Goal' \in South(r)$.

Lemma 8. *Let r and r' be two robots that are in the configuration East/West, with $r' \in East(r)$, and $r \in West(r')$ at some time t_0. Then, there is a finite time $\bar{t}$ in which this configuration is transformed into North/East or North/South configuration. Moreover, r and r' cannot enter any other configuration except the North/East or North/South configuration.*

Proof (Lemma 8). We distinguish several cases depending on the movement of each robot. We assume that both r and r' always reach their final destinations. All other cases where r or r' end their moves before destination are easy to deduce from previous lemmas.

1. **r moves/ r' does not move:** By the algorithm, r will perform a *side move up*. Let *Goal* be the destination of r and $\bar{t}$ be the time when r reaches its target. At $\bar{t}$, we have $r' \in South(r)$ (since at $\bar{t}$, r' becomes below $\Lambda_E(r)$). In addition, by Lemma 4, $Goal \in North(r')$. Then, at $\bar{t}$, $r \in North(r')$. Consequently, r and r' become in the configuration *North/South* in a finite time.
2. **r' moves/ r does not move:** By the algorithm, r' will perform a *side move down*. Let $Goal'$ be its destination and $\bar{t'}$ be the time when r' reaches $Goal'$. At time $\bar{t'}$, r is above $\Lambda_W(r')$, thus $r \in North(r')$. In addition, by Lemma 7, $r' \in East(r)$ or $r' \in South(r)$ at $\bar{t'}$. Consequently, r and r' leave the configuration *East/West* in a finite number of steps, and become in the configuration *East/North* or *North/South*.
3. **both r and r' move:** By the algorithm, r will perform a *side move up* and r' will perform a *side move down*. Let *Goal* and $Goal'$ be their respective destinations and $\bar{t}$ and $\bar{t'}$ be the time when they end their moves, respectively. At $\bar{t}$, $\forall p$ that is below $\Lambda_E(r(\bar{t}))$, $p \in South(r)$. Since, at $\bar{t}$, $r' \in \overline{r'Goal'}$, and by Lemma 7, $Goal' \in East(r(t_0))$ or $Goal' \in South(r(t_0))$. Thus, $r' \in South(r)$ at $\bar{t}$ because $\Lambda_E(r(\bar{t}))$ is above $Goal'$ and r'.
   When r' reaches $Goal'$, r is above $\Lambda_W(r')$. Consequently, at $\bar{t'}$, $r \in North(r')$. Since, r and r' reach their respective target in a finite time, we hence conclude that they become in the configuration *North/South* in a finite time.

From Lemma 8, Lemma 5 and Theorem 2, we conclude:

Theorem 3. *Any East/West configuration of two robots equipped with $\pi/8$-Inaccurate compasses is transformed after a finite time to the gathering.*

4.4 Transition of *North/West* Configuration to Gathering

Lemma 9. *Given two robots r and r' at some time t_0, where r and r' are in the configuration North/West, with $r \in West(r')$ and $r' \in North(r)$, then the destination $Goal'$ of r' (resulting from its side move down) belongs to $East(r)$.*

The proof is very similar to the proof of Lemma 7, thus omitted here.

Lemma 10. *Let r and r' be two robots that are in the configuration North/West, with $r \in West(r')$, and $r' \in North(r)$ at some time t_0. Then, there is a finite time $\bar{t}$ in which this configuration is transformed into North/East or East/West or North/South configuration. Moreover, r and r' can not enter any other configuration except the North/East or East/West or North/South configuration.*

Proof (Lemma 10).

By the algorithm, r' will make a *side move down*. Let $Goal'$ be its destination. Then, by Lemma 9, $Goal' \in East(r)$. As long as $r' \in North(r)$, r remains stationary. While r' is moving toward its target, it crosses $East(r)$ sector. Then, r and r' become in the configuration *East/West* if $\Lambda_W(r')$ is still above r. Otherwise, they enter the configuration *North/East*, with $r \in North(r')$ if r' reaches $Goal'$ and r still did not move. Finally, r and r' enter the configuration *North/South* if r performs a look operation when $r' \in East(r)$, and moves to it destination. From Lemma 3, these transformations are done in a finite time, and the lemma holds.

From Lemma 10, Theorem 2 and Theorem 3, we conclude:

Theorem 4. *Any North/West configuration of two robots equipped with $\pi/8$-Inaccurate compasses is transformed after a finite time to gathering.*

4.5 Transition of *East/South* Configuration to Gathering

Lemma 11. *Let r and r' be two robots that are in the configuration East/South at some time t_0, with $r' \in East(r)$ and $r \in South(r')$ Then, there is a finite time t in which this configuration is transformed into North/South or North/East or East/West or the gathering.*

Proof (Lemma 11). By the algorithm, r' will make a *direct move* toward r, and r will make a *side move up*. Then, we distinguish several cases, depending on where each robot sees the other one, and where it ends its move. By using similar arguments as in previous lemmas, it is easy to show that r and r' shift to the *North/South* or *North/East* or *East/West* configuration or the gathering in a finite time.

From Lemma 5, Lemma 11, Theorem 2 and Theorem 3, we conclude that:

Theorem 5. *Any East/South configuration of two robots equipped with* $\pi/8$*-Inaccurate compasses is transformed in a finite time to gathering.*

Theorem 6. *In a system, with* 2 *anonymous, oblivious mobile robots relying on inaccurate compasses, the gathering problem is solvable in a finite time for* $\pi/8$*-Inaccurate compasses.*

Proof (Theorem 6).

Theorem 1 states the different valid configurations by the algorithm. Also, from Lemma 5, Theorem 2, Theorem 3, Theorem 4 and Theorem 5, any valid configuration is transformed into gathering in a finite time (see Fig. 3), thus the theorem holds.

5 Conclusion

In this paper, we concentrate on the gathering of autonomous mobile robots when their compasses are subject to errors. In particular, we have studied the solvability of the gathering of two asynchronous mobile robots in the face of compass inaccuracies, and relying on oblivious computations. We thus provided an algorithm that gathers in a finite number of steps, two asynchronous oblivious mobile robots equipped with compasses that can differ by as much as $\pi/4$.

The benefit of our algorithm is that we solve the problem with inaccurate compasses. Moreover, our algorithm is self-stabilizing and tolerates any number of transient errors. We can also argue that even with weaker compasses that fluctuate for some arbitrary periods, and eventually they become constant with bounded errors that are less than or equal to $\pi/4$, our algorithm is still valid and solves the problem in a finite time.

Acknowledgments

We are especially grateful to Hirotaka Ono, Matthias Wiesmann and Rami Yared for their insightful comments regarding this work.

References

1. Agmon, N., Peleg, D.: Fault-tolerant gathering algorithms for autonomous mobile robots. In: Proc. 15th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA'04), Philadelphia, PA, USA (2004) 1070–1078
2. 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) (1999) 818–828
3. Cieliebak, M.: Gathering non-oblivious mobile robots. In: Proc. 6th Latin American Symp. on Theoretical Informatics (LATIN'04). (2004) 577–588
4. Cieliebak, M., Flocchini, P., Prencipe, G., Santoro, N.: Solving the robots gathering problem. In: Proc. Intl. Colloquium on Automata, Languages and Programming (ICALP'03). (2003) 1181–1196

5. Cohen, R., Peleg, D.: Convergence of Autonomous Mobile Robots With Inaccurate Sensors and Movements. In: Proc. 23rd International Symposium on Theoretical Aspects of Computer Science (STACS'06). LNCS (2006)
6. Défago, X., Gradinariu, M., Messika, S., Raipin-Parvédy, P.: Fault-tolerant and self-stabilizing mobile robots gathering. In: Proc. 20th Intl. Symp. on Distributed Computing (DISC'06). LNCS 4167 (2006) 46–60
7. Dolev, S.: Self-Stabilization. MIT Press (2000)
8. Fisher, M., J., Lynch, N., A., Paterson, M. S.: Impossibility of distributed consensus with one faulty process. J. ACM. **32**(2) (1985) 374–382
9. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous robots with limited visibility. Theor. Comput. Sci. **337**(1–3) (2005) 147–168
10. Imazu, H., Itoh, N., Katayama, Y., Inuzuka, N., Wada, K.: A Gathering Problem for Autonomous Mobile Robots with Disagreement in Compasses (in Japanese). In: 1st Workshop on Theoretical Computer Science in Izumo, Japan (2005) 43–46
11. Prencipe, G.: Instantaneous Actions vs. Full Asynchronicity: Controlling and Coordinating a Set of Autonomous Mobile Robots. In Proc. 7th Italian Conference on Theoretical Computer Science (ICTCS'01). (2001) 154–171
12. Prencipe, G.: CORDA: Distributed coordination of a set of autonomous mobile robots. In: Proc. 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01), Bertinoro, Italy (2001) 185–190
13. Prencipe, G.: On the feasibility of gathering by autonomous mobile robots. In: Proc. Colloquium on Structural Information and Communication Complexity (SIROCCO'05). (2005) 246–261
14. Schneider, M.: Self-stabilization. ACM Computing Surveys **25**(1) (1993) 45–67
15. Souissi, S., Défago, X., Yamashita, M.: Using Eventually Consistent Compasses to Gather Oblivious Mobile Robots with Limited Visibility. In: Proc. 8th Intl. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS'06). LNCS 4280 (2006) 471–487
16. Souissi, S., Défago, X., Yamashita, M.: Gathering Asynchronous Mobile Robots with Inaccurate Compasses. Research Report (JAIST), IS-RR-2006-014, Hokuriku, Japan (2006)
17. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM Journal on Computing **28**(4) (1999) 1347–1363
18. Tel, G.: Introduction to Distributed Algorithms. In: Cambridge University Press (2001)

Gathering Few Fat Mobile Robots in the Plane

Jurek Czyzowicz[1,⋆], Leszek Gąsieniec[2], and Andrzej Pelc[1,3]

[1] Département d'informatique, Université du Québec en Outaouais, Gatineau, Québec J8X 3X7, Canada
{jurek, pelc}@uqo.ca
[2] Department of Computer Science, The University of Liverpool, Liverpool, L69 7ZF, UK
leszek@csc.liv.ac.uk
[3] Research partially supported by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

Abstract. Autonomous identical robots represented by unit discs move deterministically in the plane. They do not have any common coordinate system, do not communicate, do not have memory of the past and are totally asynchronous. Gathering such robots means forming a configuration for which the union of all discs representing them is connected. We solve the gathering problem for at most four robots. This is the first algorithmic result on gathering robots represented by two-dimensional figures rather than points in the plain: we call such robots *fat*.

1 Introduction

1.1 The Background and the Problem

Using teams of simple, low-cost robots is an important way of accomplishing large mechanical tasks in dangerous or hostile environments. Systems of such autonomous robots have been extensively studied in the robotics and artificial intelligence community [3,4,11,12,13,14,15]. Recently, algorithmic aspects of distributed coordination of teams of robots freely moving in the plane have been investigated by many researchers [1,2,5,6,7,8,9,10,17,18,19,20]. Models of perception and motion of robots aimed at grasping the intuition that these are weak-performance devices that can be cheaply mass-produced. One of the most extensively studied is the asynchronous model of [6,7,8,9,18]. In this model robots are identical, anonymous, do not have any common coordinate system, do not communicate, do not have memory of the past and operate asynchronously in Look-Compute-Move cycles. Each robot, represented as a point in the plane, wakes up at times controlled by the adversary, observes positions of all other robots *at this time* then computes a target point and starts moving towards it at a speed controlled by the adversary. The adversary may also stop the robot before it reaches its target point, thus finishing the cycle. The aim is gathering

⋆ Research partially supported by NSERC discovery grant.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 350–364, 2006.

all robots in one point of the plane. Gathering is one of the basic primitive operations in controlling teams of autonomous moving robots and has been also studied in robotics and artificial intelligence [3,11,12].

This scenario is indeed very weak and thus can be potentially applied to a large class of autonomous moving devices. However, the model is not realistic in one aspect: representation of robots by points. In reality, even very small robots occupy some space, which results in two important complications: some robots may prevent full visibility of others and some robots may mechanically obstruct the motion of others, staying or getting in their line of move. The aim of the present paper is to study the gathering problem in the asynchronous model, at the same time addressing the above-mentioned issues. We represent robots as unit discs in the plane (we call them "fat" robots to distinguish our scenario from the previous point representation). We keep the Look-Compute-Move paradigm but add the two aspects resulting from the "fatness" of robots. First, robot R_1 can see robot R_2, if there exist points x and y in circles bounding R_1 and R_2, respectively, such that the segment xy does not contain any point of any other robot. Second, if a robot R touches another robot (i.e., the circles representing these robots become tangent) then both robots stop and this ends their current cycle.

Since for fat robots it is impossible to gather them in one point (robots stop at a touch and thus cannot penetrate each other) we change the definition of gathering accordingly. Gathering fat robots means forming a configuration for which the union of all discs representing them is connected. Moreover, all robots must have full visibility to be aware that gathering is accomplished.

It turns out that adding the realistic aspect of fatness significantly complicates the task of gathering. To see this, consider a team of 4 robots whose centers are situated on intersecting non-perpendicular lines, one robot in each of the four halflines. This is what is called in [7] a biangular configuration . The center of biangularity (which in this case is the intersection point x of the lines) is invariant under straight moves in its direction of all robots, regardless at what relative speed they move towards it. Hence if robots are represented by points, a gathering algorithm in this particular case is straightforward: each robot computes the point x and moves towards it. Eventually, all robots will reach the point x. However, for fat robots, this is not a correct algorithm. Indeed, since lines are not perpendicular, the adversary may control the speed of robots so that two pairs of robots are formed, the robots in each of them obstructing each other's moves, without forming a connected configuration.

The above example shows how mechanical obstruction of one robot by another may cause problems. Visibility issues also significantly complicate gathering algorithms. For example, it was proved in [8] that when robots are represented by points then the algorithm consisting in always going towards the gravity center converges, i.e., permits to get all robots inside an arbitrarily small circle. However, for fat robots the center of gravity of the entire system may be impossible to compute by some robots that do not have full visibility. Even when forming a connected configuration is possible using a variant of this algorithm, consisting

in going towards the gravity center of the *visible part* of the configuration, some robots may never be aware that the task is accomplished. This is the case for 3 robots in a straight line. They will eventually form a connected configuration by a sequence of moves along this line but the two external robots will never know when this is done.

The aim of this paper is to present a gathering algorithm for three and for four robots (gathering one or two fat robots is straightforward). We first describe our model in detail, recalling features of the Look-Compute-Move paradigm and emphasizing differences between our model and the point-representation model from [6,7,8,9,18]. Then we present the gathering algorithm for three robots, which is much simpler than for four robots and provides a good introduction to it, as it already has to cope with some of the main difficulties in an easier situation. Most of the paper is devoted to the design and analysis of the gathering algorithm for 4 robots. The remaining challenge is to generalize it to an arbitrary number of robots. We were unable to do it.

1.2 Related Work

A heuristic approach to the gathering problem, from the point of view of robotics and artificial intelligence was presented in [3,4,11,12,13,14,15]. In most of the algorithmic literature robots were represented by points and full visibility was assumed. Gathering algorithms in the semi-synchronous model, where robots operate in synchronous cycles but some robots may skip some cycles, were investigated in [20]. The asynchronous model was first described in [9]. In [7] the authors showed a gathering algorithm for $n > 2$ robots assuming *multiplicity detection*, i.e., the capability of a robot to tell if a given point contains one or more robots. In [18] it was proved that without multiplicity detection the gathering problem is unsolvable. In [8] it was proved that the gravitational algorithm consisting in moving towards the center of gravity enables getting all robots in an arbitrarily small circle. Fault-tolerant algorithms for gathering were studied in [1]. The gathering problem was also studied with limited visibility [2,10]. However, limitation was not caused by other robots obstructing the view but by imposing for each robot a radius of vision. In [2] the authors provide theoretical analysis of a gathering algorithm with robots represented by points and then perform simulations in the more realistic setting where robots are represented by discs. To the best of our knowledge ours is the first paper presenting a provably correct gathering algorithm for fat robots.

2 The Model

Robots are represented by closed unit discs in the plane. Robots are identical, anonymous and undistinguishable. Since all robots have radius 1, they have a common measure of unit (unlike in the model from [9]) but do not have any common system of coordinates, similarly as in this model. Robot R_1 can see robot R_2, if there exist points x and y in circles bounding R_1 and R_2, respectively, such that the segment xy does not contain any point of any other robot. (In

particular, each robot can see itself). Notice that if a robot R_1 can see robot R_2, it can always see some non-zero arc of its bounding circle and thus it may compute its center. This definition of visibility results from the "fatness" of our robots and differs from [9], where full visibility of robots represented by points was assumed. We say that a robot has full visibility if it can see all the robots, and we say that there is full visibility in a given position of robots, if all of them have full visibility.

Each of the robots R_i in the system executes asynchronously simple cycles consisting of three steps.

- **Look:** Identify locations of all robots *visible* to R_i. The result of this step is a set $\mathcal{P}$ of centers of these robots, called a *configuration*.
- **Compute:** Execute the algorithm on input $\mathcal{P}$, and output a target point p.
- **Move:** Move on a straight line towards point p. If during this motion the robot touches some other robot, it stops and finishes the current cycle. (This is another difference from the model in [9], where robots modeled by points could "pass through each other" not noticing it. This is hardly the case in a physical environment). The adversary may also stop the robot at any point before the target (thus finishing the current cycle), as long as at least a specified distance ϵ has been traversed. The robots do not know ϵ. If the robot does not encounter another robot on its way and the adversary does not stop it, the robot stops when its center reaches point p, and finishes the current cycle.

Notice that the above asynchronous cycle paradigm imposes inherent limitations on the perception, computing and moving capabilities of the robots. A robot computes a target point on the basis of a previously perceived configuration and may start moving towards it when the configuration has already changed. In fact, a robot R_1 can see robot R_2 while R_2 is moving, and R_1 does not realize this fact. Also robots do not have any memory from the past cycles. For example, two robots at some distance cannot accomplish the simple task of meeting in the middle of the segment joining them because they may be stopped on the way to it and, not remembering their previous positions, are unable to recompute this middle point. Moreover, robots cannot communicate directly: their only way of communication is by observing the positions of others, which, as we mentioned, may become obsolete at the time of moving.

The adversary in this model has a lot of power and is restricted only in two ways. First, each of the steps takes an unspecified but finite time. This is the usual restriction on adversaries in asynchronous systems, otherwise no task can be accomplished in finite time. The other restriction is that the adversary has the obligation to let the robot traverse at least a distance ϵ during a move step, unless the target point was closer. Otherwise no gathering is possible because the adversary could exercise its stopping prerogative in consecutive cycles after $1, \frac{1}{2}, \frac{1}{4}, \ldots$, thus keeping each robot from traversing a distance of more than 2 (which reminds the Achilles and tortoise paradox).

A *connected configuration* of robots is their position in the plane, such that between any two points of any two robots there exists a polygonal line each of

whose points belongs to some robot. Robots accomplish *gathering*, if they get to some connected configuration and all of them can see all robots (and thus be aware that a connected configuration is achieved). A gathering algorithm for n robots stops if robots accomplish gathering. (We assume that robots know n). A gathering algorithm is correct if it accomplishes gathering starting from any initial position of the robots, i.e. any configuration in which no pair of robots shares an internal point.

Note that look and move steps in each cycle do not depend on the gathering algorithm. A given algorithm prescribes only how to compute the target point p depending on any possible configuration $\mathcal{P}$.

In the sequel we will often identify robots with their centers, thus saying, e.g., "robots form a triangle" instead of "centers of robots form a triangle", "robots are at distance D" instead of "centers of robots are at distance D", etc.

3 Gathering Three Robots

In this section we design an algorithm for gathering three robots in the plane. As we will see, this is a much easier task, nevertheless it already exhibits some of the difficulties with which we will have to cope later for four robots. First notice that at each time one of two situations may happen: either robots form a triangle and then full visibility of all robots is assured, or robots are collinear, and then all robots are aware of it, although two of them do not have full visibility.

We will use the following geometric fact.

Fact 3.1. *Inside every triangle abc with all angles smaller than* $120°$*, there exists a unique point* R *such that all angles* $\angle ARB, \angle BRC, \angle CRA$ *are exactly* $120°$.

Algorithm ThreeRobots
if robots form a triangle with all angles smaller than $120°$
then compute the point R such that $\angle ARB = \angle BRC = \angle CRA = 120°$
and compute target point p between your center and R,
at distance $2\sqrt{3}/3$ from R
else
if robots form a triangle with an angle $\angle ABC$ at least $120°$
then compute the target point p on the segment between your center and B, at distance 2 from B {B remains idle}
else {robots are collinear and their centers form line L}
if two other robots are visible
then compute target point at distance 1, such that the segment between this point and your center is perpendicular to L.
else compute target point equal to your own center

Remark. Note that, when the three robots are aligned along line l, the algorithm sends the middle robot perpendicularly to l at distance 1. By symmetry the robot may choose each of the two directions to move away from l. We assume that in such circumstances the robot breaks the symmetry individually by using its local

system of coordinates, unknown to other robots. Similar situation will occur in the case of the algorithm for four robots.

Theorem 1. *Algorithm ThreeRobots is a correct gathering algorithm for three robots.*

Proof. If robots form a triangle with all angles smaller than 120° then the unique point R from Fact 3.1 can be computed by all robots. Robots move towards this point until they get at distance $2\sqrt{3}/3$ from R. Note that the point R remains the same during the whole process and the characteristic of the triangle that all of its angles are smaller than than 120° remains unchanged during the process. (This is important because it implies that all robots execute the same clause of the algorithm, regardless of their relative speeds and possible interruptions of the moves, controlled by the adversary.) In this way robots get at distance $2\sqrt{3}/3$ from R. Since all angles between trajectories of robots are 120°, one robot cannot obstruct another one on its way to the target point, regardless of their relative speed. When all robots get at distance $2\sqrt{3}/3$ from R, they form a connected configuration and the algorithm stops. During the entire process all robots have full visibility, hence when they eventually form a connected configuration, all of them are aware of it. Note that, in case some robot is initially closer that $2\sqrt{3}/3$ from R, because of the asynchrony of the process the gathering configuration is not deterministic. Indeed, if the robot which is closer than $2\sqrt{3}/3$ to R is slow in reaching its target point, the other two robots may bump on it, preventing it from terminating its move step.

If robots form a triangle with an angle $\angle ABC$ at least 120°, such an angle must be unique. Robot B is then unique (known to all) and it stays idle. Other robots move towards it and stop at distance 2 from it. Notice that, regardless of the relative speed of the robots, the property $\angle ABC \geq 120°$ is satisfied during the entire process, hence the same clause of the algorithm is executed by all robots. Since the angle is large, moving robots do not obstruct each other on their way to the target. Again, during the entire process all robots have full visibility, thus they notice when gathering is completed.

Finally, if robots are collinear, only one of them (call it central) has full visibility (can see two other robots). This robot departs perpendicularly from the line L of the robots, at distance 1. Before it starts moving, the two other robots stay idle. After it started moving, other robots, after seeing it, start moving towards it. Notice an important subtlety. The move of the central robot may be slow. In the meantime, other robots may execute their look step. Seeing the central robot "on its way" to the target point, they start moving towards its temporary location and not towards its final destination. Thus the trajectory of the other robots will in general be a polygonal convex line whose final segment will be towards the final destination of the central robot. This final destination does not need to be the initially calculated point at distance 1 from line L because one of the other robots may hit the central one on its way to this point, thus stopping it for good. The crucial observation is that, due to the initial choice of distance 1 from line L, the triangle formed by the robots will have one angle (at the central robot) at least 120°. Thus, while the central robot still executes

the third clause of the algorithm (perpendicular departing from line L), the two other robots may already execute the second clause (going towards the robot at the large angle). Since the robot at the large angle is always the central one, other robots eventually hit it and the algorithm stops.

4 Gathering Four Robots

4.1 Overview of the Algorithm

The rest of the paper is devoted to the design and analysis of a gathering algorithm for four robots. As we will see, this task is incomparably more complicated than for three robots, and the algorithm is accordingly more complex. Thus it would not be convenient to present it at a low level, indicating which target point is computed for which configuration (as we did in the simple case of three robots). Instead, we identify nine *situations* which form a partition of all possible positions in which robots can be. These situations are not configurations seen by the robots because in some cases some robots do not have full visibility. However, for each situation, we indicate what a robot should do depending on what it sees. More precisely, for each situation we describe a procedure treating this situation. In most cases, the procedure applied in a given situation brings robots to the same situation (in a different position of the robots, monotonically approaching a specific configuration), until some condition is met, defining a new situation. Hence, most often, all robots execute the same procedure in any moment of the algorithm execution. There are, however, a few important exceptions from this rule, when some robots still execute a procedure treating situation A, while other robots already treat situation B because they have seen robots executing procedure treating situation A, on their way to a target, and perceived the configuration as already satisfying conditions of situation B. This complication, due to the asynchrony of the process, could be already seen for three robots, when one robot was still in the process of departing from a line (situation A) and the other robots (already seeing a triangle) perceived the situation as B. In the case of four robots, a lot of care will be needed in the design of the algorithm, to guarantee that these seemingly incoherent actions do not prevent robots from finally gathering.

The idea of the algorithm is the following. Intuitively speaking, two most general situations are: all robots form a convex *quadrilateral*, or three robots form a *triangle* with the fourth robot inside it. In the latter case, the idea is to gather robots forming the triangle around the internal robot. The corresponding procedure makes the internal robot wait, while the robots at the vertices of the triangle move towards the internal robot until they meet it. However, because of the fatness of the robots, external robots may not succeed in becoming tangent to the internal one, unless the angles between the trajectories of the external robots are large enough. Some special preparation is then necessary. In the case of *quadrilateral* configuration, robots move along diagonals of the quadrilateral until they form a rectangle consisting of two symmetric pairs tangent to the same line L and tangent to the other robot in the pair. (For some

angles between diagonals and some positions of robots this may be complicated by mutual obstructions of robots on their way.) In the special case of perpendicular diagonals, gathering is already achieved. In other cases, however, the two pairs are far apart and they must approach each other by "sliding" along line L. Hence there is a situation *sliding* and a procedure to treat it. The task is further complicated by the fact that the initial position of the robots may prevent full visibility. Such is the case, for example when three or all four robots are aligned, or two robots prevent visibility of the other two (the corresponding situations are *three aligned*, *four aligned* and *partial visibility*). In the case of the *four aligned* situation, the external robots wait and the internal robots move perpendicularly to the line of their alignment. However, because of symmetry, it is impossible to predict in which of the two possible directions the robots that are to move will decide to go. Moreover, in the case when the two internal robots decide to go in the same direction, in view of asynchronicity the robots could reach a *quadrilateral* or *triangle* configuration in uncontrollable way. This nondeterminism made us introduce for this case an intermediate situation *leaving line*. Finally, a special situation, occurring at the end of the gathering process is when two robots are very close to each other and two others try to "lock" the position by touching them from both sides. This is the *locking* situation.

The most difficult problem in the design of the algorithm is to prevent robots from "unexpectedly" transiting to a situation B while treating situation A. Suppose that when treating the situation *quadrilateral* robots momentarily enter in situation *sliding*. Then a robot that performs the *look* step at this point, starts treating situation *sliding* and the other robots keep treating situation *quadrilateral*. This can potentially lead to complete disintegration of the process. In order to synchronize the behavior of the robots, we will attempt to define some specific positions, at which the moving robots must stop. Arriving at such positions, the moving robots may change the procedure they perform or even their function (i.e. whether they move or wait immobile). However, to design such synchronizing positions is not always possible and sometimes a robot may still perform a procedure corresponding to some situation while other robots may already recognize a subsequent situation. Such events, when they happen, will be carefully monitored and the movement of the robots momentarily performing different procedures will be coordinated.

4.2 Description of Situations

We define the following 9 situations.

1. **gathering**
   - robots form a connected configuration
   - there is full visibility
2. **four aligned**
   - centers of all four robots belong to the same line
3. **partial visibility**
   - robots form a convex quadrilateral
   - two robots collectively obstruct visibility of two other robots

4. **locking**
   - no *partial visibility* situation
   - robots form a convex quadrilateral
   - one of the diagonals has length $d \leq \sqrt{8}$
5. **leaving line**
   - no situation 2-4
   - there are two robots A and D (we call them *external*) and we suppose, without loss of generality, that segment P, joining the centers of external robots is horizontal.
   - each other robot B and C (the two robots B and C are called *internal*) intersects the convex hull of A and D
   - The distance of each external robot from the vertical projection on P of the center of each internal robot equals at least 2
   - the following positions are excluded: (1) each internal robot has exactly one point in common with the convex hull of external robots and (2) segments AD and BC are perpendicular.

   Observe that if four robots are in *leaving line* situation the distance d between the external robots is strictly greater than the distance of any other pair of robots.
6. **three aligned**
   - no *four aligned* or *leaving line* situation
   - centers of some three robots belong to the same line
7. **sliding**
   - no *locking* situation
   - the bounding circles of the four robots are tangent to a line s (called *sliding line*)
   - there are two pairs of robots A, B and C, D, such that in each pair, robots (called *partners*) are separated by the line s, and in each pair the distance of their tangency points with s is at most $1/3$

   In this situation there is full visibility.
8. **quadrilateral**
   - no situation 2-7
   - robots form a convex quadrilateral

   In this situation there is full visibility.
9. **triangle**
   - no leaving line situation
   - the center of one of the robots (called the *internal robot*) belongs to the interior of the triangle formed by the centers of the remaining three robots (called the *external robots*)

   In this situation there is full visibility. The three angles $\angle APB$, $\angle BPC$, $\angle CPA$, where P is the internal robot, are called *internal angles*.

   Moreover, in each situation 2-9 the conditions of the *gathering* situation do not occur, as all robots are to wait when the gathering is accomplished.

Lemma 1. *Situations 1 – 9 form a partition of all possible positions of robots.*

Proof. It is easy to check that each situation excludes the conditions of all smaller-numbered situations, either by definition or by exclusiveness of the conditions. Hence situations are disjoint. On the other hand, in any possible configuration either at least three robots are aligned, or the convex hull of the robots forms a triangle or a quadrilateral. Hence, each configuration is classified into one of the situations *three aligned*, *quadrilateral* or *triangle*, unless it has been classified into some smaller-numbered situation. Thus each configuration must belong to some of the situations 1 – 9.

4.3 Description and Correctness of the Algorithm

The algorithm consists of eight procedures, treating each of the situations 2 – 9, and can be shortly formulated as follows

Algorithm FourRobots
loop
  $i :=$ CURRENT SITUATION;
  if $i > 1$ **then** TREAT SITUATION i;
    else STOP;
end-loop

In Figure 1 we show a diagram of all possible transitions between situations. An arrow from situation i to situation j means that procedure TREAT SITUATION i may lead to situation j. Notice that the diagram in Figure 1 is an acyclic graph with a single sink in node *gathering*.

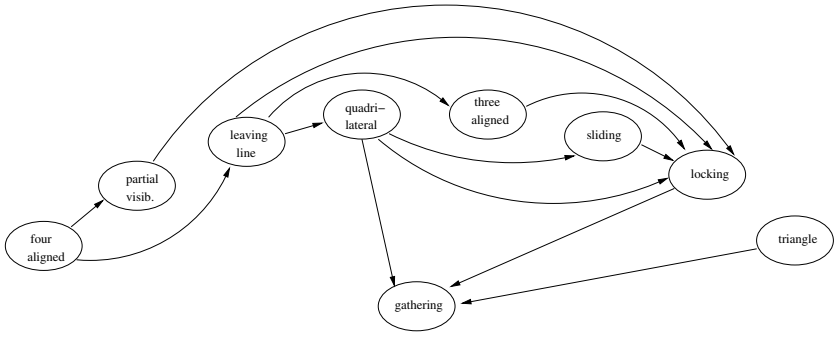


Fig. 1. Diagram of transitions between situations. Circles depict situations 1 – 9 and arrows depict possible transitions between them.

The proof of correctness of Algorithm FOUR ROBOTS is done by showing that each procedure TREAT SITUATION i ends after a finite number of cycles arriving at a new situation according to some outgoing arc from Figure 1. Due to lack of space we present only two of these procedures: TREAT FOUR ALIGNED and TREAT TRIANGLE. The remaining procedures and a detailed proof of correctness of Algorithm FOUR ROBOTS will appear in a full version of this paper.

Situation *four aligned* is the only one which is quit instantaneously, i.e. as soon as the first robot starts moving. Hence, during this movement, while some robots may perform procedure TREAT FOUR ALIGNED other robots may already recognize some further situation and they act accordingly. We suppose, without loss of generality, that the line containing the centers of four robots is horizontal. In this situation we can identify two *external* robots, each one seeing only one other robot and two *internal* robots, each of them seeing two other robots. The procedure TREAT FOUR ALIGNED keeps external robots immobile, while each internal robot moves vertically by a small distance called a *notch*. By symmetry and because of asynchronicity we cannot say whether each internal robot decides to move up or down. The value of notch is defined in such a way that, in case internal robots decide to move in opposite directions, the full visibility is never achieved during this movement. Moreover, during the movement, the distance between the internal robots will never be equal to $\sqrt{8}$. We have the following geometric lemma.

Lemma 2. *Suppose that the four robots are aligned at line l. Then there exists a function $f : R^+ \times R^+ \longrightarrow R^+$ such that, if each internal robot that sees two neighbor robots at distances x and y, moves perpendicularly to the line joining all centers, at distance at most $f(x, y)$ from this line, and both internal robots move on different sides of this line then during this movement*

1. *full visibility of external robots is never obtained, regardless of the initial positions of the robots*
2. *the distance of the two internal robots never equals $\sqrt{8}$ during the movement*

Proof. Suppose that four robots A, B, C and D appear in this order on a horizontal line l. Let x_B and y_B denote the distances of the internal robot B to the two other robots visible by B. Similarly we define x_C and y_C for the internal robot C. Let $d_B = max(x_B, y_B)$ and $d_C = max(x_C, y_C)$. We prove first that if robot B moves vertically by the distance $g(x_B, y_B) = 4/\sqrt{d_B^2 + 8d_B + 4}$, while robot C moves in the opposite direction by the distance $g(x_C, y_C) = 4/\sqrt{d_C^2 + 8d_C + 4}$, the full visibility is never achieved. The external robots would obtain full visibility at the earliest opportunity if they were touching internal robots, i.e. $x_B = 2$ and $y_B = 2$. In other words, if we prove that our values for $g(x, y)$ are working for this case, they will work for any other configuration. Suppose then, that $|AB| = |CD| = 2$ and $|BC| = d$. Suppose that B moves downwards and C moves upwards by the same distance. Consider the critical moment when the full visibility is being obtained. It corresponds to the situation when B and C are becoming tangent to the line l', which is the separating tangent of A and B having negative slope $-\alpha$. Note that $\sin\alpha = 2/d+4$ and $\tan\alpha = x/2$, where x is the distance of each of the internal robots from line l. By solving these equations we obtain $x = 4/\sqrt{d^2 + 8d + 4}$.

Let $e_B = min^+(\sqrt{8}-x_B, \sqrt{8}-y_B, +\infty)$, where $min^+(\sqrt{8}-x_B, \sqrt{8}-y_B, +\infty)$ denotes the smallest positive value among $\sqrt{8} - x_B$, $\sqrt{8} - y_B$ and $+\infty$. Similarly $e_C = min^+(\sqrt{8} - x_C, \sqrt{8} - y_C, +\infty)$. We prove, that if the internal robots go vertically in different directions, each one traversing the distance

$h(x,y) = 1/4\sqrt{e(4\sqrt{2}-e)}$, the distance between the internal robots can never be equal to $\sqrt{8}$. Note that, if the original distance of B from C is larger or equal to $\sqrt{8}$, since the robots move in the opposite directions, this distance increases and the second claim of the lemma is obviously true. Suppose now that $e = e_B = e_C = \sqrt{8} - x_B = \sqrt{8} - y_C$, i.e. the value of $\sqrt{8} - e$ equals the distance between B and C. If each internal robot traverses equal length, starting from their original positions, their distance achieves $\sqrt{8}$ when each of them traverses a vertical segment of length $1/2\sqrt{e(4\sqrt{2}-e)}$. Since $h(x,y)$ equals half that value, the distance of $\sqrt{8}$ between the internal robots will never be reached. If the original distance of the internal robots is smaller than $\sqrt{8} - e_B$ and/or $\sqrt{8} - e_C$ the second claim of the lemma is verified even more so.

By setting $f(x,y) = min(g(x,y), h(x,y))$ we assure that both claims of the lemma are verified at the same time.

Call the distance obtained for an internal robot in Lemma 2, the *notch* of this robot.

Procedure TREAT FOUR ALIGNED;

- the external robots do not move
- each internal robot moves by a notch

Lemma 3. *In a final number of cycles the procedure* TREAT FOUR ALIGNED *brings a set of four robots into one of the situations:* leaving line*,* locking *or* partial visibility*.*

Proof. If both internal robots move vertically in the same direction or if only one of them moves, the leaving line conditions are immediately met. Suppose then that the internal robots move simultaneously in the opposite directions. If their original distance was less than $\sqrt{8}$, by lemma 2 and by the continuity argument they stay at distance less than $\sqrt{8}$ through this movement. The locking situation is reached. If the original distance of the internal robots was greater or equal to $\sqrt{8}$ and they start moving in the opposite directions, the *partial visibility* situation is achieved. Note that, since the *leaving line* situation is quit instantaneously, as soon as any robot starts moving, the other robots recognize already the subsequent situation, while the first robot still completes its residual movement of the procedure TREAT FOUR ALIGNED. As it will be seen later, this residual movement is consistent with the behavior of this robot in the subsequent situation.

We now introduce the procedure TREAT TRIANGLE. The goal of this procedure is to achieve *gathering* in a finite number of cycles while remaining in *triangle situation* throughout the execution of the procedure. Some care must be taken in order not to inadvertently achieve the situation *leaving line* in which a robot may also be centered inside the convex hull of the other three robots.

Procedure TREAT TRIANGLE;

1 **if** all internal angles are equal 120° **then** internal robot waits and each external robot moves towards the internal robot
2 **else**
  2.1 **if** the triangle formed by the external robots has a unique longest side (call it AB) **then** the third external robot C moves perpendicularly to AB until some other pair of robots, say A, C are at distance $|AB|$
  2.2 **else** [the triangle formed by the external robots has two longest sides AB and AC]
    2.2.1 robot C moves perpendicularly to line AB until it is at distance $|AB|\sqrt{3}/6 + 2$ from line AB or further
    2.2.2 all external robots wait and internal robot D moves to the point of triangle ABC for which all internal angles become equal 120°

Remarks. Note that the height $|AB|\sqrt{3}/6 + 2$ in the isosceles triangle ABC is the smallest one permitting the internal robot D to reach the point of equal internal angles of 120°.

Lemma 4. *In a final number of cycles the procedure* TREAT TRIANGLE *ends up in* gathering.

Proof. Since throughout this procedure the internal robot D remains in the interior of triangle ABC, the only other situation which may "unexpectedly" arise is *leaving line*.

If all internal angles are equal to 120° the robots perform step [1]. Since throughout this step the internal angles stay the same, and each external robot remains outside the convex hull of the other two external robots, the *leaving line* condition will never come up. This step finishes in *gathering*. To achieve equal internal angles the internal robot moves to a special, unique point of the triangle ABC in step [2.2.2]. Since the triangle ABC remains isosceles with two longer sides equal, the condition of *leaving line* will not arise. In order that D can reach a point of equal interior angles, a preparation is made in steps [2.1] and [2.2.1]. In step [2.1] one of the external robots moves so that the triangle ABC becomes isosceles. Note that the moving robot is going along the line perpendicular to the longest side of the triangle so the *leaving line* condition will not come forth. In step [2.2.1] the moving external robot C converts the original isosceles triangle ABC into another isosceles triangle which admits the internal point of equal internal angles. It is easy to see that a triangle admits such a point if all triangle angles are smaller than 120°, i.e. in the case of isosceles triangle ABC the height CX must be greater than $|AB|\sqrt{3}/6$. In our case of fat robots, a constant of 2 has to be added so that robots C and D will not overlap. Since during step [2.2.1] the triangle ABC remains isosceles with maximal sides equal, the *leaving line* condition is prevented.

Theorem 2. *Algorithm FourRobots is a correct gathering algorithm for four robots.*

Proof. Consider any initial configuration of four robots. By Lemma 1, this configuration corresponds to a unique situation (a node of the diagram from Figure 1). By a corresponding lemma, in a finite number of cycles this node will be left according to one of the outgoing arcs. It is also easy to observe, that a deadlock is never possible, since in any configuration at least one robot was programmed to make a move. By acyclicity of the diagram the sink node (*gathering*) must be eventually reached.

5 Conclusion

We presented gathering algorithms for three or four robots represented as unit discs in the plane, in a realistic model featuring visibility and move constraints due to the non-zero size of the robots, and at the same time keeping the whole generality of the asynchronous *look-compute-move* paradigm from [9]. The natural problem of generalizing our algorithm to the case of an arbitrary finite number of robots remains open. Another related problem is to obtain full visibility among an arbitrary finite set of robots. As we have seen in the case of four robots, achieving full visibility may be a natural first step towards gathering.

References

1. N. Agmon, D. Peleg, Fault-tolerant gathering algorithms for autonomous mobile robots, Proc. 15th ACM-SIAM Symp. on Discrete Algorithms (SODA 2004), 1063-1071.
2. H. Ando, Y. Oasa, I. Suzuki, M. Yamashita, A distributed memoryless point convergence algorithm for mobile robots with limited visibility, IEEE Transactions on Robotics and Automation 15 (1999), 818-828.
3. T. Balch, R.C. Arkin, Behavior based formation control for multi-robot teams, IEEE Transactions on Robotics and Automation 14 (1998).
4. Y.U. Cao, A.S. Fukunaga, A.B. Kahng, Cooperative mobile robotics: antecedents and directions, Autonomous Robots 4 (1997), 7-23.
5. M. Cieliebak, Gathering non-oblivious mobile robots, Proc. 6th Latin American Theoretical Informatics (LATIN'2004), LNCS 2976, 577-588.
6. M. Cieliebak, G. Prencipe, Gathering autonomous mobile robots, Proc. 9th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2002), 57-72.
7. M. Cieliebak, P. Flocchini, G. Prencipe, N. Santoro, Solving the robots gathering problem, Proc. 30th Colloquium on Automata, Languages and Computing (ICALP 2003), 1181-1196.
8. R. Cohen, D. Peleg, Convergence properties of the gravitational algorithm in asynchronous robot systems. Proc. 12th Annual European Symposium on Algorithms (ESA 2004), LNCS 3221, 228-239.
9. P. Flocchini, G. Prencipe, N. Santoro, P. Widmayer, Hard tasks for weak robots: the role of common knowledge in pattern formation by autonomous mobile robots, Proc. 10th Annual International Symposium on Algorithms And Computation (ISAAC 1999), LNCS 1741, 93-102.
10. P. Flocchini, G. Prencipe, N. Santoro, P. Widmayer, Gathering of autonomous mobile robots with limited visibility, Proc. 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2001), LNCS 2010, 247-258.

11. D. Jung, G. Cheng, A. Zelinsky, Experiments in realising cooperation between autonomous mobile robots, Proc. Symp. on Experimental Robotics, 1997.
12. M.J. Mataric, Designing emergent behaviors, from local interaction to collective intelligence. In: From Animals to Animals 2: Int. Conf. on simulation of Adaptive Behavior (1993), 423-441.
13. L.E. Parker, Designing control laws for cooperative agent teams, Proc. IEEE Conf. on Robotics and Automation (1993), 582-587.
14. L.E. Parker, On the design of behavior-based multi-robot teams, J. of Advanced Robotics 10 (1996).
15. L.E. Parker, C. Touzet, Multi-robot learning in a cooperative observation task, Proc. Distributed Autonomous Robotic Systems 4 (2000), 391-401.
16. G. Prencipe, CORDA: Distributed coordination of a set of autonomous mobile robots, Proc. 4th European Research Seminar on Advances in Distributed Systems (2001), 185-190.
17. G. Prencipe, Instantaneous actions vs. full asynchronicity : controlling and coordinating a set of autonomous mobile robots, Proc. 7th Italian Conference on Theoretical Computer Science (ICTCS 2001), LNCS 2202, 154-171.
18. G. Prencipe, On the feasibility of gathering by autonomous mobile robots, Proc. 12th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2005), LNCS 3499, 246-261.
19. D. Sugihara, I. Suzuki, Distributed algorithms for formation of geometric patterns with many mobile robots, J. of Robotic Systems 13 (1996), 127-139.
20. I. Suzuki, M. Yamashita, Distributed anonymous mobile robots: formation of geometric patterns, SIAM J. on Computing 28 (1999), 1347-1363.

Hop Chains: Secure Routing and the Establishment of Distinct Identities

Rida A. Bazzi[1], Young-ri Choi[2], and Mohamed G. Gouda[2]

[1] School of Computing and Informatics
Arizona State University
Tempe, Arizona, 85287
bazzi@asu.edu
[2] Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{yrchoi, gouda}@cs.utexas.edu

Abstract. We present a secure routing protocol that is immune to Sybil attacks, and that can tolerate initial collusion of Byzantine routers, or runtime collusion of non-adjacent Byzantine routers in the absence of collusion between adjacent routers. For these settings, the calculated distance from a destination to a node is not smaller than the actual shortest distance from the destination to the node. The protocol can also tolerate initial collusion of Byzantine routers and runtime collusion of adjacent Byzantine routers but in the absence of runtime collusion between non-adjacent routers. For this setting, there is a bound on how short the calculated distance is compared to the actual shortest distance. The protocol makes very weak timing assumptions and requires synchronization only between neighbors or second neighbors. We propose to use this protocol for secure localization of routers using hop-count distances, which can be then used as a proof of identity of nodes.

1 Introduction

In peer-to-peer networks, physical entities (or hosts) communicate with each other using pseudonyms or logical identities. Logical identities are assumed by software processes that execute on the hosts to provide or request services from other hosts. To the outside world, a host is identified with the software process that provides the logical functionality. In the absence of direct physical knowledge of a remote host, or certification by a central authority, it is not possible to tell whether or not two distinct logical identities reside on the same host (physical entity) and it is possible for one entity to appear in the system under different names or *counterfeit* identities. Douceur [4] was the first to thoroughly study this problem, and he says that an entity launches a *Sybil attack* when it appears under different identities. He claims that in the absence of a central certifying authority, the Sybil attack cannot be solved in practice.

Bazzi and Konjevod [2] proposed the use of geometric techniques to determine how many identities amongst a group of identities belong to distinct entities and

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 365–379, 2006.

thereby reducing the harm due to Sybil attack. Their work is based on existing evidence that roundtrip delays in the Internet exhibit geometric properties [9]. They provided solutions under a variety of adversarial assumptions including colluding entities and beacons without assuming a central certifying authority with direct knowledge of entities in the system. While the work of Bazzi and Konjevod is a significant step forward, it makes some restricting assumptions. For instance, the results about the geometry of roundtrip delays apply to systems in which routers are honest and they are not necessarily applicable in systems in which routers are corrupt. Also, their solutions require accurate measurements of roundtrip delays and clock synchronizations between routers that can be far apart physically. This is not always possible given the variability of network load and delays.[1]

In this work, our goal is to present a solution to the Sybil attack problem under the weakest possible system assumptions and in the absence of a central authority with direct knowledge of entities in the system. Our solution can tolerate stronger adversarial settings while making weaker system assumptions. In particular, we assume that routers can be dishonest and we allow for more collusion between the routers. Also, we require very rough synchronization between non-adjacent routers (in a sense that we will define precisely later). For some settings, we require synchronization, but only between adjacent (or almost adjacent) routers, which means that the synchronization we require is local. Relaxing the synchrony and synchronization assumptions is a major improvement over the results of [2] and we believe that it is an improvement that brings the results closer to a practical setting.

At the heart of our approach is a secure distance vector routing protocol that can tolerate Byzantine routers, Sybil attack by routers and collusion between routers. The protocol assumes that there are no shared keys between any two nodes, and that only the destination's public key is known a priori by nodes in the network. Under the assumption of no collusion between corrupt nodes, a first version of the protocol guarantees that no node can have a calculated hop-count distance, or simply distance, to destination that is shorter than its real or actual shortest hop-count distance to destination. In the presence of initial collusion, in which corrupt nodes can share information initially, but not afterwards, the second version of protocol guarantees that no honest router can have a calculated hop-count distance to destination that is shorter than its real or actual shortest hop-count distance to destination. In the presence of initial collusion between any two nodes and runtime collusion between adjacent corrupt nodes, in which corrupt nodes can communicate with each other at any time, the second protocol guarantees the following: for any path P from destination to an honest node u, the calculated hop-count distance of node u is not less than the number of honest nodes on P plus the number of corrupt components of P (a corrupt component is a maximal subpath that contains corrupt nodes). In other words, every sequence of adjacent corrupt nodes on a path can appear to be one node. In the presence

[1] In their work, they propose approaches to handle inaccuracies, but these approaches are incomplete.

of remote collusion between nodes and if there are no colluding adjacent nodes, then a further modification guarantees that the hop-count distance calculated by an honest node is not shorter than the shortest distance from destination to the node. The protocol has two basic components. The first is a practical and simple protocol that enables a node to determine if another node is its physical neighbor. The second is a novel use of key chains, which we call *hop-chains*, that enable the destination to certify *remotely* its distance to nodes in the network. To tolerate initial collusion, we introduce *mistrust hop-chains* to prevent nodes from cheating by initially agreeing on keys. This secure routing protocol we present is a significant contribution on its own. The protocol is more secure and requires less assumptions than other secure routing protocols in the literature (see Section 9).

Our solution to the Sybil attack problem proposes to use the secure routing protocol in order to come up with a secure localization protocol for networks in which hop-count distances from a number of beacons (or anchor points) can be used to localize nodes. This is along the lines of the approach of [2], but replacing roundtrip delays with hop-count distance.

2 Identities and Public Keys

In our model, only the destination has a public key that needs to be known by all other nodes. Other nodes need to have identities that cannot be forged by faulty nodes. This can be achieved by having each node randomly choose its own public key and corresponding private key. We assume that the keys are large enough so that corrupt nodes can with negligible probability guess or generate keys identical to those of honest nodes. Also, correct honest nodes generate different keys with high probability. This guarantees that with high probability nodes cannot forge messages, but does not rule out that corrupt nodes can replay messages.

In our framework, the identity of a node is its public key. For an honest node, this identity is unique and does not change over time. For a corrupt node, there can be multiple identities, one for each public key that the node chooses. We spell out our assumptions about keys in Section 5.

3 Neighbor Computation

The ability of a node to determine whether another node is its neighbor is an important ingredient for our secure routing protocol. Before we explain how that determination can be done, we need to precisely define what we mean by "determining if a node is the neighbor of another node".

We say that a node is the neighbor of another node if the two nodes can communicate directly and not through an intermediate node. In wireless networks, this requires that the nodes are in each other's range. In wired networks, this requires that the nodes either have access to a shared communication link or share a private link. In our model, nodes are known to other nodes through their public keys. So, determining whether a node is the neighbor of another node

reduces to determining if the owner of the private key corresponding to a given public key is in the neighborhood of the node. What we determine is something subtly different from the foregoing. We determine if a node with *access to* the private key corresponding to a given public key is in the neighborhood of the node. The distinction is subtle, but important. A node has *access to* the private key if it has the private key or it is in collusion with a node that has the private key. If there is no collusion other than initial collusion between nodes, then having access to a key and having a key are the same thing.

3.1 Immediate Neighbors

A first step in neighbor determination is to broadcast a message requesting from neighbors to provide their public keys. The goal of neighbor determination is to determine if a neighbor of the node has access to the private key corresponding to the provided public key.

A naive approach for determining whether a node is the neighbor of another node is to send a request message and wait for a reply within a short period of time. The reply should allow for the transmission time, roundtrip delay and any local computation at the node to encrypt and decrypt messages exchanged to prevent third parties for interfering with the communication. Unfortunately, the time for computations can be substantial especially if public key encryption is involved which makes the approach vulnerable to a man-in-the-middle attack.

A better approach is similar to the one taken by [3] in which communication is done in the clear to eliminate high processing time. In a first phase, a node sends a random bit in a message encrypted with the destination's key. The destination decrypts the message and recovers the bit. In a second phase, the node broadcasts a message in the clear to all its neighbors. Upon receipt of the message, the destination performs XOR on the first bit of the message with the random bit and resends the message in the clear to the sender. The extra processing time is minimal. The probability that the destination sends the correct answer without knowing the random bit is $1/2$. This probability can be made arbitrarily small by repeating the two phases multiple times. A corrupt node B cannot compromise this scheme by launching a man-in-the-middle attack in a timely manner. But, a corrupt node B can execute the first phase by colluding with another node C, which decrypts the first phase message and provides the value of the bit to B. This way, B can execute the second phase. Thus, this two-phase approach guarantees that B *has access* to the private key corresponding to the public key it sends to A.

3.2 Neighbors of Neighbors

To tolerate runtime collusion between non-adjacent nodes and if there are no colluding adjacent nodes, our routing protocol requires the ability for a node to determine if another node is the neighbor of its neighbor. We propose to use the same approach we propose for determining neighbors to also determine neighbors of neighbors by allowing more time for the message to be forwarded to a neighbor of a neighbor and then sent back.

3.3 Effects of Congestion

If two nodes have a dedicate link between them, then the determination of neighbors can be done without interference by other nodes. In a wireless medium, other nodes can interfere in the communication by launching denial of service attacks. We do not address denial of service attacks in this paper. Our assumption about congestion is fundamentally different from the assumptions in [2]. In our work, we make the realistic and practical assumption that two *adjacent* (immediate neighbors) nodes can communicate with no congestion for some periods of time, whereas in [2], a similar assumption is made for nodes that are many hops apart.

3.4 Timing Consideration

In our neighbor computations, we assume that the dominant factor in the delay is due to transmission and processing, but not propagation delay. The transmission rate between two adjacent nodes is determined by their hardware and it is not unreasonable that the nodes can measure time to an accuracy of 1 bit. In wireless networks, speeds of 100 Mbps can be considered high. At this speed, a 4 KByte frame takes around 0.32 msec. During that time a signal can propagate up to 96 km which is way beyond the range of node to node transmission in ad-hoc networks. In wired networks, propagation delay can be substantial for transatlantic communication, but such communication has to go through known entities that charge for their services and cannot be part of any ad-hoc network.

4 Distance Vector Routing and Its Vulnerabilities

In a traditional distance vector routing protocol, nodes in a network collaborate to build a spanning tree whose root is the ultimate destination node d. Initially, no node u other than d has a parent in the routing tree, and the distance of node u is infinite. Only the ultimate destination node d is in the routing tree, and it periodically broadcasts an advertisement message of the form $adv(d, 0)$ to its neighbors.

When a node u whose current distance is greater than $s + 1$ receives an $adv(v, s)$ message, node u makes node v its parent in the tree, and sets its distance to be $s+1$. Once node u has a parent in the tree, node u becomes connected to the tree and starts sending an $adv(u, s')$ message periodically, where s' is the current distance of u. When u stops receiving advertisement messages from its parent for a certain time period, it stops sending advertisement messages.

A node can cause harm if it drops packets it is supposed to forward[2] or if it reports a false (shorter) distance to destination. Such misbehavior is illustrated in Figure 1 where the ultimate destination is node (0,0) and node (8,8) misbehaves. The left side of the figure illustrates the case where node (8,8) drops packets and the right side illustrates the case where node (8,8) reports a distance of 2 to

[2] There are techniques to detect nodes that do not forward messages [1], but in this paper we do not consider the problem of detecting such nodes.

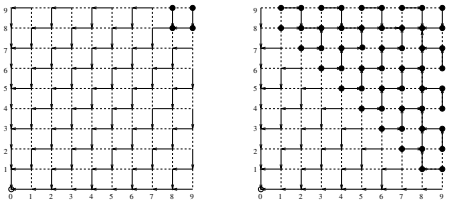


Fig. 1. Routing tree when (8,8) does not lie (left) and when (8,8) lies (right)

destination. The black circles illustrate the affected nodes and node (8,8) (in general, the number of affected nodes depends on the difference between the reported distance and the real distance). Our goal for a secure distance vector routing protocol is to prevent corrupt nodes from reporting distances that are smaller than their actual distances to destination. The proposed secure routing protocol tolerates strong adversaries. We consider the following failures.

Byzantine failures. Corrupt nodes can behave arbitrarily. In particular, they can advertise multiple public keys (attempt Sybil attacks) and they can replay or resend messages received from others.

Initial collusion of nodes. Corrupt nodes that initially collude can share information before the execution of the protocol, but they cannot communicate information that they learn during the execution of the protocol. To our knowledge, this model has not been considered by others. It makes sense to consider it in our setting because nodes are known to others through their public keys. It is not clear how a node can find or *trust* another node to collude with. If a node is corrupted with a virus, for example, then two corrupted nodes share the common information that the virus carries and therefore they initially collude.

Runtime collusion of adjacent nodes. Even though it is not clear how nodes can find other nodes to collude with, it makes sense to consider adjacent nodes that are colluding at runtime. In fact, if two adjacent nodes are initially colluding, they could discover that they have the same keys by communicating with each other and then decide to collude. Our protocol tolerates run-time collusion between adjacent nodes (or connected component consisting of corrupt colluding nodes).

Runtime collusion of non-adjacent nodes. We consider runtime collusion between non-adjacent nodes, but in the absence of collusion between adjacent nodes. We assume that non-adjacent corrupt nodes can communicate information with each other at any time.

5 Tolerating Non-colluding Byzantine Failures

Every node u in the network creates its own public key(s) (BK_u), and corresponding private key(s) (RK_u). The public key BK_d is known by all nodes. We denote a message m encrypted with a key BK_u with $BK_u\langle m\rangle$. Similarly,

$BK_u\langle m\rangle$ is the decryption of m using the public key of u, and $BK_u\langle RK_u\langle m\rangle\rangle = m$. To reduce the size of messages, we assume the existence of a message digest, or one way hash, function MD. The use of MD is not needed for the correctness of the protocol. If a node receives a message (m, m') such that $BK_u\langle m'\rangle = BK_u\langle RK_u\langle m\rangle\rangle$, then m' must have been encrypted by u, which has RK_u. Similarly if a node receives a message (m, m') such that $BK_u\langle m'\rangle = BK_u\langle RK_u\langle MD(m)\rangle\rangle$, then m' must have been encrypted by u.

In the protocol, each node u that is connected to the routing tree maintains a hop-chain that verifies its hop-count distance to destination. The hop-chain contains a sequence of public keys and a sequence of certificates. The public keys are supposed to be keys of a sequence of nodes $d = u_0, u_1, \ldots, u_k = u$ that form a path from d to u. The certificates vouch that every node in the sequence is the neighbor of the next node in the sequence. Finally, a hop-chain has a date dt that specifies the period of validity of the chain. The date changes infrequently relative to the communication delays in the network, and it does not require any tight synchronization between the nodes. A hop-chain has the following format: $\langle dt, BK_{u_0}, BK_{u_1}, \ldots, BK_{u_{k-1}}, BK_{u_k}, C_{u_0}, C_{u_1}, \ldots C_{u_{k-1}}, C_{u_k}\rangle$, where $C_{u_0} = RK_{u_0}\langle MD(dt, BK_{u_0})\rangle$, and $C_{u_i} = RK_{u_{i-1}}\langle MD(dt, BK_{u_i})\rangle$, $0 < i \leq k$.

Definition 1. *The* length *of a hop-chain H_u of a node u, denoted $len(H_u)$, is the number of certificates in H_u.* [3]

It is straightforward to see that only a node that has RK_{u_0} can generate C_{u_0}, and only a node that has $RK_{u_{i-1}}$ can generate C_{u_i}. We say that a hop chain is valid if it is of the form $\langle dt, K_0, K_1, \ldots, K_k, C_0, C_1, \ldots, C_k\rangle$ and

- $K_0 = BK_d$ and $BK_d\langle C_0\rangle = MD(dt, BK_d)$
- $BK_{i-1}\langle C_i\rangle = MD(dt, K_i)$, $0 < i \leq k$.

A node u that receives a hop-chain can locally check its validity. The protocol ensures that the owners of successive keys in the sequence are neighbors or the same node (creating successive bogus nodes). It will follow that the hop-chain length of node u minus one cannot be less than the shortest distance from u to d.

Initially, no node u other than d has a parent in the routing tree, and the distance of node u is infinite. Only node d is initially connected to the routing tree.

Each hop-chain contains a date field dt that indicates the date (time) at which the chain is generated. The root of the routing tree, node d, periodically updates dt every P seconds. Thus, node d periodically recomputes its chain $\langle dt, BK_d, RK_d\langle MD(dt, BK_d)\rangle\rangle$ every P seconds. The period of time P is chosen to be larger than the delay between nodes in the network.

A node u that is connected to the tree periodically broadcasts an advertisement message to its neighbors every p seconds where $p \ll P$. The advertisement message of node u has the form $adv(BK_u, dt, H_u)$, where dt is the latest date that u is aware of, and H_u is the latest chain that u has.

When a node u receives an advertisement message, $adv(bk, t, h)$, where bk is a public key, t is a date, and h is a hop-chain, u ignores the message if t is smaller

[3] Note that $len(H_u)$ minus one is equal to the hop-distance from destination to u.

```
1:  var   dt          : integer,      // date
2:        H_u, ph     : integer,      // current/potential hop-chain
3:        ds          : 0..dmax+1,    // distance, initially dmax+1
4:        BK_p        : integer,      // parent's permanent public key
5:        trc         : 0..tmax,      // time to remain connected
6:        wait        : boolean,      // wait for ack msg or not, init. false
7:        pdt, t      : integer,      // potential and received date
8:        h, c        : integer,      // received hop-chain/certificate
9:        bk, bk'     : integer       // received keys
```

Fig. 2. Variables of a node u

than the latest date u is aware of – the message is too old. If the message has a date that is more recent than the latest date at node u, u verifies that the message is valid. If the message is valid, u tentatively decides to use the received hop-chain to calculate its distance to destination. An *adv* message is valid, if bk is the public key of a neighbor of u, the date of the message is the same as the date of the hop-chain h, the hop-chain h is valid, and the last key in the chain is equal to bk. In the case that the date of the message is the same as the most recent date u is aware of, if the message is valid, and the length of h is smaller than the current distance of u to destination, u tentatively decides to use the received hop-chain to calculate its distance to destination. When a node u tentatively decides to use the received chain to calculate its distance to destination, u makes bk its potential parent, assigns h to the potential chain ph, and assigns t to the potential date pdt. Finally, u sends a reply message to node bk and waits to receive a certificate from bk – its potential parent. The reply message is of the form $rpl(bk, BK_u, t)$. While node u is waiting to receive a certificate, u ignores any advertisement messages u receives until u receives a certificate or u times out. (This is only to keep our code easy to follow.)

There is no loss in ignoring advertisement messages, since advertisement messages will be sent periodically, and so node u can receive them later.

When a node u that is connected to the routing tree receives a reply message $rpl(bk, bk', t)$, where $bk = BK_u$, and t equals to dt in its own hop-chain, node u first computes a certificate $c = RK_u\langle MD(dt, bk')\rangle$, and then u sends an acknowledgment message $ack(bk', t, c)$ to node bk'.

When a node u that is waiting to receive a certificate from its potential parent receives an acknowledgment message $ack(bk', t, c)$, where $bk' = BK_u$, and $t = pdt$, node u checks the validity of the certificate in the message. The certificate c is valid if it is encrypted with the corresponding private key of the last key $tpbk$ in the potential hop-chain ph (the key of its potential parent) and so $c = tpbk\langle MD(dt, BK_u)\rangle$. If the certificate in the message is valid, u makes node $tpbk$ its parent in the routing tree and updates its distance to destination. Finally, u computes its (new) hop-chain by adding BK_u and c to the hop-chain of its parent, ph.

When a node u has a parent and does not receive any valid advertisement message from its parent for a time period of $tmax \times p$ seconds, u concludes that it is not connected to its parent anymore. Thus, u disconnects from the tree by making its distance infinite, and stops sending advertisement messages.

After node u sends a reply message to its potential parent v, u starts a timer and times out after w seconds, at which time u no longer considers v as its potential parent. The value of w is chosen to be large enough to accommodate roundtrip delay to a neighbor including time for public key encryption and decryption. The protocol variables and specification of a node u are given in Figures 2 and 3. We state without proof the properties of the protocol.

Lemma 1 (len(hop-chain) $\geq$ len(shortest path)). *For every honest node u, $len(H_u) \geq len(S)$, where S is the shortest path from node d to node u and $len(S)$ is the number of nodes in path S.*

Lemma 2 (len(hop-chain) $\leq$ len(good path)). *For every honest node u, if there exists a "good" path G from node d to node u such that each node in the path is honest, then eventually $len(H_u) \leq len(G)$ holds.*

6 Tolerating Initial Collusion

The protocol of the previous section is vulnerable to initial collusion. Consider two nodes u and v that share the public and private keys BK_u and RK_u. Assume that v is a farther node from the destination. Since v has u's keys, the neighbors of v will consider the owner of the private key RK_u to be their neighbor. If at some point, node v receives an advertisement message with a chain that contains the public key of u, v can *cut* the chain, only keep the portion of the chain that is identical to u's chain, and present that portion to its neighbors. Node v can then advertise u's chain to its neighbor, and the neighbors of v will find the received chain to be valid, in effect v manages to claim a distance to destination that is shorter than its actual distance to destination.

The reason for the success of this attack is that a node is certified based only on the initial information of the node, and initially colluding nodes share all their initial information. To get around this difficulty, we need to certify nodes based on information that they do not have initially. This can be achieved by having a parent in the routing tree create public/private key pairs for its children.

These *temporary* keys will be used alongside the *permanent* keys of a node. We say that they are temporary because their values depend on the identity of the parent of a node at a given time. For a node u, we denote these keys with TBK_u and TRK_u. For the destination node d, we really need no temporary keys, but we introduce them to make the protocol more uniform.

In the modified protocol, nodes use temporary keys and permanent keys to check the validity of a certificate and therefore of a chain. The *mistrust* hop-chain of a node has the format: $\langle dt, (BK_{u_0}, TBK_{u_0}), \ldots, (BK_{u_k}, TBK_{u_k}), C_{u_0}, C_{u_1}, \ldots, C_{u_{k-1}}, C_{u_k} \rangle$, where $C_{u_0} = RK_{u_0}\langle TRK_{u_0}\langle MD(dt, BK_{u_0}, TBK_{u_0})\rangle\rangle$, and $C_{u_i} = RK_{u_{i-1}}\langle TRK_{u_{i-1}}\langle MD(dt, BK_{u_i}, TBK_{u_i})\rangle\rangle$, $0 < i \leq k$.

It is straightforward to see that only a node that has RK_{u_0} and TRK_{u_0} can generate C_{u_0}, and only a node that has $RK_{u_{i-1}}$ and $TRK_{u_{i-1}}$ can generate C_{u_i}.

```
1:    timeout DATE expires → // d periodically updates date
2:                           if (u = d) then
3:                               dt := UPDATE_DT;
4:                               H_u := ⟨dt, BK_u, RK_u⟨MD(dt, BK_u)⟩⟩;
5:                               timeout DATE after P

6:  [] timeout ADV expires →  // u periodically sends advertisement
7:                           if (u = d) then
8:                               send adv(BK_u, dt, H_u);
9:                               timeout ADV after p
10:                          elseif (u ≠ d) then
11:                              trc := MAX(trc − 1, 0);
12:                              if (trc > 0) then
13:                                  send adv(BK_u, dt, H_u);
14:                                  timeout ADV after p
15:                              elseif (trc = 0) then
16:                                  ds := dmax+1

17: [] timeout RPL expires→   // no longer wait for ack from potential parent
18:                           wait := false

19: [] rcv adv(bk, t, h)  →   // if valid adv received from a node closer to d
20:                           // update potential parent and reply to sender
21:                           if ¬wait ∧ (t > dt ∨ (t = dt ∧ len(h) < ds))
                              ∧ valid(adv(bk, t, h)) then
22:                               pdt := t; ph := h;
23:                               wait := true;
24:                               send rpl(bk, BK_u, t) to bk;
25:                               timeout RPL after w
26:                           // if valid advertisement received from parent
27:                           // stay connected for a longer period
28:                           if ((trc > 0) ∧ (bk = BK_p) ∧ (len(h) = ds)
                              ∧ valid(adv(bk, t, h))) then
29:                               trc := tmax

30: [] rcv rpl(bk, bk′, t) →  // if valid reply received from a node
31:                           // compute a certificate and send it to sender
32:                           if ((BK_u = bk) ∧ (t = dt) ∧ (trc > 0) then
33:                               send ack(bk′, dt, RK_u⟨MD(dt, bk′)⟩) to bk′

34: [] rcv ack(bk, t, c)  →   // if valid ack received from potential parent
35:                           // update its parent, distance, chain, and send adv
36:                           if wait ∧ (BK_u = bk) ∧ (pdt = t)
                              ∧ valid(ack(bk, t, c)) then
37:                               dt := pdt;
38:                               H_u := COMP_CERT(ph, c);
39:                               ds := len(ph);
40:                               BK_p := GET_BKP(ph);
41:                               wait := false;
42:                               trc := tmax;
43:                               send adv(BK_u, dt, H_u);
44:                               timeout ADV after p
```

Fig. 3. A specification of a node u

We say that a mistrust hop-chain is valid if the chain is of the form $\langle dt, (K_0, T_0), (K_1, T_1), \ldots, (K_k, T_k), C_0, C_1\ , \ldots, C_k\rangle$ and

- $K_0 = BK_d$ and $TBK_d\langle BK_d\langle C_0\rangle\rangle = MD(dt, K_0, T_0)$
- $TBK_{i-1}\langle BK_{i-1}\langle C_i\rangle\rangle = MD(dt, K_i, T_i)$, $0 < i \leq k$.

Using permanent and temporary keys, the protocol ensures that the owner of every pair of permanent and temporary public keys in the sequence encrypted the certificate of the owner of the next pair of public keys in the sequence. Thus,

```
1:  [] rcv rpl(bk, bk', t)  →  // if valid reply received from a node
2:                             // choose temp. keys and send a cert. to sender
3:                             if ((BK_u = bk) ∧ (t = dt) ∧ (trc > 0) then
4:                                 (tcbk, tcrk) := GEN_KEYS;
5:                                 send ack((bk', tcbk), dt, bk'⟨tcrk⟩,
                                   RK_u⟨TRK_u⟨MD(dt, bk', tcbk)⟩⟩) to bk'

6:  [] rcv ack(bk, bk', t, r, c) →  // if valid ack received from potential parent
7:                                  // update its parent, distance, chain, and send adv
8:                                  if wait ∧ (BK_u = bk) ∧ (pdt = t)∧
                                    valid(ack(bk, bk', t, r, c)) then
9:                                      dt := pdt;
10:                                     TBK_u := bk'; TRK_u := RK_u⟨r⟩;
11:                                     H_u := COMP_CERT(ph, c);
12:                                     ds := len(ph);
13:                                     BK_p := GET_BKP(ph);
14:                                     wait := false;
15:                                     trc := tmax;
16:                                     send adv(BK_u, dt, H_u);
17:                                     timeout ADV after p
```

Fig. 4. *rpl* and *ack* processing to handle initial collusion

a corrupt node u that initially colludes with another node v closer to destination cannot use the hop-chain of v, since u has no access to the temporary private key of v.

In the modified protocol, *rpl* and *ack* processing is modified as in Figure 4. When a node u that is connected to the routing tree receives a reply message $rpl(bk, bk', t)$, where $bk = BK_u$, and t equals to dt in its own hop-chain, node u randomly chooses temporary public/private key pair $(tcbk, tcrk)$ for node bk'. Node u then computes a certificate $c = RK_u\langle TRK_u\langle MD(dt, bk', tcbk)\rangle$, and also computes an encrypted temporary private key $r = bk'\langle tcrk\rangle$. Finally, node u sends an acknowledgment message $ack((bk', tcbk), t, r, c)$ to node bk'.

When a node u that is waiting to receive a certificate from its potential parent receives an acknowledgment message $ack((bk', bk''), t, r, c)$, where $bk' = BK_u$, and $t = pdt$, node u first checks the validity of the certificate in the message. The certificate c is valid if it is encrypted with the corresponding permanent and temporary private keys of the last key pair $(pbk, tpbk)$ in the potential chain ph (the key of its potential parent) and so $tpbk\langle pbk\langle c\rangle\rangle = MD(t, BK_u, bk'')$. If the *ack* message is valid, u makes node pbk its parent in the routing tree and updates its distance to destination. Finally, u computes its temporary public and private keys, TBK_u and TRK_u by assigning bk'' to TBK_u, and $RK_u\langle r\rangle$ to TRK_u, and then computes its (new) hop-chain by adding (BK_u, TBK_u) and c to the hop chain of its parent, ph. We state without proof the properties of the protocol.

Lemma 3 (Initial collusion: len(hop-chain) ≥ len(shortest path)). *For every honest node u, $len(H_u) \geq len(S)$, where S is the shortest path from node d to node u.*

Lemma 4 (Initial collusion + Collusion of adjacent nodes). *For every honest node u, if there exists a path P from node d to node u, then $len(H_u) \geq len(P) - (|cor(P)| - |cor\_comp(P)|)$, where $cor(P)$ is the set of corrupt nodes in P and $cor\_comp(P)$ is the set of maximal connected components of P whose elements are all corrupt.*

7 Tolerating Runtime Collusion of Non-adjacent Nodes

The protocol in the previous section will not work if two colluding nodes can share both their permanent as well as their temporary keys provided by the parent of a node. In order to tolerate runtime collusion of non-adjacent nodes, we make the parent of a node u keep the temporary private key of node u. However, implementing this idea is not straightforward. Consider the following modification. The hop-chain format does not change, except that the temporary private key of u is kept at the parent of u. The *rpl* message sent by u contains a nonce that the potential parent v of u should forward to its parent to encrypt with the temporary private key of v. When v receives this *rpl* message, v forwards the nonce to its parent (the potential grandparent of u). Later when v receives the encrypted nonce from its parent, v forwards it to u. Clearly, this will not work because two non-adjacent colluding nodes v and v' can cheat as follows. The farther node v forwards the nonce sent by u to node v' closer to destination, and in turn v' forwards it to its parent. When the parent of v' sends the encrypted nonce to v', v' forwards it to v that forwards it to u.

In the above modification, we need to ensure that the parent of v, the grandparent of u, is a neighbor of v. Thus, we need to run the *neighbor of neighbor* determination protocol described in Section 3 for the owner of the temporary private key of v, which should be a neighbor of a neighbor of u.

The protocol is modified as follows. First, an *ack* message sent from u to v does not contain the temporary private key of v generated by u. Second, when u receives an $adv(bk, t, h)$ message, u needs to check that the owner of the last temporary private key in the chain is a neighbor of a neighbor of u, as well as the validity of the received hop-chain.

Lemma 5 (Collusion: len(hop-chain) $\geq$ len(shortest path)). *For every honest node u, $len(H_u) \geq len(S)$, where S is the shortest path from node d to node u.*

8 Preventing and Mitigating the Sybil Attack

We say that a solution prevents Sybil attacks if no entity can successfully pretend to have more than one identity. We say that a solution mitigates Sybil attacks if the solution limits the number of identities that an entity can have. This is done under the assumption that nodes have vast resources, but we assume that corrupt nodes cannot break the public key encryption scheme in use. In the following sections, we show how the routing protocol can be used to mitigate or prevent Sybil attacks under various assumptions about the network. We start by considering restricted settings, and then we consider a general network.

8.1 Sybil Attack in an Immediate Neighborhood

In the simplest setting, we are interested in determining if two identities that are the immediate neighbors of a node reside on distinct nodes. If the nodes

are connected with physical links, then it is easy to distinguish nodes because identities that appear on the same link can be considered to be identical. In this case, the number of physical links determines the number of neighbors.

If the nodes are connected through a wireless link, then the situation is similar to the situation described in Douceur's work [4]. Roundtrip delays cannot be used to differentiate two nodes and it seems that the only way to distinguish nodes is through an approach similar to that of Douceur [4], namely requiring the node to prove that it has the resources of multiple nodes.

8.2 Sybil Attack in a Line

In this section, we consider a number of nodes connected in a line. This case is the basis for our treatment of Sybil attacks in general networks. We only consider how to detect multiple identities that correspond to the same entity or how to detect that a number of identities above some threshold correspond to the same entity. We are not concerned with a corrupt node that drops messages from nodes on its two sides thereby disconnecting them. *The result of the section will be to determine conditions under which corrupt node can successfully launch Sybil attack in a line.* We only describe the approach for the case of initial collusion and collusion between adjacent nodes.

Initial collusion. Consider a sequence of nodes $A = A_0, A_1, \ldots, A_n = B$. These nodes are such that the actual distance from A to A_i is equal to i. Also, the minimum distance from A_i to B is $n - i$. We assume that A and B are honest, the distance between A and B is n, and their public keys are known by all nodes in the line. Nodes A and B are beacons used to locate nodes in the line. Finally, we assume that only initial collusion exists in the network.

Under these assumptions, by Lemma 3, it follows that the length of a hop-chain from beacon A to A_i, $len_A(H_{A_i})$ is greater than or equal to i. Similarly, the length of a hop-chain from beacon B to A_i, $len_B(H_{A_i}) \geq n - i$. If there are no corrupt nodes, then $len_A(H_{A_i}) + len_B(H_{A_i}) = n$. It follows that a corrupt node cannot insert any bogus nodes on a hop-chain without being detected because adding bogus nodes will make the sum greater than n.

If the distance between A and B is not known, and only a lower bound n_{low} and an upper bound n_{high} on the distance are known, any node A_i such that $len_A(H_{A_i}) + len_B(H_{A_i}) \leq n_{high}$ would be accepted. If the actual distance between A and B is n_{low}, then a corrupt node can insert up to $n_{high} - n_{low}$ bogus nodes without being detected. The corrupt nodes as a group cannot insert more than $n_{high} - n_{low}$ bogus nodes (identities) without being detected. Since nodes are colluding only initially, this should also create a dilemma as to which nodes should be the ones to insert the bogus identities.

Initial collusion and collusion of adjacent nodes. Consider adjacent colluding nodes. In this case, using the protocol of Section 6, the nodes can shrink the path, but only by the number of corrupt nodes minus the number of corrupt components. This will not affect the above results, because the number of identities that can be created by corrupt nodes would still be $n_{high} - n_{low}$. The

disappeared corrupt nodes can be replaced by other corrupt nodes that appear elsewhere on the line, but the corrupt nodes cannot add more bogus identities than $n_{high} - n_{low}$.

We summarize the results of these sections with the following lemma.

Lemma 6. *On a line, the number of new identities that can be added is not more than $n_{high} - n_{low}$, where n_{high} and n_{low} are upper and lower bounds on the hop-count distance between the end nodes assumed honest.*

8.3 Sybil Attack in a Network

In a general network, under the assumption of no collusion between adjacent nodes, the path from a beacon node to any node cannot be made shorter than it really is. Also, the length of a path from a beacon to any node is not more than the length of the shortest good path. If the network has enough redundant paths, then the distances between nodes are not affected by corrupt routers. In this case, we propose to use hop-count distances of a node from a number of beacons as the identity of the node. There is already a good amount of work on hop-count based coordinates (see [5] for example), and it is not our goal in this paper to study this topic. We simply propose to use our secure routing protocol in conjunction with hop-count based coordinates in order to assign identities to nodes. If the number of corrupt nodes is not large, then corrupt nodes cannot practically affect changes in the location of other nodes.

9 Secure Routing Related Work

Secure distance vector routing protocols have been proposed by many researcher [11,10,12,6,7]. Existing protocols are based on assumptions that are stronger than the ones we make. The protocol proposed in [11] uses a set of the intrusion detection sensors to detect routing attacks, and requires knowledge of the network topology and sensor positions. SEAD [6] does not prevent corrupt nodes from replying advertisement messages, and does not consider colluded attackers. In [7], nodes are assigned to unique identifiers (by a central authority), the destination knows all these identifiers, and the clocks of all nodes are tightly synchronized (to use [8]). RIP-RT [10] assumes that corrupt nodes cannot modify the value of the time-to-live field in a probing message and that any two nodes share a key. Moreover, this protocol assumes that each node knows the identifiers of adjacent nodes. The problem of detecting misbehaving nodes was considered in [1], which also proposes an on-demand secure routing protocol. Our routing protocol focuses on reducing harm caused by corrupt nodes that lie their distances, and does not consider to detect such nodes.

References

1. B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *WiSE '02: Proceedings of the 3rd ACM workshop on Wireless security*, pages 21–30. ACM Press, 2002.

2. R. Bazzi and G. Konjevod. On the stabilishment of distinct identities in overlay networks. In *Proceedings of ACM Symposium on Principles of Distributed Computing*.
3. S. Brands and D. Chaum. Distance-bounding protocols (extended abstract). In *Proceedings of EUROCRYPT*, pages 344–359, 1993.
4. J. Douceur. The sybil attack. In *Proceedings of IPTPS*, pages 251–260, 2002.
5. R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, 2005.
6. Y.-C. Hu, D. B. Johnson, and A. Perrig. Sead: Secure efficient distance vector routing for mobile wireless ad hoc networks. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*.
7. Y.-C. Hu, A. Perrig, and D. B. Johnson. Efficient security mechanisms for routing protocols. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS 2003)*, February 2003.
8. Y.-C. Hu, A. Perrig, and D. B. Johnson. Packet leashes: A defense against wormhole attacks in wireless ad hoc networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, April 2003.
9. T. Ng and H. Zhang. Predicting internet network distance with coordinate-based approaches. In *Proceedings of INFOCOM*, 2002.
10. D. Pei, D. Massey, and L. Zhang. Detection of invalid routing announcements in the rip protocol. In *Proceedings of GLOBECOM 2003*, 2003.
11. V. M. tal and G. Vigna. Sensor-based intrusion detection for intra-domain distance-vector routing. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 127–137. ACM Press, 2002.
12. T. Wan, E. Kranakis, and P. V. Oorschot. S-rip: A secure distance vector routing protocol. In *Proceedings of Applied Cryptography and Network Security*, 2004.

Computing on a Partially Eponymous Ring*

Marios Mavronicolas[1], Loizos Michael[2], and Paul Spirakis[3]

[1] Department of Computer Science, University of Cyprus, Nicosia CY-1678, Cyprus, and Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, 33102 Paderborn, Germany
mavronic@cs.ucy.ac.cy

[2] Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, U.S.A.
loizos@eecs.harvard.edu

[3] Department of Computer Engineering and Informatics, University of Patras, 265 00 Patras, Greece, and Research and Academic Computer Technology Institute, 265 00 Patras, Greece
spirakis@cti.gr

Abstract. We study the *partially eponymous* model of distributed computation, which simultaneously generalizes the *anonymous* and the *eponymous* models. In this model, processors have *identities*, which are neither necessarily all identical, nor necessarily unique; processors receive *inputs* and must reach *outputs* that respect a *relation*. We focus on the *partially eponymous ring* R, and we are interested in the computation of *circularly symmetric* relations on it.

• We distinguish between solvability and computability: in *solvability*, processors must *always* reach outputs that respect the relation; in *computability*, they must reach outputs that respect the relation whenever possible, and report impossibility otherwise.
- We provide an *efficient* characterization of solvability of an arbitrary (circularly symmetric) relation on an arbitrary set of rings. The characterization is topological and can be expressed as a number-theoretic property that can be checked efficiently.
- We present a *universal* distributed algorithm for computing any arbitrary (circularly symmetric) relation on any set of rings.

• Towards obtaining *message complexity* bounds, we derive a distributed algorithm for a natural generalization of *Leader Election*, in which a (non-zero) number of leaders are elected. We use this algorithm as a subroutine of our universal algorithm for collecting views; hence, we prove, as our main result, an upper bound on the message complexity of this particular instantiation of our universal algorithm to compute an *arbitrary* (circularly symmetric) relation on an *arbitrary* set of rings. The shown upper bound demonstrates a graceful degradation with the *Least Minimum Base*, a parameter indicating the degree of topological compatibility between the relation and the set of rings. We employ this universal upper bound to identify two interesting cases where an arbitrary relation can

* This work has been partially supported by the IST Program of the European Union under contract number IST-15964 (AEOLUS).

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 380–394, 2006.

be computed with an efficient number of $O(|R| \cdot \lg |R|)$ messages: The set of rings is *universal* (which allows the solvability of Leader Election), or *logarithmic* (where each identity appears at most $\lg |R|$ times).

1 Introduction

Motivation and Framework. Two of the best studied models in Distributed Computing Theory are the *eponymous* model and the *anonymous* model. In both models, *processors* may receive *inputs* and must reach *outputs* that are related to the inputs according to some (*recursive*) *relation*.

• In the eponymous model, processors have unique *identities*. This availability enables the computability of all relations: processors first solve *Leader Election* [11] to elect a leader among them; then, the leader undertakes computation of the relation and communicates the solution to the other processors.

• In the anonymous model, processors have *identical* identities and they run the same local algorithm. The impossibility of breaking this initial symmetry retains many relations unsolvable in the anonymous model; the prime example is the impossibility of solving *Leader Election* on an anonymous ring [1].

This long-known separation between the eponymous model and the anonymous model invites the investigation of an intermediate model, where there are identities available to the processors, but these are neither necessarily unique, nor necessarily all identical. Call this intermediate model the *partially eponymous* model. We consider a particular case of the partially eponymous model, that of the (*asynchronous*) *partially eponymous ring* R with *bidirectional* communication and *orientation*. Which relations are solvable on the partially eponymous ring? For *which message complexity* can an arbitrary relation be computed? (*Bit complexity* remains beyond the scope of this work.)

We focus on *circularly symmetric relations*, the broadest class of relations that are natural to consider for rings. Roughly speaking, in a circularly symmetric relation, shifting any *output vector* for a given *input vector* must yield a correct output vector for the correspondingly shifted input vector.

An essential attribute of most previous work on anonymous networks has been the requirement that the distributed algorithm for a particular relation runs on *all* networks and occasionally reports impossibility (exactly, of course, when it is impossible to return *admissible* outputs — ones that respect the relation). This concept will be called *computability* in this work. An orthogonal viewpoint is to actually isolate the subclass of networks on which it is always possible to return admissible outputs, in order to obtain tailored algorithms that are possibly more *efficient* (in terms of message complexity) than those running on *all* networks. This motivation leads to the concept of *solvability*: a relation is *solvable* on a set of networks if there is a distributed algorithm which, when run on any network in the set, leads all processors to reach admissible outputs (and *never* report impossibility).

Previous Work. Computation on anonymous networks was first studied in the seminal work of Angluin [1], where the impossibility of solving Leader Election

was first established. Yamashita and Kameda [12,13] have considered the solvability of several representative distributed computing problems on anonymous networks and characterized the class of (anonymous) networks on which each problem is solvable under different assumptions on the network attributes (e.g., size, topology, etc.) that are made available to the processors.

The more general model of an arbitrary partially eponymous network has been first considered by Yamashita and Kameda [15]. They focused on Leader Election and provided a graph-theoretic characterization of its solvability under different assumptions on the communication mode and the available (a)synchrony. Further work on the partially eponymous ring has been carried out in [6,7]. Chalopin *et al.* [6] very recently considered some specific generalizations of Leader Election in an arbitrary partially eponymous network. Under the assumption that processors have an approximate knowledge of the ring size, Dobrev and Pelc [7] presented both lower and upper bounds on message complexity for both the synchronous and the asynchronous cases of a partially eponymous ring.

Boldi and Vigna [3] have considered the more general solvability problem for an arbitrary relation on an arbitrary network and for any level of knowledge and anonymity (or *eponymity*) of the processors.

Attiya *et al.* [2] initiated the study of computing functions on the asynchronous anonymous ring. Flocchini *et al.* [8] consider the problems of Leader Election, *Edge Election* and *Multiset Sorting* on the asynchronous anonymous ring R where processors are distinguished by *input values* that are not necessarily distinct. So, input values are treated in the partially eponymous model of Flocchini *et al.* [8] as either identities or as inputs. We emphasize that the partially eponymous model of Flocchini *et al.* [8] does not simultaneously consider identities and inputs. Under the assumptions that input values are binary and $|R|$ is prime, Flocchini *et al.* [8, Theorems 4.1 and 4.2] provide lower and upper bounds on message complexity for these three problems. The lower and upper bounds are $\Omega(\sum_j (z_j^2 + t_j^2))$ and $O(\sum_j (z_j^2 + t_j^2) + |R| \cdot \lg |R|)$, respectively, where z_j and t_j are the lengths of consecutive blocks of 1's and 0's, respectively, in the vector of binary inputs.

The first efficient algorithm for Leader Election in the eponymous ring is based on the intuitive idea of domination in neighborhoods with progressively doubling size, which is due to Hirschberg and Sinclair [9]; it achieves an $O(|R| \cdot \lg |R|)$ upper bound on message complexity. A corresponding lower bound of $\Omega(|R| \cdot \lg |R|)$ has been established in [5].

Contribution. We start by studying solvability and computability.

• We discover (Theorem 1) that solvability is equivalent to *compatibility*, a new abstract, topological concept we introduce to capture the *possibility* that symmetries present in the initial configuration, comprised of the identities and the inputs, persist to the reached outputs.

To measure the initial symmetry, we use the *period* of a vector consisting of the identities and the inputs; the smaller the symmetry, the longer the period, and we call it the *Minimum Base*. It turns out that Minimum Base enjoys an elegant number-theoretic expression allowing for its efficient evaluation. Similarly, we

measure the symmetry in an output vector using its period. The possibility of persistence of initial symmetries to the final symmetries amounts to demanding that the period of *some* admissible output vector divides that of the initial vector, and this is our definition of compatibility. Compatibility can be checked efficiently (since it reduces to a number-theoretic property that can be checked efficiently); hence, our characterization of solvability is an *efficient* one.

• We present a *universal* algorithm for computing any arbitrary (circularly symmetric) relation on any set of rings (Theorem 2). This algorithm is comprised of any distributed algorithm for collecting *views* at each processor, followed by local steps specific to the particular relation.

• As an application of our characterization of solvability, we derive a particular characterization of solvability for (circularly symmetric) *aperiodic* relations (Theorem 3); Leader Election is an example of such relations. So, this determines a topological characterization of the class of relations that are equivalent to Leader Election with respect to solvability.

We then study message complexity (with respect to computability).

• As our chief algorithmic instrument, we present a distributed algorithm for a natural generalization of Leader Election in which a (non-zero) number of leaders must be elected; call it *Multiple Leader Election* (Proposition 2). This algorithm works correctly on a given configuration when advised with a lower bound k on the Minimum Base for that configuration. This distributed algorithm exploits the idea of *doubling neighborhoods* from the distributed algorithm of Hirschberg and Sinclair [9] for solving Leader Election on the eponymous ring; it achieves message complexity $O(|R| \cdot \lg k)$ (Proposition 1) for any advice k.

• In turn, we use the distributed algorithm for Multiple Leader Election to construct a *universal* algorithm to compute an arbitrary (circularly symmetric) relation Ψ on a set of rings $\mathbf{ID}$ (consisting of all rings with the same, arbitrarily chosen, identity multiplicities). The universal algorithm has message complexity $O\left((n^2/\mathsf{LMB}(\mathbf{ID},\Psi)) + n \cdot \lg \mathsf{LMB}(\mathbf{ID},\Psi)\right)$, where $\mathsf{LMB}(\mathbf{ID},\Psi)$ is the *Least Minimum Base* — the least value of Minimum Base over all configurations with rings coming from $\mathbf{ID}$ and input vectors coming from the domain of Ψ (Theorem 4). Here, $\mathsf{LMB}(\mathbf{ID},\Psi)$ is used as the advice k for the distributed algorithm to solve Multiple Leader Election. (Note that this is permissible when designing a distributed algorithm to compute the relation Ψ on the set of rings $\mathbf{ID}$.) Interestingly, the established upper bound demonstrates that the message complexity on rings of size n degrades gracefully with the Least Minimum Base, ranging from $O(n \cdot \lg n)$ for the eponymous ring to $O(n^2)$ for the anonymous ring. So, our universal upper bound is *tight* for these two extreme models.

• We are finally interested in determining sets of rings on which the universal upper bound on message complexity from Theorem 4 is low. In particular, on which sets of rings (of size n) is an upper bound of $O(n \cdot \lg n)$ possible? We identify two such (*incomparable*) classes of sets:

– Say that a set of rings is *universal* if Leader Election is solvable on it. So, every relation is solvable on such a universal set. We prove that a relation is computable with $O(n \cdot \lg n)$ messages on a universal set of rings (Theorem 5).

Hence, surprisingly, Leader Election is either unsolvable on a given set of rings, or efficiently computable on the given set with $O(n \cdot \lg n)$ messages.
- Say that a set of rings is *logarithmic* if each identity appears at most $\lg n$ times. We prove that a relation is computable with $O(n \cdot \lg n)$ messages on a logarithmic set of rings (Theorem 6); note that this holds even if the relation is *not* solvable on that set of rings.

Comparison to Directly Related Work. Whereas Boldi and Vigna [3] provide an **effective** characterization of anonymous solvability for any arbitrary relation on any arbitrary network, our work provides the *first* **efficient** characterization of partially eponymous solvability for any arbitrary relation on the ring (Theorem 4.1). It is not evident how the effective graph-theoretic characterization from [3] (involving *graph coverings* and *graph fibrations*) could yield an efficient characterization for the special case of the partially eponymous ring. In fact, our main goal has been to derive a *direct* solvability characterization for the particular case of the partially eponymous ring that bypasses the complex framework of graph coverings and graph fibrations developed in [3] for the general case of an arbitrary network. Although the work in [3] invests a great effort in translating concepts of Distributed Computing into some complex graph-theoretic form, our proof techniques for the solvability characterization are elementary.

Theorem 4 improves [8, Theorem 4.2] in three fronts. First, it works for an *arbitrary* ring size $|R|$, while [8, Theorem 4.2] assumes that $|R|$ is prime. Second, [8, Theorem 4.2] assumes binary inputs, while Theorem 4 makes no assumption on either inputs or identities. Third, and most important, Theorem 4 applies to *any* arbitrary relation, while [8, Theorem 4.2] is tailored to three specific relations (Leader Election, Edge Election and Multiset Sorting). We remark, however, that the worst-case message complexity in both Theorem 4 and [8, Theorem 4.2] is $\Theta(|R|^2)$. Note also that [8, Theorem 5.1] is the special case of Theorem 3 where Ψ is the Leader Election Relation.

Our definition for Least Minimum Base is built on top of *Minimum Base*, originally defined in [4,10] and used in [3,6] to obtain characterizations of solvability in anonymous networks. However, we exploit the very simple structure of the ring network to derive and use a particularly simple version of Minimum Base. For the case of the anonymous ring, Attiya *et al.* [2] defined the *Symmetry Index* to measure the symmetry in an initial configuration (containing only inputs); in contrast, (Least) Minimum Base measures asymmetry, while also taking identities into account.

Dobrev and Pelc [7, Theorem 3.1] prove an $\Omega(M \cdot n)$ lower bound on message complexity for the computability of Leader Election on the partially eponymous ring, where M is an upper bound on the ring size known to the processors; this implies a corresponding $\Omega(n^2)$ lower bound when the ring size is known exactly. This lower bound applies to the class of all rings; hence, it does not contradict the upper bound in Theorem 4, which applies to a set of rings **ID**.

2 Mathematical Preliminaries

Denote $\mathbb{N} = \{0, 1, 2, \ldots\}$, $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$, and $[n] = \{0, 1, \ldots, n-1\}$ for each integer $n \geq 1$. Denote GCD and LCM the functions mapping a set of integers to their *Greatest Common Divisor* and *Least Common Multiple*, respectively. We assume a global, possibly infinite set Σ (containing 0 and 1), and we consider a *vector* $\mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in \Sigma^n$. λ denotes the empty vector, while $\mathbf{x} \diamond \mathbf{y}$ denotes the concatenation of vectors $\mathbf{x}$ and $\mathbf{y}$. With each vector $\mathbf{x}$, we associate a multiset $\mathsf{M}(\mathbf{x})$ with the multiplicities of the entries of $\mathbf{x}$. We use the function M to partition Σ^n into equivalence classes, where all vectors in an equivalence class have the same image under M. Denote $\mathbf{X}$ the equivalence class containing the vector $\mathbf{x}$. By abuse of notation, $\mathsf{M}(\mathbf{X})$ will denote $\mathsf{M}(\mathbf{x})$ for any $\mathbf{x} \in \mathbf{X}$.

For any integer $k \in [n]$, the ***(cyclic) shift*** $\sigma_k(\mathbf{x})$ of vector $\mathbf{x}$ is the vector $\langle x_k, x_{k+1}, \ldots, x_{k+n-1} \rangle$, with indices taken modulo n; so, σ_k shifts $\mathbf{x}$, k places anti-clockwise. The definition is extended to all integers k in the natural way.

The ***period*** $\mathsf{T}(\mathbf{x})$ of vector $\mathbf{x}$ is the *least* integer $k, 0 < k \leq n$, such that $\sigma_k(\mathbf{x}) = \mathbf{x}$. Say that $\mathbf{x}$ is $\mathsf{T}(\mathbf{x})$-periodic; $\mathbf{x}$ is ***aperiodic*** if $\mathsf{T}(\mathbf{x}) = n$, and $\mathbf{x}$ is ***uniperiodic*** if $\mathsf{T}(\mathbf{x}) = 1$. Say that $\mathbf{x}$ is ***eponymous*** if each entry of $\mathbf{x}$ is unique; clearly, an eponymous vector is aperiodic, but not vice versa. Say that $\mathbf{x}$ is ***anonymous*** if all entries of $\mathbf{x}$ are identical; so, a vector is anonymous if and only if it is uniperiodic. Clearly, the period of a vector is invariant under shifting. So, the period captures the degree of circular asymmetry of a vector: the smaller the period, the more circular symmetries the vector has. We prove:

Lemma 1. *For each vector $\mathbf{x} \in \Sigma^n$, and $l, m \in \mathbb{N}$, $\sigma_l(\mathbf{x}) = \sigma_m(\mathbf{x})$ if and only if $l \equiv m \pmod{\mathsf{T}(\mathbf{x})}$.*

Call a vector $\widetilde{\mathbf{x}} \in \mathbf{X}$ a ***min-period*** vector of $\mathbf{X}$ if it minimizes period among all vectors in $\mathbf{X}$. We prove:

Lemma 2. *For each equivalence class $\mathbf{X} \subseteq \Sigma^n$, (1) $\mathsf{T}(\widetilde{\mathbf{x}}) = n/\mathsf{GCD}(\mathsf{M}(\mathbf{X}))$, and (2) for each vector $\mathbf{x} \in \mathbf{X}$, $\mathsf{T}(\widetilde{\mathbf{x}})$ divides $\mathsf{T}(\mathbf{x})$.*

Say that $\mathbf{X}$ is ***aperiodic*** if each vector $\mathbf{x} \in \mathbf{X}$ is aperiodic; say that $\mathbf{X}$ is ***uniperiodic*** if each vector $\mathbf{x} \in \mathbf{X}$ is uniperiodic. Say that $\mathbf{X}$ is ***k-periodic*** if the min-period vector $\widetilde{\mathbf{x}}$ of $\mathbf{X}$ is k-periodic. (So, aperiodic and uniperiodic are identified with n-periodic and 1-periodic, respectively.) Clearly, Lemma 2 (condition (1)) implies that $\mathbf{X}$ is $(n/\mathsf{GCD}(\mathsf{M}(\mathbf{X})))$-periodic. Hence, $\mathbf{X}$ is aperiodic if and only if $\mathsf{GCD}(\mathsf{M}(\mathbf{X})) = 1$, and $\mathbf{X}$ is uniperiodic if and only if $\mathsf{GCD}(\mathsf{M}(\mathbf{X})) = n$. Say that $\mathbf{X}$ is ***anonymous*** if all vectors $\mathbf{x} \in \mathbf{X}$ are anonymous; say that $\mathbf{X}$ is ***eponymous*** if all vectors $\mathbf{x} \in \mathbf{X}$ are eponymous.

We use the standard lexicographical ordering $\preceq$ on Σ^n. We write $\mathbf{x} \prec \mathbf{y}$ to mean $\mathbf{x} \preceq \mathbf{y}$ and $\mathbf{x} \neq \mathbf{y}$. For each $k \in [n+1]$, the ***prefix*** of order k of $\mathbf{x}$, denoted $P_k(\mathbf{x})$, is given by $P_k(\mathbf{x}) = \langle x_0, x_1, \ldots, x_{k-1} \rangle$, with $P_0(\mathbf{x}) = \lambda$. Clearly, for $k < l$, $P_k(\mathbf{x}) \prec P_k(\mathbf{y})$ implies $P_l(\mathbf{x}) \prec P_l(\mathbf{y})$ (and, in particular, $\mathbf{x} \prec \mathbf{y}$). The ***shuffle*** of two vectors $\mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ and $\mathbf{y} = \langle y_0, y_1, \ldots, y_{n-1} \rangle$, denoted by $\mathbf{x} \| \mathbf{y}$, is the vector $\mathbf{x} \| \mathbf{y} = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}) \rangle$. We observe:

Lemma 3. *For each pair of vectors* $\mathbf{x}, \mathbf{y} \in \Sigma^n$, $\mathsf{T}(\mathbf{x} \| \mathbf{y}) = \mathsf{LCM}(\mathsf{T}(\mathbf{x}), \mathsf{T}(\mathbf{y}))$.

A (recursive) *relation* is a subset $\Psi \subseteq \Sigma^n \times \Sigma^n$. For a vector $\mathbf{x} \in \Sigma^n$, $\Psi(\mathbf{x}) = \{\mathbf{y} \mid (\mathbf{x}, \mathbf{y}) \in \Psi\}$; every vector $\mathbf{y} \in \Psi(\mathbf{x})$ is an *image* of $\mathbf{x}$ under Ψ. The set $\mathsf{Dom}(\Psi)$ of all vectors $\mathbf{x} \in \Sigma^n$ with at least one image under Ψ is the *domain* of Ψ; the set of all images of all vectors $\mathbf{x} \in \Sigma^n$ is the *image* of Ψ, denoted as $\mathsf{Im}(\Psi)$. The relation Ψ is *total* if $\mathsf{Dom}(\Psi) = \Sigma^n$. Given two relations $\Psi_1, \Psi_2 \subseteq \Sigma^n \times \Sigma^n$, their *composition* is the relation $\Psi_1 \circ \Psi_2 = \{(\mathbf{x}, \mathbf{y}) \mid (\mathbf{x}, \mathbf{z}) \in \Psi_2$ and $(\mathbf{z}, \mathbf{y}) \in \Psi_1$ for some $\mathbf{z} \in \Sigma^n\}$. For a relation $\Psi \subseteq \Sigma^n \times \Sigma^n$, note that $\sigma_1 \circ \Psi = \{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} = \sigma_1(\mathbf{z})$ for some $\mathbf{z} \in \Psi(\mathbf{x})\}$; note also that $\Psi \circ \sigma_1 = \{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} \in \Psi(\mathbf{z})$ where $\mathbf{z} = \sigma_1(\mathbf{x})\}$. In other words, for a vector $\mathbf{x}$, $\sigma_1 \circ \Psi(\mathbf{x}) = \{\mathbf{y} \mid \mathbf{y} = \sigma_1(\mathbf{z})$ for some $\mathbf{z} \in \Psi(\mathbf{x})\}$ and $\Psi \circ \sigma_1(\mathbf{x}) = \{\mathbf{y} \mid \mathbf{y} \in \Psi(\sigma_1(\mathbf{x}))\}$. Thus, $\sigma_1 \circ \Psi$ maps inputs to shifts of their images, while $\Psi \circ \sigma_1$ maps inputs to images of their shifts. The relation Ψ is ***circularly symmetric*** if $\sigma_1 \circ \Psi \subseteq \Psi \circ \sigma_1$. Intuitively, in a circularly symmetric relation, shifts of images are always images of shifts. A direct induction implies that $\sigma_k \circ \Psi \subseteq \Psi \circ \sigma_k$ for any circularly symmetric relation Ψ and for all integers $k \in \mathbb{N}$.

The relation $\Psi \subseteq \Sigma^n \times \Sigma^n$ is ***aperiodic*** if each vector in $\mathsf{Im}(\Psi)$ is aperiodic; so, each image under Ψ has no circular symmetries. On the other extreme, the relation $\Psi \subseteq \Sigma^n \times \Sigma^n$ is ***uniperiodic*** if each vector in $\mathsf{Im}(\Psi)$ is uniperiodic; so, each image under Ψ is constant. In the middle, the relation $\Psi \subseteq \Sigma^n \times \Sigma^n$ is ***k-periodic*** if each vectors in $\mathsf{Im}(\Psi)$ is k-periodic. Thus, n-periodic and 1-periodic relations are precisely the aperiodic and uniperiodic relations, respectively.

In the ***Leader Election Relation*** $LE \subseteq \Sigma^n \times \Sigma^n$, the set of images of an input vector $\mathbf{x}$ is the set of all vectors with exactly one 1 and $n-1$ 0's; 1 and 0 correspond to "elected" and "non-elected", respectively. Clearly, the Leader Election Relation is both circularly symmetric and aperiodic.

We now discuss a generalization of the Leader Election Relation. Consider a function $\Phi : \Sigma^n \to \mathbb{Z}^+$. In the ***$\Phi$-Leader Election Relation*** Φ-$LE \subseteq \Sigma^n \times \Sigma^n$, the set of images of an input vector $\mathbf{x}$ is the set of all binary output vectors with the number of 1's ranging from 1 to $\Phi(\mathbf{x})$ (both inclusive). The special case where $\Phi(\mathbf{x}) = 1$ for all vectors $\mathbf{x} \in \Sigma^n$ is precisely the Leader Election Relation. A ***Multiple Leader Election Relation*** is a Φ-Leader Election Relation for some such function Φ.

3 The Partially Eponymous Ring

General. We start with the standard model of an *asynchronous*, *anonymous ring* as studied, for example, in [2,8]. We assume that the ring is *oriented* and *bidirectional*. We augment this model so that processors have ***identities*** that are neither necessarily all identical, nor necessarily unique. Call it a ***partially eponymous ring***. In the ***anonymous ring*** identities are all identical, while in the ***eponymous ring*** identities are unique.

A ***ring*** R is a cyclic arrangement of $|R|$ identical processors $0, 1, \ldots, |R|-1$. Processor j has an identity id_j and receives an input in_j. The ***identity vector*** is $\mathbf{id} = \langle id_0, id_1, \ldots, id_{|R|-1} \rangle$; the ***input vector*** is $\mathbf{in} = \langle in_0, in_1, \ldots, in_{|R|-1} \rangle$.

Note that for the anonymous ring, $\mathsf{T}(\mathbf{id}) = 1$; for the eponymous ring, $\mathsf{T}(\mathbf{id}) = |R|$. The ***(initial) configuration*** of the ring R is the pair $\langle \mathbf{id}, \mathbf{in} \rangle$. Each processor must reach an output out_j by running a local algorithm and communicating with its two neighbors. The ***output vector*** is $\mathbf{out} = \langle out_0, out_1, \ldots, out_{|R|-1} \rangle$.

There is a *single* local *algorithm* A run by all processors; A is represented as a state machine. Each computation step of A at processor j is dependent on the current state of j, the messages currently received at j and on the local identity id_j and input in_j. A ***distributed algorithm*** $\mathcal{A}$ is a collection of local algorithms, one for each processor. We restrict our attention to ***non-uniform*** distributed algorithms, where the size of the ring is "hard-wired" into the single local algorithm. So, we consider rings of a certain size n. The distributed algorithm $\mathcal{A}$ induces a set of (*asynchronous*) *executions*.

Each identity vector $\mathbf{id} \in \Sigma^n$ specifies a single ring; by abuse of notation, denote as $\mathbf{id}$ the specified ring. An equivalence class $\mathbf{ID} \subseteq \Sigma^n$ induces a set of rings, each corresponding to some particular identity vector $\mathbf{id} \in \mathbf{ID}$; by abuse of notation, denote as $\mathbf{ID}$ the induced set.

Solvability and Computability. Consider a configuration $\langle \mathbf{id}, \mathbf{in} \rangle$. Say that the distributed algorithm $\mathcal{A}$ ***solves the set of output vectors*** $\mathcal{OUT}$ ***on the configuration*** $\langle \mathbf{id}, \mathbf{in} \rangle$ if each execution of $\mathcal{A}$ on the ring $\mathbf{id}$ with input $\mathbf{in}$ results to an output vector $\mathbf{out} \in \mathcal{OUT}$. Say that the set of output vectors $\mathcal{OUT}$ is ***solvable on the configuration*** $\langle \mathbf{id}, \mathbf{in} \rangle$ if there is a distributed algorithm $\mathcal{A}$ that solves $\mathcal{OUT}$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. Say that the relation Ψ is ***solvable on the set of rings*** $\mathcal{R}$ if there is a distributed algorithm $\mathcal{A}$ such that for each configuration $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(\Psi)$, $\mathcal{A}$ solves $\Psi(\mathbf{in})$ on $\langle \mathbf{id}, \mathbf{in} \rangle$.

Say that the distributed algorithm $\mathcal{A}$ ***computes the set of output vectors*** $\mathcal{OUT}$ ***on the configuration*** $\langle \mathbf{id}, \mathbf{in} \rangle$ if the following holds: if $\mathcal{OUT}$ is solvable on the configuration $\langle \mathbf{id}, \mathbf{in} \rangle$, then $\mathcal{A}$ solves $\mathcal{OUT}$ in $\langle \mathbf{id}, \mathbf{in} \rangle$; else $\mathcal{A}$ solves $\{\perp^n\}$ on $\langle \mathbf{id}, \mathbf{in} \rangle$ (an *unsolvability* output). We now develop the notion of a distributed algorithm working for a set of rings and on the entire domain of the relation Ψ; intuitively, the set of rings represents the "knowledge" that the algorithm requires. Formally, the distributed algorithm $\mathcal{A}$ ***computes the relation*** Ψ ***on a set of rings*** $\mathcal{R}$ ***with*** $g(n)$ ***messages*** if for each configuration $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(\Psi)$, $\mathcal{A}$ computes $\Psi(\mathbf{in})$ on the configuration $\langle \mathbf{id}, \mathbf{in} \rangle$. The relation Ψ is ***computable on a set of rings*** $\mathcal{R}$ ***with*** $g(n)$ ***messages*** if there is a distributed algorithm $\mathcal{A}$ that computes Ψ on $\mathbf{ID}$ with $g(n)$ messages. Note that solvability of a relation Ψ on a set of rings $\mathbf{ID}$ implies computability of Ψ on $\mathbf{ID}$ (with some number of messages). However, the inverse does not necessarily hold.

The Least Minimum Base. The ***Minimum Base*** $\mathsf{MB}(\mathbf{id}, \mathbf{in})$ of a configuration $\langle \mathbf{id}, \mathbf{in} \rangle$ is defined by $\mathsf{MB}(\mathbf{id}, \mathbf{in}) = \mathsf{T}(\mathbf{id} \parallel \mathbf{in})$ (cf. [4,10]). For a set of rings $\mathbf{ID}$ with a common input vector $\mathbf{in}$, the ***Min-Period Minimum Base*** $\mathsf{MB}(\mathbf{ID}, \mathbf{in})$ is defined by $\mathsf{MB}(\mathbf{ID}, \mathbf{in}) = \mathsf{MB}(\widetilde{\mathbf{id}}, \mathbf{in})$, where $\widetilde{\mathbf{id}}$ is the min-period vector of $\mathbf{ID}$. Recall that $\mathsf{T}(\widetilde{\mathbf{id}}) = n/\mathsf{GCD}(\mathsf{M}(\mathbf{ID}))$. Hence, Lemma 3 implies that $\mathsf{MB}(\mathbf{ID}, \mathbf{in}) = \mathsf{LCM}(\mathsf{T}(\widetilde{\mathbf{id}}), \mathsf{T}(\mathbf{in})) = \mathsf{LCM}\,(n/\mathsf{GCD}(\mathsf{M}(\mathbf{ID})), \mathsf{T}(\mathbf{in}))$. By Lemma 2 (condition (2)), $\mathsf{T}(\widetilde{\mathbf{id}})$ divides $\mathsf{T}(\mathbf{id})$ for each ring $\mathbf{id} \in \mathbf{ID}$. Thus, it follows that

$\mathsf{MB}(\mathbf{ID}, \mathbf{in})$ divides $\mathsf{LCM}(\mathsf{T}(\mathbf{id}), \mathsf{T}(\mathbf{in}))$ for each ring $\mathbf{id} \in \mathbf{ID}$. Note that if $\mathbf{ID}$ is the set of anonymous rings, then $\mathsf{MB}(\mathbf{ID}, \mathbf{in}) = \mathsf{LCM}(1, \mathsf{T}(\mathbf{in})) = \mathsf{T}(\mathbf{in})$; if $\mathbf{ID}$ is the set of eponymous rings, then $\mathsf{MB}(\mathbf{ID}, \mathbf{in}) = \mathsf{LCM}(\mathsf{T}(\widetilde{\mathbf{id}}), \mathsf{T}(\mathbf{in})) \geq \mathsf{T}(\mathbf{in})$. So, intuitively, the Minimum Base is an indicator of computability. The ***Least Minimum Base*** $\mathsf{LMB}(\mathbf{ID}, \Psi)$ of a set of rings $\mathbf{ID}$ and a relation Ψ is defined by $\mathsf{LMB}(\mathbf{ID}, \Psi) = \mathsf{min}\{\mathsf{MB}(\mathbf{id}, \mathbf{in}) \mid \langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(\Psi)\}$.

Views. We conclude with a definition that extends one in [15, Section 3] to a ring where processors receive inputs. Given a ring $\mathbf{id}$ with input vector $\mathbf{in}$, the ***view*** of processor j is $view_j(\mathbf{id}, \mathbf{in}) = \sigma_j(\mathbf{id} \parallel \mathbf{in}) = \sigma_j(\mathbf{id}) \parallel \sigma_j(\mathbf{in})$. Clearly, views of processors are cyclic shifts of each other. There is a direct, non-uniform distributed algorithm $\mathcal{A}_{\mathrm{CV}}$ with message complexity $\Theta(|R|^2)$ that allows each processor to construct its own view on a ring R. It is simple to prove:

Lemma 4. *Consider a ring* $\mathbf{id}$ *with input* $\mathbf{in}$ *and two processors* j *and* k *with* $view_j(\mathbf{id}, \mathbf{in}) = view_k(\mathbf{id}, \mathbf{in})$. *Then, in a synchronous execution of a distributed algorithm with output vector* $\mathbf{out}$, $out_j = out_k$.

4 Solvability and Computability

Preliminaries. We provide a definition of compatibility between a set of output vectors $\mathcal{OUT}$ and a configuration $\langle \mathbf{id}, \mathbf{in} \rangle$. The set of output vectors $\mathcal{OUT}$ is ***compatible*** with the configuration $\langle \mathbf{id}, \mathbf{in} \rangle$ if there is an output vector $\mathbf{out} \in \mathcal{OUT}$ such that $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\mathbf{id}, \mathbf{in})$. We prove:

Lemma 5. *Assume that a set of output vectors* $\mathcal{OUT}$ *is solvable on a configuration* $\langle \mathbf{id}, \mathbf{in} \rangle$. *Then,* $\mathcal{OUT}$ *is compatible with* $\langle \mathbf{id}, \mathbf{in} \rangle$.

Proof (sketch). By assumption, there is a distributed algorithm $\mathcal{A}$ that solves $\mathcal{OUT}$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. Fix a synchronous execution of $\mathcal{A}$ on $\langle \mathbf{id}, \mathbf{in} \rangle$, and consider the associated vector $\mathbf{out} \in \mathcal{OUT}$. Recall that $view_j(\mathbf{id}, \mathbf{in}) = \sigma_j(\mathbf{id} \parallel \mathbf{in})$, so that $view_{j+\mathsf{T}(\mathbf{id}\parallel\mathbf{in})}(\mathbf{id}, \mathbf{in}) = \sigma_{j+\mathsf{T}(\mathbf{id}\parallel\mathbf{in})}(\mathbf{id} \parallel \mathbf{in})$. By Lemma 1, it holds that $\sigma_j(\mathbf{id} \parallel \mathbf{in}) = \sigma_{j+\mathsf{T}(\mathbf{id}\parallel\mathbf{in})}(\mathbf{id} \parallel \mathbf{in})$. So $view_j(\mathbf{id}, \mathbf{in}) = view_{j+\mathsf{T}(\mathbf{id}\parallel\mathbf{in})}(\mathbf{id}, \mathbf{in})$, and Lemma 4 implies that $out_j = out_{j+\mathsf{T}(\mathbf{id}\parallel\mathbf{in})}$. Since j was chosen arbitrarily, this implies that $\sigma_{\mathsf{T}(\mathbf{id}\parallel\mathbf{in})}(\mathbf{out}) = \mathbf{out}$. By definition of period, Lemma 1 implies that $\mathsf{T}(\mathbf{out})$ divides $\mathsf{T}(\mathbf{id} \parallel \mathbf{in}) = \mathsf{MB}(\mathbf{id}, \mathbf{in})$. □

We now introduce the distributed (non-uniform) algorithm $\mathcal{A}_\Psi$ associated with an arbitrary circularly symmetric relation $\Psi \in \Sigma^n \times \Sigma^n$; the algorithm is described in Figure 1. Note that the distributed algorithm $\mathcal{A}_\Psi$ does *not* specify how the views are constructed in *Step* 2. The views can be constructed by invoking, for example, the distributed algorithm $\mathcal{A}_{\mathrm{CV}}$ which collects the identities and inputs of processors using n^2 messages. All remaining steps are local. *Steps* 3–6 enable processors to choose a common output vector, while *Step* 6 enables processor j to output its individual coordinate in this vector. The set *Choices* contains all candidates for the common output vector; in *Step* 6, processors use a common function (e.g., min) to single out one of the candidates. We prove:

$\mathcal{A}_\Psi$: CODE FOR PROCESSOR j WITH IDENTITY id_j AND INPUT in_j

1: Upon receiving message $\langle wake \rangle$ **do**
2: Construct $view_j(\mathbf{id}, \mathbf{in}) = vid_j \,\|\, vin_j$. $\backslash * \; vid_j = \sigma_j(\mathbf{id})$, $vin_j = \sigma_j(\mathbf{in}) \; * \backslash$
3: $Views_j[i] := \langle \sigma_i(vid_j), \sigma_i(vin_j) \rangle$ for each $i \in [|R|]$.
4: $Choices := \{(\mathbf{x}, \mathbf{y}, \mathbf{z}) \mid \langle \mathbf{x}, \mathbf{y} \rangle \in Views_j;\ \mathbf{z} \in \Psi(\mathbf{y});\ \mathsf{T}(\mathbf{z}) \text{ divides } \mathsf{MB}(\mathbf{x}, \mathbf{y})\}$.
5: **If** $Choices = \emptyset$ **then** $out_j := \perp$ and terminate.
6: (a) $(\underline{\mathbf{x}}, \underline{\mathbf{y}}, \underline{\mathbf{z}}) := \mathsf{min}(Choices)$; (b) $k_j := \mathsf{min}\{i \in [|R|] \mid Views_j[i] = \langle \underline{\mathbf{x}}, \underline{\mathbf{y}} \rangle\}$.
7: Set out_j to be the first entry of $\sigma_{-k_j}(\underline{\mathbf{z}})$ and terminate.

Fig. 1. Algorithm $\mathcal{A}_\Psi$: code for processor j

Lemma 6. *For a configuration* $\langle \mathbf{id}, \mathbf{in} \rangle$*, either* $\mathcal{A}_\Psi$ *solves* $\Psi(\mathbf{in})$ *on* $\langle \mathbf{id}, \mathbf{in} \rangle$*, or* $\mathcal{A}_\Psi$ *solves* $\{\perp^n\}$ *on* $\langle \mathbf{id}, \mathbf{in} \rangle$*.*

The proof of Lemma 6 requires Ψ to be circularly symmetric. It is not evident whether this assumption is *essential* for the partially eponymous ring, although it is known to be so for computing functions on the anonymous ring [2].

Main Results. Our first result concerns the solvability of an arbitrary (circularly symmetric) relation on an arbitrary set of rings **ID**. We provide a definition of compatibility between a relation $\Psi \subseteq \Sigma^n \times \Sigma^n$ and a set of rings $\mathbf{ID} \subseteq \Sigma^n$. The relation Ψ is ***compatible*** with the set of rings **ID** if for each input vector $\mathbf{in} \in \mathsf{Dom}(\Psi)$, there is some output vector $\mathbf{out} \in \Psi(\mathbf{in})$ such that $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\mathbf{ID}, \mathbf{in})$. Recall that $\mathsf{MB}(\mathbf{ID}, \mathbf{in})$ can be computed efficiently; thus, compatibility can be checked efficiently. We prove:

Theorem 1 (Partially Eponymous Solvability Theorem). *A circularly symmetric relation* Ψ *is solvable on a set of rings* **ID** *if and only if* Ψ *is compatible with* **ID***.*

Proof (sketch). Assume that Ψ is solvable on **ID**. By definition of solvability, there is a distributed algorithm $\mathcal{A}$ such that for each configuration $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(\Psi)$, $\mathcal{A}$ solves the set of vectors $\Psi(\mathbf{in})$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. So, fix the configuration $\langle \widetilde{\mathbf{id}}, \mathbf{in} \rangle$ for an arbitrary vector $\mathbf{in} \in \mathsf{Dom}(\Psi)$. If follows that the set of vectors $\Psi(\mathbf{in})$ is solvable on $\langle \widetilde{\mathbf{id}}, \mathbf{in} \rangle$. By Lemma 5, this implies that $\Psi(\mathbf{in})$ is compatible with $\langle \widetilde{\mathbf{id}}, \mathbf{in} \rangle$. By definition of compatibility between a set of output vectors and a configuration, it follows that there is some output vector $\mathbf{out} \in \Psi(\mathbf{in})$ such that $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\widetilde{\mathbf{id}}, \mathbf{in}) = \mathsf{MB}(\mathbf{ID}, \mathbf{in})$ (by definition of Min-Period Minimum Base). By definition of compatibility, the claim follows.

Assume now that Ψ is compatible with **ID**. By definition of compatibility, for each input vector $\mathbf{in} \in \mathsf{Dom}(\Psi)$, there is an output vector $\mathbf{out} \in \Psi(\mathbf{in})$ such that $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\mathbf{ID}, \mathbf{in})$. Fix an arbitrary configuration $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(\Psi)$. Recall that $\mathsf{MB}(\mathbf{ID}, \mathbf{in})$ divides $\mathsf{MB}(\mathbf{id}, \mathbf{in})$. It follows that there is some output vector $\mathbf{out} \in \Psi(\mathbf{in})$ such that $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\mathbf{id}, \mathbf{in})$. Recall the distributed algorithm $\mathcal{A}_\Psi$. Clearly, $\langle \mathbf{id}, \mathbf{in} \rangle$ is an entry of $Views_j$ for each processor $j \in [n]$. Since $\mathbf{out} \in \Psi(\mathbf{in})$ and $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\mathbf{id}, \mathbf{in})$, it follows

that $(\mathbf{id}, \mathbf{in}, \mathbf{out}) \in Choices$; so $Choices \neq \emptyset$. By the algorithm $\mathcal{A}_\Psi$ (*Step* 5), it follows that the algorithm does not solve $\{\perp^n\}$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. Hence, Lemma 6 implies that $\mathcal{A}_\Psi$ solves $\Psi(\mathbf{in})$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. Since the configuration $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(\Psi)$ was chosen arbitrarily, it follows that Ψ is solvable on $\mathbf{ID}$. □

Since compatibility is efficiently checkable, Theorem 1 provides an efficient characterization of solvability for the partially eponymous ring. We now prove:

Theorem 2 (Partially Eponymous Computability Theorem). *Algorithm $\mathcal{A}_\Psi$ computes the circularly symmetric relation Ψ on a set of rings* $\mathbf{ID}$.

Proof (sketch). Fix an arbitrary configuration $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(\Psi)$. We will prove that $\mathcal{A}_\Psi$ computes $\Psi(\mathbf{in})$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. We proceed by case analysis: Assume first that $\Psi(\mathbf{in})$ is solvable on $\langle \mathbf{id}, \mathbf{in} \rangle$. By Lemma 5 it follows that $\Psi(\mathbf{in})$ is compatible with $\langle \mathbf{id}, \mathbf{in} \rangle$. By definition of compatibility of a set of output vectors with a configuration, this implies that there is some output vector $\mathbf{out} \in \Psi(\mathbf{in})$ such that $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\mathbf{id}, \mathbf{in})$. From the distributed algorithm $\mathcal{A}_\Psi$, $\langle \mathbf{id}, \mathbf{in} \rangle$ is an entry of $Views_j$ for each processor $j \in [n]$. Since, $\mathbf{out} \in \Psi(\mathbf{in})$ and $\mathsf{T}(\mathbf{out})$ divides $\mathsf{MB}(\mathbf{id}, \mathbf{in})$, it follows that $(\mathbf{id}, \mathbf{in}, \mathbf{out}) \in Choices$; so $Choices \neq \emptyset$. By the algorithm $\mathcal{A}_\Psi$ (*Step* 5), it follows that $\mathcal{A}_\Psi$ does not solve $\{\perp^n\}$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. Hence, Lemma 6 implies that $\mathcal{A}_\Psi$ solves $\Psi(\mathbf{in})$ on $\langle \mathbf{id}, \mathbf{in} \rangle$, as needed. Assume now that $\Psi(\mathbf{in})$ is not solvable on $\langle \mathbf{id}, \mathbf{in} \rangle$. This implies that $\mathcal{A}_\Psi$ does not solve $\Psi(\mathbf{in})$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. By Lemma 6, $\mathcal{A}_\Psi$ solves $\{\perp^n\}$ on $\langle \mathbf{id}, \mathbf{in} \rangle$. □

Applications. For uniperiodic relations, Theorem 1 immediately implies that every circularly symmetric, uniperiodic relation is solvable on any set if rings $\mathbf{ID}$. As a natural application of Theorem 1 on aperiodic relations, we prove:

Theorem 3 (Solvability of Aperiodic Relations). *A total, circularly symmetric, aperiodic relation Ψ is solvable on a set of rings* $\mathbf{ID}$ *if and only if* $\mathbf{ID}$ *is aperiodic.*

Actually, Theorem 3 applies more generally to a *non-total*, circularly symmetric, aperiodic relation Ψ, as long as there is at least one constant vector in $\mathsf{Dom}(\Psi)$. Many relations from the literature assume no inputs, so that their domain consists of a single, constant input vector; Theorem 3 applies to all such relations. Finally, note that Theorem 3 can be further generalized to prove that a circularly symmetric, k-periodic relation Ψ, with at least one constant vector in $\mathsf{Dom}(\Psi)$, is solvable on a set of rings $\mathbf{ID}$ if and only if $\mathbf{ID}$ is l-periodic, and k divides l.

5 Message Complexity

Multiple Leader Election as a Tool. We present an asynchronous distributed algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ with advice k. Here, k is an integer that is available to each processor (e.g., it is "hard-wired" into its local algorithm much in the same way the ring size is in a non-uniform distributed algorithm). k will act as a parameter

to $\mathcal{A}_{\mathrm{MLE}}(k)$: $\mathcal{A}_{\mathrm{MLE}}(k)$ will satisfy a particular correctness property for certain specific advices k; furthermore, the message complexity of $\mathcal{A}_{\mathrm{MLE}}(k)$ will depend on k. The algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ is similar in spirit to the well-known (*neighborhood doubling*) asynchronous distributed algorithm of Hirschberg and Sinclair [9] that computes Leader Election on the eponymous ring R. (Recall that the algorithm of Hirschberg and Sinclair uses $O(|R| \cdot \lg |R|)$ messages.) So, each processor explores neighborhoods around it whose size doubles in each *phase*; in phase r, the processor collects identities of other processors in the neighborhood that are 2^r to the left (counter-clockwise) of it, or $2^{r+1} - 1$ to the right (clockwise) of it. It then uses these identities to locally compute the prefixes of length 2^r of the views of all processors that are 2^r to the left or to the right of it. Then, the processor compares the prefix of length 2^r of its own view to those prefixes it has computed; it survives the phase (so that it can proceed to the next phase) if and only if its own prefix is *the lexicographically least* among all 2^{r+1} prefixes of the views of its neighbors that it has computed. Note that by *Step* 21, $r \leq \lceil \lg k \rceil - 1$; thus, k determines the size of the largest neighborhood that each processor will explore before terminating. Also, note that the major difference between our algorithm for the partially eponymous ring and the classical algorithm of Hirschberg and Sinclair to compute Leader Election on the eponymous ring is that our algorithm awards processors to proceed to the next phase on the basis of the *computed prefixes* of processors' views (which change across phases), as opposed to processors' identities (that remain constant across phases). Note that the lexicographic ordering provides the property that views and their corresponding prefixes are consistently ordered. Hence, the comparison of prefixes in *Step* 20 is essentially a comparison of views (hence, an efficient one since it avoids the full construction of views). This is an essential feature of our algorithm, and its achieved message efficiency is due to this feature. The distributed algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ appears in pseudocode in Figure 2. We proceed to prove certain message complexity and correctness properties of $\mathcal{A}_{\mathrm{MLE}}(k)$.

Using an analysis similar to the analysis of the message complexity for the algorithm of Hirschberg and Sinclair [9], we prove an upper bound on the message complexity of the distributed algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$. Since k determines the size of the largest neighborhood that each processor will explore before terminating, the message complexity of the distributed algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ increases with k.

Proposition 1. *Algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ uses $O(|R| \cdot \lg k)$ messages on the ring R.*

We continue with a correctness property of the algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ for specific advices k. For any k that divides n consider the function $\Phi_k : \Sigma^n \to \mathbb{Z}^+$ such that $\Phi_k(\mathbf{in}) = 2n/k$ for each input vector $\mathbf{in} \in \Sigma^n$. We continue to prove:

Proposition 2. *Algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ with advice $k, 1 \leq k \leq \mathsf{MB}(\mathbf{id}, \mathbf{in})$ solves the set of output vectors Φ_k-$LE(\mathbf{in})$ on the configuration $\langle \mathbf{id}, \mathbf{in} \rangle$.*

Intuitively, we wish to elect as few leaders as possible, since each leader will be subsequently asked to undertake an additional (message intensive) distributed computation. Proposition 2 establishes that the larger the advice k is, the less leaders are elected; on the other hand, k cannot be chosen to be arbitrarily large.

$\mathcal{A}_{\mathrm{MLE}}(k)$: CODE FOR PROCESSOR j WITH IDENTITY id_j AND INPUT in_j

Initially, $label_j = segment_j = left\_segment_j = right\_segment_j = \lambda$.

1: Upon receiving message $\langle wake \rangle$ **do**
2: Send message $\langle probe, 0, 1 \rangle$ to left.

3: Upon receiving message $\langle probe, r, d \rangle$ from right **do**
4: **If** $d < 2^r$ **then** send message $\langle probe, r, d+1 \rangle$ to left.
5: **If** $d = 2^r$ **then** send message $\langle reply, \langle (id_j, in_j) \rangle, r, 1 \rangle$ to right.

6: Upon receiving message $\langle reply, s, r, d \rangle$ from left **do**
7: **If** $d < 2^r$ **then** send message $\langle reply, s \diamond \langle (id_j, in_j) \rangle, r, d+1 \rangle$ to right.
8: **If** $d = 2^r$ **then do**
9: $left\_segment_j := s$.
10: Send message $\langle probe, r, 1 \rangle$ to right.

11: Upon receiving message $\langle probe, r, d \rangle$ from left **do**
12: **If** $d < 2^{r+1} - 1$ **then** send message $\langle probe, r, d+1 \rangle$ to right.
13: **If** $d = 2^{r+1} - 1$ **then** send message $\langle reply, \langle (id_j, in_j) \rangle, r, 1 \rangle$ to left.

14: Upon receiving message $\langle reply, s, r, d \rangle$ from right **do**
15: **If** $d < 2^{r+1} - 1$ **then** send message $\langle reply, \langle (id_j, in_j) \rangle \diamond s, r, d+1 \rangle$ to left.
16: **If** $d = 2^{r+1} - 1$ **then do**
17: $right\_segment_j := s$.
18: $segment_j := left\_segment_j \diamond \langle (id_j, in_j) \rangle \diamond right\_segment_j$.
19: $label_j := P_{2^r}(\sigma_{2^r}(segment_j))$.
20: **If** $label_j \prec P_{2^r}(\sigma_i(segment_j))$ and $label_j \preceq P_{2^r}(\sigma_{2^r+i+1}(segment_j))$, for all $i \in [2^r]$ **then do**
21: **If** $2^{r+1} \geq k$ **then** terminate as a leader.
22: **Else** send message $\langle probe, r+1, 1 \rangle$ to left.
23: **Else** terminate as a non-leader.

Fig. 2. Algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$: code for processor j

A Universal Upper Bound on Message Complexity. Recall that we have established correctness guarantees for the algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ only when $k \leq \mathsf{MB}(\mathbf{id}, \mathbf{in})$ on configuration $\langle \mathbf{id}, \mathbf{in} \rangle$. Recall also that we wish to maximize k, so as to minimize the number of elected leaders. Furthermore, if algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ is to be used for electing a number of leaders, the value of k must be "known" to the processors. This raises the natural question of what such an appropriate value for k is. In what follows, we choose $k = \mathsf{LMB}(\mathbf{ID}, \Psi)$, the least upper bound imposed on k by Proposition 2, across all configurations $\langle \mathbf{id}, \mathbf{in} \rangle$. We next ask how this advice k relates to the message complexity of computing an arbitrary (circularly symmetric) relation $\Psi \subseteq \Sigma^n \times \Sigma^n$ on an arbitrary set of rings $\mathbf{ID} \subseteq \Sigma^n$; recall that Ψ is computable on $\mathbf{ID}$ with $O(n^2)$ messages (Theorem 2). We prove:

Theorem 4 (Partially Eponymous Message Complexity Theorem). *The circularly symmetric relation Ψ is computable on a set of rings* $\mathbf{ID}$ *with* $O\left((n^2/\mathsf{LMB}(\mathbf{ID}, \Psi)) + n \cdot \lg \mathsf{LMB}(\mathbf{ID}, \Psi)\right)$ *messages.*

Proof (sketch). Here is a distributed algorithm $\mathcal{A}$ (which is an instantiation of $\mathcal{A}_\Psi$) to compute Ψ on $\mathbf{ID}$ with that many messages. Consider an arbitrary configuration $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \Sigma^n$. $\mathcal{A}$ proceeds as follows:

• On top, the distributed algorithm $\mathcal{A}_\Psi$ (see Figure 1) is invoked to compute Ψ on $\mathbf{ID}$. *Step* 2 is implemented by the following steps:
- First, the processors run the distributed algorithm $\mathcal{A}_{\mathrm{MLE}}(k)$ with advice $k = \mathsf{LMB}(\mathbf{ID}, \Psi) \leq \mathsf{MB}(\mathbf{id}, \mathbf{in})$ using $O(n \cdot \lg \mathsf{LMB}(\mathbf{ID}, \Psi))$ messages (by Proposition 1); by Proposition 2, there are now elected at least 1 and at most $O(n/\mathsf{LMB}(\mathbf{ID}, \Psi))$ leaders.
- All elected leaders run the algorithm $\mathcal{A}_{\mathrm{CV}}$ (see Section 4) to collect their views, for a total of $O((n/\mathsf{LMB}(\mathbf{ID}, \Psi)) \cdot n) = O\left(n^2/\mathsf{LMB}(\mathbf{ID}, \Psi)\right)$ messages. Then, the leaders communicate the collected views to all processors for a total of $O((n/\mathsf{LMB}(\mathbf{ID}, \Psi)) \cdot n) = O\left(n^2/\mathsf{LMB}(\mathbf{ID}, \Psi)\right)$ messages. Now, each processor locally derives its view, which it returns to top.

So, the message complexity of $\mathcal{A}$ is as claimed. □

Applications. For which sets of rings $\mathbf{ID}$ is the upper bound on message complexity from Theorem 4 low? We identify two such classes of sets.

Say that a set of rings $\mathbf{ID} \subseteq \Sigma^n$ is ***universal*** if Leader Election is solvable on $\mathbf{ID}$. Clearly, *every* (circularly symmetric) relation is solvable on a universal set of rings. Recall that Leader Election is both circularly symmetric and aperiodic. Hence, Theorem 3 implies that $\mathbf{ID}$ is aperiodic. Thus, for each $\langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(LE)$, $\mathsf{MB}(\mathbf{id}, \mathbf{in}) = n$. Hence, $\mathsf{LMB}(\mathbf{ID}, LE) = \mathsf{min}\{\mathsf{MB}(\mathbf{id}, \mathbf{in}) \mid \langle \mathbf{id}, \mathbf{in} \rangle \in \mathbf{ID} \times \mathsf{Dom}(LE)\} = n$. Theorem 4 now immediately implies:

Theorem 5 (Message Complexity on Universal Set of Rings). *A circularly symmetric relation is computable on a universal set of rings* $\mathbf{ID}$ *with* $O(n \cdot \lg n)$ *messages.*

Consider an arbitrary circularly symmetric relation Ψ. Consider a ring $\mathbf{id} \in \Sigma^n$ where each identity has multiplicity at most $\lg n$; call it a ***logarithmic*** ring. The corresponding set of rings $\mathbf{ID}$ will be called a ***logarithmic*** set of rings. Lemma 2 (condition (1)) implies that $\mathsf{T}(\widetilde{\mathbf{id}}) = n/\mathsf{GCD}(\mathsf{M}(\mathbf{ID})) \geq n/\lg n$. So, for each input vector $\mathbf{in} \in \mathsf{Dom}(\Psi)$, $\mathsf{MB}(\widetilde{\mathbf{id}}, \mathbf{in}) = \mathsf{LCM}(\mathsf{T}(\widetilde{\mathbf{id}}), \mathsf{T}(\mathbf{in})) \geq n/\lg n$. This implies that $\mathsf{LMB}(\mathbf{ID}, \Psi) \geq n/\lg n$. Since also $\mathsf{LMB}(\mathbf{ID}, \Psi) \leq n$, Theorem 4 immediately implies:

Theorem 6 (Message Complexity on Logarithmic Set of Rings). *A circularly symmetric relation is computable on a logarithmic set of rings* $\mathbf{ID}$ *with* $O(n \cdot \lg n)$ *messages.*

We can obtain additional upper bounds on the message complexity of computing a circularly symmetric relation by further generalizing Theorem 6. Towards this end, we generalize the definition of a logarithmic set of rings to a set of rings with an upper bound m on the multiplicity of each identity in any ring from the set. The corresponding upper bound on message complexity is $O(n \cdot \max\{m, \lg n\})$.

6 Epilogue

We presented a comprehensive study of solvability, computability and message complexity for the partially eponymous ring. Several interesting questions remain. For example, is there a *matching* lower bound to the universal upper bound on message complexity from Theorem 4? Can we characterize the class of sets of rings of size n for which this (universal) upper bound becomes $O(n \cdot \lg n)$? Finally, a challenging task is to extend our theory for the partially eponymous ring to other network architectures (such as *hypercubes* and *tori*).

References

1. D. Angluin, "Local and Global Properties in Networks of Processors," *Proceedings of the 12th Annual ACM Symposium on Theory of Computing,* pp. 82–93, 1980.
2. H. Attiya, M. Snir and M. Warmuth, "Computing on an Anonymous Ring," *Journal of the ACM,* Vol. 35, No. 4, pp. 845–875, 1988.
3. P. Boldi and S. Vigna, "An Effective Characterization of Computability in Anonymous Networks," *Proceedings of the 15th International Symposium on Distributed Computing,* Vol. 2180, pp. 33–47, LNCS, Springer-Verlag, 2001.
4. P. Boldi and S. Vigna, "Fibrations of Graphs," *Discrete Mathematics,* Vol. 243, pp. 21–66, 2002.
5. J. E. Burns, "A Formal Model for Message Passing Systems," Technical Report TR-91, Department of Computer Science, Indiana University, 1980.
6. J. Chalopin, S. Das and N. Santoro, "Groupings and Pairings in Anonymous Networks," *Proceedings of the 20th International Symposium on Distributed Computing,* Vol. 4167, pp. 105–119, LNCS, Springer-Verlag, 2006.
7. S. Dobrev and A. Pelc, "Leader Election in Rings with Nonunique Labels," *Fundamenta Informaticae,* Vol. 59, No. 4, pp. 333–347, 2004.
8. P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio and N. Santoro, "Sorting and Election in Anonymous Asynchronous Rings," *Journal of Parallel and Distributed Computing,* Vol. 64, No. 2, pp. 254–265, 2004.
9. D. Hirschberg and J. B. Sinclair, "Decentralized Extrema-Finding in Circular Configurations of Processes," *Communications of the ACM,* Vol. 23, No. 11, pp. 627–628, 1980.
10. F. T. Leighton, "Finite Common Coverings of Graphs," *Journal of Combinatorial Theory (Series B),* Vol. 33, pp. 231–238, 1982.
11. G. LeLann, "Distributed Systems - Towards a Formal Approach," *Information Processing 77,* Proceedings of the IFIP Congress, pp. 155–160, 1977.
12. M. Yamashita and T. Kameda, "Computing on Anonymous Networks, Part I: Characterizing the Solvable Cases," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 7, No. 1, pp. 69–89, 1996.
13. M. Yamashita and T. Kameda, "Computing on Anonymous Networks, Part II: Decision and Membership Problems," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 7, No. 1, pp. 90–96, 1996.
14. M. Yamashita and T. Kameda, "Computing Functions on Asynchronous Anonymous Networks," *Mathematical Systems Theory,* Vol. 29, No. 4, pp. 331–356, 1998. Erratum in *Theory of Computing Systems,* Vol. 31, No. 1, pp. 109, 1998.
15. M. Yamashita and T. Kameda, "Leader Election Problem on Networks in Which Processor Identity Numbers are not Distinct," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 10, No. 9, pp. 878–887, 1999.

Self-stabilizing Leader Election in Networks of Finite-State Anonymous Agents

Michael Fischer and Hong Jiang*

Department of Computer Science, Yale University

Abstract. This paper considers the self-stabilizing leader-election problem in a model of interacting anonymous finite-state agents. Leader election is a fundamental problem in distributed systems; many distributed problems are easily solved with the help of a central coordinator. Self-stabilizing algorithms do not require initialization in order to operate correctly and can recover from transient faults that obliterate all state information in the system. Anonymous finite-state agents model systems of identical simple computational nodes such as sensor networks and biological computers. Self-stabilizing leader election is easily shown to be impossible in such systems without additional structure.

An *eventual leader detector* $\Omega?$ is an oracle that eventually detects the presence or absence of a leader. With the help of $\Omega?$, uniform self-stabilizing leader election algorithms are presented for two natural classes of network graphs: complete graphs and rings. The first algorithm works under either a local or global fairness condition, whereas the second requires global fairness. With only local fairness, uniform self-stabilizing leader election in rings is impossible, even with the help of $\Omega?$.

Keywords: anonymous, failure detector, fairness, finite-state, impossibility result, leader election, population protocols, ring network, self-stabilization, sensor networks.

1 Introduction

Leader election is a fundamental problem in distributed systems. Many problems that are hard otherwise become easy to solve once a central coordinator is available. In reality, the availability and reliability of a leader both depend on a variety of factors: the feasibility to deploy or elect a leader, the possibility that an existing leader crashes, and the possibility that transient faults generate multiple leaders.

In many scenarios, a reasonable expectation is that when the network eventually becomes well-behaved and remains so, a leader is elected and remains reliable. This behavior is captured by failure detector Ω [1] also known as an *eventual leader elector*. With Ω, every process i has a local oracle $\texttt{leader}_i$. When invoked, $\texttt{leader}_i$ returns a process ID which process i considers to be its current leader. Ω guarantees that there is a time after which all processes have the same

* Supported by NSF grant ITR-0331548.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 395–409, 2006.

non-faulty leader. Ω is important because it was shown to be the weakest failure detector required to solve consensus in the conventional model of asynchronous distributed systems [1], and it has been used in many other algorithms.

Ω presupposes a model in which agents have unique process ID's. In this paper, we study a model of distributed systems called *population protocols* which was developed in a series of papers [2,3,4]. A network consists of an unbounded but finite population of identical anonymous finite-state agents. The protocols we present are uniform over natural classes of networks: They are independent of the network size, and the agents do not need to know the size of the network. The agents in our model are strongly anonymous. Not only do agents lack unique process ID's, but an agent cannot even determine whether two distinct messages are from the same process, nor can it direct two outgoing messages to the same recipient. By way of contrast, some related work on anonymous networks assumes an underlying port-to-port communication model in which processes are permanently assigned to ports, and a process sends and receives messages through distinguished ports. Such a model gives agents the ability to tell whether a set of messages come from different neighbors and to direct messages to the same or distinct neighbors, which is generally impossible in our model. Identical devices are easy to manufacture in large quantity. In addition, population protocols use $O(1)$ space per node, which is highly desirable in networks of memory-limited devices such as ad hoc mobile networks.

We introduce $\Omega?$, an analog of Ω appropriate to anonymous networks, which we call an *eventual leader detector*. Instead of electing a leader, $\Omega?$ simply reports to each agent a guess about whether or not one or more leaders are present in the network. The guess may be correct or not, and different inconsistent guesses may be reported to different agents. The only guarantee is that from some point onward in any infinite execution, if there is continuously a leader, or if there is continuously no leader, $\Omega?$ eventually accurately reports that fact to each agent.

Using $\Omega?$, we give uniform self-stabilizing leader election algorithms for fully connected networks (assuming local fairness) and for rings (assuming global fairness). We also show that uniform leader election is impossible in rings with only local fairness, even with the help of $\Omega?$. The different fairness conditions are defined in section 3.1

2 Related Work

A self-stabilizing algorithm does not depend on initialization of process states, and an execution of a self-stabilizing algorithm converges to a set of pre-defined stable configurations starting from any arbitrary configuration. The first self-stabilizing algorithms are introduced by Dijkstra[5]. Schneider[6] presents a survey on early research on self-stabilization.

Itkis, Lin, and Simon [7] present a deterministic constant-space self-stabilizing protocol for leader election on uniform bidirectional asynchronous rings of prime size. In their model, there is a central daemon that picks an enabled processor each time to make an atomic move. The chosen processor can read the

states of its two neighbors at the same time to determine its next state. Higham and Myers [8] give a randomized self-stabilizing algorithm that solves token circulation and leader election on anonymous, uniform, synchronous, and unidirectional rings of arbitrary but known size, in which each processor state and message has size in $O(\log n)$. Dolev, Israeli, and Moran [9] present a randomized self-stabilizing leader election protocol that tolerates addition or deletion of processors and links. Their protocol uses $O(\log n)$ bits per node. Beauquier, Gradinariu, and Johnen [10] present a silent and deterministic self-stabilizing leader-election protocol requiring constant memory space on unidirectional, ID-based rings where the ID values are bounded. They also prove that a non-constant lower bound on space is required by a (deterministic or randomized) self-stabilizing leader-election protocol on unidirectional anonymous rings under an unfair daemon. Based on the observation that in a stabilized system, a transient fault usually affect a small number of processes, Ghosh and Gupta [11] introduce a self-stabilizing leader-election algorithm that recovers quickly from small scale transient faults. Their algorithm assumes the existence of unique IDs. Fernández, Jiménez, and Raynal [12] present two eventual leader-election algorithms in networks where nodes have limited global information. Their algorithms are implementations of Ω in a hybrid model and require unique and totally-ordered IDs. Angluin et al. [4] present a non-uniform leader-election algorithm for rings in the population protocols model. They also show in the same paper that there does not exist a self-stabilizing leader-election protocol for general connected networks.

Chandra and Toueg [13] introduce the concept of unreliable failure detectors and study how they can be used to solve the asynchronous consensus problem with crash failures. In a related paper, Chandra, Hadzilacos, and Toueg [1] prove that one of the failure detectors in [13] is the weakest failure detector for solving asynchronous consensus with a majority of reliable processes. They also show that Ω is a weakest failure detector with which one can solve consensus. Aguilera et al. present an algorithm to implement Ω and to solve consensus in partially synchronous systems [14]. Ω can be implemented in a system with up to f process crashes, if there exists some correct process with f outgoing links that are eventually timely. The focus of these papers is the consensus problem, so the underlying network is assumed to be fully connected.

3 Model and Definitions

We introduce the population-protocol model to the extent required to present the results in this paper. A more detailed description is available in [4].

We represent a network by a directed graph $G = (V, E)$ with n vertices numbered 0 through $n-1$ and no multi-edges or self-loops. Each vertex represents a finite-state sensing device called an agent, and an edge (u, v) indicates the possibility of a communication between u and v in which u is the *initiator* and v is the *responder*. An "undirected" network refers to a communication graph in which edge (v, u) is present if and only if edge (u, v) is present. The number

associated with each node is used solely for the ease of description and is not known to the agents.

A *protocol* $P(Q, \mathcal{C}, X, Y, O, \delta)$ consists of a finite set of *states* Q, a set of *initial configurations* $\mathcal{C}$, a finite set X of *input symbols*, an *output function* $O : Q \to Y$, where Y is a finite set of *output symbols*, and a *transition function* δ mapping each element of $(Q \times X) \times (Q \times X)$ to a nonempty subset of $Q \times Q$. If $(p', q') \in \delta((p, x), (q, y))$, we call $((p, x), (q, y)) \to (p', q')$ a *transition*. A transition does not necessarily cause either of the nodes to change its state. The transition function, and the protocol, is *deterministic* if $\delta((p, x), (q, y))$ always contains just one pair of states. The inputs provide a way for a protocol to interact with an external entity, be it the environment, a user, or another protocol. In this paper, an agent i interacts with its leader detector through the input port.

A *configuration* is a mapping $C : V \to Q$ specifying the state of each device in the network, and an *input assignment* is a mapping $\alpha : V \to X$. A *trace* $T_G(Z)$ on a graph $G(V, E)$ is an infinite sequence of assignments from V to the symbol set Z: $T_G = \lambda_0, \lambda_1, \ldots$ where λ_i is an assignment from V to Z. The set Z is called the *alphabet* of T_G. If $Z = X$, then each λ_i is an input assignment, and we say T_G is an *input trace* of the protocol.

An *action* is a pair $\sigma = (r, e)$, where r is a transition $((p, x), (q, y)) \to (p', q')$ of δ and $e = (u, v)$ is an edge of G. Let C and C' be configurations, α be an input assignment, and u, v be distinct nodes. We say that σ is *enabled* in (C, α) if $C(u) = p$, $\alpha(u) = x$, $C(v) = q$, and $\alpha(v) = y$. We say that (C, α) *goes to* C' *via* σ, denoted $(C, \alpha) \xrightarrow{\sigma} C'$, if σ is enabled in (C, α), $C'(u) = p'$, $C'(v) = q'$, and $C'(w) = C(w)$ for all $w \in V - \{u, v\}$. In words, C' is the configuration that results from C by applying the transition rule r to the node pair e. Finally, we say that (C, α) *can go to* C' *in one step*, denoted $(C, \alpha) \to C'$, if $(C, \alpha) \xrightarrow{\sigma} C'$ for some action σ, and we say that σ is *taken* during that step. It is possible for more than one action to be taken during the same step.

Given an input trace $IT = \alpha_0, \alpha_1, \ldots$ we write $C \xrightarrow{*} C'$ if there is a sequence of configurations $C = C_0, C_1, \ldots, C_k = C'$, such that $(C_i, \alpha_i) \to C_{i+1}$ for all i, $0 \le i < k$, in which case we say that C' is *reachable* from C given input trace IT.

An *execution* is an infinite sequence of configurations and input assignments (C_0, α_0), (C_1, α_1), ... such that $C_0 \in \mathcal{C}$ and for each i, $(C_i, \alpha_i) \to C_{i+1}$. In the rest of this paper, all occurrences of "execution" refer to an infinite sequence as defined here. We extend the output function O to take a configuration C and produce an *output assignment* $O(C)$ defined by $O(C)(v) = O(C(v))$. Let $E = (C_0, \alpha_0)$, (C_1, α_1), ..., (C_i, α_i), ... be an execution of P. We define the *output trace* of an execution as $OT(E) = O(C_0), O(C_1), \ldots, O(C_i), \ldots$.

3.1 Fairness

We consider fairness conditions of different strengths. Let $E = (C_0, \alpha_0)$, (C_1, α_1), ..., $(C_i, \alpha_i), \ldots$ be an execution. The following conditions apply to E.

Strong global fairness. For every C, α, and C' such that $(C, \alpha) \to C'$, if $(C, \alpha) = (C_i, \alpha_i)$ for infinitely many i, then $(C_i, \alpha_i) = (C, \alpha)$ and $C_{i+1} = C'$

for infinitely many i. (Hence, the step $(C, \alpha) \to C'$ is taken infinitely many times in E.)

Strong local fairness. For every action σ, if σ is enabled in (C_i, α_i) for infinitely many i, then $(C_i, \alpha_i) \xrightarrow{\sigma} C_{i+1}$ for infinitely many i. (Hence, the action σ is taken infinitely many times in E.)

Global fairness asserts that each *step* $(C, \alpha) \to C'$ that can be taken infinitely often is actually taken infinitely often. By way of contrast, local fairness only asserts that each *action* σ that can be taken infinitely often is actually taken infinitely often. This differs from global fairness in the case of an action that is enabled infinitely often in more than one context. Global fairness would insist that it be taken infinitely often in all such contexts, whereas local fairness only requires that it occur infinitely often in one such context. For example, if σ is enabled in both (C_1, α_1) and (C_2, α_2), where $(C_1, \alpha_1) \neq (C_2, \alpha_2)$, an execution in which σ was never taken from (C_2, α_2) would not be globally fair, but it would be locally fair if σ were taken infinitely often from (C_1, α_1).

Theorem 1. *Global fairness implies local fairness.*

Proof. Suppose E satisfies strong global fairness. Because there are only finitely many distinct (C_i, α_i) pairs in E, if σ is enabled in (C_i, α_i) for infinitely many i, then σ is enabled in some particular (C, α) that occurs infinitely often in E. Let $(C, \alpha) \xrightarrow{\sigma} C'$. By strong global fairness, the step $(C, \alpha) \to C'$ is taken infinitely many times in E; hence, E satisfies strong local fairness.

These fairness definitions talk about certain steps that must be taken infinitely many times in E. For many purposes, it is immaterial whether a goal configuration C' is reached in one step or in many. This leads us to define corresponding weak fairness conditions.

Weak global fairness. For every C, α, and C' such that $(C, \alpha) \to C'$, if (C, α) occurs infinitely often in E, then C' occurs infinitely often in E.

Weak local fairness. For every action σ, if σ is enabled infinitely often in E, then there exist C, α, C' such that $(C, \alpha) \xrightarrow{\sigma} C'$, (C, α) occurs infinitely often in E, and C' occurs infinitely often in E.

The weak forms of fairness do not insist that particular steps occur infinitely often in E but only that the configurations that would result from those steps occur infinitely often. Thus, whereas strong fairness insists that a particular action occurs in a single step, weak fairness allows the configuration that would result from that action to be reached in many steps.

Obviously, the weak forms of fairness are implied by the corresponding strong forms. But the relationship between the weak and strong forms is even closer.

Theorem 2. *Every execution sequence that satisfies weak global (resp. local) fairness has an infinite subsequence that satisfies strong global (resp. local) fairness. Moreover, the sets of infinitely occurring pairs* (C, α) *are the same in both sequences.*

Proof (Sketch). The intuition is that if $(C_i, \alpha_i) \rightarrow C_j$ for $j > i$, then the segment $(C_{i+1}, \alpha_{i+1}), \ldots, (C_{j-1}, \alpha_{j-1})$ can be removed from E and the result is still an execution sequence. In the new sequence, (C_i, α) is adjacent to C_j, so $(C_i, \alpha_i) \rightarrow C_j$ occurs as a single step as required by strong fairness. Details are left to the full paper.

Discussion. A fair question to ask is, "Which is the 'right' definition of fairness?" In light of Theorem 2, it makes little difference whether one works with the strong or weak forms of fairness, for a weakly fair execution has embedded in it a corresponding strongly fair execution. In subsequent sections, we do not explicitly distinguish between the strong and weak versions of the fairness conditions when the difference is immaterial.

Whether global or local fairness is more realistic depends on how scheduling decisions are made. If the next step to take is chosen randomly, with each possible step having a non-zero probability of being chosen, then a globally fair execution will result with probability 1.

However, systems are often viewed as consisting of a collection of semi-autonomous components. The scheduler activates each component infinitely often, but the scheduling decision is not assumed to be independent of the states of the other components. Thus, component A might be permitted to execute when component B is in state 1 but not when it is in state 2. As long as A is given infinitely many chances to run, the scheduler would be considered to be fair, even though A never gets to run at a time when B is in state 2. For such a system, local fairness (in one of its many varieties) is the appropriate notion of fairness.

3.2 Behavior, Implementation and Self-stabilization

A self-stabilizing system can start at an arbitrary configuration and eventually exhibit "good" behavior. We define a *behavior* B on a network $G(V, E)$ to be a set of traces on G that have the same alphabet. We write $B(Z)$ to be explicit about the common alphabet Z. A behavior B is *constant* if every trace in B is constant. If the output trace of every fair execution of a protocol $P(Q, \mathcal{C}, X, Y, O, \delta)$ starting from any configuration in $\mathcal{C}$ is in some behavior $B_{\text{out}}(Y)$, we say P is an *implementation* of output behavior B_{out}. Given a behavior $B(Z)$, we define the corresponding *stable behavior* $B^s(Z)$: $T \in B^s$ if and only if Z is T's alphabet, and there exists $T' \in B$ such that T' is a suffix of T. Thus, an execution in a stable behavior may have a completely arbitrary finite prefix followed by an execution with the desired properties. If $P(Q, \mathcal{C}, X, Y, O, \delta)$ is an implementation of B^s, and $\mathcal{C}$ is the set of all possible configurations, we say that P is a *self-stabilizing implementation* of B.

The leader-election behavior LE on graph $G = (V, E)$ is the set of all constant traces $\beta, \beta, \ldots$ such that for some $v \in V$, $\beta(v) = L$ and for all $u \neq v$, $\beta(u) = N$. Informally, there is a static node with the leader mark L, and all other nodes have the non-leader mark N in every configuration.

3.3 Eventual Leader Detector $\Omega?$

A failure detector is a kind of oracle that provides some information to the system that it is unable to compute on its own, thereby extending the power of the system. Traditionally, failure detectors have been viewed as diagnostic devices that test nodes and inform the system when failures are detected, hence the name. However, failure detectors are a more general concept. In this paper, we use them to supply global semantic information about the protocol, namely, whether or not a leader is present in the system. Our failure detectors are weak in the sense that they do not respond immediately to the presence or absence of a leader but only after some indeterminate delay. Moreover, they do not report their findings to all nodes simultaneously, so some nodes might learn of the loss or gain of a leader before others do. Traditionally, failure detectors are modeled as local procedure calls in each process. In this paper we model a failure detector as a black box. Instead of each node invoking a local procedure, the failure detector feeds inputs to each node at each step.

The *eventual leader detector* $\Omega?$ supplies a Boolean input to each process at each step so that the following conditions are satisfied by every execution E:

1. If all but finitely many configurations of E lack a leader, then each process receives input `false` at all but finitely many steps.
2. If all but finitely many configurations of E contain one or more leaders, then each process receives input `true` at all but finitely many steps.

3.4 Implementation of $\Omega?$

The weak guarantees of $\Omega?$ allow it to be simply implemented in practice using timeouts. Each leader periodically propagates a "keep-alive" signal. Each agent keeps a timer and resets the timer whenever it receives a signal from a leader. On timeout, the agent sets the leader detector flag to `false` to indicate the absence of a leader. It sets the flag back to `true` whenever it receives a signal from a leader. In a good environment where the links are reliable and timely, each agent will eventually detect the absence or presence of leaders correctly. In an adverse environment where nodes malfunction and links may drop or unduly delay messages, the leader detector may give incorrect answers and the system may become unstable. (For example, multiple leaders may be generated.) However, eventually after the environment becomes good again, the leader detector will produce correct information and the system will become stable.

4 Leader Election in Complete Network Graphs

We give a simple leader-election algorithm for complete network graphs using a leader detector $\Omega?$. Each node has a memory slot that can hold either a leader mark "♟" or nothing "−" for a total of two states. Each node receives its current input `true` (T) or `false` (F) from $\Omega?$. A non-leader becomes a leader, when the leader detector signals the absence of a leader, and the responder is not a leader.

When two leaders interact, the responder becomes a non-leader. Otherwise, no state change occurs.

We formally describe the algorithm by pattern rules which are matched against the state and input of the initiator and responder, respectively. If the match succeeds, the states of the two interacting nodes are replaced by the respective states on the right side of the rule. In performing the match, "$*$" is a "don't care" symbol that always matches the slot or the input. On the right hand side, "$*$" specifies that the contents of the corresponding slot do not change. If no explicit rules match, a null transition in which neither node changes state is implied. Therefore every (configuration, input assignment) pair has an admissible successor.

Algorithm 1

Rule 1. $((♟, *), (♟, *)) \to ((♟), (-))$
Rule 2. $((-, \mathsf{F}), (-, *)) \to ((♟), (-))$
Rule 3. $((-, \mathsf{T}), (-, *)) \to ((-), (-))$

Each node outputs L when it holds a ♟, otherwise it outputs N.

To establish the correctness of a self-stabilizing algorithm, we define a notion of "safe configuration" and prove two things:

1. Starting from an arbitrary configuration, a safe configuration will eventually be reached.
2. Starting from an arbitrary *safe* configuration, the output traces of all possible executions have a suffix in the desired behavior.

For Algorithm 1, the desired behavior is the leader-election behavior LE, and the *safe configurations* are those in which at least one agent outputs L.

Lemma 1. *Let E be an execution of Algorithm 1 starting from an arbitrary configuration. Then E contains a safe configuration.*

Proof. Suppose no configuration of E is safe. This means there are no leaders in E, so from some point on, every node receives `false` from the leader detector. By rule 2, the initiator of the next interaction will declare itself a leader, a contradiction. Hence, E contains a safe configuration.

Lemma 2. *Let E be an infinite globally or locally fair execution of Algorithm 1 starting from an arbitrary safe configuration. Then the output trace of E has a suffix in LE.*

Proof. Notice that the only way for the number of leaders to decrease is via rule 1. The number of leaders decreases by one only when two leaders interact, so there is always at least one leader in subsequent configurations. Eventually all agents will receive `true` from the leader detector, after which new leaders cease being generated. By either local or global fairness, every two agents interact with

each other infinitely often, so eventually the number of leaders will decrease to one. The last leader cannot disappear, no new leaders are created, and the leader status cannot be transferred to another agent. Hence, the output trace of the suffix of E from this point on is in LE.

Theorem 3. *Given $\Omega?$, Algorithm 1 is a self-stabilizing implementation of the leader-election behavior LE that is correct under both global and local fairness.*

Proof. The correctness of Theorem 3 follows from Lemma 1 and Lemma 2.

5 Leader Election in Rings

The ring is an important network topology. The leader-election problem in rings has been extensively studied in the literature [4,7,8,10,11]. Most of those results assume local fairness or a similar fairness condition. It has been shown that there is no uniform[1] self-stabilizing leader-election algorithm in anonymous rings under the assumption of local fairness.

We refine these results in two ways. First, we show that uniform leader election in anonymous rings remains impossible under local fairness, even with the help of the leader detector $\Omega?$. Second, we exhibit a uniform self-stabilizing leader election algorithm using $\Omega?$ that works in rings of arbitrary size under the assumption of global fairness. We leave open the question of whether such an algorithm exists without the help of $\Omega?$.

5.1 Impossibility Under Local Fairness

Theorem 4. *No uniform leader-election algorithm exists in a ring assuming local fairness, even with the help of leader detector $\Omega?$*

Proof. Assume to the contrary that there is a uniform leader-election algorithm for rings that works under local fairness with the help of $\Omega?$. We consider the type of ring that is the most powerful in terms of computation: The ring is a directed cycle, so each node in an interaction knows whether it is talking to its next or previous neighbor in clockwise order around the ring.

For any $n \geq 2$, we look at two rings: R_1 has n nodes labeled $0, 1, \ldots, n-1$ and R_2 has $2n$ nodes labeled $0, 1, \ldots, 2n-1$. Given a locally fair execution E_1 of R_1, we describe a locally fair execution E_2 of R_2 such that all but finitely many configurations of E_2 have exactly two leaders. Hence, any algorithm with correct leader-election behavior on R_1 fails to have leader-election behavior on R_2, showing that there is no uniform leader-election algorithm.

Intuitively, we regard R_2 as two copies of R_1 spliced together into a single ring, as is shown in Figure 1. Each step of E_1 is applied separately to the two copies of R_1 in R_2. After every such pair of steps, the two copies of R_1 that comprise R_2 will be in the same configuration as each other and as R_1. Hence, if R_1 has one leader, then R_2 has two leaders.

[1] An algorithm is *uniform* if it works in rings of all sizes.

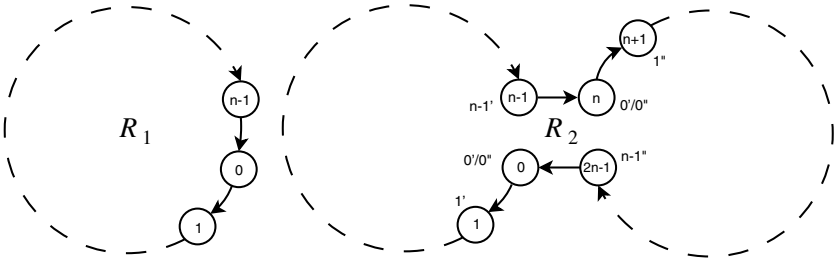


Fig. 1. The rings R_1 and R_2

Formally, node u of R_1 *corresponds* to the two nodes u and $u+n$ of R_2. An edge $e=(u,v)$ of R_1 *corresponds* to the two edges $e'=(u',v')$ and $e''=(u'',v'')$ of R_2 such that u corresponds to u' and u'' and v corresponds to v' and v''. For example, if $n=7$, then $(2,3)$ of R_1 corresponds to $(2,3)$ and $(9,10)$ of R_2, and $(6,0)$ of R_1 corresponds to $(6,7)$ and $(13,0)$ of R_2.[2] We say that configurations C of R_1 and D of R_2 are *compatible* if $C(u)=D(u)=D(u+n)$ for each $u=0,\ldots,n-1$, i.e., the states of corresponding nodes are the same.

Let $E_1=(C_0,\alpha_0),(C_1,\alpha_1),\ldots$ be a fair execution of R_1. Let D_0 be a configuration of R_2 that is compatible with C_0. We construct an execution

$$E_2=(D_0,\alpha_0),(D'_0,\alpha_0),(D_1,\alpha_1),(D'_1,\alpha_1),\ldots$$

of R_2 such that C_t and D_t are compatible for all t. We then argue that E_2 is locally fair and satisfies $\Omega?$, from which we derive a contradiction to the assumption that a uniform leader-election algorithm exists.

At each stage t of the construction, we assume that C_t and D_t are compatible. Let $\sigma_t=(r_t,e_t)$ be an action such that $(C_t,\alpha_t)\xrightarrow{\sigma_t}C_{t+1}$, where r_t is a transition of δ and $e_t=(u_t,v_t)$ is an edge of R_1. Let $\sigma'_t=(r,e')$ and $\sigma''_t=(r,e'')$ be actions, where $e'_t=(u'_t,v'_t)$ and $e''_t=(u''_t,v''_t)$ are the two edges of R_2 that correspond to e_t. Both σ'_t and σ''_t are enabled in D_t. This is because σ_t is enabled in C_t, and since C_t is compatible with D_t, $D_t(u'_t)=D_t(u''_t)=C_t(u_t)$ and $D_t(v'_t)=D_t(v''_t)=C_t(v_t)$.

Let D'_t be the unique configuration such that $(D_t,\alpha_t)\xrightarrow{\sigma'_t}D'_t$. Because $n\geq 2$, the nodes u',u'',v',v'' are all distinct, so the states of u'' and v'' are the same in D_t and in D'_t. Hence, σ''_t is also enabled in D'_t. Let D_{t+1} be the unique configuration such that $(D'_t,\alpha_t)\xrightarrow{\sigma''_t}D_{t+1}$. It is easily shown that C_{t+1} and D_{t+1} are compatible. By induction, C_t and D_t are compatible for all t.

It remains to show that E_2 is a locally fair execution of R_2. It is obviously an execution (since each configuration follows from the previous one by a legal action). We must argue that it is locally fair and that the inputs are consistent with $\Omega?$. Local fairness follows from the correspondence between steps of R_2

[2] Note that $(6,0)$ does not correspond to pairs $(6,0)$ and $(13,7)$ obtained by interchanging the second components since these latter pairs are not edges of R_2.

and R_1. If some action σ' of R_2 is infinitely often enabled in E_2, then the corresponding action σ of R_1 is infinitely often enabled in E_1. By local fairness of E_1, σ is taken infinitely often in E_1. By the above construction, σ' is taken infinitely often in E_2.

Finally, we argue that the inputs in E_2 satisfy the conditions for $\Omega?$. For each t, if configurations C_t and C_{t+1} both have leaders, then D_t, D'_t, and D_{t+1} all have leaders. Similarly, if C_t and C_{t+1} both lack leaders, then D_t, D'_t, and D_{t+1} all lack leaders. Hence, if all but finitely many configurations of E_1 lack leaders, then all but finitely many configurations of E_2 lack leaders, and similarly, if all but finitely many configurations of E_1 have leaders, then all but finitely many configurations of E_2 have leaders. Hence, the sequence of input assignments $\alpha_0, \alpha_0, \alpha_1, \alpha_1, \ldots$ in E_2 is correct for $\Omega?$.

It follows that E_2 is a locally fair execution of R_2 with leader detector $\Omega?$. However, the output trace of E_2 is not in LE since all but finitely many configurations of E_2 have two leaders. Thus, the assumed algorithm is not a uniform leader-election algorithm.

5.2 Leader Election Under Global Fairness

A non-uniform self-stabilizing leader-election algorithm assuming global fairness was given in [4]. Here we give a uniform algorithm with the help of $\Omega?$.

We assume that the ring is *directed*, which means each node has a sense of "forward" (clockwise) and "backward" (counter-clockwise), and every interaction takes place between the initiator and its forward neighbor. A self-stabilizing algorithm to direct an undirected ring was given in [4], so our algorithm can be applied to any weakly connected cycle, whether directed or not.

Each node can store zero or one of each of three kinds of tokens: a bullet "◀", a leader mark "♟", and a shield "▌", for a total of eight possible states. Corresponding to each kind of token is a *slot* which is *empty* if the corresponding token is not present, and *full* if it is present. An empty slot is denoted by "−"; a full slot is denoted by the corresponding token. The slots in each node are ordered with the bullet first, leader mark second, and shield third. Extending this to a clockwise ordering of all slots in the ring, the shield slot of one node is followed by the bullet slot of the next node in clockwise order.

Algorithm 2

Rule 1. $((*\,*\,*, \mathsf{F}), (*\,*\,*, *)) \to ((\blacktriangleleft\,♟\,▌), (*\,*\,*))$
Rule 2. $((*\,-\,▌, \mathsf{T}), (*\,*\,*, *)) \to ((*\,-\,-), (-\,*\,▌))$
Rule 3. $((*\,♟\,▌, \mathsf{T}), (*\,*\,*, *)) \to ((\blacktriangleleft\,♟\,-), (-\,*\,▌))$
Rule 4. $((*\,♟\,-, \mathsf{T}), (-\,*\,*, *)) \to ((\blacktriangleleft\,♟\,-), (-\,*\,*))$
Rule 5. $((*\,*\,-, \mathsf{T}), (\blacktriangleleft\,*\,*, *)) \to ((\blacktriangleleft\,-\,-), (-\,*\,*))$

Each node outputs L when it holds a ♟, otherwise it outputs N.

When two nodes interact and the initiator's input is `false` (F), a leader and a shield are created. At the same time, a bullet is fired (rule 1). This is the only

way for leaders and shields to be created. When the initiator's input is `true` (T), the following rules apply: Shields move forward around the ring (rules 2 and 3), and bullets move backward (rule 5). Bullets are absorbed by any shield they encounter (rules 2 and 3) but kill any leaders along the way (rule 5). If a bullet moves into a node already containing a bullet, the two bullets merge into one. Similarly, when two shields meet, they merge into one. A leader fires a bullet whenever it is the initiator of an interaction (rules 3 and 4).

A node i in a configuration is called a *protected leader*, and node j is called its *protecting shield*, if i has a leader mark, j has a shield, and all of the slots between i's leader mark and j's shield in clockwise order are empty. A node can be both a protected leader and its own protecting shield. We show that eventually there is exactly one protected leader, one protecting shield, and no unprotected leader.

Let E be an execution. Define S_E to be the maximal suffix of E such that every (configuration, input assignment) pair in S_E occurs infinitely often. S_E is well-defined and infinite since there are only finitely many distinct (configuration, input assignment) pairs. Define IRC_E[3] to be the set of configurations that occur in S_E.

The follows lemmas are all qualified by "for any execution E".

Lemma 3. *If any configuration in* IRC_E *has a protected leader, then every configuration in* IRC_E *has a protected leader.*

Proof. Let $C \in \mathrm{IRC}_E$ be a configuration with a protected leader, and suppose $(C, \alpha) \to C'$. We show that C' has a protected leader, regardless of which transition rule was applied.

- Rule 1 creates a new protected leader.
- Rule 2 moves the shield forward. If the responder is a leader, it becomes protected. If not, the protected leader is still protected after the move.
- Rule 3 fires a bullet and moves the shield forward. If the responder is a leader, it becomes protected. If not, the initiator remains a protected leader.
- Rule 4 fires a bullet. This does not affect the protected leader, for the bullet cannot be the only non-empty slot between the protected leader and its protecting shield.
- Rule 5 moves a bullet from the responder to the initiator. If the initiator is a leader, it is killed by the bullet. However, the initiator was not protected beforehand since there was no shield between the leader mark and the bullet token. This rule does not affect the protected status of any other leader.

Thus, every configuration in S_E following the first one having a protected leader also has a protected leader. Because every pair in S_E occurs infinitely often, all configurations in IRC_E have a protected leader.

Let α_{F} and α_{T} be input assignments such that α_{F} assigns `false` and α_{T} assigns `true` to every node.

[3] IRC stands for Infinitely Recurring Configurations.

Lemma 4. *If no configuration in* IRC_E *has a leader, then every input assignment in* S_E *is* α_{F}. *If every configuration in* IRC_E *has a leader, then every input assignment in* S_E *is* α_{T}.

Proof. Immediate from the definition of leader detector and the fact that every pair in S_E occurs infinitely often.

Lemma 5. *Every configuration in* IRC_E *has at least one leader.*

Proof. Suppose some configuration $C \in \mathrm{IRC}_E$ lacks a leader. There are two cases: If no configuration in IRC_E has a leader, then by Lemma 4, every input assignment in S_E is α_{F}, so every step in S_E is via rule 1. On the other hand, if some configuration $C' \in \mathrm{IRC}_E$ has a leader, then there is a sequence of steps in S_E that goes from (C, α) to (C', α') for some input assignments α and α' since both C and C' occur infinitely often in S_E. One of the steps must be via rule 1 since it is the only leader-creating rule. In either case, the application of rule 1 creates a configuration with a *protected* leader. Lemma 3 then implies that all configurations in IRC_E have protected leaders, contradicting the assumption that C lacks a leader.

Lemma 6. *Every input assignment in* S_E *is* α_{T}.

Proof. By Lemma 5, every configuration in IRC_E has a leader. The result follows from Lemma 4.

Lemma 7. *Suppose* $C \in \mathrm{IRC}_E$, $C = C_0$, C_1, ..., $C_r = C'$ *are configurations, and* $(C_i, \alpha_{\mathsf{T}}) \to C_{i+1}$, *for* $i = 0, \ldots, r-1$. *Then* $C' \in \mathrm{IRC}_E$.

Proof. An easy induction shows that each $C_i \in \mathrm{IRC}_E$. Suppose $C_i \in \mathrm{IRC}_E$. By Lemma 6, every pair in S_E has input assignment α_{T}. Since $(C_i, \alpha_{\mathsf{T}})$ occurs infinitely often in S_E, $C_{i+1} \in \mathrm{IRC}_E$ by global fairness.

Lemma 8. *Every configuration in* IRC_E *contains the same number of leaders and the same number of shields.*

Proof. By Lemma 6, every input assignment in S_E is α_{T}, so rule 1 is never applied in S_E. Therefore, no step can increase the number n of leaders or the number m of shields. But also, no step can decrease n or m since otherwise S_E would contain only finitely many configurations with n leaders or m shields, a contradiction to the definition of S_E. Hence, no step changes n or m, so all configurations have the same number of leaders and the same number of shields.

Lemma 9. *No configuration in* IRC_E *contains an unprotected leader.*

Proof. Suppose $C \in \mathrm{IRC}_E$ contains an unprotected leader. By Lemma 6, (C, α_{T}) occurs in S_E. From (C, α_{T}), there exists a finite sequence of steps to kill the unprotected leader by applying the rules 3, 4, and 5. By Lemma 7, the resulting configuration C' is in IRC_E. By Lemma 6, no step in S_E can create a new leader, so C' has fewer leaders than C. This contradicts Lemma 8.

Lemma 10. *Every configuration in* IRC_E *contains exactly one shield and exactly one leader.*

Proof. By Lemmas 5 and 9, every configuration in IRC_E contains at least one protected leader. This implies that each configuration contains at least one shield. There must be exactly one, for if any configuration has two or more shields, there exists a finite sequence of steps to merge two shields by applying rules 2 and 3 (possible by Lemma 6), resulting in a configuration C' with fewer shields. By Lemma 7, $C' \in \mathrm{IRC}_E$, contradicting Lemma 8. Finally, each shield is the protecting shield of at most one leader, so each configuration contains exactly one leader.

Theorem 5. *Given* $\Omega?$*, Algorithm 2 is a self-stabilizing implementation of the leader-election behavior* LE *in rings under global fairness.*

Proof. By Lemma 10, every configuration in IRC_E has exactly one leader. The *same* node is leader in every such configuration since none of the five rules can move the leader mark from one node to another in a single step. Hence, $OT(S_E) \in LE$, and Theorem 5 follows.

Algorithm 2 is at the same time a self-stabilizing token-circulation algorithm. After an execution stabilizes, there is exactly one shield moving around the ring, which could provide a token-circulation service.

6 Conclusion

We study the problem of self-stabilizing leader election in a model of finite-state anonymous agents. We consider this problem under two fairness conditions and with two interaction graph topologies. Our protocols utilize a leader detector $\Omega?$ that eventually correctly detects the presence or absence of a leader in the network. We show that the difficulty of leader election in the population-protocol model is due to the difficulty of detecting the presence and absence of leaders. It is an open problem for future research whether $\Omega?$ can be implemented in rings and other families of network graphs in the population-protocol model.

References

1. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **20**(4) (1996) 685 – 722
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Twenty-Third ACM Symposium on Principles of Distributed Computing. (2004) 290–299
3. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In Prasanna, V.K., Iyengar, S., Spirakis, P., Welsh, M., eds.: Distributed Computing in Sensor Systems: First IEEE International Conference, DCOSS 2005, Marina del Rey, CA, USE, June/July, 2005, Proceedings. Volume 3560 of Lecture Notes in Computer Science., Springer-Verlag (2005) 63–74

4. Angluin, D., Aspnes, J., Fischer, M.J., Jiang, H.: Self-stabilizing population protocols. In: Ninth International Conference on Principles of Distributed Systems. (2005) 79–90
5. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17**(11) (1974) 643–644
6. Schneider, M.: Self-stabilization. ACM Computing Surveys **25**(1) (1993) 45–67
7. Itkis, G., Lin, C., Simon, J.: Deterministic, constant space, self-stabilizing leader election on uniform rings. In: Workshop on Distributed Algorithms. (1995) 288–302
8. Higham, L., Myers, S.: Self-stabilizing token circulation on anonymous message passing rings. Technical report, University of Calgary (1999)
9. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 424–440
10. Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. In: Eighteenth ACM Symposium on Principles of Distributed Computing. (1999) 199–207
11. Ghosh, S., Gupta, A.: An exercise in fault-containment: Self-stabilizing leader election. Information Processing Letters (59) (1996) 281–288
12. Fernández, A., Jiménez, E., Raynal, M.: Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In: 2006 International Conference on Dependable Systems and Networks. (2006)
13. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
14. 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 ACM Symposium on Principles of Distributed Computing. (2004) 328–337

Robust Self-stabilizing Clustering Algorithm

Colette Johnen and Le Huy Nguyen

LRI–Université Paris Sud, CNRS UMR 8623
Bâtiment 490, F91405, Orsay Cedex, France
colette@lri.fr, lehuy@lri.fr

Abstract. Ad hoc networks consist of wireless hosts that communicate with each other in the absence of a fixed infrastructure. Such networks cannot rely on centralized and organized network management. The clustering problem consists in partitioning network nodes into groups called clusters, giving a hierarchical organization of the network. A self-stabilizing algorithm, regardless of the initial system configuration, converges to legitimates configurations without external intervention. Due to this property, self-stabilizing algorithms tolerate transient faults.

In this paper we present a robust self-stabilizing clustering algorithm for ad hoc network. The robustness property guarantees that, starting from an arbitrary configuration, in one round, network is partitioned into clusters. After that, the network stays partitioned during the convergence phase toward a legitimate configuration where the clusters partition ensures that any neighborhood has at most k clusterheads (k is a given parameter).

Keywords: Self-stabilization, Distributed algorithm, Clustering, Ad hoc networking.

1 Introduction

An *ad hoc* network is a self-organized network, especially those with wireless or temporary plug-in connections. Such a network may operate in a standalone fashion, or may be connected to the larger Internet [1]. In these networks, mobile routers may move arbitrary often; thus, the network's topology may change rapidly and unpredictably. Ad hoc networks cannot rely on centralized and organized network management. Significant examples include establishing survivable, efficient, dynamic communication for emergency/rescue operations, disaster relief efforts, and military networks. Meetings where participants aim at creating a temporary wireless ad hoc network is another typical example. Quick deployment is needed in these situations.

Clustering means partitioning network nodes into groups called clusters, providing the network with a hierarchical organization. A cluster is a connected subgraph of the global networks composed of a clusterhead and ordinary nodes. Each node belongs to only one cluster. In addition, a cluster is required to obey to certain constraints that are used for network management, routing methods, resource allocation, etc. By dividing the network into non-overlapped clusters,

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 410–424, 2006.

intra-cluster routing is administered by the clusterhead and inter-cluster routing can be achieved in a reactive manner between clusterheads. Clustering-based routing reduces the amount of routing information propagated in the network. Clustering facilitates the reuse of resources, which improves the system capacity. Members of a cluster can share resources such as software, memory space, printer, etc. Moreover, clustering can be used to reduce the amount of information that is used to store the network state. Distant nodes outside of a cluster usually do not need to know the details of specific events occurring inside this cluster. Indeed, an overview of the cluster's state is generally sufficient for those distant nodes to make control decisions. Thus, the clusterhead is typically in charge of collecting the state of nodes in its cluster and constructing an overview of its cluster state.

For the above mentioned reasons, it is not surprising that several distributed clustering algorithms have been proposed during the last ten years [2,3,4,5,6,7,8]. The clustering algorithms in [2,6] construct a spanning tree. Then the clusters are constructed on top of the spanning tree. The clusterheads set do not necessarily form a dominating set (i.e., a node can be at distance greater than 1 from its clusterhead). Two network architectures for MANET (Mobile Ad hoc Wireless Network) are proposed in [7,8] where nodes are organized into clusters. The clusterheads form an independent set (i.e., clusterheads are not neighbors) and a dominating set. The clusterheads are selected according to the value of their IDs. In [5], a weight-based distributed clustering algorithm taking into account several parameters (node's degree, transmission and battery power, node mobility) is presented. In a neighborhood, the selected nodes are those that are the most suitable for the clusterhead role (i.e., a node optimizing all the parameters). In [4], a Distributed and Mobility-Adaptive Clustering algorithm, called DMAC, is presented. The clusterheads are selected according to a node's parameter (called *weight*). The higher is the weight of a node, the more suitable this node is for the role of clusterhead. An extended version of this algorithm, called Generalized DMAC (GDMAC), is proposed in [3]. In this latter algorithm, the clusterheads do not have to form an independent set. This implies that, when, due to the mobility of the nodes, two or more clusterheads become neighbors, none has to resign. Thus, in highly mobile environment the clustering management with GDMAC requires less overhead than the clustering management with DMAC. A self-stabilizing version of DMAC and GDMAC is presented in [9].

A system is self-stabilizing when regardless of its initial configuration, it is guaranteed to reach a legitimate configuration in a finite number of steps. A system which is not self-stabilizing may stay in an illegitimate configuration forever. The correctness of self-stabilizing algorithms does not depend on initialization of variables, and a self-stabilizing algorithm converges to some predefined stable configuration starting from an arbitrary initial one. Self-stabilizing algorithms are thus inherently tolerant to transient faults in the system. Many self-stabilizing algorithms can also adapt dynamically to changes in the network topology or system parameters (e.g., communication speed, number of nodes). A new configuration resulting from a topological changes is viewed as an

inconsistent configuration from which the system will converge to a configuration consistent with the new topology. [10] presents a self-stabilizing algorithm that constructs a maximal independent set (i.e., members of the set are not neighbors, and the set is maximal to this property). Note that a maximal independent set is a good candidate for the clusterheads set because a maximal independent set is also a dominating set (i.e., any node is member of the dominating set or has a neighbor that is member of the set). In [11], a self-stabilizing algorithm that creates a minimal dominating set (i.e., if any member of the set leaves the set, the set is no more a dominating set) is presented. Note that a minimal dominating set is not necessarily an independent set. Several self-stabilizing algorithms for cluster formation and clusterhead selection have been proposed [9,12,13,14]. In [12], a self-stabilizing link-cluster algorithm under an asynchronous message-passing system model is presented (no convergence proofs are presented). The definition of cluster is not exactly the same as ours: an ordinary node can be at distance two of its clusterhead. The presented clustering algorithm requires three types of messages, our algorithms adapted to message passing model require one type of message. A self-stabilizing algorithm for cluster formation is presented in [13]. A density criteria (defined in [14]) is used to select clusterhead: a node v chooses in its neighborhood the node having the highest density. A v's neighborhood contains all nodes at distance less or equal to 2 from v. Therefore, to choose clusterhead, communication at distance 2 is required. Our algorithms build clusters on local information; thus it requires only communication between nodes at distance 1 of each others.

In this paper, we present a robust and self-stabilizing version of GDMAC. The obtained clusters satisfy the "ad hoc clustering properties":

> (1) each node is at most at distance 1 from the clusterhead of its cluster.
>
> (2) in a neighborhood there are at most k clusterheads (k being a given parameter).
>
> (3) the clusterhead of a node is nearly the best choice: its clusterhead was a nearly optimal weight.

These properties are formally defined in section 3. Starting from an arbitrary configuration, the system satisfies the safety predicate in one synchronous computation step. Once the system satisfies the safety predicate, the system performs correctly its task (i.e., the network is partitioned into clusters). The partition may have to change to get a partition satisfying the ad hoc clustering properties. During the construction of the final clusters the safety predicate stay verified: the networks is always partitioned. That is why we call this algorithm robust. The algorithm in [9] is not robust: a node may not belong to a cluster during the stabilization phase even if it belongs initially to a well-formed cluster. In [15], a robust self-stabilizing version of DMAC is presented under the synchronous schedule.

Our algorithm is designed for the state model. Nevertheless, it can be easily transformed into an algorithm for the message-passing model. For this purpose, each node v periodically broadcasts to its neighbors a message containing its

state. Based on this message, v's neighbors decide to update or not their variables. After a change in the value of v's state, node v broadcasts to its neighbors its new state.

The paper is organized as follows. In section 2, the formal definition of self-stabilization is presented. The clustering problem is discussed in the section 3. A robust version of [9] is described in section 4. The self-stabilization proof is presented in section 5. Section 6 discusses about the robustness of our algorithm. Finally, the time complexity is analyzed in section 7.

2 Model

We model a distributed system by an undirected graph $G = (V, E)$ in which V, is the set of nodes and there is an edge $\{u, v\} \in E$ if and only if u and v can communicate u and v are said neighbors. The set of neighbors of a node $v \in V$ will be denoted by N_v. Every node v in the network is assigned an unique identifier (ID). For simplicity, here we identify each node with its ID and we denote both with v. We assume the locally shared memory model of communication. Thus, each node i has a finite set of *local variables* such that the variables at a node i can be read by i and any neighbors of i, but can be only modified by i. The nodes execute their programs - code asynchronously. We assume that the code of each node i consists of a finite set of guarded statements of the form $Rule : Guard \rightarrow Action$, where $Guard$ is a boolean predicate involving the local variables of i and the local variables of its neighbors, and $Action$ is an assignment that modifies the local variables in i. A *rule* is executed by node p only if the guard rule evaluates to true, in which case we say the node p is enabled. The *state* of a node is defined by the values of its local variables. A *configuration* of a distributed system G is an instance of the node states. A computation e of a system G is a sequence of configurations $c_1, c_2, ...$ such that for $i = 1, 2, ...$, the configuration c_{i+1} is reached from c_i by a single step of one or several enabled nodes. A computation is *fair* if any node in G that is continuously enabled along the computation, will eventually perform an action. Let $\mathcal{C}$ be the set of possible configurations and $\mathcal{E}$ be the set of all possible computations of a system G. The set of computations of G starting with the particular *initial configuration* $c \in \mathcal{C}$ will be denoted $\mathcal{E}_c$. The set of computations of $\mathcal{E}$ whose initial configurations are all elements of $B \in \mathcal{C}$ is denoted as $\mathcal{E}_B$.

In this paper, we use the notion *attractor* [16] to define self-stabilization.

Definition 1. *(**Attractor**). Let B_1 and B_2 be subsets of $\mathcal{C}$. Then B_1 is an attractor for B_2 if and only if:*

1. $\forall e \in \mathcal{E}_{B_2}, (e = c_1, c_2, ...), \exists i \geq 1 : c_i \in B_1$ *(convergence).*

2. $\forall e \in \mathcal{E}_{B_1}, (e = c_1, c_2, ...), \forall i \geq 1, c_i \in B_1$ *(closure).*

The set of configurations matching the specification of problems is called the set of *legitimate* configurations, denoted as $\mathcal{L}$. $\mathcal{C} \backslash \mathcal{L}$ denotes the set of illegitimate configurations.

Definition 2. *(**Self-stabilization**). A distributed system S is called self-stabilizing if and only if there exists a non-empty set $\mathcal{L} \subseteq \mathcal{C}$ such that the following conditions hold:*

1. *$\mathcal{L}$ is an attractor for $\mathcal{C}$.*

2. *$\forall e \in \mathcal{E}_{\mathcal{L}}, e$ verifies the specification problem.*

One motivation for our robust stabilization is that a system should react gracefully to the input changes - preserving a safety predicate in the presence of the input changes. The safety predicate is chosen to ensure that the system still perform correctly its task during the period of convergence. A self-stabilizing protocol is robust with respect to input changes, if starting from a legitimate configuration followed by input changes, the safety predicate holds continuously until the protocol converges to a legitimate configuration.

Definition 3. *(**Robustness under Input Change [16]**). Let $\mathcal{SP}$ be a predicate on configurations called safety predicate, let $\mathcal{IC}$ be a set of input changes in the system. A self-stabilizing distributed system $\mathcal{S}$ is robust under $\mathcal{IC}$ if and only if a set of configurations satisfying $\mathcal{SP}$ (i) is closed, and (ii) is closed under any input change of $\mathcal{IC}$.*

3 Clustering for Ad Hoc Network

Clustering an ad hoc network means partitioning its nodes into *clusters*, each one with a *clusterhead* and some *ordinary nodes*. In order to meet the requirements imposed by the wireless, mobile nature of these networks, nodes in the same cluster has to be at distance at most 1 of their clusterhead. Thus, the following *clustering property* has to be satisfied:

1. Every ordinary node has at least a clusterhead as neighbor (*dominance* property).

We consider weighted networks, i.e., a weight w_v is assigned to each node $v \in V$ of the network. In ad hoc networks, amount of bandwidth, memory space or battery power of a node could be used to determine weight values. For simplicity, in this paper we assume that each node has a different weight. Note that if several nodes have the same weight, one may use the couple $(weight, ID)$ to give distinct 'weights' to each node. The choice of the clusterheads is based on the *weight* associated to each node: the higher the weight of a node, the better this node is suitable to be a clusterhead.

Assume that the clusterheads are bound to never be neighbors. This implies that, when due to the mobility of the nodes two or more clusterheads become neighbors, those with the smaller weights have to *resign* and affiliate with the now higher neighboring clusterhead. Furthermore, when a clusterhead v becomes the neighbor of an ordinary node u whose current clusterhead has weight smaller than v's weight, u has to affiliate with (i.e., *switch* to the cluster of) v. These "resignation" and "switching" processes due to node's mobility are a consistent

part of the clustering management overhead that should be minimized in ad hoc network where the topology changes fairly often. To overcome the above limitations, in [3] Basagni introduced a generalization of the previous clustering property called *Ad hoc clustering properties* defined as follow:

> **1.** Every ordinary node always affiliates with a neighbor that is clusterhead and which has higher weight than the weight of the ordinary node (*affiliation* condition).
>
> **2.** For every ordinary node v, for every clusterhead $z \in N_v : w_z \leq w_{Clusterhead_v} + h$ (*clusterhead* condition).
>
> **3.** A clusterhead has at most k neighboring clusterheads (k being an integer, $0 \leq k < n$) (*k-neighborhood* condition).

The first requirement ensures that each ordinary node has direct access to at least one clusterhead (the one of the cluster to which it belongs), thus allowing fast intra and inter cluster communications. The second requirement guarantees that each ordinary node always stays with a clusterhead that gives it a "good" service. By varying the threshold parameter h it is possible to reduce the switching overhead associated to the passage of an ordinary node from its current clusterhead to a new neighboring one when it is not necessary. When $h = 0$ we simply obtain that each ordinary node affiliates with the neighboring clusterhead with the highest weight. Finally, the third requirement allows us to have up to k neighboring clusterheads, $0 \leq k < n$. When $k = 0$ we obtain that two clusterhead can not be neighbors.

Safety property for clustering algorithm. The safety property has to ensure that the network is partitioned into clusters and each cluster has a leader that performs clusterhead tasks. In a clustered network, the role of clusterhead is to act as a local coordinator within a cluster, performing information aggregation and exchange to neighboring clusters.

4 Robust Self-stabilizing Clustering Algorithm

In this section, we present a clustering algorithm (variables, predicates and rules are formally presented in Algorithm 1). This algorithm is self-stabilizing and robust to the input changes. Even during the stabilization phase, it is desired that network is correctly partitioned, i.e., each node belongs to only a cluster. This property, called "safety", guarantees functioning of the applications using the hierarchical structure established by Algorithm 1, because each node belongs to a cluster.

A node has 3 possible states. It can be a truly clusterhead, in this case its Ch value is T. It can be an ordinary node, in this case its Ch value F. Otherwise, it can be a nearly ordinary node, in this case its Ch value is NF.

After the R_1 action, v is a truly clusterhead. After the R_2 action, v is an ordinary node. After the R_3 action, v is a nearly ordinary node.

A truly clusterhead v checks the number of its neighbors that are clusterheads. If this number is lesser or equal to k then SR_v should have the value 0 (R_4 action).

Constants

$w_v : \mathbb{N}$; // the weight of node v

Local variables of node v

Ch_v: {T, F, NF}; // indicates the role of node v

$Clusterhead_v : IDs$; // the clusterhead of node v

$SR_v : \mathbb{N}$;// the highest weight which violates the 3^{th} condition in v's neighbor

Macros

$N_v^+ = \{z \in N_v : (Ch_z = T) \wedge (w_z > w_v)\}$; // the set of v's neighboring clusterhead that has higher weight than v's weight

$Cl_v = |N_v^+|$; // the number of v's neighboring clusterhead that has higher weight than v's weight

Predicates

$\mathbf{G_1}(v) = \mathbf{G_{11}}(v) \vee \mathbf{G_{12}}(v)$

$\mathbf{G_{11}}(v) \equiv (Ch_v \neq T) \wedge (N_v^+ = \emptyset)$

$\mathbf{G_{12}}(v) \equiv (Ch_v = T) \wedge (Clusterhead_v \neq v) \wedge (\forall z \in N_v^+ : w_v > SR_z) \wedge (Cl_v \leq k)$

$\mathbf{G_2}(v) = \mathbf{G_{21}}(v) \vee \mathbf{G_{22}}(v)$

$\mathbf{G_{21}}(v) \equiv (Ch_v = F) \wedge \{(\exists z \in N_v^+ : w_z > w_{Clusterhead_v} + h) \vee (Clusterhead_v \notin N_v^+)\}$

$\mathbf{G_{22}}(v) \equiv (Ch_v = NF) \wedge \{(\forall z \in N_v : Clusterhead_z \neq v) \wedge (N_v^+ \neq \emptyset)$

$\mathbf{G_3}(v) = \mathbf{G_{31}}(v) \vee \mathbf{G_{32}}(v)$

$\mathbf{G_{31}}(v) \equiv (Ch_v = T) \wedge \{(\exists z \in N_v^+ : (w_v \leq SR_z)) \vee (Cl_v > k)\}$

$\mathbf{G_{32}}(v) \equiv (Ch_v = NF) \wedge (Clusterhead_v \neq v)$

$\mathbf{G_4}(v) \equiv (Ch_v \neq T) \wedge (SR_v \neq 0)$

$\mathbf{G_5}(v) \equiv (Ch_v = T) \wedge (SR_v \neq max(0,\ k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\}))$

Rules

$\mathbf{R_1}(v) : \mathbf{G_1}(v) \rightarrow Ch_v := T;\ Clusterhead_v := v;$
$SR_v := max(0,\ k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\})$

$\mathbf{R_2}(v) : \mathbf{G_2}(v) \rightarrow Ch_v := F;\ Clusterhead_v := max_{w_z}\{z \in N_v^+\};\ SR_v := 0$

$\mathbf{R_3}(v) : \mathbf{G_3}(v) \rightarrow Ch_v := NF;\ Clusterhead_v = v;\ SR_v := 0$

// update the value of SR_v

$\mathbf{R_4}(v) : (\neg\mathbf{G_1}(v) \wedge \neg\mathbf{G_2}(v) \wedge \neg\mathbf{G_3}(v)) \wedge \mathbf{G_4}(v) \rightarrow SR_v := 0$

$\mathbf{R_5}(v) : (\neg\mathbf{G_1}(v) \wedge \neg\mathbf{G_2}(v) \wedge \neg\mathbf{G_3}(v)) \wedge \mathbf{G_5}(v) \rightarrow$
$SR_v := max(0,\ k + 1^{\text{th}}\{w_z : z \in N_v \wedge (Ch_z = T)\})$

Algorithm 1. Robust Self-stabilizing Clustering Algorithm

If this number is greater than k, then the clusterhead sets up the value of SR_v to the weight of the first neighbor clusterhead having to resign, the one having the $(k+1)$th highest weight (R_5 action). All clusterheads having smaller and equal weight than SR_v will have to resign to ensure k-neighborhood condition. SR_v value of an ordinary node is 0 or R_4 is enabled.

A truly clusterhead ($Ch_v = T$) has to resign its role iff it violates the k-neighborhood condition. A clusterhead v having to resign takes the nearly ordinary state ($Ch_v = NF$) - it performs R_3 action. v stays in this nearly ordinary state until all of nodes in its cluster have joined another cluster.

A node v that has the state "nearly ordinary" is requiring that the members of its cluster join another cluster. Thus, the members of v's cluster are enabled (G_{11} or G_{21} predicate is verified), till v is nearly ordinary. As the scheduler is fair, the members of v's cluster will perform the rule R_1 or R_2. Thus, they will leave the v's cluster. Eventually v's cluster contains one member: v. ($\forall z \in N_v : Clusterhead_z \neq v$). After that time, v will become an ordinary node (rule R_2) if v has at least a neighbor clusterhead whose weight is higher than v's weight. Otherwise, v will become a clusterhead (rule R_1).

Due to an incorrect initial configuration, a node v may have to correct the value of $Clusterhead_v$ and/or SR_v. In this case it verifies one of the following predicates: G_{12}, G_{32}, G_4.

The safety predicate $\mathcal{SP}$ is defined as follow:

$$\mathcal{SP} \equiv \forall v \in V : (Clusterhead_v \in N_v \cup \{v\}) \wedge (Ch_{Clusterhead_v} \neq F).$$

$\mathcal{SP}$ predicate ensures that (i) each node belongs to a cluster and that (ii) each cluster has a clusterhead that performs its tasks correctly. Because a nearly ordinary node and truly clusterhead node acts as a clusterhead. Thus, the hierarchical structure exists if the $\mathcal{SP}$ is verified.

We denote z the clusterhead of node v. The safety predicate $\mathcal{SP}$ ensures that z is a neighbor of v and z is not an ordinary node. Thus, the safety predicate $\mathcal{SP}$ is only violated in cases of a z's removal (or a crash of z), a failure of link between v and z. Therefore, the safety predicate $\mathcal{SP}$ is preserved in the following input changes:

1. Change of node's weight (illustrated in Figure 1).

2. Crash of ordinary nodes.

3. Joining of subnetworks that verify $\mathcal{SP}$.

4. Failures of link between two ordinary nodes or between two clusterhead nodes.

After proving that $\mathcal{SP}$ is closed we conclude that Algorithm 1 is robust under input changes presented above.

Algorithm 1 is illustrated in Figure 1, in this example, $k = 1$. Initially, node 5 has 2 clusterheads in its neighborhood. It assigns its SR to 9. 9 is the weight of the first clusterhead which violates the 1-neighborhood condition in node 5's

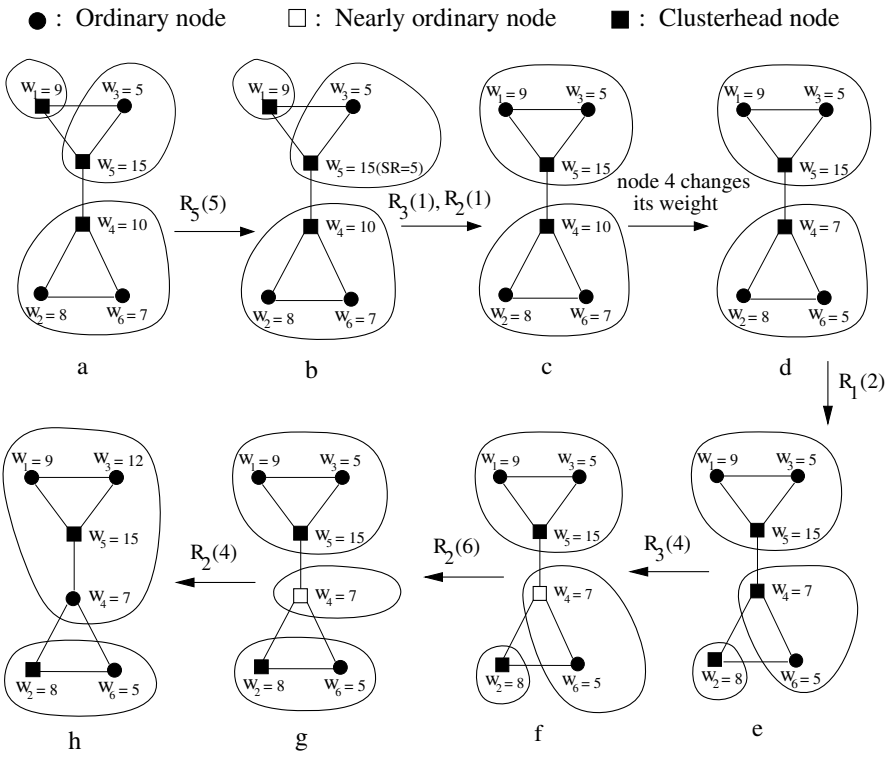


Fig. 1. Convergence to a legitimate configuration in the case k = 1, h = 0

neighborhood (Figure 1.b). Node 1 does not stay clusterhead because $SR_5 \geq w_1$: node 1 resigns to nearly ordinary state. No node has chosen node 1 as clusterhead (i.e., node is in the cluster led by node 1). Thus, during the next computation step, node 1 can join the cluster led by node 5 (Figure 1.c). Due to the change of the weight of node 4 (Figure 1.d), node 2 cannot stay ordinary : all clusterheads in the node 2's neighborhood have a weight that is smaller than node 2's weight. Thus, node 2 becomes clusterhead (Figure 1.e). Node 4 resigns to nearly ordinary state (Figure 1.f). It cannot keep the state 'truly clusterhead', because it violates the 1-neighborhood condition: there are two clusterheads in its neighborhood which have a higher weight than its weight (node 2 and 5). Node 6 does not verify the affiliation condition ($Ch_{clusterhead_6} = Ch_4 = NF$). Node 6 switches of cluster, it goes int eh cluster led by node 2 (Figure 1.g). After that, node 4 can take the state 'ordinary' and stop to behave as a clusterhead. Node 4 joins the cluster led by 5 (Figure 1.h). The network is stabilized. During the convergence phase, the safety property $\mathcal{SP}$ is always verified: at any time the network is partitioned into cluster, and each cluster has a leader ready to do the leadership tasks (i.e., a leader which has the state truly clusterhead or nearly ordinary).

5 Proofs of self-stabilization

5.1 Proof of Convergence

We first prove that the system reaches a terminal configuration.

Lemma 1. *Let v be a node. The value of Ch_v cannot be NF forever.*

Proof: We prove by contradiction. Assume that $Ch_v = NF$ is verified forever. Assume that there is a node $u \in N_v$ such that $Clusterhead_u = v$.

Case 1. $Ch_u = F$. Since $Ch_v = NF$ then $Clusterhead_u \notin N_u^+$ (see the definition of N_u^+). Thus, $G_{21}(u)$ is verified. As all computations are fair, u will perform R_2. After doing R_2, $Clusterhead_u \neq v$ is verified forever.

Case 2. $Ch_u = T$. Since $Clusterhead_u = v \neq u$. Thus, $G_{12}(u)$ or $G_{31}(u)$ is verified. As all computations are fair, u will perform R_1 or R_3. After doing R_1 or R_3, $Clusterhead_u \neq v$ is verified forever.

Case 3. $Ch_u = NF$. Since $Clusterhead_u = v \neq u$. Thus, $G_{32}(u)$ is verified. As all computations are fair, u will perform R_3. After doing R_3, $Clusterhead_u \neq v$ is verified forever.

Therefore, $\forall u \in N_v,\ Clusterhead_u \neq v$ is verified. Thus, $G_{11}(v)$ or $G_{22}(v)$ is verified. As all computations are fair, v will perform R_1 or R_2. After doing R_1 or R_2, $Ch_v = NF$ is not verified. That is a contrary. □

Lemma 2. *$A_1 = \{C \mid \forall v : (G_{12}(v) = F) \wedge (G_{32}(v) = F)\}$ is an attractor.*

Proof: If v verifies predicate G_{12} (resp G_{32}) then v is enabled and will stay enabled up to the time where v performs R_1 (resp R_3). As all computations are fair, v eventually performs R_1 (resp R_3). After that G_{12} (resp G_{32}) is never verified. □

Lemma 3. *In A_1, once v had performed the rule R_1, v does not perform R_1, R_2 or R_3 unless a node u such that $w_u > w_v$ had performed R_1.*

Proof: In A_1, $G_{12}(v)$ and $G_{32}(v) = F$ is never true.
Once v had performed the rule R_1, we have that $Ch_v = T$ and $Clusterhead_v = v$. Thus, the next rule performed by v will be R_3.
Before doing R_1, $G_{11}(v)$ is verified, we have $N_v^+ = \emptyset$. At time where v performs R_3, $G_{31}(v)$ is verified, implies that $N_v^+ \neq \emptyset$. Thus in meantime, a node $u \in N_v$, $w_u > w_v$ performed the rule R_1. □

Lemma 4. *In A_1, once v had performed the rule R_2, v does not perform R_1, R_2 or R_3 unless a node u such that $w_u > w_v$ had performed a rule R_1 or R_3.*

Proof: Once v had performed the rule R_2, we have that $Ch_v = F$ and $Cluster\ head_v := max_{w_z}\{z \in N_v^+\}$. Denote u the clusterhead of v, we have $u \in N_v^+$ and $w_u = max_{w_z}\{z \in N_v^+\} > w_v$. Next time that v will perform a rule, $G_{11}(v)$ **or** $G_{21}(v)$ is verified.

Case 1. $G_{11}(v)$ is verified. At time where v performs R_1, $N_v^+ = \emptyset$, implies that u performed the rule R_3 in meantime.

Case 2. $G_{21}(v)$ is verified. We have $(\exists z \in N_v^+ : w_z > w_u + h) \vee (u \notin N_v^+)$, implies that in meantime u performed R_3 or a node $z \in N_v$ such that $w_z > w_u + h > w_u$ performed R_1. □

Corollary 1. *In A_1, once v had performed the rule R_3, v will certainly perform R_1 or R_2.*

Lemma 5. *Every fair computation e that starts in A_1 has a suffix where in any reached configuration $\forall v \in V : (G_i(v) = F),\ i = \{1..3\}$.*

Proof: We will prove by contradiction. Assume that e has not a suffix in which $\forall v \in V : (G_i(v) = F),\ i = \{1..3\}$. A node cannot verify forever $G_1(v) \vee G_2(v) \vee G_3(v)$ (this node would be enabled forever and never performs a rule). Thus along a maximal computation there is a node v that infinitely often verifies $G_1(v)$, $G_2(v)$ or $G_3(v)$ and also infinitely often does not verify $G_1(v)$, $G_2(v)$ and $G_3(v)$. Meaning that v executes infinitely often R_1, R_2 or R_3. Following Corollary 1, if v executes infinitely often R_3 then v executes also infinitely often R_1 or R_2. Following Lemma 2, 3 and 4, once v have performed a rule R_1, R_2 or R_3, it will perform R_1, R_2 or R_3 again if there exists a node u ($w_u > w_v$) that performs R_1, R_2 or $R_3(u)$. Since the set of nodes is finite, then v performs R_1, R_2 or R_3 infinitely often only if there exists a node u ($w_u > w_v$) that performs R_1, R_2 or R_3 infinite many times. Using a similar argument we have a infinite sequence of nodes having increasing weight that performs R_1, R_2 or R_3 infinitely often. Since the number of nodes is finite, this is a contrary. Hence our hypothesis is false, and for every node v, $G_i(v) : i = 1, 2, 3$ becomes false forever. □

Theorem 1. *The system eventually reaches a terminal configuration.*

Proof: By Lemma 5, $G_i(v), i = \{1..3\}$ is not verified, node v would only update of SR_v one time if necessary. When $G_i(v) = F,\ i = \{1..5\}$ for every node v, the system reaches a terminal configuration. □

5.2 Proof of Correctness

Theorem 2. *Once a terminal configuration is reached, the ad hoc clustering properties are satisfied.*

Proof: In a terminal configuration, for every node v, we have $G_i(v) = F :\ i = \{1..5\}$. Following Lemma 1, in a terminal configuration there is not a node v such that $Ch_v = NF$.
Case 1. $Ch_v = F$.
$G_1(v) = F$ implies N_v^+ is not empty. $G_2(v) = F$ implies $(\nexists z \in N_v^+ : (w_z > w_{Clusterhead_v} + h))$ and $(Clusterhead_v \in N_v^+)$. Thus v satisfies property 1 and 2.
Case 2. $Ch_v = T$.
$(G_2(v) = F) \equiv (\forall z \in N_v^+ : w_v > SR_z) \wedge (Cl_v \leq k)$. $G_1(v) = F$ implies that $Clusterhead_v = v$. We now prove that v has at most k neighboring clusterheads. Since $Cl_v \leq k$, then v has at most k neighboring clusterheads with higher weight than v's weight. Assume that v has more than k neighboring clusterheads, thus there exits at least a neighboring clusterhead u of v such that $w_u \leq SR_v < w_v$. Hence, $G_{22}(u) = T$ because $v \in N_u^+(w_u \leq SR_v)$, that is a contrary. □

6 Robustness

On a configuration that satisfies $\mathcal{SP}$, the clusterhead of any node performs its task correctly, because it is not an ordinary node. Thus, the hierarchical structure is kept up. Let us remind the definition of $\mathcal{SP}$: $\mathcal{SP} \equiv \forall v \in V : (Clusterhead_v \in N_v \cup \{v\}) \wedge (Ch_{Clusterhead_v} \neq F)$.

Let v a node. We define $\mathcal{SP}_v$ as the safety predicate $\mathcal{SP}$ on v.

$$\mathcal{SP}_v \equiv (Clusterhead_v \in N_v \cup \{v\}) \wedge (Ch_{Clusterhead_v} \neq F).$$

Lemma 6. *$\mathcal{SP}_v$ is closed.*

Proof: Assume that we have a computation step $c_1 \xrightarrow{cs} c_2$, we will prove that if $\mathcal{SP}_v$ is verified in c_1, then in c_2, $\mathcal{SP}_v$ is verified.
We will prove by contrary. Assume that in c_2, $(Clusterhead_v \notin \{N_v \cup v\}) \vee (Ch_{Clusterhead_v} = F)$. Thus, in cs there are two possibilities.

Case 1. v changed its clusterhead during the execution cs. Note that the rules R_4 and R_5 do not change the value of clusterhead of v. If v performs R_1 or R_3 in cs then $\mathcal{SP}_v$ is always verified because after doing R_1 or R_3, $(Clusterhead_v = v) \wedge (Ch_v \neq F)$. Thus, v performed R_2 during the execution of cs. We denote z the clusterhead selected by v in cs. In c_1, $Ch_z = T$ and in c_2, $Ch_z = F$. In cs, z cannot perform R_2. Thus, there is a contrary because R_2 is the only rule that changes the Ch_z value to F.

Case 2. v did not change its clusterhead during the execution of cs. Denote z the clusterhead of v. In c_1, $\mathcal{SP}_v$ is verified implies that $Ch_z \neq F$. In c_2, $\mathcal{SP}_(v)$ is not verified implies that $Ch_z = F$. Thus, during the execution cs, z performed R_2. But z can perform R_2 only when $G_{22}(z)$ is verified, that implies $Clusterhead_v \neq z$ in cs. That is a contrary. □

Theorem 3. *$\mathcal{SP}$ is closed.*

Proof: The theorem follows directly from Lemma 6. □

7 Time Complexity

We consider synchronous computation, in which every process performs its code simultaneously. Thus, all enabled process perform a rule in a computation step.

Theorem 4. *The system verifies $\mathcal{SP}$ in one round under a synchronous schedule.*

Proof: Assume that we have a computation step $c_1 \xrightarrow{cs} c_2$. There are two possibilities:

Case 1. In c_1, $G_i(v) = F$, $\forall i \in \{1..3\}$. We denote $z = Clusterhead_v$ in c_1.
1. If $Ch_v = T$. Since $G_{12}(v)$ and $G_{31}(v)$ are not verified, that implies $z = v$, thus $\mathcal{SP}_v$ is verified in c_1.

2. If $Ch_v = F$. Since $G_{21}(v)$ is not verified, that implies $z \in N_v^+$, thus $\mathcal{SP}_v$ is verified in c_1.

3. If $Ch_v = NF$. Since $G_{11}(v)$ and $G_{22}(v)$ are not verified, that implies $z = v$, thus $\mathcal{SP}_v$ is verified in c_1.

Thus, in c_1, $\mathcal{SP}_v$ is verified. Since $\mathcal{SP}_v$ is closed (Lemma 6), then in c_2, $\mathcal{SP}_v$ is verified.

Case 2. In c_1, $\exists i \in \{1..3\} : \ G_i(v) = T$.

1. If $G_1(v) = T$. v will performs $R_1(v)$ in cs. After performing $R_1(v)$, $(Clusterhead_v = v) \wedge (Ch_v = T)$, thus $\mathcal{SP}_v$ is verified in c_2.

2. If $G_3(v) = T$. v will performs $R_3(v)$ in cs. After performing $R_3(v)$, $(Clusterhead_v = v) \wedge (Ch_v = NF)$, thus $\mathcal{SP}_v$ is verified in c_2.

3. If $G_2(v) = T$. v will performs $R_2(v)$ in cs. We denote z' the clusterhead selected by v in cs. Using the same argument in case 2 of Lemma 6: z' could not perform R_2 in cs. Therefore, $\mathcal{SP}_v$ is verified in c_2.

□

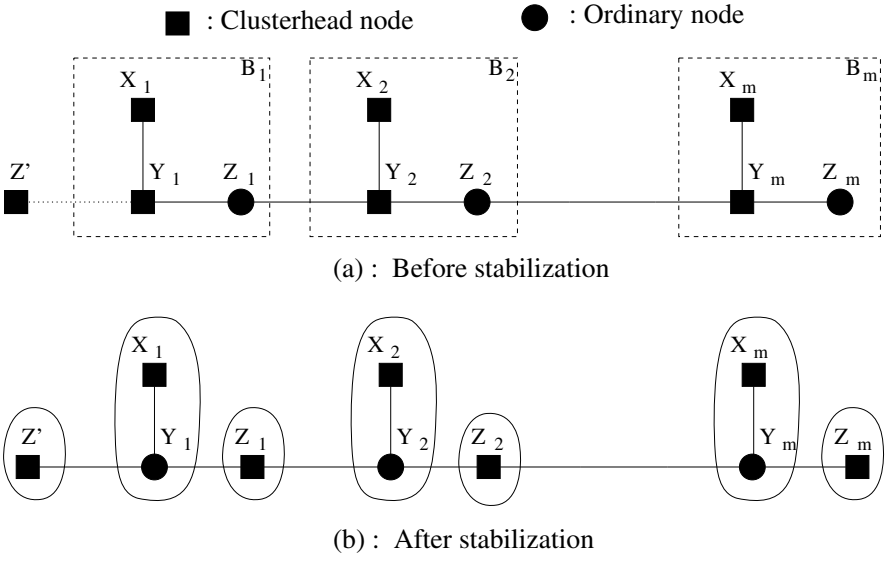


Fig. 2. Stabilization time

The stabilization time is the maximum number of computation steps needed to reach a stabilized configuration from an arbitrary initial one. Figure 2 presents a scenario to measure stabilization time in the case $k = 1, \ h = 0$. Note that this example can be generalized at any value of k and the initial configuration is the worst one. We have a configuration C composed by m blocs as depicted in Figure 2(a). Each bloc B_i includes two clusterheads X_i, Y_i and an ordinary node Z_i. We assume that the weight of nodes are ordered as the following:

$X_i > Y_i > Z_i > Y_{i+1}$. A clusterhead node Z', $Z' > Y_1$ is a neighbor of Y_1. The largest convergence time under any weight-based clustering algorithm happens with this initial configuration. We denote N the number of nodes in the system S, $N = m(k+2)+1$. Following Algorithm 1, from the initial configuration, each bloc B_i will one after another takes two computation steps to reconstruct. Thus, $2m+1$ rounds are needed to converge under the synchronous schedule. The stabilization time is $O(2N/(k+2))$.

References

1. Frodigh, M., Johansson, P., Larsson, P.: Wireless ad hoc networking: The art of networking without a network. In: Ericsson Review, No. 4. (2000)
2. Banerjee, S., Khuller, S.: A clustering scheme for hierarchical control in multi-hop wireless networks. In: The 20th Conference of the IEEE Communications Society(INFOCOM'01). (2001) 1028–1037
3. Basagni, S.: Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks. In: Proceedings of the IEEE 50th International Vehicular Technology Conference(VTC'99). (1999) 889–893
4. Basagni, S.: Distributed clustering for ad hoc networks. In: Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms, and Networks(ISPAN'99). (1999) 310–315
5. Chatterjee, M., Das, S., Turgut, D.: WCA: A weighted clustering algorithm for mobile ad hoc networks. Journal of Cluster Computing, Special issue on Mobile Ad hoc Networking **5**(2) (2002) 193–204
6. Fernandess, Y., Malkhi, D.: K-clustering in wireless ad hoc networks. In: Proceedings of the second ACM international workshop on Principles of mobile computing(POMC'02). (2002) 31–37
7. Gerla, M., Tsai, J.T.: Multicluster, mobile, multimedia radio network. Wireless Networks **1**(3) (1995) 255–265
8. Lin, C.R., Gerla, M.: Adaptive clustering for mobile wireless networks. IEEE Journal on Selected Areas in Communications **15**(7) (1997) 1265–1275
9. Johnen, C., Nguyen, L.: Self-stabilizing weight-based clustering algorithm for ad hoc sensor networks. In: Proceedings of the Second International Workshop on Algorithmic Aspects of Wireless Sensor Networks(AlgoSensors'06). (2006)
10. 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: Proceedings of the 5th IPDPS Workshop on Advances in Parallel and Distributed Computational Models(WAPDCM'03). (2003)
11. Xu, Z., Hedetniemi, S.T., Goddard, W., Srimani, P.K.: A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In: Proceedings of the 5th International Workshop on Distributed Computing, Springer LNCS 2918(IWDC'03). (2003)
12. Bein, D., Datta, A.K., Jagganagari, C.R., Villain, V.: A self-stabilizing link-cluster algorithm in mobile ad hoc networks. In: Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks(ISPAN'05). (2005) 436–441
13. Mitton, N., Fleury, E., Lassous, I.G., Tixeuil, S.: Self-stabilization in self-organized multihop wireless networks. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops(WWAN'05). (2005) 909–915

14. Mitton, N., Busson, A., Fleury, E.: Self-organization in large scale ad hoc networks. In: Proceedings of the third Annual Mediterranean Ad Hoc Networking Workshop(MED-HOC-NET'04). (2004)
15. Kakugawa, H., Masuzawa, T.: A self-stabilizing minimal dominating set algorithm with safe convergence. In: Proceedings of the 8th IPDPS Workshop on Advances in Parallel and Distributed Computational Models(APDCM'06). (2006)
16. Johnen, C., Tixeuil, S.: Route preserving stabilization. In: Proceedings of the 6th International Symposium on Self-stabilizing System, Springer LNCS 2704(SSS'03). (2003) 184–198

Self-stabilizing Wireless Connected Overlays

Vadim Drabkin[1], Roy Friedman[1], and Maria Gradinariu[2,⋆]

[1] Technion, Israel
{vdrabkin, roy}@cs.technion.ac.il
[2] LIP6, Université Paris 6 (Pierre et Marie Currie) - INRIA
Maria.Gradinariu@lip6.fr

Abstract. We propose the correctness proofs and the complexity analysis for the first self-stabilizing constructions of connected overlays for wireless networks (eg. MANETs, WSN) based on the computation of *Connected Dominating Set* (CDS). The basic idea is to construct an overlay that contains a small number of nodes, but still obtain full connectivity of the network while only relying on local exchanges of information and knowledge. We adopt two methodologies of construction: the first methodology consists of two parallel tasks, namely, computing a *maximal independent set* (MIS) and then adding bridge nodes between the MIS nodes. The second methodology computes a connected dominating set using the observation that a dominator is a bridge between nodes that do not share the same neighborhood.

The proposed algorithms are fully decentralized and are designed in a self-stabilizing manner in order to cope with transient faults, mobility and nodes join/leave. In particular, they do not need to be (re)initialized after a fault or a physical topology change. That is, whatever the initial configuration is, the algorithms satisfy their specification after a stabilization period. The convergence time of our algorithms is linear in the size of the network and they use only one extra bit of memory. We also present an optimization of our algorithms that reduces the number of nodes in the cover. However, the optimization increases the convergence time with a constant factor.

1 Introduction

Wireless adhoc or sensor networks, unlike cellular networks, do not rely on a pre-existing cellular infrastructure. In these networks it is beneficial to set-up an overlay (backbone) that will help economizing the energy of the system while routing, clustering, deploying replicas or scheduling the execution of nodes in order to avoid collisions. It is folklore nowadays that the computation of connected dominating sets is the basis of such backbones. A dominating set of a graph is a subset of the graph nodes such that every node in the graph is either member of the dominating set or adjacent to a member of the dominating set. A dominating set is connected if there is a path between any two nodes in the dominating set that includes only nodes in the dominating set.

⋆ This work was done while the author was with IRISA, Université Rennes 1 - INRIA.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 425–439, 2006.

In a large scale network deployed in a fault prone environment a connected backbone should verify additional properties such as: consume the minimum resources in the system, have a decent size and finally to be able to self-organize and recover from transient or permanent failures.

One way to construct a connected overlay is to exploit the common need of the applications that will share the local resources of each node in the network. Maximal independent sets (MIS) are often used in order to design efficient clustering, provide collision free transmission, address scheduling or resource allocation issues. Therefore, a service that provides a maximal independent set is mandatory in the future middlewares dedicated to wireless networks. This service can also be exploited in order to obtain connected overlays by simply connecting the nodes selected to be part of the maximal independent set. Therefore, one of the design methods proposed in our paper uses the above approach. However, in systems that are designed to respond to very simple tasks like for example gathering and transmission of data, the middleware part may not include a maximal independent set service. Consequently, in these systems, it is more interesting to compute connected overlays based on the construction of dominating sets from scratch. Therefore in our work we also explore the fault-tolerance and self-stabilization issues of this second design methodology.

Related Work. A first algorithm that computes a weakly connected overlay based on the idea of connecting MIS nodes was proposed in [1]. The proposed algorithm performs first a leader election then constructs a spanning tree rooted at the leader. Then the leader starts a coloring process. In this phase the levels of the tree color alternately gray and black (the black nodes form a MIS while the gray nodes are the bridges). The proposed solution uses inefficiently the network resources since it requires a leader election and the construction of a spanning tree. Additionally it introduces an unique point of failure (the leader). Moreover, the solution is not fault-tolerant. A distributed local, fault tolerant and self-healing solution for the computation of a weakly connected overlay based on connecting MIS nodes via bridges was proposed in [12]. The proposed solution is only evaluated via simulations, no correctness proofs are provided.

Several algorithms have been proposed for finding connected dominating sets in adhoc networks or sensor networks [19,23,4,5,21,3,12,18,16]. All these solutions are distributed and apply the pruning strategy: a subset of nodes is selected to be part of the connected cover and from this set all redundant nodes are removed. However none of the above algorithms is self-stabilizing [1] or fault-tolerant. Self-organizing issues in the construction of connected dominating sets is addressed in [20]. The fault tolerance issues are addressed in [25] by the study of k-coverage and k-connectivity. The proposed solution involves the computation of a Voronoi diagram for independent sensor nodes. Neither the implementation of local Voronoi diagrams nor the transient faults are addressed. In [6,7,8] self-stabilizing solutions are proposed for computing connected covers of query

[1] An algorithm is said to be self-stabilizing if started in any arbitrary state, the algorithm reaches some predefined legitimate state in a finite number of steps.

regions in sensor networks. As stated in [8] the above problem is not equivalent to computing a connected overlay. However, solutions proposed in [6,8] use as building blocks in our solutions to the connected overlays proposed in [12].

To the best of our knowledge the first self-stabilizing algorithm for the computation of connected dominating sets was proposed in our previous work [12] in the context of designing efficient lookup algorithms for MANETs. In this work several overlays are experimentally evaluated with respect to the lookup specific metrics. However the proposed overlays are not proved correct neither analyzed from the complexity perspective. In [2] the authors proposed and prove correct a solution for connected dominating sets based on the self-stabilizing minimum domination protocol proposed in [24]. The solution proposed in [2] needs a synchronous scheduler and converges in a polynomial time. In parallel with our work, [17] proposed an algorithm for computing a self-stabilizing minimal dominating set algorithm with safe convergence. The algorithm works under a synchronous scheduler.

Contributions. In this paper we propose novel self-stabilizing and fault-tolerant algorithms for the computation of connected overlays in wireless networks. Unlike the solutions proposed in [2,17], our algorithms do not make use of any synchrony (i.e. they work in asynchronous environments). Additionally, they are able to self-organize, self-heal and cope with both transient and permanent faults. Therefore they are appealing for large scale networks. Two methodologies of design are used. The fist one uses as basis a maximal independent set overlay and computes in a self-stabilizing manner bridges between MIS nodes, while the second one computes a connected dominating set from scratch. We proposed two different algorithms that follow the first strategy. The convergence time of both of them is linear in the size of the network. The first algorithm converges quicker while the second algorithm provides a smaller overlay. Both algorithms need only one additional state provided a MIS. Additionally, we prove correct and analyze the complexity of the self-stabilizing algorithm we first proposed in [12] that computes a connected dominating set. The algorithm needs only 2 states and converges under an asynchronous scheduler in a linear time in the size of the network. Due to space limitation the detailed proofs of our results are available in [11].

2 System Model and Definitions

Network Model. In this work we focus on wireless mobile systems. A *node* in the system is a device that uses wireless communication. A transmission of a node p is received by all nodes within a disk centered on p whose radius depends on the transmission power, referred to in the following as the *transmission disk*; the radius of the transmission disk is called the *transmission range*. Thus, there is a single communication primitive, `send`(m), allowing a node to transmit a message m to all nodes inside its transmission disk. The combination of the nodes and the transitive closure of their transmission disks forms a wireless ad-hoc network.

The network described above can also be modeled as a graph $G = (V, E)$ where V is the set of network nodes and E models the one-to-one neighboring relations. A node q is a *neighbor* of another node p if q is located within the transmission disk of p. In the following, $\mathcal{N}(p)$ refers to the set of neighbors of a node p. By considering $\mathcal{N}(p)$ as a relation (defining the set $\mathcal{N}(p)$), we say that a node p has a *path* to a node q if q appears in the transitive closure of the $\mathcal{N}(p)$ relation.

Finally, we assume an abstract entity called an *overlay*, which is simply a collection of nodes. Nodes that belong to the overlay are called *overlay backbone nodes*. Additionally we consider a generic function which associates with each node some value from an ordered domain, which represents the node's appropriateness to serve in the overlay. We call this value the *goodness number*. This way, it is possible to compare any two nodes using their goodness number and to prefer to elect the one whose value is higher to the overlay. For example, it is easy to evaluate and compare the battery level of nodes, or to obtain and compare the number of objects for which a node is proxy. In the following, e_i denotes the goodness number of node i. In order to simplify the algorithms presentation we introduce the goodness relation denoted in the following by $\prec$. Node j is better than node i according to the $\prec$ relation if either the goodness number of i is superior to the goodness number of j or the nodes have the same goodness number but the identifier of j is greater than the identifier of i. Formally, $i \prec j$ iff $e_i < e_j \vee e_i = e_j \wedge id_i < id_j$. Note that $\prec$ defines a total order on the nodes when the goodness values are comparable. Part of what we do in this paper is investigating various protocols for deciding which nodes should be in the overlay.

Program. In this paper, we consider the local shared memory model of communication as used by Dijkstra [9]. The program of every processor consists of a set of *shared variables* (henceforth, referred to as variables) and a finite set of actions. Every processor (or sensor) can only write to its own variables, but can read its own variables and the variables owned by the neighboring nodes. The above can be easily implemented in a wireless environment using a pool or push strategy and the assumption that each node locally maintains a copy of the shared variables of its neighbors. With the pool strategy each node periodically synchronizes its local copies with the real variables of its neighbors. In the push strategy a node, whenever its local variables change, pushes the changes to its neighborhood.

Each action is of the following form: $< label >::< guard > \longrightarrow < statement >$. 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.

The *state* of a node is defined by the values of its variables. The *state* of a system is the product of the states of all nodes. We will refer to the state of a node and system as a (*local*) *state* and (*global*) *configuration*, respectively.

Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations 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* [2]. The *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. The set of all possible computations of $\mathcal{P}$ in system S is denoted as $\mathcal{E}$. A node p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists an action A such that the guard of A is true in γ. Similarly, an action A is said to be enabled (in γ) at p if the guard of A is true at p (in γ).

We assume a *weakly fair and distributed daemon(scheduler)* also referred in the following as asynchronous or arbitrary scheduler. *Weak fairness* means that if a node p is continuously enabled, then p will be eventually chosen by the daemon to execute an action. A *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.

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.

Self-stabilization [10]. Let $\mathcal{L}_{\mathcal{A}}$ be a non-empty *legitimacy predicate*[3] of an algorithm $\mathcal{A}$ with respect to a specification predicate *Spec* such that every configuration satisfying $\mathcal{L}_{\mathcal{A}}$ satisfies *Spec*. Module $\mathcal{A}$ is *self-stabilizing* with respect to *Spec* iff the following two conditions hold:
(i) Every computation of $\mathcal{A}$ starting from a configuration satisfying $\mathcal{L}_{\mathcal{A}}$ preserves $\mathcal{L}_{\mathcal{A}}$ (*closure*).
(ii) Every computation of $\mathcal{A}$ starting from an arbitrary configuration contains a configuration that satisfies $\mathcal{L}_{\mathcal{A}}$ (*convergence*).

We require the algorithms to be self-organizing, self-stabilizing and self-healing [10]. That is, regardless of the initial state (wrong initialization of the local variables, memory or program counter corruptions) nodes *self-configure/self-organize* using only local information in order to make the system *self-stabilize* to a *legitimate state*. The legitimate state is defined with respect to a 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 connected cover should be able to *self-heal* without any external intervention and the impact should be confined within a tightly bounded region around the perturbed area.

[2] In a terminal configuration no action is enabled.
[3] A legitimacy predicate is defined over the configurations of a system and is an indicator of its correct behavior.

3 Self-∗ MIS-Based Connected Overlay

We start this section by giving a formal definition of a *Maximal Independent Set* (MIS) and then discuss how to obtain an MIS-based connected overlay.

Let $G = (V, E)$ be a communication graph. Two nodes i and j in G are said to be *independent* if $(i, j) \notin E$. A subset $S \subseteq V$ of nodes is *independent* if every pair of nodes in S are independent. A set S is a *maximal independent set* (MIS) if S is independent, yet for any node $k \in V \setminus S$, $S \cup \{k\}$ is not independent.

The MIS-based connected overlay proposed in this paper is constructed in two phases that may be executed in parallel. In the first phase, the MIS is computed. Since by definition, the set of nodes in an MIS cannot directly communicate with each other, a second phase identifies bridge nodes that connect the MIS nodes. Of course, the goal is to find as few bridges as possible, yet to do this in a completely decentralized manner.

Several self-stabilizing constructions of MIS overlays are proposed in the literature ([14,15,12]). In the next section we analyze the self* properties and the complexity of the solution we first proposed in [12]. This solution extends the algorithm proposed in [15] to systems where the selection of MIS nodes is based on their goodness number. The time complexity of our solution is $O(n)$ and it uses only one memory bit which is the optimal memory requirement for computing a MIS. Nodes with respect to the MIS can be *active* or *passive*. An active node is a node part of the MIS while a passive node is not member of the MIS.

3.1 Self-stabilizing Maximal Independent Set (MIS)

We are interested in a distributed algorithm for computing a MIS in such a way that every node makes local calculations based only on the knowledge of its neighbors. Recall that the neighbors of p are the nodes that appear in the transmission disk of p, and thus p can communicate directly with them, and every message p sends is received by all of them. Additionally, we would like to influence the overlay construction process such that the overlay nodes will be the "best" nodes under a given metric. For example, since in mobile systems nodes are often battery operated, we may wish to use the energy level as the metric, in order to have the nodes with highest energy levels members of the overlay. Alternatively, we might use the number of objects for which a node is proxy as the metric, in order to reduce the average number of hops a search message has to travel. Similarly, we might use bandwidth, transmission range, or local storage capacity, or some combination of several such metrics. This is achieved by the *goodness number*.

The MIS algorithm consists of computation steps that are taken periodically and repeatedly by each node. In each computation step, each node makes a local computation about whether it thinks it should be in the MIS or not, and then exchanges its local information with its neighbors. For simplicity, we concentrate below on the local computation steps only.

Module 1. Goodness-based MIS executed by node i

Input Parameters: $\mathcal{N}(i)$: set of i's neighbors (including i itself);
$status_j, \forall j \in \mathcal{N}(i), j \neq i$: the status of each i neighbor;
$e_j, \forall j \in \mathcal{N}(i)$: the goodness number of neighbor j;

InOut Parameters: $status_i$: the status of node i
Predicates: $i \prec j \equiv e_i < e_j \vee e_i = e_j \wedge id_i < id_j$
$MAXE(i) \equiv \forall j \in \mathcal{N}(i), j \neq i, j \prec i$
Actions:
$\mathcal{R}_1$: $MAXE(i) \wedge status_i \neq active \rightarrow status_i := active$
$\mathcal{R}_2$: $status_i = active \wedge \exists j \in \mathcal{N}(i), status_j = active \wedge i \prec j \rightarrow status_i := passive$
$\mathcal{R}_3$: $\neg MAXE(i) \wedge \forall j \in \mathcal{N}(i), status_j \neq active \rightarrow status_i := active$

The local state of each node includes a *status*, which is either *active* or *passive*, its goodness number, and its knowledge of the local states of all its neighbors (based on the last local state they reported to it), and for each neighbor, the list of its active neighbors. The *active* status means that the node believes it is in the MIS, while *passive* means that it believes that it is not part of the MIS.

The local execution of the MIS part of the protocol includes the rules $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ of Module 1. The first rule, $\mathcal{R}_1$, is used to elect nodes that have the maximal goodness number in their neighborhood. The second rule, $\mathcal{R}_2$, is used to ensure that we do not have a situation in which two neighboring nodes are in the MIS. This could happen, for example, due to movement of nodes, which changes their neighborhood. Thus, if an active node i finds that one of its neighbors is active and has higher goodness number, then i gives up and becomes *passive*. The third rule, $\mathcal{R}_3$, is executed by not active nodes that do not have an *active* neighbor even though they do not have the highest goodness number. This situation can occur, for example, if all neighbors of i that have higher goodness number than i gave up being *active* due to the fact that they had other neighbors with even higher goodness numbers.

Rules $\mathcal{R}_1$ and $\mathcal{R}_3$ are based on the evaluation of the maximal goodness number in a neighborhood. In case two nodes have the same goodness number, the symmetry is broken using ids. In the following, e_i denotes the goodness number of node i. In order to simplify the algorithms presentation we introduce the goodness relation denoted in the following by $\prec$. Node j is better than node i according to the $\prec$ relation if either the goodness number of i is superior to the goodness number of j or the nodes have the same goodness number but the identifier of j is greater than the identifier of i. Formally, $i \prec j$ iff $e_i < e_j \vee e_i = e_j \wedge id_i < id_j$. Note that $\prec$ defines a total order on the nodes when the goodness values are comparable. Also, $MAXE(i)$ is a boolean predicate that evaluates to true if and only if i is maximal in its neighborhood w.r.t. the relation $\prec$ defined above.

Definition 1 (legitimate configuration for Module 1). *A legitimate configuration of Module 1 is a configuration where active nodes define a maximal independent set. Let $\mathcal{L}_{MIS}$ be the set of legitimate configurations of Module 1.*

Module 2. Local Asynchronous Bridge Construction executed by node i

InOut Parameters: $status_i$: the status of i
Input Parameters: $\mathcal{N}(i)$: set of i's neighbors;
Predicates: $i \prec j \equiv e_i < e_j \vee e_i = e_j \wedge id_i < id_j$
$BridgeCandidate(i) \equiv \exists j \in \mathcal{N}(i), status_j = active \wedge \neg(\mathcal{N}(i) \subseteq \mathcal{N}(j))$
$Covered(i) \equiv \exists j \in \mathcal{N}(i), j \neq i, \big(\mathcal{N}(i) \subset \mathcal{N}(j) \vee (\mathcal{N}(i) = \mathcal{N}(j) \wedge i \prec j)\big)$

Actions:
$\mathcal{R}_1$: $status_i = passive \wedge BridgeCandidate(i) \wedge \neg Covered(i) \rightarrow status_i := bridge$
$\mathcal{R}_2$: $status_i = bridge \wedge (\neg BridgeCandidate(i) \vee Covered(i)) \rightarrow status_i := passive$

Note that once in a legitimate configuration none of the rules of Module 1 are enabled.

Lemma 1 (convergence of Module 1). *Let $\mathcal{S}$ be a system executing Module 1 under an arbitrary scheduler. Any execution of $\mathcal{S}$ converges to a configuration in $\mathcal{L}_{MIS}$.*

Lemma 2 (convergence time of Module 1). *The convergence time of Module 1 is O(n) steps under an arbitrary weakly fair scheduler.*

In the next two sections we propose and prove correct self-stabilizing constructions of MIS-based connected overlays.

3.2 Local Asynchronous MIS-Based Connected Overlay

The algorithm shown as Module 2 provides the bridges between the active nodes computed by an underlying self-stabilizing MIS algorithm (recall that MIS nodes are not neighbors). An example of a MIS algorithm can be Module 1. Module 2 has two rules and is designed to work on top of a self-stabilizing algorithm that computes a maximal independent set. The local state of each node includes a *status*, which is either *active*, *passive* or *bridge*, its goodness number, and its knowledge of the local states of all its neighbors (based on the last local state they reported to it), and for each neighbor, the list of its active neighbors. The *active* status means that the node believes it is in the MIS, *bridge* means that it is acting as bridge, and *passive* means that it is neither.

The first rule, $\mathcal{R}_1$, is used to designate nodes that potentially connect two nodes in the MIS. Specifically, if node i is *passive*, and has at least one neighbor which is *active*, then i becomes a *bridge* node only if it is not covered by other node. Rule $\mathcal{R}_2$ is used to eliminate redundant *bridge* nodes (covered by other bridges). That is, whenever there are two neighboring *bridge* nodes i and j such that the set of neighbors of i is included in the set of j's neighbors, or both have the same neighbors but i has lower goodness number than j, then i switches back to *passive*. Rule $\mathcal{R}_2$ also acts as a correction rule. Whenever, bridges are erroneously initialized (they are not neighbors of an active node or are covered by an active neighbor) they are demoted to passive.

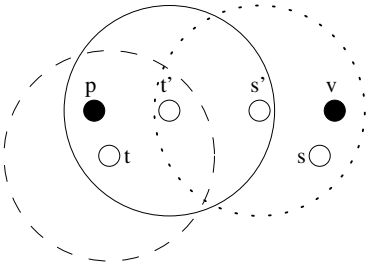


Fig. 1. Topology Example

Example 1. In the following we show via an example an asynchronous execution of Module 2 under a weakly fair arbitrary scheduler.

The topology of the network in Figure 1 is as follows: p, v are active; t, t', s, s' are passive; $t, t' \in \mathcal{N}(p)$; $p, t' \in \mathcal{N}(t)$; $p, t, s' \in \mathcal{N}(t')$; $s, s' \in \mathcal{N}(v)$; $v, s' \in \mathcal{N}(s)$; $v, s, t' \in \mathcal{N}(s')$. Let t', t passive and let s and s' passive as well. Since the scheduler is weakly fair, t' eventually executes rule $\mathcal{R}_1$ of Module 2 and becomes *bridge*. t is covered by t', t will not execute rule $\mathcal{R}_1$ of Module 2 and remains *passive*. In a similar way, s' becomes *bridge* and s stays *passive*.

Definition 2 (legitimate configuration for Module 2). *A legitimate configuration of Module 2 is a configuration where any good neighbor of an active node is a bridge. A good neighbor is a neighbor that is not covered by any other neighbor. Let $\mathcal{L}_{BRIDGE}$ be the set of legitimate configurations of Module 2.*

Lemma 3 (convergence of Module 2). *Let $\mathcal{S}$ be a system executing Module 2 under an asynchronous scheduler. Any execution of $\mathcal{S}$ converges to a configuration in $\mathcal{L}_{BRIDGE}$.*

Lemma 4. *The convergence time of Module 2 started from a configuration verifying $\mathcal{L}_{MIS}$ is (n-m) steps in the worst case, where m is the size of the MIS.*

Let WCMIS be the hierarchical composition [13] of a self-stabilizing maximal independent set (eg. Algorithm shown as Module 1) and Module 2.

Definition 3 (legitimate configuration for WCMIS). *A legitimate configuration of WCMIS is a configuration where active nodes define a maximal independent set and the subgraph containing only active and bridge nodes is connected. Let $\mathcal{L}_{WCMIS}$ be the set of legitimate configurations of WCMIS.*

Lemma 5 (convergence of WCMIS). *Let $\mathcal{S}$ be a system executing WCMIS under an asynchronous scheduler. Any execution of $\mathcal{S}$ converges to a configuration in $\mathcal{L}_{WCMIS}$.*

Lemma 6. *Let O(f(n)) be the convergence time of a self-stabilizing MIS algorithm. The convergence time of WCMIS is O(f(n)+n) steps. The convergence of WCMIS obtained as the hierarchical composition between Module 1 and Module 2 is $O(n)$.*

Module 3. Asynchronous Bridge Construction executed by node i

InOut Parameters: $status_i$: the status of i
Input Parameters: $\mathcal{N}(i)$: set of i's neighbors;
$ActiveN^d(i)$: set of i's active neighbors up to distance d;
Predicates: $i \prec j \equiv e_i < e_j \vee e_i = e_j \wedge id_i < id_j$
$BridgeCandidate(i) \equiv \exists j \in \mathcal{N}(i), status_j = active$
$Covered(i) \equiv \exists j \in \mathcal{N}(i), j \neq i, status_j = bridge \wedge \big((ActiveN^2(i) \subset ActiveN^2(j)) \vee (ActiveN^2(i) = ActiveN^2(j) \wedge i \prec j)\big)$
Actions:
$\mathcal{R}_1$: $status_i = passive \wedge BridgeCandidate(i) \wedge \neg Covered(i) \rightarrow status_i := bridge$
$\mathcal{R}_2$: $status_i = bridge \wedge (Covered(i) \vee \neg BridgeCandidate(i)) \rightarrow status_i := passive$

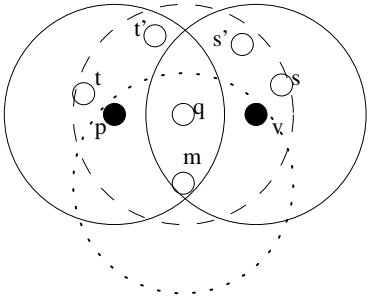


Fig. 2. Topology Example

3.3 Self* MIS-Based Connected Overlay with Restricted Knowledge

The algorithm shown as Module 2 stabilizes under an weakly fair scheduler and is fault tolerant. Unfortunately, the cover may include almost all the nodes in the network. In the following we propose a slightly modification of this algorithm, Module 3, that reduces via pruning the size of the overlay. The essential difference between the two algorithms is the choice of redundant bridges via the Cover predicate. A bridge i is covered by another bridge j if j has as neighbors all the *active neighbors* of i up to distance two. The algorithm idea is as follows. A node, neighbor of an active node, promotes to bridge via $\mathcal{R}_1$ if it has no neighbor bridge that cover it. A bridge is demoted to passive via $\mathcal{R}_2$ if it is covered by another bridge or it has no active neighbor.

Example 2. In the following we show via two examples an asynchronous execution of Module 3 under a weakly fair arbitrary scheduler.

Consider first the topology showed in Figure 1. Assume t' and t passive (assume s and s' passive as well). Since the scheduler is weakly fair, t and t' eventually execute rule $\mathcal{R}_1$ of Module 3 and become *bridge*. Since t' covers t then t can execute $\mathcal{R}_2$ and demotes to passive. The same scenario is true for the node s

that is covered by the node s'. So, s can execute $\mathcal{R}_2$ and demotes to passive. t' and s' cannot execute their actions. So, the system is stabilized.

Consider now Figure 2 with the following topology: p, v active; t, t', s, s', m, q passive; $t, t', m, q \in \mathcal{N}(p)$; $p, t', q \in \mathcal{N}(t)$; $p, t, s', q \in \mathcal{N}(t')$; $s, s', m, q \in \mathcal{N}(v)$; $v, s', q \in \mathcal{N}(s)$; $v, s, t', q \in \mathcal{N}(s')$. $p, t', s', v, t, s, m \in \mathcal{N}(q)$ $p, v, q \in \mathcal{N}(m)$. Assume $m \prec q$, $t \prec t'$ and $s \prec s'$. Assume all nodes but p and v passive. Since the scheduler is weakly fair, t and t' eventually execute rule $\mathcal{R}_1$ of Module 3 and become *bridge*. Since t' covers t, t will execute rule $\mathcal{R}_2$ of Module 3 and will demote to passive. In a similar way, s becomes passive. Finally, m, q execute $\mathcal{R}_1$. So, m, s', t' are enabled for $\mathcal{R}_2$ ($m \prec q$ and s', t' are covered by q). The overlay stabilizes and each node executes its actions a finite number of steps.

Lemma 7 (convergence of Module 3). *Let $\mathcal{S}$ be a system executing Module 3 under an asynchronous scheduler. Any execution of $\mathcal{S}$ converges to a configuration in $\mathcal{L}_{BRIDGE}$.*

Lemma 8. *The convergence time of Module 3 started from a configuration verifying $\mathcal{L}_{MIS}$ is $\Delta(n-m))$ steps, where m is the size of the MIS and Δ is the diameter of the network.*

Let WCMIS be the hierarchical composition of Modules shown as Module 1 and Module 3. Using Lemmas 2 and 8 and the same arguments as for the proof of Lemma 5 we obtain the following results.

Lemma 9 (convergence of WCMIS). *Let $\mathcal{S}$ be a system executing WCMIS under an asynchronous scheduler. Any execution of $\mathcal{S}$ converges to a configuration in $\mathcal{L}_{WCMIS}$. The convergence time of WCMIS is $(\Delta + 2)n$ steps in the worst case or $O(\Delta n)$.*

Optimization. The convergence time of Module 3 can be reduced (for the worst case) with a slightly modification of the Coverage predicate.

$CoveredModified(i) \equiv \exists j \in \mathcal{N}(i), j \neq i, status_j = bridge \wedge \big((Active\_or\_BridgeN(i) \subset Active\_or\_BridgeN(j)) \vee (Active\_or\_BridgeN(i) = Active\_orBridgeN(j) \wedge i \prec j)\big)$ where $Active\_or\_BridgeN(i)$ is the set of active i's neighbors up to distance 2 and bridge neighbors up to distance 1, $Active\_or\_Bridge(i) = ActiveN^2(i) \bigcup BridgeN^1(i)$ where $BridgeN^d(i)$ denotes the bridge neighbors of the node i up to distance d.

Lemma 10. *In each execution of Module 3 where Covered predicate is replaced by $CoveredModified$ predicate a non active process executes its actions at most 2 times. The convergence time of Module 3 is $2(n-m)$ steps in the worst case, where m is the size of the MIS.*

Corolary 1. *The convergence time of WCMIS obtained as the composition of Module 3 optimized and Module 1 is $2n$ steps in the worst case or O(n).*

4 Self-∗ Connected Dominating Set (DS) Based Overlay

We start this section by giving a formal definition of a *Connected Dominating Set* (DS), and then discuss how to obtain a DS based overlay.

Let $G = (V, E)$ be a communication graph. A set $S \subset V$ is a *dominating set* if any node in V is a member of S or has a neighbor in S. S is *connected* if for any two nodes in S there is a communication path between them including only nodes in S.

In the following, we present a self-stabilizing version of the DS-algorithm construction of [22,23]. The algorithm (Module 4) requires each node to know about its neighbors at distance two, or in other words, the neighbors of its direct neighbors. The protocol also uses the neighbors independence predicate, as defined below:

Definition 4 (independent neighbors). *Let $G = (V, E)$ be the communication graph and let $i \in V$. $Independent\_Neighbors(i) \equiv \exists y, k \in \mathcal{N}(i), y \neq k \neq i, k \notin \mathcal{N}(y) \wedge y \notin \mathcal{N}(k)$.*

Intuitively, the predicate $Independent\_Neighbors(i)$ evaluates to true if there are two neighbors of i, y and k, that are not direct neighbors of each other. If this predicate is true, then i should be in the dominating set, unless there is another node j that is a neighbor of both y and k and has a higher goodness number.

Thus, a node i executes the rule $\mathcal{R}_1$ if the predicate $Independent\_Neighbors$ is true for i and i is not dominated by another node then i becomes *active*. However, by the rule $\mathcal{R}_2$, if an *active* node i finds another *active* neighbor j and both share the same neighbors or the set of i's neighbors is included in the set of j's neighbors, yet the other node has a higher goodness number, then i gives up. Note that at the beginning of the protocol all nodes might be *passive*. Also, if the graph is fully connected (a clique), then no node will become *active* by $\mathcal{R}_1$. This is taken care of by the rule $\mathcal{R}_3$, in which if all the nodes in i's neighborhood are *passive* and i has the maximal goodness number, then i becomes *active*.

The system is in a legitimate configuration if the set of nodes is split in two classes: the class of active nodes and the class of passive nodes. The active nodes form a connected overlay and each passive node has an active neighbor.

Definition 5 (legitimate configuration for Module 4). *A legitimate configuration of Module 4 is a configuration where active nodes define a connected overlay and each passive node has an active neighbor. Let $\mathcal{L}_{CDS}$ be the set of legitimate configurations of Module 4.*

Lemma 11. *Let e be an arbitrary execution of Module 4. e converges to a terminal configuration in a finite number of steps. The set of active nodes in the terminal configuration is connected.*

Lemma 12. *A terminal configuration for Module 4 is in $\mathcal{L}_{CDS}$.*

Lemma 13. *Module 4 converges in n steps.*

Module 4. Goodness based DS executed by node i

Input Parameters: $\mathcal{N}(i)$: set of i's neighbors;
$status_j, \forall j \in \mathcal{N}(i), j \neq i$: active,passive;
$e_j, \forall j \in \mathcal{N}(i)$: the goodness number of neighbor j;
$\mathcal{N}(j), \forall j \in \mathcal{N}(i)$: the set of neighbors of neighbor j of i;
InOut Parameter: $status_i$: the status of node i
Predicates:
$Independent\_Neighbors(i) \equiv \exists y, k \in \mathcal{N}(i), y \neq k \neq i, k \notin \mathcal{N}(y) \wedge y \notin \mathcal{N}(k)$
$i \prec j \equiv e_i < e_j \vee e_i = e_j \wedge id_i < id_j$
$MAXE(i) \equiv \forall j \in \mathcal{N}(i), j \neq i, j \prec i$
$Dominator(j, i) \equiv (\mathcal{N}(i) \subset \mathcal{N}(j)) \vee (\mathcal{N}(i) = \mathcal{N}(j) \wedge i \prec j)$
Actions:
$\mathcal{R}_1$: $status_i = passive \wedge Independent\_Neighbors(i) \wedge \exists j \in \mathcal{N}(i), Dominator(j, i) \rightarrow status_i := active$
$\mathcal{R}_2$: $status_i = active \wedge \exists j \in \mathcal{N}(i), status_j = active \wedge Dominator(j, i) \rightarrow status_i := passive$
$\mathcal{R}_3$: $status_i = passive \wedge \forall j \in \mathcal{N}(i), \mathcal{N}(j) = \mathcal{N}(i) \wedge MAXE(i) \rightarrow status_i := active$

5 Self-healing Properties of the Proposed Algorithms

Note 1. Note that Module 1 converges despite local fluctuations of the goodness value.

- If an active node suffers a drop in its goodness value then it will be automatically demoted and replaced by the node with the maximal goodness in its neighborhood.
- If the goodness value of a passive node increases such that it becomes maximal in its neighborhood then the node is promoted active and replaces the current active node.
- If in a neighborhood a new node is injected and this node has the maximal goodness then the new node is promoted active and replaces the existing active node in its neighborhood.

Lemma 14. *WCMIS obtained as the composition of Module 2 and Module 1 self-heals under an asynchronous scheduler.*

Lemma 15. *WCMIS obtained as the composition of Module 3 and Module 1 self-heals under an asynchronous scheduler.*

Lemma 16. *Module 4 self-heals.*

6 Conclusions

In this paper we proposed the correctness proofs and analyzed the stabilization time and the memory cost of the fist three self-stabilizing algorithms that

compute connected overlays in wireless network under asynchronous schedulers. These algorithms are designed following two different methods: the first two algorithms use an underlying maximal independent set while the third one constructs a connected dominating set from scratch. The first method may be tempting in systems where a maximal independent service already exists and only local computations are required (eg. byzantine prone environments). However, this method generates a coverage set that may include (in sparse topologies) almost all nodes in the network (see Module 2). Therefore we proposed Module 3 that computes a better connected overlay using pruning strategies. However, the pruning proposed in this paper increase the convergence time with a constant factor and need communications at two hops distance. The most appealing approach is Module 4 with a linear convergence time in the worst case and a cost of one memory bit. However, in order to execute this algorithm each node has to know the neighbors of their neighbors. An open question is to find self-stabilizing solutions to the connected overlays problem that converge in a sub-linear time. Another open question would be to find a relation between the cost of pruning and its impact on the drop of the coverage size. Finally, a third open question is to design the byzantine robust version of the proposed algorithms.

References

1. K. Alzoubi, Wan Peng-Jun, and Ophir Frieder. Weakly connected dominating sets and sparse spanners in wireless adhoc networks. *ICDCS03 Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 96–104, 2003.
2. J. Ankur and A. Gupta. A distributed self-stabilizing algorithm for finding a connected dominating set in a graph. *PDCAT 2005*, 2005.
3. F Dai and J Wu. Distributed dominant pruning in ad hoc networks. *Proceedings of ICC'03*, 2003.
4. Bevan Das and Vaduvur Bharghavan. Routing in ad-hoc networks using minimum connected dominating sets. In *ICC (1)*, pages 376–380, 1997.
5. Bevan Das, Raghupathy Sivakumar, and Vaduvur Bharghavan. Routing in ad hoc networks using a spine. In *ICCCN*, pages 34–41, 1997.
6. AK Datta, M Gradinariu, P Linga, and P Raipan-Parvéde. Self-stabilizing query covers in sensor networks. *SRDS*, 2005.
7. AK Datta, M Gradinariu, and R. Patel. Optimal self* query region covers in sensor networks. *ISPAN*, 2005.
8. AK Datta, M Gradinariu, and R Patel. Dominating-sets based self-stabilizing minimum query covers in sensor networks. Technical Report 1803, IRISA/Universite Rennes 1, 2006.
9. EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11):643–644, Nov 1974.
10. S Dolev. *Self-Stabilization*. MIT Press, 2000.
11. V. Drabkin, R. Friedman, and M. Gradinariu. Self-stabilizing wireless connected overlay. Technical report, LIP6, Universite Paris 6, 2006.
12. R. Friedman, M. Gradinariu, and G. Simon. Locating cache proxies in manets. *MobiHoc04 Proceedings of the Thifth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2004.

13. Mohamed G. Gouda and Ted Herman. Adaptive programming. *IEEE Trans. Software Eng.*, 17(9):911–921, 1991.
14. M. Gradinariu and S. Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *Proceedings of OPODIS 2000, STUDIA INFORMATICA*, pages 55–70, 2000.
15. T. Herman and S. Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. *Algosensors 2004*, 2004.
16. F Ingelrest, D Simplot-Ryl, and I Stojmenovic. Smaller connected dominating sets in ad hoc and sensor networks based on coverage by two-hop neighbors. Technical report, Institut National De Recherche En Informatique Et En Automatique, April 2005.
17. H. Kakugawa and T Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. *IEEE Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006.
18. H Liu, Y Pan, and J Cao. An improved distributed algorithm for connected dominating sets in wireless ad hoc networks. *Proceedings of the ISPA'04*, Dec 2004.
19. Wan Peng-Jun, K. Alzoubi, and Ophir Frieder. Distributed construction of connected dominating sets in wireless adhoc networks. *INFOCOM02 Proceedings of the Conference on Computer Communications*, 2002.
20. Fabrice Theoleyre and Fabrice Valois. About the self-stabilization of a virtual topology for self-organization in ad hoc networks. In *Self-Stabilizing Systems*, pages 214–228, 2005.
21. J Wu. Extended dominating-set-based routing in ad hoc wireless networks with unidirectional links. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):866–881, Sep 2002.
22. J. Wu, M. Gao, and Stojmenovic. On calculating power-aware connected dominating sets for efficient routing in ad hoc wireless networks. *Proc. of the 30th International Conference on Parallel Processing (ICPP'01)*, pages 346–353, 2001.
23. J. Wu and H. Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. *Proc. of the 3th Int. Workshop on Discrete Algothrithms and Methods for MOBILE Computing and Communications (DialM'99)*, pages 7–14, 1999.
24. Z. Xu, S.T. Hedetniemi, W. Goddard, and P.K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. *IWDC 2003*, 2003.
25. S. Das Z. Zhou and H. Gupta. Fault tolerant connected sensor cover with variable sensing and transmission. In *SECON*, 2005.

Author Index