Gadi Taubenfeld (Ed.)

LNCS 5218

Distributed Computing

22nd International Symposium, DISC 2008
Arcachon, France, September 2008
Proceedings

Springer

Lecture Notes in Computer Science 5218

Gadi Taubenfeld (Ed.)

Distributed Computing

22nd International Symposium, DISC 2008
Arcachon, France, September 22-24, 2008
Proceedings

Springer

Volume Editor

Gadi Taubenfeld
School of Computer Science
The Interdisciplinary Center
P.O.Box 167
Herzliya 46150, Israel
E-mail: tgadi@idc.ac.il

Library of Congress Control Number: 2008935023

CR Subject Classification (1998): C.2.4, C.2.2, F.2.2, D.1.3, F.1.1, D.4.4-5

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

ISSN 0302-9743
ISBN-10 3-540-87778-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-87778-3 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12525812 06/3180 5 4 3 2 1 0

Preface

DISC, the International Symposium on Distributed Computing, is an annual forum for presentation of research on all aspects of distributed computing, including the theory, design, implementation and applications of distributed algorithms, systems and networks. The 22nd edition of DISC was held during September 22-24, 2008, in Arcachon, France.

There were 101 submissions submitted to DISC this year and this volume contains 33 15-page-long regular papers selected by the Program Committee among these submissions. Every submitted paper was read and evaluated by Program Committee members assisted by external reviewers. The final decisions regarding acceptance or rejection of each paper were made during the electronic Program Committee meeting held during June 2008. Revised and expanded versions of a few best selected papers will be considered for publication in a special issue of the journal *Distributed Computing*.

The Program Committee selected Robert Danek and Wojciech Golab as the recipients of this year's Best Paper Award for their paper "Closing the Complexity Gap Between FCFS Mutual Exclusion and Mutual Exclusion."The Program Committee selected Wojciech Wawrzyniak as the recipient of this year's Best Student Paper Award for the paper "Fast Distributed Approximations in Planar Graphs" coauthored with Andrzej Czygrinow and Michal Hańćkowiak.

This volume of the proceedings also contains 11 two-page-long brief announcements (BA). These BAs present ongoing work or recent results whose full description is not yet ready; it is expected that full papers containing these results will soon appear in other conferences or journals. The main purpose of the BA track is to announce ongoing projects to the distributed computing community and to obtain feedback for the authors. Each BA was also read and evaluated by the Program Committee.

The support of the sponsors, which are mentioned later, is gratefully acknowledged.

July 2008 Gadi Taubenfeld

The 2008 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing is awarded for an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade.

The Dijkstra Award Committee has selected Baruch Awerbuch and David Peleg as the recipients of this year's Edsger W. Dijkstra Prize in Distributed Computing. The prize is given to them for their outstanding paper: "Sparse Partitions" published in the proceedings of the 31st Annual Symposium on Foundations of Computer Science, pp. 503–513, 1990.

The "Sparse Partitions" paper by Awerbuch and Peleg signified the coming-of-age of the area of distributed network algorithms. A line of research that started with Awerbuch's synchronizer and Peleg's spanner has culminated in this ground-breaking paper that has had a profound impact on algorithmic research in distributed computing and in graph algorithms in general.

The paper presents concrete definitions of the intuitive concepts of locality and load, and gives surprisingly effective constructions to trade them off. The fundamental technical contribution in the paper is the algorithm of coarsening, which takes, as input, a decomposition of the graph to possibly overlapping components, and generates a new decomposition whose locality is slightly worse, but whose load is far better. The desired balance between locality and load is controlled by a parameter provided by the user. While many other underlying ideas were present in prior work of Awerbuch and Peleg (separately), in the "Sparse Partitions" paper these ideas have come together, with a unified view, resulting in a new powerful toolkit that is indispensable for all workers in the field.

The magnitude of the progress achieved by the new techniques was immediately recognized, and its implications spawn much research to this day. In the "Sparse Partitions" paper itself, the authors improve on the best known results for two central problems of network algorithms, and many other applications of the results followed, quite a few of them in applications that were visionary at their time. To mention just a few, these include computation of compact routing tables and location services of mobile users (in the original paper), dramatically more efficient synchronizers, effective peer-to-peer network design, and scheduling in grid-like computing models. Besides these applications of the results, the paper can be viewed as one of the important triggers to much of the fundamental research that was dedicated to exploring other variants of the basic concepts, including the notions of bounded-growth graphs, tree metrics, general and geometric spanners.

It is interesting to view the Sparse Partitions paper in a historical context. The area of network algorithms has its roots in classical graph algorithms. Distributed algorithms have proved to be an algorithmically rich field with the “Minimum Spanning Tree” paper of Gallager, Humblet and Spira. Motivated by the asynchronous nature of distributed systems, Awerbuch invented the concept of a synchronizer. Peleg, coming from the graph theoretic direction, generalized the notion of spanning tree and invented the concept of spanners. In the “Sparse Partitions” paper, the additional ingredient of load was added to the combination, yielding a powerful conceptual and algorithmic tool. The results superseded the best known results for classical graph algorithms, thus showing the maturity of the field, which closed a circle by becoming a leading source for graph algorithms of any kind.

Award Committee 2008:

Yehuda Afek	Tel-Aviv University
Faith Ellen	University of Toronto
Shay Kutten	Technion
Boaz Patt-Shamir	Tel-Aviv University
Sergio Rajsbaum	UNAM
Gadi Taubenfeld, Chair	IDC

Organization

DISC is an international symposium on the theory, design, analysis, implementation and application of distributed systems and networks. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biannual International Workshop on Distributed Algorithms on Graphs (WDAG). The scope was soon extended to cover all aspects of distributed algorithms as WDAG came to stand for International Workshop on Distributed AlGorithms, and in 1989 it became an annual symposium. To reflect the expansion of its area of interest, the name was changed to DISC (International Symposium on DIStributed Computing) in 1998. The name change also reflects the opening of the symposium to all aspects of distributed computing. The aim of DISC is to reflect the exciting and rapid developments in this field.

Program Committee Chair

Gadi Taubenfeld	IDC Herzliya, Israel

Organization Committee Chair

Cyril Gavoille	University of Bordeaux, France

Steering Committee Chair

Rachid Guerraoui	EPFL, Switzerland

Program Committee

James Anderson	UNC at Chapel Hill, USA
Bernadette Charron-Bost	Ecole Polytechnique, France
Cyril Gavoille	University of Bordeaux, France
Chryssis Georgiou	University of Cyprus, Cyprus

Phil Gibbons	Intel Research Pittsburgh, USA
Seth Gilbert	EPFL, Switzerland
Tim Harris	Microsoft Research Cambridge, UK
Danny Hendler	Ben Gurion University, Israel
Dariusz Kowalski	University of Liverpool, UK
Amos Korman	CNRS and University of Paris 7, France
Shay Kutten	Technion, Israel
Marios Mavronicolas	University of Cyprus, Cyprus
Michel Raynal	IRISA, University of Rennes, France
Luis Rodrigues	INESC-ID, IST, Portugal
Tami Tamir	IDC Herzliya, Israel
Gadi Taubenfeld (Chair)	IDC Herzliya, Israel
Sébastien Tixeuil	University of Paris 6, France
Philippas Tsigas	Chalmers University, Sweden
Mark Tuttle	Intel, USA
Roger Wattenhofer	ETH Zurich, Switzerland
Haifeng Yu	National University of Singapore, Singapore

Organization Committee

Cyril Gavoille (Chair)	University of Bordeaux, France
Nicolas Hanusse	CNRS, University of Bordeaux, France
David Ilcinkas	CNRS, University of Bordeaux, France
Ralf Klasing,	CNRS, University of Bordeaux, France
Michael Montassier	University of Bordeaux, France
Akka Zemmari	University of Bordeaux, France
Sergio Rajsbaum (publicity)	UNAM, Mexico

Steering Committee

Shlomi Dolev	BGU, Israel
Antonio Fernández	University Rey Juan Carlos, Spain
Rachid Guerraoui (Chair)	EPFL, Switzerland
Andrzej Pelc	University of Quebec, Canada
Sergio Rajsbaum	UNAM, Mexico
Nicola Santoro	Carleton University, Canada
Gadi Taubenfeld	IDC, Israel

External Reviewers

Ittai Abraham
Uri Abraham
Filipe Araujo
James Aspnes
Amos Beimel
Samuel Bernard
Nicolas Bonichon
Olivier Bournez
Nicolas Burri
Ran Canetti
Nuno Carvalho
Daniel Cederman
Wei Chen
Gregory Chokler
Andrzej Czygrinow
Paolo D'Arco
Shantanu Das
Gianluca De Marco
Roberto De Prisco
Bilel Derbel
Shlomi Dolev
Raphael Eidenbenz
Michael Elkin
Yuval Emek
Hugues Fauconnier
Antonio Fernandez
Paola Flocchini
Roland Flury
Felix Freiling
Zhang Fu
Eli Gafni
Leszek Gasieniec
Georgios Georgiadis
Anders Gidenstam
Olga Goussevskaia
Vincent Gramoli
Fabiola Greve
Phuong Ha
Nicolas Hanusse
David Hay
Ted Herman
Martin Hutle
David Ilcinkas
Sidharth Jaggi
Yuh-Jzer Joung
Erez Kantor
Jasleen Kaur
Ralf Klasing
Marina Kopeetsky
Adrian Kosowski
Evangelos Kranakis
Danny Krizanc
Ajay Ksemkalyani
Michael Kuhn
Arnaud Labourel
Emmanuelle Lebhar
João Leitão
Christoph Lenzen
Andrzej Lingas
Zvi Lotker
Olivier Ly
Urmi Majumder
Lior Malka
Dahlia Malkhi
Jean-Philippe Martin
Toshimistu Masuzawa
Remo Meier
Michael Merideth
Stephan Merz
Alessia Milani
Dorian Miller
Alan Mislove
Neeraj Mittal
José Mocito
Peter Musial
Mikhail Nesterenko
Nicolas Nicolaou
Tim Nieberg
Yvonne Anne Oswald
Charis Papadakis
Marina Papatriantafilou
Boaz Patt-Shamir
Andrzej Pelc
David Peleg
Ljubomir Perkovic
Maria Potop-Butucaru
Sergio Rajsbaum
Pascal von Rickenbach

Etienne Riviere
Paolo Romano
Adi Rosen
Matthieu Roy
Eric Ruppert
Johannes Schneider
Alex Shvartsman
Vasu Singh
Philipp Sommer
Håkan Sundell

Corentin Travers
Frédéric Tronel
Sara Tucci-Piergiovanni
Milan Vojnovic
Jennifer Welch
Josef Widder
Eric Winfree
Piotr Zielinski

Sponsors

CNRS

INRIA

LaBRI

REGION AQUITAINE

BORDEAUX 1

cost 293

ENSEIRB
ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

ASR
Architecture
Systèmes
Réseaux
GdR - CNRS
Pôle RésCom

Table of Contents

Regular Papers

Brief Announcements

The Mailbox Problem

(Extended Abstract)

Marcos K. Aguilera[1], Eli Gafni[1,2], and Leslie Lamport[1]

[1] Microsoft Research Silicon Valley
[2] UCLA

Abstract. We propose and solve a synchronization problem called the *mailbox problem*, motivated by the interaction between devices and processor in a computer. In this problem, a postman delivers letters to the mailbox of a housewife and uses a flag to signal a non-empty mailbox. The wife must remove all letters delivered to the mailbox and should not walk to the mailbox if it is empty. We present algorithms and an impossibility result for this problem.

1 Introduction

Computers typically use interrupts to synchronize communication between a processor and I/O devices. When a device has a new request, it raises an interrupt line to get the processor's attention. The processor periodically checks if the interrupt line has been raised and, if so, it interrupts its current task and executes an interrupt handler to process unhandled device requests. The interrupt line is then cleared so that it can be used when new requests come from the device. (This is a slight simplification, since there is typically an interrupt controller between the device and processor. In this case, we consider the interrupt controller as the "device" that interrupts the processor.) It is imperative that the processor eventually execute the interrupt handler if there are unhandled requests. Furthermore, it is desirable to avoid *spurious interrupts*, in which the processor executes the interrupt handler when there is no unhandled request. A closely related problem occurs in multi-threaded programming, in which the processor and the devices are separate threads and the interrupt is some type of software signal [8,10].

In this paper, we study a theoretical synchronization problem that arises from this setting, which we call the *mailbox problem*. From time to time, a postman (the device) places letters (requests) for a housewife (the processor) in a mailbox by the street.[1] The mailbox has a flag that the wife can see from her house. She looks at the flag from time to time and, depending on what she sees, may decide to go to the mailbox to pick up its contents, perhaps changing the position of the flag. The wife and postman can leave notes for one another at the mailbox. (The notes cannot be read from the house.) We require a protocol to ensure that (i) the wife picks up every letter placed in the mailbox and (ii) the wife never goes to the mailbox when it is empty (corresponding to a spurious interrupt). The protocol cannot leave the wife or the postman stuck at the mailbox, regardless of what the other does. For example, if the wife and postman are

[1] This problem originated long ago, when all mail was delivered by men and only women stayed at home.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 1–15, 2008.

both at the mailbox when the postman decides to take a nap, the wife need not remain at the mailbox until the postman wakes up. We do not require the wife to receive letters that are still in the sleeping postman's bag. However, we interpret condition (i) to require that she be able to receive mail left by the postman in previous visits to the mailbox without waiting for him to wake up.

The following simple protocol was once used in computers. The postman/device raises the flag after he delivers a letter/request; the wife/processor goes to the mailbox if the flag is raised and lowers the flag after emptying the mailbox. It is easy to see that this can cause a spurious interrupt if the postman goes to the mailbox while the flag is still raised from a previous visit and falls asleep after putting a letter in the box and before raising the flag.

There are obviously no spurious interrupts with this protocol if the postman can deliver mail to the box and raise the flag in an indivisible atomic action, while the wife can remove mail from the box and lower the flag in an indivisible atomic action. Moreover, the problem is solvable if the wife and postman can leave notes for one another, and the reading or writing of a note and the raising or lowering of the flag can be performed atomically. Here is a simple algorithm that uses a single note written by the postman and read by the wife. The postman stacks letters in delivery order in the box. After delivering his letters, the postman as a single action writes the total number of letters he has delivered so far on his note and raises the flag. When she sees the flag up, the wife as a single action lowers the flag and reads the postman's note. Then, starting from the bottom of the stack, the wife removes only enough letters so the total number she has ever removed from the box equals the value she read on the note.

What if a single atomic action can only either read or write a note or read or write a flag? Then, we show that there are no algorithms that use only two Boolean flags, one writable by the wife and one by the postman. However, perhaps surprisingly, there is a wait-free algorithm that uses two 14-valued flags, as we show. We do not know if there is an algorithm that uses smaller flags.

The mailbox problem is an instance of a general class of problems called *bounded-signaling problems*. We give a general algorithm for any problem in this class. The algorithm is non-blocking but not wait-free. It is an open problem whether there are general wait-free algorithms in this case.

The paper is organized as follows. We first define the mailbox problem in Section 2. In Section 3, we give a wait-free algorithm for the problem. To do so, we first explain the *sussus* protocol in Section 3.1. We then give a non-blocking algorithm that uses flags with large timestamps in Section 3.2. We show how to shrink these timestamps in Section 3.3. We then explain how to change the non-blocking algorithm into a wait-free algorithm in Section 3.4. In Section 4, we show that there are no non-blocking (or wait-free) algorithms that use only two Boolean flags. Next, we consider general bounded-signaling problems in Section 5. We describe related work in Section 6. Because of space limitations, most proofs are omitted from the paper.

2 Problem Definition

We now state the mailbox problem more precisely. For simplicity, we let only one letter at a time be delivered to or removed from the mailbox. It is easy to turn a solution to this problem into one in which multiple letters can be delivered or removed.

We assume a *postman* process and a *wife* process. There are three operations: the postman's *deliver* operation, the wife's *check* operation, which returns a Boolean value, and her *remove* operation. The postman can invoke the *deliver* operation at any time. The wife can invoke the *remove* operation only if the last operation she invoked was *check* and it returned TRUE. We describe the execution of these operations in terms of the mailbox metaphor—for example "checking the flag" means executing the *check* operation. Remember that *deliver* and *remove* respectively delivers and removes only a single letter.

Safety properties. We must implement *deliver*, *check*, and *remove* so that in every system execution in which the wife follows her protocol of checking and obtaining TRUE before removing, the following safety properties hold.

If the wife and postman never execute concurrently, then the value returned by an execution of *check* is TRUE if and only if there are more *deliver* than *remove* executions before this execution of *check*. This is the *sequential specification of safety*.

Neither the wife nor the postman can execute multiple operations concurrently, but the wife can execute concurrently with the postman. The allowable behaviors are specified by requiring that they act as if each operation were executed atomically at some point between its invocation and its completion—a condition known as linearizability [4].

Liveness properties. A process executes an operation by performing a sequence of atomic steps. A solution should also satisfy a liveness property stating that, under some hypothesis, a process's operation executions complete. We now state two possible liveness properties we can require of an algorithm. We number the two processes, letting the wife be process 0 and the postman be process 1. Thus, for each process number i, the other process number is $1-i$.

- *(Non-blocking)* For each i, if process i keeps taking steps when executing an operation, then either that operation execution completes or process $1-i$ completes an infinite number of operations.
- *(Wait-free)* For each i, every operation execution begun by process i completes if i keeps taking steps—even if process $1-i$ halts in the middle of an operation execution [3]. The algorithm is said to be *bounded wait-free* [3] or *loop-free* [6] if each operation completes before the process executing it has taken N steps, for some fixed constant N.

Process communication and state. A solution requires the two processes to communicate and maintain state. For that, processes have *shared variables*. We assume that there are two shared variables: *Flag* and *Notes*. It is desirable that *Flag* assume only a small number of values, but *Notes* can assume infinitely many values.

Operation *check* should be efficient: its execution should access a small amount of persistent state. We consider two alternative interpretations of this requirement:

- *(Weak access restriction)* Operation *check* accesses at most one shared variable, *Flag*, and it only accesses this variable by reading.
- *(Strong access restriction)* Operation *check* accesses at most one shared variable, *Flag*, it only accesses this variable by reading, and it returns a value that depends only on what it reads from *Flag*.

With the weak access restriction, $check$ can remember and use process-local state across its executions, while with the strong access restriction, $check$ is a memoryless operation that is a function of $Flag$ alone.

We are interested in solutions in which variables are atomic registers or arrays of atomic registers, and an atomic step can read or write at most one atomic register.

3 Algorithms

We now give a solution to the mailbox problem with the strong access restriction and, a fortiori, with the weak access restriction as well. It is easy to find such a solution if $Flag$ can hold an unbounded number of values. For example, we can use the algorithm mentioned in the introduction in which the postman writes his note and raises the flag in one atomic step, except having him write his note in $Flag$. We now present a solution in which $Flag$ is an array $Flag[0..1]$ with two single-writer atomic registers (a single-writer atomic register is an atomic register writable by a single process), each of which can assume only 14 values. We do not know if there is a solution that uses fewer values.

We explain our algorithm in several steps. We first present an auxiliary protocol in Section 3.1. Then, in Section 3.2, we give a solution to the mailbox problem that is non-blocking and uses flags with unbounded timestamps. In Section 3.3, we show how to bound the timestamps. Finally, we show how to make the algorithm wait-free in Section 3.4.

3.1 The *sussus* Protocol

The *sussus* protocol is defined in terms of an operation $sussus(v)$ that can be invoked at most once by each process i. Intuitively, when a process i invokes $sussus(v)$ with $v = v_i$, the process tries to communicate value v_i to the other process and learn any value communicated by the other process. The operation returns an outcome and a value to process i. This value is either $\perp$ or the value v_{1-i} with which the other process invokes $sussus$. The outcome is either *success* or *unknown*. A *success* outcome indicates that process i communicates its value successfully to the other process, provided the other process invokes operation $sussus$ and completes it. An *unknown* outcome indicates that process i does not know whether it communicates its value successfully. More precisely, the protocol is bounded wait-free and satisfies the following safety properties:

- *(SU1)* If both processes complete their operation execution,[2] then at least one obtains the outcome $success$.
- *(SU2)* For each i, if process i completes the operation execution before process $1-i$ invokes the operation, then process i obtains the outcome $success$.
- *(SU3)* For each i, if both processes complete the operation execution and process i obtains the outcome $success$, then process $1-i$ obtains the value v_i with which process i invoked the operation.

Figure 1 shows the sussus protocol, written in +CAL [7]. Procedure $sussus$ shows the code for operation $sussus$, while the code at the bottom shows an invocation to

[2] A process may not complete the operation execution if it stops taking steps.

```
     variables $A = [i \in 0\,..\,1 \mapsto \bot]$,
               $B = [i \in 0\,..\,1 \mapsto \bot]$;
      (* $A$ and $B$ are shared arrays indexed by $0\,..\,1$ with $A[i] = B[i] = \bot$ for each $i$ *)

     procedure $sussus(v)$                                (* output: $outcome$, $outvalue$ *)
     {
s1:    $A[self] := v$;                                    (* $self$ is the process id: 0 or 1*)
s2:    $outvalue := A[1 - self]$;
       if $(outvalue = \bot)$
         $outcome :=$ "success";                                                  (* Case A *)
       else {
s3:      $B[self] :=$ "done";
s4:      if $(B[1 - self] = \bot)$
             $outcome :=$ "unknown";                                              (* Case B *)
         else $outcome :=$ "success";                                             (* Case C *)
       };
s5:    return;
     };

     process $(Proc \in 0\,..\,1)$
       (* process−local variables *)
       variables $outcome$, $outvalue$;
     {
m1:    with $(v \in Int)$ { call $sussus(v)$; }
     }
```

Fig. 1. The *sussus* protocol

sussus with a value v chosen non-deterministically from the set *Int* of all integers. The outcome and value returned by operation *sussus* are placed in variables *outcome* and *outvalue*, respectively. Labels in $^{+}$CAL indicate the grain of atomicity: an atomic step consists of executing all code from one label to the next. In the first step of procedure *sussus*, process i sets array element $A[i]$ of shared variable A to value v. In the next step, process i reads $A[1-i]$ and stores the result in local variable *outvalue*. If the value read is $\bot$ then process i sets *outcome* to "success". Otherwise, in a third step, process i sets $B[i]$ to "done" and, in a fourth step, it reads $B[1-i]$; if the result is $\bot$, process i sets *outcome* to "unknown", otherwise it sets *outcome* to "success". Observe that each atomic step accesses at most one array element of one shared variable.

To see why the protocol satisfies properties SU1–SU3, observe that there are three possibilities for the values of variables *outcome* and *outvalue* when a process completes its operation:

$$\begin{array}{lll} \text{Case A:} & outcome = \text{“success”}, & outvalue = \bot \\ \text{Case B:} & outcome = \text{“unknown”}, & outvalue \neq \bot \\ \text{Case C:} & outcome = \text{“success”}, & outvalue \neq \bot \end{array}$$

These cases are indicated by comments in the code.

Figure 2 shows these cases as six pairs, where each pair $\langle i, \rho \rangle$ represents process i ending up in case ρ. Beneath each such pair, we indicate the outcome that process i

obtains, with S standing for $success$ and U for $unknown$. Two adjacent pairs indicate the results obtained by each process in some execution. For example, we see the adjacent pairs $\langle 1, B\rangle$ and $\langle 0, C\rangle$ and the letters U and S beneath them. This indicates that, in some execution, process 1 ends up in case B with outcome $unknown$, while process 0 ends up in case C with outcome $success$. It turns out that *every* execution in which both processes complete their execution of $sussus$ corresponds to some adjacent pair in the figure. It is easy to prove this by straightforward case analysis, and even easier by model checking the $^{+}$CAL code. Properties SU1–SU3 follow easily from this fact together with the observation that v_{1-i} is the only value other than $\perp$ that process i can possibly obtain. (Remember that each process invokes operation $sussus$ at most once.)

$$\begin{array}{cccccc} \langle 0, A\rangle & \langle 1, B\rangle & \langle 0, C\rangle & \langle 1, C\rangle & \langle 0, B\rangle & \langle 1, A\rangle \\ \text{S} & \text{U} & \text{S} & \text{S} & \text{U} & \text{S} \end{array}$$

Fig. 2. Possibilities when both processes complete execution of the *sussus* protocol

3.2 Non-blocking Algorithm with Large Flag Values

We now present a solution to the mailbox problem that is non-blocking and uses flags that keep large, unbounded timestamps. In this algorithm, the postman and wife each keep a private counter with the number of times that they have executed $deliver$ and $remove$, respectively. To deliver or remove a letter, a process increments its counter and executes a procedure to compare its counter with the other process's counter (see procedures $deliver$ and $remove$ in Figure 3). The comparison procedure is explained in detail below. Its effect is to write to $Flag[i]$ a record with two fields, Rel and $Timestamp$. Rel is either "=" or "≠", according to the result of the comparison. $Timestamp$ indicates how recent the result in Rel is; this information is used elsewhere to determine which of $Flag[0]$ or $Flag[1]$ has the most recent result.

The wife checks if the mailbox has letters or not by reading $Flag[0]$ and $Flag[1]$, choosing the flag with highest timestamp, and verifying if that flag says "=" or "≠". If it says "=" then the wife considers the mailbox to be empty, otherwise, to be non-empty (see procedure $check$ in Figure 3).

In the comparison procedure, a process i executes one or more rounds numbered $1, 2, \ldots$, starting with the smallest round it has not yet executed. In each round k, process i executes an instance of the *sussus* protocol to try to communicate the value of its counter and, possibly, learn the value of the other process's counter. If the outcome of *sussus* is *success*, process i compares its counter with the most recent value that it learned from the other process. The comparison result is written to $Flag[i]$ together with timestamp k, the process's current round. The process is now done executing the $compare$ procedure. If, on the other hand, the outcome of *sussus* is *unknown* then process i proceeds to the next round $k+1$. This continues until, in some round, the outcome of *sussus* is *success*.

The detailed code for the comparison procedure is shown in Figure 4. It invokes a multi-instance version of the *sussus* protocol in procedure $multisussus$, which is a

```
     variables                                                            (* shared variables *)
       $A = [k \in Int, i \in 0..1 \mapsto \bot]$,      (* A is an array indexed by the integers and $0..1$ *)
       $B = [k \in Int, i \in 0..1 \mapsto \bot]$,
       $Flag = [i \in 0..1 \mapsto [Timestamp \mapsto 0, Rel \mapsto$ "="]];   (* $Flag$ is an array of records with
                                         fields $Timestamp$ and $Rel$ initialized to 1 and "=" *)
     process $(proc \in 0..1)$
       variables                                                     (* process-local variables *)
         $counter = 0$,                                          (* # times removed/delivered *)
         $round = 0$,                                                (* current round number *)
         $otherc = 0$,                                   (* last known counter of other process *)
         $outcome$,                                       (* output of procedure $multisussus$ *)
         $outvalue$,                                      (* output of procedure $multisussus$ *)
         $hasmail$;                                            (* output of procedure $check$ *)
     {
m1:    while (TRUE) {
         if $(self = 0)$ {                                              (* wife-specific code *)
m2:        call $check()$;
m3:        if $(hasmail)$ call $remove()$;
         }
         else call $deliver()$;                                      (* postman-specific code *)
       } (* while *)
     }
     procedure deliver(){
d1:    $counter := counter + 1$;
d2:    call $compare(counter)$;
d3:    return;
     };
     procedure $remove()${
r1:    $counter := counter + 1$;
r2:    call $compare(counter)$;
r3:    return;
     };
     procedure $check()$                                                 (* output: $hasmail$ *)
       variables $t\_f0$, $t\_f1$;                              (* procedure-local variables *)
     {
c1:    $t\_f0 := Flag[0]$;
c2:    $t\_f1 := Flag[1]$;
c3:    if $(t\_f0.Timestamp > t\_f1.Timestamp)${
         if $(t\_f0.Rel =$ "=") $hasmail :=$ FALSE;
         else $hasmail :=$ TRUE;
       } else {
         if $(t\_f1.Rel =$ "=") $hasmail :=$ FALSE;
         else $hasmail :=$ TRUE;
       };
c4:    return;
     };
```

Fig. 3. Non-blocking algorithm with large flag values (1/2). Top: shared and global variable definitions. Middle: starting code. Bottom: procedures.

```
      procedure compare($c$)
      {
s1:     $outcome$ := “unknown”;
s2:     while ($outcome \neq$ “success”) {
          (* advance round *)
s6:       $round := round + 1$;
s7:       call multisussus($round, c$);
s8:       if ($outvalue \neq \bot$) {
            $otherc := outvalue$;                    (* remember $outvalue$ *)
          };
        }; (* while *)
s9:     if ($c \neq otherc$)
          $Flag[self] := [Timestamp \mapsto round, Rel \mapsto$ “$\neq$”];
        else $Flag[self] := [Timestamp \mapsto round, Rel \mapsto$ “=”];
s10:    return;
      };

      procedure multisussus($rnd, v$)          (* output: $outcome$ and $outvalue$ *)
      {
ss1:    $A[rnd, self] := v$;
ss2:    $outvalue := A[rnd, 1 - self]$;
ss3:    if ($outvalue = \bot$)
          $outcome$ := “success”;
        else {
ss4:      $B[rnd, self]$ := “done”;
ss5:      if ($B[rnd, 1 - self] = \bot$)
            $outcome$ := “unknown”;
          else $outcome$ := “success”;
        };
ss6:    return;
      };
```

Fig. 4. Non-blocking algorithm with large flag values (2/2)

trivial extension of the code in Figure 1. Shared variable *Notes*, used in the mailbox problem definition, is not shown in the code: for clarity, we replaced it with two shared variables, A and B. These variables should be regarded as fields $Notes.A$ and $Notes.B$ of *Notes*. Procedure *check* writes its return value to process-local variable *hasmail*, since in $^{+}$CAL, a procedure call has no mechanisms for returning a value.

Intuitively, the algorithm works because the rounds provide a way to order operation executions, ensuring linearizability. Roughly speaking, we can assign each operation execution to a round, as follows:

- An execution of *remove* or *deliver* by a process is assigned the first round in its execution in which the other process learns the process's value or the process obtains outcome *success* from *sussus*.

– An execution of $check$ is assigned the larger of the timestamps it reads from $Flag[0]$ and $Flag[1]$.

We now order operation executions according to their assigned round number. If two operation executions are assigned the same round number, we order $deliver$ before $remove$ before $check$ operations. This ordering ensures that if some operation execution op completes before another operation execution op' starts then op is ordered before op'. For example, if an execution of $deliver$ by the postman completes in round k then a subsequent execution of $remove$ by the wife cannot be assigned to round k or smaller. This is because it is impossible for the postman to learn the wife's new value in round k or smaller since the postman already executed them.

Theorem 1. *The algorithm in Figures 3 and 4 is a non-blocking algorithm that solves the mailbox problem with the strong access restriction.*

A fortiori, the algorithm is also a non-blocking algorithm that solves the mailbox problem with the weak access restriction.

3.3 Non-blocking Algorithm with Small Flag Values

We now give an algorithm that uses flags with small values. We do so by modifying the algorithm in the previous section, which uses unbounded timestamps, to use instead timestamps that assume only 7 different values.

In the new algorithm, as in the previous one, processes execute in (asynchronous) rounds. However, in the new algorithm, the timestamp that a process uses in round k is not k; it is a value chosen dynamically at the end of round $k-1$ according to what the process sees in that round.

Let $ts_{k,i}$ be the timestamp that process i uses in round k. To understand how $ts_{k,i}$ is chosen, we consider some properties that it must have. Let us assume that the *sussus* protocol in round k returns outcome *success* for process i—otherwise $ts_{k,i}$ does not get written to $Flag[i]$ and so it is irrelevant. In the previous algorithm of Section 3.2, $ts_{k,i}=k$. Such a timestamp has the property that it is larger than any timestamps from previous rounds. This is too strong a property to try to satisfy with bounded timestamps. However, closer inspection reveals that it is sufficient for $ts_{k,i}$ to be larger than previous-round timestamps that could appear in $Flag[1-i]$ at the same time that $ts_{k,i}$ appears in $Flag[i]$. It turns out that there are only two such timestamps: the timestamp already in $Flag[1-i]$ when process i ends round $k-1$, and the last timestamp learned by process i when process i ends round $k-1$. Thus, at the end of round $k-1$, process i needs to pick $ts_{k,i}$ so that it dominates these two timestamps.

Therefore, to bound the number of timestamps, we must choose them from a finite set TS with an antisymmetric total relation $\succeq$ such that, for any two elements $t_1, t_2 \in TS$, there is an element $s \in TS$ that strictly dominates both t_1 and t_2 under $\succeq$. This would be impossible if we required the relation $\succeq$ to be transitive, but we do not. A computer search reveals that the smallest set with the requisite relation $\succeq$ contains 7 elements. We take $TS = 1\,..\,7$ to be our 7-element set and define

$$\begin{aligned}
Array &\triangleq \langle\langle 1,0,1,1,1,0,0\rangle,\\
&\quad\ \langle 1,1,1,0,0,0,1\rangle,\\
&\quad\ \langle 0,0,1,0,1,1,1\rangle,\\
&\quad\ \langle 0,1,1,1,0,1,0\rangle,\\
&\quad\ \langle 0,1,0,1,1,0,1\rangle,\\
&\quad\ \langle 1,1,0,0,1,1,0\rangle,\\
&\quad\ \langle 1,0,0,1,0,1,1\rangle\rangle\\
v \succeq w &\triangleq (Array[v][w] = 1)\\
v \succ w &\triangleq v \succeq w \wedge v \neq w\\
dominate(v, w) &\triangleq \text{CHOOSE } x \in 1\,..\,7 : x \succ v \wedge x \succ w
\end{aligned}$$

Figures 5 and 6 shows the detailed code of the algorithm sketched above. Figure 5 is very similar to Figure 3. The significant changes to the algorithm are in Figure 6, where asterisks indicate a difference relative to Figure 4.

Theorem 2. *The algorithm in Figures 5 and 6 is a non-blocking algorithm that solves the mailbox problem with the strong access restriction. It uses a Flag with two 14-valued single-writer atomic registers.*

```
variables                                                    (* shared variables *)
  same as before except for this minor change:
    $Flag = [i \in 0..1 \mapsto [Timestamp \mapsto 1, Rel \mapsto \text{"="}]];$
process ($proc \in 0..1$)
  variables                                           (* process-local variables *)
    same as before, with the following additions
    $ts = 1,$                                                (* current timestamp *)
    $nextts = 2,$                                       (* next timestamp to use *)
    $otherts = 1,$                       (* last known timestamp of other process *)
{
    same as before
}
procedure $deliver()$  same as before
procedure $remove()$  same as before
procedure $check()$
  same as before, except replace
      if ($t\_f0.Timestamp > t\_f1.Timestamp$){
  with
      if ($t\_f0.Timestamp \succ t\_f1.Timestamp$) {
procedure $multisussus(rnd, v)$  same as before
```

Fig. 5. Non-blocking algorithm with small flag values (1/2). This part is very similar to Figure 3.

```
          procedure $compare(c)$
          {
     s1:    $outcome := $ "unknown";
     s2:    while ($outcome \neq$ "success") {
              (* advance round *)
     s6:      $round := round + 1$;
*             $ts := nextts$;                      (* use timestamp chosen at end of last round *)
*    s7:      call $multisussus(round, [Timestamp \mapsto ts, Count \mapsto c])$;
                             (* record with $Timestamp$ and $Count$ fields set to $ts$ and $c$ *)
     s8:      if ($outvalue \neq \perp$) {
*               $otherts := outvalue.Timestamp$;   (* remember timestamp of other process *)
*               $otherc := outvalue.Count$;          (* remember counter of other process *)
              };
*             $nextts := dominate(otherts, Flag[1 - self].Timestamp)$;   (* for next round *)
            }; (* while *)
     s9:    if ($c \neq otherc$)
*             $Flag[self] := [Timestamp \mapsto ts, Rel \mapsto$ "$\neq$"];      (* use $ts$ as timestamp *)
*           else $Flag[self] := [Timestamp \mapsto ts, Rel \mapsto$ "="];
     s10:   return;
          };
```

Fig. 6. Non-blocking algorithm with small flag values (2/2). Asterisks indicate changes relative to Figure 4.

3.4 Wait-Free Algorithm with Small Flag Values

The algorithms of Sections 3.2 and 3.3 are non-blocking but not wait-free, because a process completes a *deliver* or *remove* operation only when it obtains outcome *success* from the *sussus* protocol. Thus, if the process keeps getting outcome *unknown* in every round, the process never completes its operation. Closer examination reveals this could only happen with the wife, because of the way processes invoke operations: if the postman got stuck forever in a *deliver* execution, the wife would execute enough *remove* operations for the mailbox to be empty, which would cause her to stop invoking *remove* (since she invokes *remove* only if *check* returns TRUE), and this would allow the postman to eventually obtain outcome *success* and complete his operation.

Therefore, the algorithm fails to be wait-free only in executions in which the postman executes infinitely many *deliver* operations while the wife gets stuck executing *remove*. But there is a simple mechanism for the wife to complete her operation. Because the postman's counter is monotonically increasing, if the wife knows that the postman's counter is larger than her own, she can simply complete her operation and leave her flag unchanged, since her flag already indicates that her counter is smaller than the postman's — otherwise she would not be executing *remove* in the first place. This mechanism is shown in Figure 7 in the statement labeled "s3".

We have also included a simple optimization in which, if process i sees that its round r_i is lagging behind the other process's round r_{1-i}, then process i jumps to round $r_{1-i} - 1$. The reason it is possible to jump in this case is that process i will obtain

```
        $max(x, y) \triangleq$ IF $x > y$ THEN $x$ ELSE $y$

        **procedure** $compare(c)$
          **variables** $t\_round, t\_otherround$;
        {
   s1:    $outcome$ := “unknown”;
   s2:    **while** ($outcome \neq$ “success”) {
*  s3:      **if** ($self = 0 \wedge c < otherc$) **return**;            (* wife process *)

            (* advance or skip round *)
*  s4:      $t\_otherround := Round[1 - self]$;
*  s5:      $t\_round := max(Round[self] + 1, t\_otherround - 1)$;
*  s6:      $Round[self] := t\_round$;

            $ts := nextts$;
   s7:      **call** $multisussus(t\_round, [Timestamp \mapsto ts, Count \mapsto c])$;
   s8:      **if** ($outvalue \neq \bot$) {
              $otherts := outvalue.Timestamp$;
              $otherc := outvalue.Count$;
            };
            $nextts := dominate(otherts, Flag[1 - self].Timestamp)$;
          }; (* while *)
   s9:    **if** ($c \neq otherc$)
            $Flag[self] := [Timestamp \mapsto ts, Rel \mapsto$ “$\neq$”];
          **else** $Flag[self] := [Timestamp \mapsto ts, Rel \mapsto$ “=”];
   s10:   **return**;
        };
```

Fig. 7. Wait-free algorithm with small flag values: *compare* procedure. Asterisks indicate changes relative to the non-blocking algorithm with small flag values.

an outcome *unknown* from the *sussus* protocol in every round from r_i to $r_{1-i}-1$. In each of these rounds, the process would learn the value of the other process, but what it learns in a round is subsumed by what it learns in a higher round. Therefore, the process only needs to execute round $r_{1-i}-1$. This optimization is shown in Figure 7 in the statements labeled “s4” through “s7”. It uses an additional shared array $Round[i]$ that stores the current round of process i (this used to be in process-local variable $round$, which no longer is used), where initially $Round[i] = 0$ for $i = 0, 1$.

Theorem 3. *The algorithm in Figures 5 and 7 is a wait-free algorithm that solves the mailbox problem with the strong access restriction. It uses a Flag with two 14-valued single-writer atomic registers.*

4 Impossibility

We now show that it is impossible to solve the mailbox problem when *Flag* has only two bits, each writable by a single process. This result holds even if *Notes* can hold unbounded values.

Theorem 4. *There is no non-blocking algorithm that solves the mailbox problem with the strong access restriction when Flag is an array with two 2-valued single-writer atomic registers.*

Proof sketch. We show the result by contradiction: suppose there is such an algorithm $\mathcal{A}$. Let $\mathit{Flag}[0]$ and $\mathit{Flag}[1]$ denote the two 2-valued single-writer atomic registers. We show how to use $\mathcal{A}$ to solve consensus using only registers, which is impossible [2,9].

If $\mathit{Flag}[0]$ and $\mathit{Flag}[1]$ are writable by the same process, it is easy to get a contradiction. Without loss of generality we can assume $\mathit{Flag}[0]$ is writable by the wife (process 0) and $\mathit{Flag}[1]$ is writable by the postman (process 1).

A *solo execution* of an operation is one where only one process takes steps (the other does nothing).

We define a function C such that $C(F_0, F_1)$ is the value returned by a solo execution of check when $\mathit{Flag}[i] = F_i$ at the beginning of the execution. This is well-defined because (1) with the strong access restriction, operation check returns a value that depends only on what it reads from Flag, and (2) in a solo execution of check, the value of Flag does not change.

Assume without loss of generality that initially $\mathit{Flag}[0]{=}\mathit{Flag}[1]{=}0$.

Claim. $C(0,0){=}C(1,1){=}$FALSE *and* $C(0,1){=}C(1,0){=}$TRUE.

To show this claim, note that initially check returns FALSE as no letters have been delivered. Moreover, initially $\mathit{Flag}[0]{=}\mathit{Flag}[1]{=}0$. Therefore $C(0,0) =$ FALSE.

From the initial system state, a solo execution of $\mathit{deliver}$ by the postman must set $\mathit{Flag}[1]$ to 1 (otherwise a subsequent execution of check incorrectly returns $C(0,0) =$ FALSE) and we have $C(0,1) =$ TRUE.

After this solo execution of $\mathit{deliver}$, suppose there is a solo execution of remove by the wife. This execution sets $\mathit{Flag}[0]$ to 1 (otherwise a subsequent execution of check incorrectly returns $C(0,1) =$ TRUE) and we have $C(1,1) =$ FALSE.

After these solo executions of $\mathit{deliver}$ and remove, suppose there is a solo execution of $\mathit{deliver}$. Then, it sets $\mathit{Flag}[1]$ to 0 and we have $C(1,0) =$ TRUE. This shows the claim.

Let S be the system state after a solo execution of $\mathit{deliver}$ from the initial state. In state S, $\mathit{Flag}[0]{=}0$ and $\mathit{Flag}[1]{=}1$.

We now give an algorithm that we will show solves consensus for the two processes. Process i first writes its proposed value into a shared variable $V[i]$. Then, starting from state S, process 0 executes operation remove of algorithm $\mathcal{A}$ and process 1 executes operation $\mathit{deliver}$ of $\mathcal{A}$. If process i ends up with a different value in $\mathit{Flag}[i]$ than when it started, then it decides on the value of $V[0]$; otherwise, it decides on the value of $V[1]$.

This algorithm solves consensus because (a) if process 0 executes by herself then remove flips the value of $\mathit{Flag}[0]$ so the process decides on $V[0]$; (b) if process 1 executes by himself then $\mathit{deliver}$ leaves $\mathit{Flag}[1]$ unchanged so the process decides on $V[1]$; (c) if both processes execute then, after they finish, the values of $\mathit{Flag}[0]$ and $\mathit{Flag}[1]$ either both flip or both remain the same (it is not possible for only one of them to flip, because $C(0,0) = C(1,1) =$ FALSE and operation check must return TRUE afterwards), and so both processes decide the same value.

This consensus algorithm uses only atomic registers and it is wait-free since $\mathcal{A}$ is non-blocking and each process invokes at most one operation of $\mathcal{A}$. This contradicts the consensus impossibility result [2,9].

5 Bounded-Signaling Problems

The mailbox problem is an instance of a broader class of problems, called *bounded-signaling problems*, which we now define. In a bounded-signaling problem, each process $i = 0, 1$ has an input v_i that can vary. From time to time, a process wishes to know the value of a finite-range function $f(v_0, v_1)$ applied to the latest values of v_0 and v_1. Each input v_i could be unbounded and, when it varies, process i can access all of shared memory. However, when a process wishes to know the latest value of f, it is limited to accessing a small amount of state.

For example, in the mailbox problem, v_0 is the number of letters that the wife has removed, v_1 is the number of letters delivered by the postman, and $f(v_0, v_1)$ indicates whether $v_0 = v_1$ or $v_0 \neq v_1$. The mailbox problem places some problem-specific restrictions on how v_0 and v_1 can change. For instance, they are monotonically nondecreasing and $v_0 \leq v_1$ because if check returns FALSE then the wife does not execute remove. Other bounded-signaling problems may not have restrictions of this type.

A precise statement of a bounded-signaling problem is the following. We are given a finite-range function $f(x, y)$, and we must implement two operations, $\mathit{change}(v)$ and $\mathit{readf}()$. If operations never execute concurrently, readf must always return the value of $f(v_0, v_1)$ where v_i is the value in the last preceding invocation to $\mathit{change}(v)$ by process i or $v_i = \perp$ if process i never invoked $\mathit{change}(v)$. The concurrent specification is obtained in the usual way from this condition by requiring linearizability. Furthermore, the implementation of readf must access a small amount of persistent state. We consider two alternative interpretations of this requirement:

- *(Weak access restriction)* Operation readf accesses at most one shared variable, of finite range; and it accesses this variable only by reading.
- *(Strong access restriction)* Operation readf accesses at most one shared variable, of finite range; it accesses this variable only by reading; and it returns a value that depends only on what it reads from the shared variable.

It turns out that the algorithm in Section 3.3 can be changed as follows to solve any bounded-signaling problem with the strong access restriction. We replace $\mathit{deliver}$ and remove with a single procedure $\mathit{change}(v)$ that sets $\mathit{counter}$ to v, and we modify the end of procedure $\mathit{compare}$ to compute f with arguments c and otherc (instead of just comparing c and otherc), and write the result and timestamp to Flag. The resulting algorithm is non-blocking. It is an open problem whether there exist wait-free algorithms for the general problem. Our wait-free algorithm in Section 3.4 does not solve the general problem since it relies on problem-specific restrictions on the inputs v_i.

6 Related Work

The mailbox problem is a type of consumer-producer synchronization problem, with the unique feature that the consumer must determine if there are items to consume by looking only at a finite-range variable.

Work on bounded timestamping shows how to bound the timestamps used in certain algorithms (e.g., [5,1]). That work considers a fixed-length array that holds some finite set of objects that must be ordered by timestamps. In our algorithms, it is not evident what this set should be. However, we believe some of the binary relations devised in that body of work could be used in our algorithms instead of the relation given by *Matrix* in Section 3.3 (but this would result in much larger timestamps than the ones we use).

Acknowledgements. We are grateful to Ilya Mironov for pointing out to us that the relation of Section 3.3 should exist for sufficiently large sets, and to the anonymous reviewers for useful suggestions.

References

1. Dolev, D., Shavit, N.: Bounded concurrent time-stamping. SIAM Journal on Computing 26(2), 418–455 (1997)
2. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J.ACM 32(2), 374–382 (1985)
3. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
4. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
5. Israeli, A., Li, M.: Bounded time-stamps. Distributed Computing 6(4), 205–209 (1993)
6. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM 17(8), 453–455 (1974)
7. Lamport, L.: The +CAL algorithm language (July 2006), http://research.microsoft.com/users/lamport/tla/pluscal.html (The page can also be found by searching the Web for the 25-letter string obtained by removing the "-" from uid-lamportpluscalhomepage)
8. Lampson, B.W., Redell, D.D.: Experience with processes and monitors in Mesa. Communications of the ACM 17(8), 453–455 (1974)
9. Loui, M.C., Abu-Amara, H.H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research 4, 163–183 (1987)
10. Saltzer, J.H.: Traffic control in a multiplexed computer system. Technical Report Project MAC Technical Report MAC-TR-30, M.I.T (June 1966)

Matrix Signatures: From MACs to Digital Signatures in Distributed Systems

Amitanand S. Aiyer[1], Lorenzo Alvisi[1,⋆], Rida A. Bazzi[2], and Allen Clement[1]

[1] Department of Computer Sciences,
University of Texas at Austin
[2] School of Computing and Informatics,
Arizona State University

Abstract. We present a general implementation for providing the properties of digital signatures using MACs in a system consisting of any number of untrusted clients and n servers, up to f of which are Byzantine. At the heart of the implementation is a novel *matrix signature* that captures the collective knowledge of the servers about the authenticity of a message. Matrix signatures can be generated or verified by the servers in response to client requests and they can be transmitted and exchanged between clients independently of the servers. The implementation requires that no more than one third of the servers be faulty, which we show to be optimal. The implementation places no synchrony requirements on the communication and only require fair channels between clients and servers.

1 Introduction

Developing dependable distributed computing protocols is a complex task. Primitives that provide strong guarantees can help in dealing with this complexity and often result in protocols that are simpler to design, reason about, and prove correct. Digital signatures are a case in point: by guaranteeing, for example, that the recipient of a signed message will be able to prove to a disinterested third party that the signer did indeed sign the message (*non repudiation*), they can discourage fraudulent behavior and hold malicious signers to their responsibilities. Weaker primitives such as message authentication codes (MACs) do not provide this desirable property.

MACs, however, offer other attractive theoretical and practical features that digital signatures lack. First, in a system in which no principal is trusted, it is possible to implement MACs that provide unconditional security—digital signatures instead are only secure under the assumption that one-way functions exist [1], which, in practical implementations, translates in turn to a series of unproven assumptions about the difficulty of factoring, the difficulty of computing discrete logarithms, or both. Second, certain MAC implementations (though not the ones that guarantee unconditional security!) can be three orders of magnitude faster to generate and verify than digital signatures [2].

[⋆] This work was supported in part by NSF grant CSR-PDOS 0720649.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 16–31, 2008.

Given these rather dramatic tradeoffs, it is natural to wonder whether, under different assumptions about the principals, it is possible to get the best of both worlds: a MAC-based implementation of digital-signatures. It is relatively easy to show that such an implementation exists in systems with a specific trusted entity [3]—in the absence of a specific trusted entity, however, the answer is unknown.

The few successful attempts to date at replacing digital signatures with MACs [2,4,5,6,7] have produced solutions specific only to the particular protocols for which the implementation was being sought—these MAC-based, signature-free protocols do not offer, nor seek to offer, a generic mechanism for transforming an arbitrary protocol based on digital signatures into one that uses MACs. Further, these new protocols tend to be significantly less intuitive than their signature-based counterparts, so much so that their presentation is often confined to obscure technical reports [2,8].

In this paper we study the possibility of implementing digital signatures using MACs in a system consisting of any number of untrusted clients and n servers, up to f of which can be Byzantine. We show that, when $n > 3f$, there exists a general implementation of digital signatures using MACs for asynchronous systems with fair channels. We also show that such an implementation is not possible if $n \leq 3f$—even if the network is synchronous and reliable.

At the heart of the implementation is a novel *matrix signature* that captures the collective knowledge of the servers about the authenticity of a message. Matrix signatures can be generated or verified by the servers in response to client requests and they can be transmitted and exchanged between clients independently of the servers.

Matrix signatures do not qualify as *unique signature schemes* [9]. Depending on the behavior of the Byzantine servers and message delivery delays, the same message signed at different times can produce different signatures, all of which are admissible. Unique signature schemes have a stronger requirement: for every message, there is a unique admissible signature. We show that unique signature schemes can also be implemented using MACs, but that any such implementation requires an exponential number of MACs if f is a constant fraction of n.

In summary, we make four main contributions:

- We introduce matrix signatures, a general, protocol-agnostic MAC-based signature scheme that provides properties, such as non-repudiation, that so far have been reserved to digital signatures.
- We present an optimally resilient implementation of a signing and verification service for matrix signatures. We prove its correctness under fairly weak system assumptions (asynchronous communication and fair channels) as long as at most one third of the servers are arbitrarily faulty.
- We show that no MAC based signature and verification service can be implemented using fewer servers, even under stronger assumptions (synchronous communication and reliable channels).
- We provide an implementation of unique signatures, and show a bound on the number of MACs required to implement them.

2 Related Work

Matrix signatures differ fundamentally from earlier attempts at using MACs in lieu of signatures by offering a general, protocol-agnostic translation mechanism.

Recent work on practical Byzantine fault tolerant (BFT) state machine replication [2,4,5,6] has highlighted the performance opportunities offered by substituting MACs for digital signatures. These papers follow a similar pattern: they first present a relatively simple protocol based on digital signatures and then remove them in favor of MACs to achieve the desired performance. These translations, however, are protocol specific, produce protocols that are significantly different from the original—with proofs of correctness that require understanding on their own— and, with the exception of [2], are incomplete.

[10] addresses the problem of allowing Byzantine readers to perform a write back without using digital signatures; however, it uses secret-sharing and relies on having a trusted writer.

Srikanth and Toueg [7] consider the problem of implementing *authenticated broadcast* in a system where processes are subject to Byzantine failures. Their implementation is applicable only to a closed system of $n > 3f$ processes, with authenticated pairwise communication between them. They do not consider the general problem of implementing signatures: in their protocol, given a message one cannot tell if it was "signed" unless one goes through the history of all messages ever received to determine whether the message was broadcast—an impractical approach if signed messages are persistent and storage is limited. In contrast, we provide signing and verification primitives for an open asynchronous system with any number of clients.

Mechanisms based on unproven number theoretic assumptions, are known to implement digital signatures using local computation without requiring any communication steps [11,12]. Some also provide unconditional security [13]; but, they bound the number of possible verifiers and allow for a small probability that a verifier may be unable to convince other verifiers that the message was signed.

If there is a trusted entity in the system signatures can be implemented over authenticated channels (or MACs) [3]. In the absence of a known trusted principal, implementing digital signatures locally requires one-way functions [1]. Our results show that even with partial trust in the system, implementing digital signatures is possible without requiring one-way functions.

3 MACs and Digital Signatures

Digital Signatures and MACs allow a message recipient to establish the authenticity of a message. Unlike MACs, digital signatures also allow a message recipient to prove this authenticity to a disinterested third party [14]—non repudiation.

3.1 Digital Signatures

A signature scheme over a set of signers S and a set of verifiers V consists of a signing procedure $\mathcal{S}_{S,V}$ and a verification procedure $\mathcal{V}_{S,V}$:

$$\mathcal{S}_{S,V} : \Sigma^* \mapsto \Sigma^* \qquad\qquad \mathcal{V}_{S,V} : \Sigma^* \times \Sigma^* \mapsto \mathbf{Boolean} \times \Sigma^*$$

The signing procedure $\mathcal{S}_{S,V}$ is used to sign a message. It outputs a signature, which can convince the verifier that the message was signed.

The set S contains all the processes that can invoke the signing procedure. The set V contains all processes that may verify a signature in the signature scheme.

The verification procedure, $\mathcal{V}_{S,V}$, takes as input a message and a signature and outputs two values. The first value is a boolean and indicates whether the verification procedure *accepts* or *rejects* the signature. The second value is a signature, whose role needs some explaining.

The signature schemes we define guarantee that (i) a verifier always accepts signatures that are generated by invoking the signing procedure and that (ii) any message whose signature is accepted was, at some point, signed by a member of S by invoking the signing procedure *although the signature that the verifier accepts may not be the one produced by the signing procedure.* We call these second type of signatures *derivative.*

In traditional, non-distributed, implementations of signatures, one does not expect that a verifier be presented with a *derivative* signature that was not explicitly generated by the signing procedure. In a distributed implementation, and for reasons that will become clear in Section 6, when we discuss the actions that Byzantine nodes can take to disrupt a MAC-based signature scheme, the existence of derivative signatures is the norm rather than the exception, and one needs to allow them in a definition of signature schemes. Furthermore, because the non-deterministic delays and Byzantine behavior of faulty servers, there exist derivative signatures that may nondeterministically be accepted or rejected by the verification procedure. It may then be impossible for a verifier who accepted the signature to prove to another the authenticity of a message.

So, from the perspective of ensuring non repudiation, derivative signatures present a challenge. To address this challenge, we require the verification procedure, every time a verifier v accepts a signed message m, to produce as output a new derivative signature that, by construction, is guaranteed to be accepted by all verifiers. This new signature can then be used by v to authenticate the sender of m to all other verifiers. Note that, if the first output value produced by the verification procedure is **false**, the second output value is irrelevant.

Digital signature schemes are required to satisfy the following properties:

Consistency. A signature produced by the signing procedure is accepted by the verification procedure.

$$\mathcal{S}_{S,V}(msg) = \sigma \qquad \Rightarrow \qquad \mathcal{V}_{S,V}(msg, \sigma) = (true, \sigma')$$

Validity. A signature for a message m that is accepted by the verification procedure cannot be generated unless a member of S has invoked the signing procedure.

$$\mathcal{V}_{S,V}(msg, \sigma) = (true, \sigma') \qquad \Rightarrow \qquad \mathcal{S}_{S,V}(msg) \text{ was invoked}$$

Verifiability. If a signature is accepted by the verification procedure for a message m, then the verifier can produce a signature for m that is guaranteed to be accepted by the verification procedure.

$$\mathcal{V}_{S,V}(msg, \sigma) = (true, \sigma') \qquad \Rightarrow \qquad \mathcal{V}_{S,V}(msg, \sigma') = (true, \sigma")$$

Verifiability is recursively defined; it ensures non-repudiation. If the verification procedure accepts a signature for a given message, then it outputs a signature that is accepted by the verification procedure for the same message. In turn, the output signature can be used to obtain another signature that will be accepted by the verification procedure and so on.

Any digital signature scheme that meets these requirements provides the general properties expected of signatures. Consistency and validity provide authentication; verifiability provides non-repudiation.

Unique Signature Schemes. Unique signature schemes are signature schemes for which only one signature can be accepted by the verification procedure for a given message. If $(\mathcal{S}_{S,V}, \mathcal{V}_{S,V})$ is a unique signature scheme, then, in addition to *consistency, validity and verifiability*, it satisfies:

$$\mathcal{V}(msg, \sigma) = (true, \sigma_{proof}) \ \wedge \ \mathcal{V}(msg, \sigma') = (true, \sigma'_{proof}) \qquad \Rightarrow \qquad \sigma = \sigma'$$

It follows from the definition that $\sigma_{proof} = \sigma'_{proof} = \sigma = \sigma'$, implying that, for unique signatures, the signature produced in output by the verification procedure is redundant . It also follows from the definition and the consistency requirement that unique signatures have deterministic signing procedures.

3.2 Message Authentication Codes

MACs are used to implement authentication between processes. A message authentication scheme consists of a signing procedure $\mathcal{S}_U$ and a verifying procedure $\mathcal{V}_U$.

$$\mathcal{S}_U : \Sigma^* \mapsto \Sigma^* \qquad\qquad \mathcal{V}_U : \Sigma^* \times \Sigma^* \mapsto \textbf{Boolean}$$

The signing procedure $\mathcal{S}_U$ takes a message and generates a MAC for that message. The verification procedure $\mathcal{V}_U$ takes a message along with a MAC and checks if the MAC is valid for that message. For a given MAC scheme, the set U contains all processes that can generate and verify MACs for the scheme.

MACs are required to ensure authentication, but not non-repudiation. Formally, they are required to satisfy:

Consistency. A MAC generated by the signing procedure will be accepted by the verifying procedure.

$$\mathcal{S}_U(msg) = \mu \qquad \Rightarrow \qquad \mathcal{V}_U(msg, \mu) = \textbf{true}$$

Validity. A MAC for a message m that is accepted by the verification procedure cannot be generated unless a member of U has invoked the signing procedure.

$$\mathcal{V}_U(msg, \mu) = \textbf{true} \qquad \Rightarrow \qquad \mathcal{S}_U(msg) \text{ was invoked}$$

3.3 Discussion

Keys, Signatures, and MACs. Formal definitions of signature schemes typically include signing and verification keys. In our work we omit the keys for simplicity and assume they are implicitly captured in $\mathcal{S}_{S,V}$ and $\mathcal{V}_{S,V}$. In our setting, S is be the set of processes that know the key needed to sign and V is the set of processes that know the key needed to verify.

MACs are also typically defined with reference to a symmetric secret key K that is used to generate and verify MACs. In our setting, processes that know K are members of the set U of signers and verifiers. In proving a lower bound on the number of MAC schemes needed to implement unique signatures, we find it convenient to identify a MAC scheme with the key K it uses. In this case, we distinguish between the name of the key, K, and the value of the key k as different schemes might use the same key value.

Signers and Verifiers. Since we will be considering Byzantine failures of servers and clients (participants), the composition of the sets S or U for a given scheme might change because a participant can give the secret signing key to another participant. To simplify the exposition, we assume that the sets of signers (verifiers) include any participant that can at some point sign (verify) a message according to the scheme.

Semantics. Formalisms for MACs and digital signatures typically express their properties in terms of probabilities that the schemes can fail. For schemes that rely on unproven assumptions, restrictions are placed on the computational powers of the adversary. In this paper we are only interested in implementing signature using a finite number of black box MAC implementations. We state our requirement in terms of properties of the executions that always hold without reference to probabilities or adversary powers. This does not affect the results, but leads to a simpler exposition.[1]

4 Model

The system consists of two sets of processes: a set of n server processes (also known as replicas) and a finite set of client processes (signers and verifiers). The set of clients and servers is called the set of *participants*. The identifiers of participants are elements of a completely ordered set.

An execution of a participant consists of a sequence of events. An event can be an internal event, a message send event or a message receive event. Two particular internal events are of special interest to us. A *message signing event* invokes a

[1] Our implementations use only finitely many MACs, consequently the probability of breaking our implementation can be made arbitrarily small if the probability of breaking the underlying MAC implementations can be made arbitrarily small. Also, our requirements restrict the set of allowable executions, which in essence place a restriction on the computational power of the verifiers. In particular, they do not allow a verifier to break the signature scheme by enumerating all possible signatures and verifying them.

signing procedure. A *message verification event* is associated with the invocation of a verification procedure. In our implementations of signature schemes we only consider communication between clients and servers to implement the signing and the verification procedures.

Clients communicate with the servers over authenticated point-to-point channels. Inter-server communication is not required. The network is asynchronous and fair—but, for simplicity, our algorithms are described in terms of reliable FIFO channels, which can be easily implemented over fair channels between correct nodes.

Each process has an internal state and follows a protocol that specifies its initial states, the state changes, and the messages to send in response to messages received from other processes. An arbitrary number of client processes and up to f of the server processes can exhibit arbitrary (Byzantine) faulty behavior [15]. The remaining processes follow the protocol specification.

5 Signatures Using MACs

We first present the high level idea assuming two trusted entities in the system. One trusted entity acts as a signing-witness and one acts as a verifying-witness. The two witnesses share a secret-key $\mathcal{K}$ that is used to generate and verify MACs.

Signing a message. A signer delegates to the signing witness the task of signing a message. This signing witness generates, using the secret key K, a MAC value for the message m to be signed and sends the MAC value to the signer. This *MAC-signature* certifies that the signer s wants to sign m. It can be presented by a verifier to the verifying-witness to validate that s has signed m.

Verifying a signature. To verify that a MAC-signature is valid, a verifier (client) delegates the verification task to the verifying witness. The verifying witness computes, using the secret key K, the MAC for the message and verifies that it is equal to the MAC presented by the verifier. If it is, the signature is accepted otherwise, it is rejected.

Since the two witnesses are trusted and only they know the secret key K, this scheme satisfies *consistency*, *validity* and *verifiability*.

6 A Distributed Signature Implementation

In an asynchronous system with $n \geq 3f + 1$ servers, it is possible to delegate the tasks of the signing witness and the verifying witness to the servers. However, achieving non-repudiation is tricky.

6.1 An Illustrative Example: Vector of MACs

Consider a scheme, for $n = 3f + 1$, where each server i has a secret key K_i is used to generate/verify MACs. The "signature" is a vector of n MACs, one for each server.

$$\boxed{\begin{array}{cccc} h_{1,1} & h_{1,2} & \mathbf{h_{1,3}} & h_{1,4} \\ \mathbf{h_{2,1}} & \mathbf{h_{2,2}} & \mathbf{h_{2,3}} & \mathbf{h_{2,4}} \\ h_{3,1} & h_{3,2} & \mathbf{h_{3,3}} & h_{3,4} \\ h_{4,1} & h_{4,2} & \mathbf{h_{4,3}} & h_{4,4} \end{array}}$$

A Matrix-signature

$$\boxed{\begin{array}{cccc} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{array}}$$

Valid Signature

$$\boxed{\begin{array}{cccc} ? & h_{1,2} & ? & ? \\ ? & h_{2,2} & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{array}}$$

Admissible Signature

Fig. 1. Example Matrix-signatures

To sign a message, the signer contacts the servers to collect the MACs. However, due to asynchrony, it cannot collect more than $(n - f) = (2f + 1)$ MACs.

To verify a signature, the verifier contacts the servers to determine which MACs are correct. Since, up to f Byzantine nodes could have sent wrong values to the signer, ensuring *consistency* requires that the verifier accept the "signature" even if only $f + 1$ MAC are accepted by the servers.

This allows the adversary to fool a non-faulty verifier into accepting a vector that contains only one correct MAC value. If that happens, the verifier will not be able to convince other verifiers that the message was signed.

6.2 Matrix Signatures

To deal with the difficulties raised in the illustrative example, we propose *matrix signatures.* A matrix signature consists of n^2 MAC values arranged in n rows and n columns, which together captures the servers' collective knowledge about the authenticity of a message.

There are n signing-witness-servers and n verifying-witness-servers; both implemented by the same n servers. Each MAC value in the matrix is calculated using a secret key $K_{i,j}$ shared between a signing-witness-server i and a verifying-witness-server j.[2]

The i^{th} row of the matrix-signature consists of the MACs generated *by* the i^{th} signing-witness-server. The j^{th} column of the matrix-signature consists of the MACs generated *for* the j^{th} verifying-witness-server.

Clients can generate (or verify) a signature by contacting all the signing-witness (or, respectively, verifying-witness) servers. The key difference with the protocol described in the previous section is that the signature being used is a matrix of $n \times n$ MACs as opposed to a single MAC value.

We distinguish between *valid* and *admissible* matrix signatures:

Definition 1 (Valid). *A matrix-signature is valid if it has at least* $(f + 1)$ *correct MAC values in every column.*

Definition 2 (Admissible). *A matrix-signature is said to be admissible if it has at least one column corresponding to a non-faulty server that contains at least* $(f + 1)$ *correct MAC values.*

[2] Although signing-witness-server i and verifying-witness-server k are both implemented by server i, for the time being, it is useful to think of them as separate entities.

Admissibility and validity respectively capture the necessary and sufficient conditions required for a matrix-signature to be successfully verified by a non-faulty verifier. Thus, every *valid* signature is *admissible*, but the converse does not hold.

6.3 Protocol Description

The protocol for generating and verifying matrix-signatures is shown in Figure 2.

Generating a Signature. To generate a matrix-signature, the signer s sends the message *Msg* to be signed, along with its identity, to all the signing-witness-servers over *authenticated channels*. Each signing-witness-server generates a row of MACs, attesting that s signs *Msg*, and responds to the signer. The signer waits to collect the MAC-rows from at least $(2f + 1)$ signing-witness-servers to form the matrix-signature.

The matrix-signature may contain some empty rows corresponding to the unresponsive/slow servers. It may also contain up to f rows with incorrect MAC values, corresponding to the faulty servers.

Verifying a Signature. To verify a matrix-signature the verifier sends (a) the matrix-signature, (b) the message, and (c) the identity of the client claiming to be the signer to the verifying-witness-servers. A verifying-witness-server admits the signature only if at least $(f + 1)$ MAC-values in the server's column are correct; otherwise, it rejects. Note that a non-faulty server will never reject a valid matrix-signature.

The verifier collects responses from the servers until it either receives $(2f+1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses to accept the signature, or it receives $(f+1)$ $\langle \text{REJECT} \rangle$ responses to reject the signature as not *valid*.

Regenerating a valid signature. Receiving $(2f+1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses does not guarantee that the signature being verified is *valid*. If some of these responses are from Byzantine nodes, the same signature could later fail the verification if the Byzantine nodes respond differently.

Verifiability requires that that if a signature passes the verification procedure, then the verifier gets a signature that will always pass the verification procedure. This is accomplished by constructing a new signature, that is a *valid* signature.

Each witness-server acts both as a verifying-witness-server and a signing-witness-server. Thus, when a witness-server admits a signature (as a verifying-witness-server), it also re-generates the corresponding row of MAC-values (as a signing-witness-server) and includes that in the response. Thus, if a verifier collects $(2f + 1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses, it receives $(2f + 1)$ rows of MAC-values, which forms a *valid* signature.

Ensuring termination. The verifier may receive $(n - f)$ responses and still not have enough admit responses or enough reject responses to decide. This can happen if the signature being verified, σ, is maliciously constructed such that some of the columns are bad. This can also happen if the signature σ is *valid*, but some non-faulty servers are slow and Byzantine servers, who respond faster, reject it.

```
Signature Client-Sign (Msg $Msg$) {
  $\forall i : \sigma_{Msg,S}[i][\,] := \perp$
  send $\langle$SIGN, $Msg, S\rangle$ to all
  do {
    // Collect MAC-rows from the servers
    rcv$\langle\sigma_i[1 \dots n]\rangle$ from server $i$
    $\sigma_{Msg,S}[i][1 \dots n] := \sigma_i[1 \dots n]$
  } until received from $\geq 2f+1$ servers
  return $\sigma_{Msg,S}$
}

(bool, Signature) Client-Verify(Msg $Msg$,
                Signer $S$, Signature $\sigma$) {
  $\forall i : \sigma_{new}[i][\,] := \perp$;   $\forall i$ : resp[$i$] $:= \perp$;
  send $\langle$VERIFY, $Msg, S, \sigma[\,][\,]\rangle$ to all
  do {
    either {
      rcv $\langle$ADMIT, $\sigma_i[1 \dots n]\rangle$ from server $i$
      $\sigma_{new}[i][1 \dots n] := \sigma_i[1 \dots n]$
      resp[$i$] := ADMIT
      if ( Count(resp, ADMIT) $\geq 2f+1$ )
        return (true, $\sigma_{new}$);
    } or {
      rcv $\langle$REJECT$\rangle$ from server $i$
      if (resp[$i$] $= \perp$) { resp[$i$] := REJECT }
      if ( Count(resp, REJECT) $\geq f+1$ )
        return (false, $\perp$);
    };
    // If can neither decide, nor wait – Retry
    if (Count(resp, $\perp$) $\leq f$)
      send $\langle$VERIFY, $Msg, S, \sigma_{new}[\,][\,]\rangle$ to
                { $r$ : resp[$r$] $\neq$ ADMIT}
  } until (false)
}
```

```
void Signing-Witness-Server(Id $i$) {
  while(true) {
    rcv $\langle$SIGN, $Msg, S\rangle$ from $S$
    $\forall j$ : compute $\sigma_i[j]$ := MAC($\mathcal{K}_{i,j}$, $S : Msg$)
    send $\langle\sigma_i[1 \dots n]\rangle$ to $S$
  }
}

void Verifying-Witness-Server(Id $j$) {
  while(true) {
    rcv $\langle$VERIFY, $Msg, S, \sigma\rangle$ from $V$
    correct_cnt := $|\{i : \sigma[i][j] ==$
                MAC($\mathcal{K}_{i,j}$, $S : Msg$)$\}|$
    if (correct_cnt $\geq f+1$)
      $\forall l$ : compute $\sigma_j[l]$ := MAC($\mathcal{K}_{j,l}$, $S : Msg$)
      send $\langle$ADMIT, $\sigma_j[1 \dots n]\rangle$ to $V$
    else
      send $\langle$REJECT$\rangle$ to $V$
  }
}
```

Fig. 2. Matrix-signatures

To ensure that the verifier gets $(2f+1)$ $\langle$ADMIT, …$\rangle$ responses it retries by sending σ_{new}, each time σ_{new} is updated, to all the servers that have not sent an $\langle$ADMIT, …$\rangle$ response. Eventually, it either receives $(f+1)$ $\langle$REJECT$\rangle$ responses from different servers (which guarantees that σ was not *valid*), or it receives $(2f+1)$ $\langle$ADMIT, …$\rangle$ responses (which ensures that the regenerated signature, σ_{new}, is *valid*).

6.4 Correctness

Algorithm described in Figure 2 for matrix-signatures satisfies all the requirements of digital signatures and ensure that the signing/verification procedures always terminate for $n \geq 3f+1$ [16].

Lemma 1. *Matrix-signature generated by the signing procedure (Fig 2) is valid.*

Lemma 2. *Valid signature always passes the verification procedure.*

Proof. A valid signature consists of all correct MAC-values in at least $(f+1)$ rows. So, no non-faulty server will send a $\langle$REJECT$\rangle$ message. When all non-faulty servers respond, the verifier will have $(2f+1)$ $\langle$ADMIT, …$\rangle$ messages.

Lemma 3. *If a matrix-signature passes the verification procedure for a non-faulty verifier, then it is admissible.*

Lemma 4. *An adversary cannot generate an admissible signature for a message Msg, for which the signer did not initiate the signing procedure.*

Proof. Consider the first non-faulty server (say j) to generate a row of MACs for the message Msg for the first time. If the signer has not initiated a the signing procedure then j would generate the row of MACs only if it has received a signature that has at least $f + 1$ correct MAC values in column j. At least one of these MAC values has to correspond to a non-faulty server (say i). $K_{i,j}$ is only known to the non-faulty servers i and j, thus it is not possible that the adversary can generate the correct MAC value.

Lemma 5. *If a signature passes the verification procedure then the newly reconstructed matrix-signature is valid.*

Lemma 6. *If a non-faulty verifier accepts that S has signed Msg, then it can convince every other non-faulty verifier that S has signed Msg.*

Theorem 1. *The Matrix-signature scheme presented in Figure 2 satisfies consistency, validity and verifiability.*

Proof. Consistency follows from Lemmas 1 and 2. Validity follows from Lemmas 3 and 4. Verifiability follows from Lemmas 2 and 5.

Theorem 2. *The signing procedure always terminates.*

Theorem 3. *The verification procedure always terminates.*

Proof. Suppose that the verification procedure does not terminate even after receiving responses from all the non-faulty servers. It cannot have received more than f $\langle \text{REJECT} \rangle$ responses. Thus, it would have received at least $(f + 1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses from the non-faulty servers that is accompanied with the correct row of MACs. These $(f + 1)$ rows of correct MACs will ensure that the new signature σ_{new} is Valid.

Thus all non-faulty servers that have not sent a $\langle \text{ADMIT}, \ldots \rangle$ response will do so, when the verifier retries with σ_{new}. The verifier will eventually have $(n - f) \geq (2f + 1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses.

6.5 Discussion

Our distributed implementation of digital signatures is based on an underlying implementation of MACs. We make no additional computational assumptions to implement the digital signatures. However, if the underlying MAC implementation relies on some computational assumptions (e.g. collision resistant hash functions, or assumptions about a non-adaptive adversary) then the signature scheme realized will be secure only as long as those assumptions hold.

7 The $n \leq 3f$ Case

We show that a generalized scheme to implement the properties of signatures using MACs is not possible if $n \leq 3f$. The lower bound holds for a much stronger system model where the network is synchronous and the point-to-point channels between the processes are authenticated and reliable.

7.1 A Stronger Model

We assume that the network is synchronous and processes communicate with each other over authenticated and reliable point-to-point channels. We also assume that processes can maintain the complete history of all messages sent and received over these authenticated channels.

This model, without any further set-up assumptions, is strictly stronger than the model described in Section 4. A lowerbound that holds in this stronger model automatically holds in the weaker model (from Section 4) where the network is asynchronous and the channels are only guaranteed to be fair.

In this model, we show that it is possible to implement MACs over authenticated channels. If, in this model, signatures can be implemented using MACs with $n \leq 3f$, then they can also be implemented over authenticated channels with $n \leq 3f$. Using signatures, it is possible to implement a reliable-broadcast channel with just $n \geq f+1$ replicas [17]. So, it would be possible to implement a reliable-broadcast channel assuming a MAC-based implementation of signatures with n servers, where $(f+1) \leq n \leq 3f$.

But, it is well known that implementing a reliable-broadcast channel in a synchronous setting over authenticated point-to-point channels, without signatures, requires $n \geq 3f+1$ [17]. We conclude that implementing signatures with MACs is not possible if $n \leq 3f$.

It only remains to show that MACs can be implemented in the strong model.

Lemma 7. *In the strong system model, MACs can be implemented amongst any set of servers, U, using authenticated point-to-point channels between them.*

Proof. (outline) To sign a message, the sender sends the message, tagged with the identity of set U, to all the servers in U over the authenticated point-to-point channels. Since the network is synchronous, these messages will be delivered to all the servers in U within the next time instance. To verify that a message is signed, a verifier looks into the history of messages received over the authenticated channel. The message is deemed to have been signed if and only if it was received on the authenticated channel from the signer.

8 Unique Signatures

We provide an implementation of unique signatures when $n > 3f$. By Lemma 7, it follows that the implementation is optimally resilient. Our implementation requires an exponential number of MAC values. We show that any implementation of unique signatures requires that an exponential number of MAC values be generated if f is a constant fraction of n. The implementation is optimal if $n = 3f + 1$; the number of MAC values it requires exactly matches the lower bound when $n = 3f + 1$.

Our implementation uses *unique MAC schemes*. These are schemes for which only one MAC value passes the verification procedure for a given message and that always generate the same MAC value for a given message. Many widely used MAC schemes are unique MAC schemes, including those that provide unconditional security.[3]

8.1 A Unique Signature Implementation

We give an overview of the implementation; detailed protocol and proofs can be found in [16].

In our unique signature scheme, the signing procedure generates signatures which are vectors of $N = \binom{n}{2f+1}$ MAC values, one for each subset of $2f+1$ servers. The i'th entry in the vector of signatures can be generated (and verified) with a key K_i that is shared by all elements of the i'th subset G_i of $2f + 1$ servers, $1 \leq i \leq \binom{n}{2f+1}$. For each K_i, the MAC scheme used to generate MAC values is common knowledge, but K_i is secret (unless divulged by some faulty server in G_i).

To sign a message, the signer sends a request to all the servers. A server generates the MAC values for each group G_i that it belongs to and sends these values to the signer. The signer collects responses until it receives $(f+1)$ identical MAC values for every group G_i. Receiving $f + 1$ identical responses for every G_i is guaranteed because each G_i contains at least $f + 1$ correct servers. Also, receiving $(f+1)$ identical MAC values guarantees that the MAC value is correct because one of the values must be from a non-faulty server.

To verify a unique signature, the verifier sends the vector of N MACs to all the servers. The i'th entry M_i is verified by server p if $p \in G_i$ and M_i is the correct MAC value generated using K_i. A verifier accepts the signature if each entry in the vector is correctly verified by $f + 1$ servers. The verifier rejects a signature if one of its entries is rejected by $f + 1$ servers. Since the underlying MAC schemes are unique and each G_i contains $2f + 1$ servers, a signature is accepted by one correct verifier if an only if it is accepted by every other correct verifier and no other signature is accepted for a given message.

8.2 Complexity of Unique Signature Implementations

Implementing MAC-based unique signature schemes requires an exponential number of keys. Here we outline the approach for the proof; details can be

[3] For many MAC schemes the verification procedure consists of running the MAC generation (signing) procedure on the message and comparing the resulting MAC value with the MAC value to be verified. Since the signing procedure is typically deterministic, only one value can pass the verification procedure.

found in [16]. We identify the MAC schemes used in the implementation with their secret keys and, in what follows, we refer to K_i instead of *the MAC scheme that uses* K_i. We consider a general implementation that uses M secret keys. Every key K_i is shared by a subset of the servers; this is the set of servers that can generate and verify MAC values using K_i. We do not make any assumptions on how a signature looks. We simply assume that the signing procedure can be expressed as a deterministic function $\mathcal{S}(msg, k_1, k_2, \ldots, k_M)$ of the message to be signed (msg), where $k_1, \ldots, k_M$ are the values of the keys $K_1, \ldots, K_M$ used in the underlying MAC schemes.

The lower bound proof relies on two main lemmas which establish that (1) every key value must be known by at least $2f + 1$ servers, and (2) for any set of f servers, there must exist a key value that is not known by any element of the set. We can then use a combinatorial argument to derive a lower bound on the number of keys.

Since we are proving a lower bound on the number of keys, we assume that the signature scheme uses the minimum possible number of keys. It follows, as shown in the following lemma, that no key is redundant. That is, for every key K_i, the value of the signature depends on the value of K_i for some message and for some combination of the values of the other keys.

Lemma 8 (No key is redundant). *For each key* K_i, $\exists msg, k_1, \ldots k_{i-1}, k_i^\alpha, k_i^\beta, k_{i+2}, \ldots, k_M$*:* $\mathcal{S}(msg, k_1, \ldots, k_{i-1}, k_i^\alpha, k_{i+1}, \ldots, k_M) = \sigma_1$*,* $\mathcal{S}(msg, k_1, \ldots, k_{i-1}, k_i^\beta, k_{i+1}, \ldots, k_M) = \sigma_2$ *and* $\sigma_1 \neq \sigma_2$

Proof. (Outline) If the signature produced for a message is always independent of the key K_i, for every combination of the other keys. Then, we can get a smaller signature implementation, by using a constant value for K_i, without affecting the resulting signature.

Lemma 9 ($2f + 1$ servers know each key). *At least* $(2f + 1)$ *servers know the value of* K_i*.*

Proof. We show by contradiction that if K_i is only known by a group G, $|G| \leq 2f$, servers. the signature scheme is not unique. If $|G| \leq 2f$, G is the union of two disjoint sets A and B of size less than $f + 1$ each. From Lemma 8, $\exists msg, k_1, \ldots k_{i-1}, k_i^\alpha, k_i^\beta, k_{i+1}, \ldots, k_M$: $\mathcal{S}(msg, k_1, \ldots, k_{i-1}, k_i^\alpha, \ldots, k_M) = \sigma_1$, $\mathcal{S}(msg, k_1, \ldots, k_{i-1}, k_i^\beta, k_{i+1}, \ldots, k_M) = \sigma_2$, and $\sigma_1 \neq \sigma_2$

Consider the following executions, where message msg is being signed. In all executions, the value of K_j is k_j for $j \neq i$.

- (Exec α) The symmetric key value for K_i is k_i^α. All servers behave correctly. The resulting signature value is σ_1.
- (Exec α') The symmetric key value for K_i is k_i^α. Servers not in B behave correctly. Servers in B set the value of K_i to be k_i^β instead of k_i^α. The resulting signature value is also σ_1 because the signature scheme is unique and tolerates up to f Byzantine failures and $|B| \leq f$.
- (Exec β) The symmetric key value for K_i is k_i^β. All servers behave correctly. The resulting signature value is σ_2.

- (Exec β') The symmetric key value for K_i is k_i^β. Servers not in A behave correctly. Servers in A set the value of K_i to be k_i^α instead of k_i^β. The resulting signature value is also σ_2 because the signature scheme is unique and tolerates up to f Byzantine failures and $|A| \leq f$.

Executions α' and β' only differ in the identities of the faulty servers and are otherwise indistinguishable to servers not in G and to clients. It follows that the same signature value should be calculated in both cases, contradicting the fact that $\sigma_1 \neq \sigma_2$.

Lemma 10 (Faulty servers do not know some key). *For every set of f servers, there exists a secret key K_i that no server in the set knows.*

Proof. If a given set of f servers has access to all the M secret keys, then, if all the elements of the set are faulty, they can generate signatures for messages that were not signed by the signer, violating validity.

We can now use a counting-argument to establish a lower bound on the number of keys required by a MAC-based unique signature implementation [16].

Theorem 4. *The number of keys used by any MAC-based implementation of a unique signature scheme is* $\geq \binom{n}{f} / \binom{n-(2f+1)}{f}$

It follows that for $n = 3f + 1$, the unique signature implementation described in Section 8.1 is optimal. In general, if the fraction of faulty nodes $\frac{f}{n} > \frac{1}{k}$, for $k \geq 3$, then the number of MACs required is at least $(\frac{k}{k-2})^f$.

References

1. Rompel, J.: One-way functions are necessary and sufficient for secure signatures. In: STOC 1990: Proceedings of the twenty-second annual ACM symposium on Theory of computing, pp. 387–394. ACM, New York (1990)
2. Castro, M.: Practical Byzantine Fault Tolerance. PhD thesis, MIT (January 2001)
3. Schneier, B.: Applied cryptography: protocols, algorithms, and source code in C, 2nd edn. John Wiley & Sons, Inc., New York (1995)
4. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (2002)
5. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In: Proc. 7th OSDI (November 2006)
6. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. In: Proc. 21st SOSP (2007)
7. Srikanth, T.K., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing 2(2), 80–94 (1987)
8. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. Technical Report TR-07-40, University of Texas at Austin (2007)
9. Goldreich, O.: Foundations of Cryptography. Volume Basic Tools. Cambridge University Press, Cambridge (2001)

10. Aiyer, A., Alvisi, L., Bazzi, R.A.: Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 443–458. Springer, Heidelberg (2007)
11. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. on Info Theory 22(6), 644–654 (1976)
12. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (1978)
13. Hanaoka, G., Shikata, J., Zheng, Y., Imai, H.: Unconditionally secure digital signature schemes admitting transferability. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 130–142. Springer, Heidelberg (2000)
14. Bishop, M.: Computer Security. Addison-Wesley, Reading (2002)
15. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
16. Aiyer, A.S., Lorenzo Alvisi, R.A.B., Clement, A.: Matrix signatures: From macs to digital signatures. Technical Report TR-08-09, University of Texas at Austin, Department of Computer Sciences (February 2008)
17. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)

How to Solve Consensus in the Smallest Window of Synchrony

Dan Alistarh[1], Seth Gilbert[1], Rachid Guerraoui[1], and Corentin Travers[2]

[1] EPFL LPD, Bat INR 310, Station 14, 1015 Lausanne, Switzerland
[2] Universidad Politecnica de Madrid, 28031 Madrid, Spain

Abstract. This paper addresses the following question: what is the minimum-sized synchronous window needed to solve consensus in an otherwise asynchronous system? In answer to this question, we present the first optimally-resilient algorithm $ASAP$ that solves consensus *as soon as possible* in an eventually synchronous system, i.e., a system that from some time GST onwards, delivers messages in a timely fashion. $ASAP$ guarantees that, in an execution with at most f failures, every process decides no later than round $GST + f + 2$, which is optimal.

1 Introduction

The problem of *consensus*, first introduced in 1980 [25,22], is defined as follows:

Definition 1 (Consensus). *Given n processes, at most t of which may crash: each process p_i begins with initial value v_i and can* decide *on an output satisfying: (1) Agreement: every process decides the same value; (2) Validity: if a process decides v, then v is some process's initial value; (3) Termination: every correct process eventually decides.*

In a seminal paper [10], Dwork et al. introduce the idea of *eventual synchrony* in order to cirumvent the asynchronous impossibility of consensus [11]. They study an asynchronous system in which, after some unknown time GST (*global stabilization time*), messages are delivered within a bounded time. They show that consensus can be solved in this case if and only if $n \geq 2t + 1$.

Protocols designed for the eventually synchronous model are appealing as they tolerate arbitrary periods of asynchrony: in this sense, they are "indulgent" [13]. Such protocols are particularly suited to existing distributed systems, which are indeed synchronous most of the time, but might sometimes experience periods of asynchrony. In practice, the system need not be permanently synchronous after GST; it is necessary only that there be a sufficienly big *window of synchrony* for consensus to complete.

This leads to the following natural question: *For how long does the system need to be synchronous to solve consensus?* In other words, how fast can processes decide in an eventually synchronous system after the network stabilizes? The algorithm presented in [10] guarantees that every process decides within $4(n+1)$ rounds of GST, i.e., the required window of synchrony is of length $4(n+1)$. On

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 32–46, 2008.

the other hand, in [7], Dutta and Guerraoui show that, in the worst case, at least $t + 2$ synchronous rounds of communication are needed. They also present an algorithm for $t < n/3$ that matches this lower bound, but they leave open the question of whether there is an optimally resilient algorithm that decides in any synchronous window of size $t + 2$. In this paper, we once and for all resolve this question by demonstrating a consensus algorithm that guarantees a decision within $t + 2$ rounds of GST.

Early decision. Even though, in the worst case, at least $t + 2$ synchronous rounds are needed to solve consensus, in some executions it it is possible to decide faster. Lamport and Fisher [21] showed that, in a synchronous system, if an execution has at most $f \leq t$ failures, it is possible to decide in $f + 2$ rounds. Dolev, Reischuk, and Strong [5] showed that this bound was optimal. It has remained an open question as to whether there is an optimally resilient early deciding protocol for eventually synchronous systems.

Intuitively, eventual synchrony requires one additional round: $t+1$ synchronous rounds to compute the decision, and one additional round to determine that the execution was, in fact, synchronous. Similarly, early-deciding algorithms require one additional round: $f + 1$ synchronous rounds to compute the decision, and one round to determine that there were only f failures. Thus, the question at hand is whether these rounds can be merged: can we verify in *just one round* both that the execution was synchronous and that there were only f failures? The algorithm presented in this paper achieves exactly that feat, terminating within $f+2$ rounds after GST in an execution with at most f failures.

Results. In this paper, we present the $ASAP$ algorithm which solves consensus and ensures the following properties: (1) *Optimal resilience*: $ASAP$ can tolerate up to $t < n/2$ crash failures; notice that no consensus algorithm can tolerate $\geq n/2$ failures in an eventually synchronous system. (2) *Early deciding*: in every execution with at most $f \leq t$ failures, every process decides no later than round $GST + f + 2$; again, notice that this is optimal.

Key ideas. The $ASAP$ algorithm consists of three main mechanisms. The first mechanism is responsible for computing a value that is safe to decide; specifically, each process maintains an *estimate*, which is updated in every round based on the messages it receives. The second mechanism is responsible for detecting asynchrony; processes maintain (and share) a log of active and failed processes which helps to discover when asynchronies have occurred. Finally, the third mechanism is responsible for determining when it is safe to decide; specifically, a process decides when it has: (1) observed $\leq f$ failures for some $f \leq t$; (2) observed at least $f + 2$ consecutive synchronous rounds; and (3) observed at least two consecutive rounds in which no process appears to have failed. The $ASAP$ algorithm combines these mechanisms within a *full information* protocol, meaning that in each round, each process sends its entire state to every other process. (Optimizing the message complexity is out of the scope of this paper.)

Perhaps the key innovation in the $ASAP$ algorithm is the mechanism by which a process updates its estimate of the decision value. We begin with the naïve approach (as in [24]) in which each process adopts the minimum estimate received in each round. In a synchronous execution with at most $f \leq t$ failures, this guarantees that every process has the same estimate no later than round $f + 1$. We augment this simple approach (generalizing on [7]) by prioritizing an estimate from a process that is about to decide. Moreover, we break ties among processes about to decide by giving priority to processes that have observed more consecutive synchronous rounds. This helps to ensure that if a process does, in fact, decide, then every process has adopted its estimate. This same prioritization scheme, however, poses a problem when a process that has been given priority (since it is about to decide), finally does *not* decide (due to a newly detected asynchrony). To resolve this issue, we sometimes *waive the priority* on an estimate: when a process p_i receives an estimate from another process p_j that is about to decide, p_i examines the messages it has received to determine whether or not p_j (or any process that has received p_j's message) can decide. If p_i can *prove* that process p_j does not decide, then p_i can treat the estimate from process p_j with normal priority. Otherwise, if p_i cannot be certain as to whether p_j will or will not decide, p_i makes the conservative decision and prioritizes the estimate from p_j. This notion of selective prioritization is at the heart of our *ASAP* algorithm, and may be of use in other contexts, such as k-set agreement and Byzantine agreement.

2 Related Work

Beginning with Dwork et al. [10], a variety of different models have been used to express eventual synchrony, including failure detectors [3,4] and round-by-round failure detectors (RRFD) [12]. These approaches have led to the concept of indulgent algorithms [7,13,14]—algorithms that tolerate unreliable failure detectors, expressed in the RRFD model. More recently, Keidar and Shraer [17,18] introduced GIRAF, a framework that extends the assumptions of RRFD.

An important line of research has approached the question we address in this paper in a different manner, asking how fast consensus can terminate if there are *no further failures* after the system stabilizes. Keidar, Guerraoui and Dutta [8] show that at least 3 rounds are needed after the system stabilizes and failures cease, and they present a matching algorithm[1]. Two further papers [17,18] also investigate the performance of consensus algorithms under relaxed timeliness and failure detector assumptions after stabilization.

Paxos-like algorithms that depend on a leader form another class of algorithms in this line of research. Work in [19,23] and [1,2] minimizes the number of "stable" synchronous communication rounds after a correct leader is elected that are

[1] It may appear surprising that we can decide within $f+2$ rounds of GST, as [8] shows that it is impossible to decide sooner than three rounds after failures cease. Indeed, a typical adversarial scenario might involve failing one processor per round during the interval $[GST+1, GST+f]$, resulting in a decision within two rounds of failures ceasing. However, this is not a contradiction as these are worst-case executions in which our algorithm does not decide until 3 rounds after failure cease.

needed to reach agreement, matching lower bounds in [20] and [16], respectively. A related algorithm is presented in [9], which guarantees termination within 17 message delays after stabilization, for the case where no failures occur after stabilization. In fact, it is conjectured there that a bound of $O(f)$ rounds is possible in the case where f failures occur after stabilization. Our paper resolves this conjecture in the affirmative.

Note that our approach to network stabilization differs from both of these previous approaches in that it focuses only on the behavior of the network, independent of failures or leader election.

Finally, Guerraoui and Dutta [6,7] have investigated the possibility of early-deciding consensus for eventual synchrony and have obtained a tight lower bound of $f + 2$ rounds for executions with $f \leq t$ failures, even if the system is initially synchronous. They also present an algorithm for the special case where $t < n/3$ (not optimally resilient) that solves consensus in executions with at most f failures within $f + 2$ rounds of GST, leaving open the question of an optimally resilient consensus algorithm, which we address in this paper.

3 Model

We consider n deterministic processes $\Pi = \{p_1, \ldots, p_n\}$, of which up to $t < n/2$ may fail by crashing. The processes communicate via an *eventually synchronous* message-passing network, modeled much as in [7, 10, 17]: time is divided into *rounds*; however, there is no assumption that every message broadcast in a round is also delivered in that round. Instead, we assume only that if all non-failed processes broadcast a message in some round r, then each process receives at least $n - t$ messages in that round [2]. We assume that the network is *eventually synchronous*: there is some round GST after which every message sent by a non-failed process is delivered in the round in which it is sent.

4 The *ASAP* Consensus Algorithm

In this section, we present an optimally-resilient early-deciding consensus algorithm for the eventually-synchronous model that tolerates $t < n/2$ failures and terminates within $f + 2$ rounds of GST, where $f \leq t$ is the actual number of failures. The pseudocode for $ASAP$ can be found in Figures 1 and 2.

4.1 High-Level Overview

The $ASAP$ algorithm builds on the idea of estimate flooding from the classical synchronous "FloodSet" algorithm (e.g., [24]) and on the idea of detecting asynchronous behavior introduced by the "indulgent" A_{t+2} algorithm of [7].

[2] A simple way to implement this would be for each node to delay its round $r + 1$ message until at least $n - t$ round r messages have been received, and ignoring messages from previous rounds; however, this affects the early-deciding properties of the algorithm, as a correct process can be delayed by asynchronous rounds in which it does not receive $n - t$ messages.

Each process maintains an estimate, along with other state, including: for each round, a set of (seemingly) active processes and a set of (seemingly) failed processes; a flag indicating whether the process is ready to decide; and an indicator for each round as to whether it appears synchronous. At the beginning of each round, processes send their entire state to every other process; *ASAP* is a *full-information protocol.* Processes then update their state and try to decide, before continuing to the next round. We briefly discuss the three main components of the algorithm:

Asynchrony Detection. Processes detect asynchrony by analyzing the messages received in preceeding rounds. Round r is marked as asynchronous by a process p if p learns that a process q is alive in a round $r' > r$, even though it believes[3] q to have failed in round r. Notice that a process p may learn that process q is still alive either directly—by receiving a message from q—or indirectly—by receiving a message from a third process that believes q to be alive. The same holds for determining which processes have failed. Thus, a process merges its view with the views of all processes from which it has received messages in a round, maximizing the amount of information used for detecting asynchrony.

Decision. A process can decide only when it is certain that every other process has adopted the same estimate. There are two steps associated with coming to a decision. If a process has observed f failures, and the previous $f + 1$ rounds are perceived as synchronous, then it sets a "ready to decide" flag to *true*. A process can decide in the following round under the following circumstances: (i) it has observed f failures; (ii) the last $f + 2$ rounds appear synchronous; and (iii) there are no new failures observed in the last two rounds. Once a process decides, it continues to participate, informing other processes of the decision.

Updating the Estimate. The procedure for updating the estimate is the key to the algorithm. Consider first the simple rule used by the classic synchronous consensus protocol, where each process adopts the minimum estimate received in every round. This fails in the context of eventual synchrony since a "slow" process may maintain the minimum estimate even though, due to network delays, it is unable to send or receive messages; this slow process can disrupt later decisions and even cause a decision that violates safety. A natural improvement, which generalizes the approach used in [7], is to prioritize the estimate of a process that is about to decide. Notice that if a process is about to decide, then it believes that it has seen at least one failure-free synchronous round, and hence its estimate should be the minimum estimate in the system. However, this too fails, as there are situations where a process has a synchronous view of $f + 1$ rounds with f failures without necessarily holding the smallest estimate in the system. Thus, we award higher priority to messages from processes that are ready to decide, but allow processes to de-prioritize such estimates if they can *prove* that no process decides after receiving that estimate in the current round.

[3] Note that, throughout this paper, we use terms like "knowledge" and "belief" in their colloquial sense, not in the knowledge-theoretical sense of [15].

```
procedure propose($v_i$)$_i$
begin
    $est_i \leftarrow v_i$; $r_i \leftarrow 1$; $msgSet_i \leftarrow \emptyset$; $sFlag_i \leftarrow false$
    $Active_i \leftarrow [\,]$; $Failed_i \leftarrow [\,]$; $AsynchRound_i \leftarrow [\,]$
    while true do
        send( $est_i$, $r_i$, $sFlag_i$, $Active_i$, $Failed_i$, $AsynchRound_i$, $decide_i$ ) to all
        wait until received messages for round $r_i$
        $msgSet_i[r_i] \leftarrow$ messages that $p_i$ receives in round $r_i$
        $Active_i[r_i] \leftarrow$ processes from which $p_i$ gets messages in round $r_i$
        $Failed_i[r_i] \leftarrow \Pi \setminus Active_i[r_i]$
        $f \leftarrow |Failed_i[r_i]|$
        updateState()     /* Update the state of $p_i$ based on messages received */
        if (checkDecisionCondition() = false) then
            $est_i \leftarrow$ getEstimate()
            if $(sCount_i \geq f + 1)$ then  $sFlag_i = true$
            else $sFlag_i = false$
        end
        $r_i \leftarrow r_i + 1$
    end
end
```

Fig. 1. The *ASAP* algorithm, at process p_i

It remains to describe how a process p can *prove* that no process decides upon receiving q's message. Consider some process s that decides upon receiving q's message. If p can identify a process that is believed by q to be alive and which *does not support* the decision being announced by q, then p can be certain that s will not decide: either s receives a message from the non-supporting process and cannot decide, or s does not receive its message and thus observes a new failure, which prevents s from deciding. Thus, a sufficient condition for discarding q's flag is the existence of a third process that: (i) q considers to be alive in the previous round, and (ii) receives a set of messages other than q's in $r-1$ (Proposition 9). Although this condition does not ensure that p discards all flags that do not lead to decision, it is enough for *ASAP* to guarantee agreement.

4.2 Detailed Description

We now describe the pseudocode in Figures 1 and 2. When consensus is initiated, each process invokes procedure propose() (see Figure 1) with its initial value. A decision is reached at process p_i when $decide_i$ is first set to *true*; the decision is stored in est_i. (For simplicity, the algorithm does not terminate after a decision; in reality, only one further round is needed.)

State Variables. A process p_i maintains the following state variables: (a) r_i is the current round number, initially 1. (b) est_i is p_i's estimate at the end of round r_i. (c) $Active_i[\,]$ is an array of sets of processes. For each round $r' \leq r_i$, $Active_i[r']$ contains the processes that p_i believes to have sent at least one message in round r'. (d) $Failed_i[\,]$ is an array of sets of processes. For each round $r' \leq r_i$, $Failed_i[r']$ contains the processes that p_i believes to have failed in round r'. (e) $msgSet_i$ is the set of messages that p_i receives in round r_i. (f) $AsynchRound_i[\,]$ is an array of flags (booleans). For each round $r' \leq r_i$, $AsynchRound_i[r'] = true$ means that

r' is seen as asynchronous in p_i's view at round r_i. (g) $sCount_i$ is an integer denoting the number of consecutive synchronous rounds p_i sees at the end of r_i. More precisely, if $sCount_i = x$, then rounds in the interval $[r_i - x + 1, r_i]$ are seen as synchronous by p_i at the end of round r_i. (h) $sFlag_i$ is a flag that is set to *true* if p_i is *ready to decide* in the next round. (i) $decided_i$ is a flag that is set to *true* if process p_i has decided.

Main algorithm. We now describe *ASAP* in more detail. We begin by outlining the structure of each round (lines 5-18, Figure 1). Each round begins when p_i broadcasts its current estimate, together with its other state, to every process (line 6); it then receives messages for round r_i (line 7). Process p_i stores these messages in $msgSet_i$ (line 8), and updates $Active_i[r_i]$ and $Failed_i[r_i]$ (lines 9–11).

Next, p_i calls the updateState() procedure (line 12), which merges the newly received information into the current state. It also updates the designation of which rounds appear synchronous. At this point, checkDecisionCondition is called (line 13) to see if a decision is possible. If so, then the round is complete. Otherwise, it continues to update the estimate (line 14), and to update its $sFlag_i$ (line 15–16). Finally, process p_i updates the round counter (line 18), and proceeds to the next round.

Procedure updateState(). The goal of the updateState() procedure is to merge the information received during the round into the existing *Active* and *Failed* sets, as well as updating the *AsynchRound* flag for each round. More specifically, for every message received by process p_i from some process p_j, for every round $r' < r_i$: process p_i merges the received set $msg_j.Active_j[r']$ with its current set $Active_i[r']$. The same procedure is carried out for the *Failed* sets. (See lines 3-8 of *updateState*(), Figure 2).

The second part of the *updateState* procedure updates the *AsynchRound* flag for each round. For all rounds $r' \leq r_i$, p_i recalculates $AsynchRound_i[r']$, marking whether r' is asynchronous in its view at round r_i (lines 9-14). Notice that a round r is seen as asynchronous if some process in $Failed_i[r]$ is discovered to also exist in the set $Active_i[k]$ for some $k > r$, i.e., the process did not actually fail in round r, as previously suspected. Finally, p_i updates $sCount_i$, with the number of previous consecutive rounds that p_i sees as synchronous (line 15).

Procedure checkDecisionCondition(). There are two conditions under which p_i decides. The first is straightforward: if p_i receives a message from another process that has already decided, then it too can decide (lines 3–6). Otherwise, process p_i decides at the end of round r_d if: (i) p_i has seen $\leq f$ failures; (ii) p_i observes at least $f + 2$ consecutive synchronous rounds; and (iii) the last two rounds appear failure-free, i.e. $Active_i[r_d] = Active_i[r_d - 1]$ (line 8). Notice that the size of $Failed_i[r_i]$ captures the number of failures that p_i has observed, and $sCount_i$ captures the number of consecutive synchronous rounds.

Procedure getEstimate(). The *getEstimate*() procedure is the key to the workings of the algorithm. The procedure begins by identifying a set of processes that have raised their flags, i.e., that are "ready to decide" (lines 3–4). The next portion of the procedure (lines 5-13) is dedicated to determining which of these flagged messages to prioritize, and which of these flags should be "discarded," i.e., treated with normal priority. Fix some process p_j whose message is being considered. Process p_i first calculates which processes have a view that is incompatible with the view of p_j (line 6); specifically, these processes received a different set of messages in round $r_i - 1$ from process p_j. None of these processes can support a decision by any process that receives a message from p_j.

Next p_i fixes f_j to be the number of failures observed by process p_j (line 7), and determines that p_j's flag should be waived if the union of the "non-supporting" processes and the failed processes is at least $f_j + 1$ (line 8). In particular, this implies that if a process p_s receives p_j's message, then one of three events occurs: (i) process p_s receives a non-supporing message; (ii) process p_s receives a message from a process that was failed by p_j; or (iii) process p_s observes at least $f_j + 1$ failures. In all three cases, process p_s cannot decide. Thus it is safe for p_i to waive p_j's flag and treat its message with normal priority (lines 9-11).

At the end of this discard process, p_i chooses an estimate from among the remaining flagged messages, if any such messages exist (lines 14-19). Specifically, it chooses the minimum estimate from among the processes that have a maximal *sCount*, i.e., it prioritizes processes that have seen more synchronous rounds. Otherwise, if there are no remaining flagged messages, p_i chooses the minimum estimate that it has received (line 18).

5 Proof of Correctness

In this section, we prove that *ASAP* satisfies validity, termination and agreement. Validity is easily verified (see, for example, Proposition 2), so we focus on termination and agreement.

5.1 Definitions and Properties

We begin with a few definitions. Throughout, we denote the round in which a variable is referenced by a superscript: for example, est_i^r is the estimate of p_i at the end of round r. First, we say that a process perceives round r to be asynchronous if it later receives a message from a process that it believes to have failed in round r.

Definition 2 (Synchronous Rounds). *Given $p_i \in \Pi$ and rounds r, r_v, we say that round r is* asynchronous in p_i's view at round r_v *if and only if there exists round r' such that $r < r' \leq r_v$ and $Active_i^{r_v}[r'] \cap Failed_i^{r_v}[r] \neq \emptyset$. Otherwise, round r is* synchronous in p_i's view at round r_v.

A process perceives a round r as failure-free if it sees the same set of processes as alive in rounds r and $r + 1$.

Definition 3 (Failure-free Rounds). *Given $p_i \in \Pi$ and rounds r, r_v, we say that round $r \leq r_v$ is* failure-free in p_i's view at round r_v *if and only if $Active_i^{r_v}[r] = Active_i^{r_v}[r + 1]$.*

```
procedure updateState()
begin
    for every $msg_j \in msgSet_i[r_i]$ do
        /* Merge newly received information */
        for round $r$ from 1 to $r_i - 1$ do
            $Active_i[r] \leftarrow msg_j.Active_j[r] \cup Active_i[r]$
            $Failed_i[r] \leftarrow msg_j.Failed_j[r] \cup Failed_i[r]$
        end
    end
    for round $r$ from 1 to $r_i - 1$ do
        /* Update $AsynchRound$ flag */
        $AsynchRound_i[r] \leftarrow false$
        for round $k$ from $r + 1$ to $r_i$ do
            if $(Active_i[k] \cap Failed_i[r] \neq \emptyset)$ then $AsynchRound_i[r] \leftarrow true$
        end
    end
    $sCount_i \leftarrow \mathbf{max}_\ell(\forall r_i - \ell \leq r' \leq r_i, AsynchRound_i[r'] = true)$
end
```

```
procedure checkDecisionCondition()
begin
    if $\exists msg_p \in msgSet_i$ s.t. $msg_p.decided_p =$ true then
        $decide_i \leftarrow$ true
        $est_i \leftarrow msg_p.est_p$
        return $decide_i$
    end
    /* If the previous $f + 2$ rounds are synchronous with at most $f$ failures */
    if $(sCount \geq |Failed_i[r_i]| + 2)$ and $(Active_i[r_i] = Active_i[r_i - 1])$ then
        $decide_i \leftarrow$ true
        return $decide_i$
    end
end
```

```
procedure getEstimate()
begin
    $flagProcSet_i \leftarrow \{p_j \in Active_i[r_i] \mid msg_j.sFlag_j = true\}$
    $flagMsgSet_i \leftarrow \{msg_j \in msgSet_i \mid msg_j.sFlag_j = true\}$
    /* Try to waive the priority on flagged messages. */
    for $p_j \in flagProcSet_i$ do
        /* Find the set of processes that disagree with $p_j$'s view. */
        $nonSupport_i^j \leftarrow \{p \in Active_i[r_i] \ : \ msg_p.Active_p[r_i - 1] \neq msg_j.Active_j[r_i - 1]\}$
        $f_j \leftarrow |msg_j.Failed_j[r_i - 1]|$
        if $(|nonSupport_i^j \cup Failed_j[r_i - 1]| \geq f_j + 1)$ then
            $msg_j.sFlag_j[r_i - 1] \leftarrow$ false
            $flagMsgSet_i \leftarrow flagMsgSet_i \setminus \{msg_j\}$
            $flagProcSet_i \leftarrow flagProcSet_i \setminus \{p_j\}$
        end
    end
    /* Adopt the min estimate of max priority; higher $sCount$ has priority. */
    if $(flagMsgSet_i \neq \emptyset)$ then
        /* The set of processes that have the highest $sCount$ */
        $highPrSet \leftarrow \{p_j \in flagMsgSet_i | msg_j.sCount_j = \mathbf{max}_{p_l \in flagMsgSet_i}(sCount_l)\}$
        $est \leftarrow \mathbf{min}_{p_j \in highPrSet}(est_j)$
    else
        $est \leftarrow \mathbf{min}_{p_j \in msgSet_i}(est_j)$
    end
    return $est$
end
```

Fig. 2. *ASAP* procedures

Note that, by convention, if a process p_m completes round r but takes no steps in round $r+1$, p_m is considered to have failed in round r. We now state two simple, yet fundamental properties of $ASAP$:

Proposition 1 (Uniformity). *If processes p_i and p_j receive the same set of messages in round r, then they adopt the same estimate at the end of round r.*

Proposition 2 (Estimate Validity). *If all processes alive at the beginning of round r have estimate v, then all processes alive at the beginning of round $r+1$ will have estimate v.*

These properties imply that if the system remains in a bivalent state (in the sense of [11]), then a failure or asynchrony has to have occured in that round. Proposition 7 combines these properties with the asynchrony-detection mechanism to show that processes with synchronous views and distinct estimates necessarily see a failure for every round that they perceive as synchronous.

5.2 Termination

In this section, we show that every correct process decides by round $GST+f+2$, as long as there are no more than $f \leq t$ failures. Recall that a process decides when there are two consecutive rounds in which it perceives no failures. By the pigeonhole principle, it is easy to see that there must be (at least) two failure-free rounds during the interval $[GST+1, GST+f+2]$; unfortunately, these rounds need not be consecutive. Even so, we can show that at least one correct node must *perceive* two consecutive rounds in this interval as failure-free.

We begin by fixing an execution α with at most f failures, and fixing GST to be the round after which α is synchronous. We now identify two failure-free rounds in the interval $[GST+1, GST+f+2]$ such that in the intervening rounds, there is precisely one failure per round.

Proposition 3. *There exists a round $r_0 > GST$ and a round $r_\ell > r_0$ such that: (a) $r_\ell \leq GST+f+2$; (b) rounds r_0 and r_ℓ are both failure free; (c) for every $r : r_0 < r < r_\ell$, there is exactly one process that fails in r; and (d) $\forall i > 0$ such that $r_0 + i < r_\ell$, there are no more than $(r_0+i) - GST - 1$ failures by the end of round $r_0 + i$.*

The claim follows from a simple counting argument. Now, fix rounds r_0 and r_ℓ that satisfy Proposition 3. For every $i < \ell$: denote by r_i the round r_0+i; let q_i be the process that fails in round r_i; let $q_\ell = \perp$. Let S_i be the set of processes that are not failed at the beginning of round r_i. We now show that, for every round r in the interval $[r_1, r_{\ell-1}]$, if a process in S_r receives a message from q_r, then it decides at the end of round r. This implies that either every process decides by the end of r_ℓ, or, for all rounds r, no process in S_r receives a message from q_r.

Lemma 1. *Assume $r_0 + 1 < r_\ell$, and some process in S_ℓ does not decide by the end of r_ℓ. Then $\forall i : 0 < i < \ell$:*

(i) For every process $p \in S_{i+1} \setminus \{q_{i+1}\}$, process p does not receive a message from q_i in round r_i.

(ii) If process $q_{i+1} \neq \perp$ receives a message from q_i in round r_i, then process q_{i+1} decides at the end of r_i.

We can now complete the proof of termination:

Theorem 1 (Termination). *Every correct process decides by the end of round $GST + f + 2$.*

Proof (sketch). If $r_0 + 1 = r_\ell$, then it is easy to see that every process decides by the end of r_ℓ, since there are two consecutive failure-free rounds. Otherwise, we conclude by Lemma 1 that none of the processes in S_ℓ receive a message from $q_{\ell-1}$ in round $r_{\ell-1}$. Thus every process receives messages from $S_{\ell-1} \setminus \{q_{\ell-1}\}$ both in rounds $r_{\ell-1}$ and r_ℓ, which implies that they decide by the end of r_ℓ.

5.3 Agreement

In this section, we prove that no two processes decide on distinct values. Our strategy is to show that once a process decides, all non-failed processes adopt the decision value at the end of the decision round (Lemma 2). Thus, no decision on another value is possible in subsequent rounds.

Synchronous Views. The key result in this section is Proposition 7, which shows that in executions perceived as synchronous, there is at least one (perceived) failure per round. The idea behind the first preliminary proposition is that if an estimate is held by some process at round r, then there exists at least one process which "carries" it for every previous round.

Proposition 4 (Carriers). *Let $r > 0$ and $p \in \Pi$. If p has estimate v at the end of round r, then for all rounds $0 \leq r' \leq r$, there exists a process $q^{r'} \in Active_p^r[r']$ such that $est_{q^{r'}}[r' - 1] = v$.*

Next, we prove that processes with synchronous views see the same information, with a delay of one round. This follows from the fact that processes communicate with a majority in every round.

Proposition 5 (View Consistency). *Given processes p_i and p_j that see rounds $r_0 + 1, \ldots, r_0 + \ell + 1$ as synchronous: $\forall r \in [r_0 + 1, r_0 + \ell]$, $Active_i^{r_0+\ell+1}[r + 1] \subseteq Active_j^{r_0+\ell+1}[r]$.*

The next proposition shows that if a process observes two consecutive synchronous rounds r and $r + 1$ with the same set of active processes S, then all processes in S receive the same set of messages during round r.

Proposition 6. *Let r, r_c be two rounds such that $r_c > r$. Let p be a process that sees round r as synchronous from round r_c. If $Active_p^{r_c}[r] = Active_p^{r_c}[r+1]$, then all processes in $Active_p^{r_c}[r]$ receive the same set of messages in round r.*

The next proposition is the culmination of this section, and shows that in periods of perceived synchrony, the amount of asynchrony in the system is limited. It captures the intuition that at least one process fails in each round in order to maintain more than one estimate in the system. Recall, this is the key argument for solving consensus in a synchronous environment.

Proposition 7. *Given processes p_i, p_j that see rounds $r_0+1, \ldots, r_0+\ell+1$ as synchronous and adopt distinct estimates at the end of round $r_0+\ell+1$, then for all $r \in [r_0+1, r_0+\ell]$, $|Active_i^{r_0+\ell+1}[r+1]| < |Active_i^{r_0+\ell+1}[r]|$.*

Proof (sketch). We proceed by contradiction: assume there exists a round $r \in [r_0+1, r_0+\ell]$ such that $Active_i^{r_0+\ell+1}[r+1] = Active_i^{r_0+\ell+1}[r]$. This implies that all processes in $Active_i^{r_0+\ell+1}[r]$ received the same set of messages in round r by Proposition 6. Proposition 1 then implies that all processes in $Active_i^{r_0+\ell+1}[r]$ have adopted the same estimate at the end of round r, that is, they have adopted $est_i^{r_0+\ell+1}$.

Proposition 4 implies that there exists a process $p \in Active_j^{r_0+\ell+1}[r+1]$ that adopts estimate $est_j^{r_0+\ell+1}$ at the end of r. By the above, this process is not in $Active_i^{r_0+\ell+1}[r]$. This, together with the fact that $est_i^{r_0+\ell+1} \neq est_j^{r_0+\ell+1}$ implies that $p \in Active_j^{r_0+\ell+1}[r+1] \setminus Active_i^{r_0+\ell+1}[r]$, which contradicts Proposition 5.

Decision Condition. In this section, we examine under which conditions a process may decide, and under what conditions a process may not decide. These propositions are critical to establishing the effectiveness of the estimate-priority mechanism. The following proposition shows that every decision is "supported" by a majority of processes with the same estimate. Furthermore, these processes have a synchronous view of the previous rounds.

Proposition 8. *Assume process p_d decides on v_d at the end of r_0+f+2, seeing $f+2$ synchronous rounds and f failures (line 10 of* checkDecisionCondition*). Let $S := Active_d^{r_0+f+2}[r_0+f+2]$. Then:*

(i) For all $p \in S$, $Active_p^{r_0+f+1}[r_0+f+1] = S$ and $est_p^{r_0+f+1} = v_d$.
(ii) At the end of r_0+f+1, processes in S see rounds $r_0+1, r_0+2, \ldots, r_0+f+1$ as synchronous rounds in which at most f failures occur.

The proposition follows from a careful examination of the decision condition. Next, we analyze a sufficient condition to ensure that a process *does not decide*, which is the basis for the flag-discard rule:

Proposition 9. *Let p be a process with $sFlag = true$ at the end of round $r > 0$. If there exists a process q such that $q \in Active_p^r[r]$ and $Active_q^r[r] \neq Active_p^r[r]$, then no process that receives p's message in round $r+1$ decides at the end of round $r+1$.*

Notice that if a process receives a message from p and not from q, then it sees q as failed; otherwise, if it receives a message from both, it sees a failure in $r-1$. In neither case can the process decide. The last proposition is a technical result that bounds a process's estimate in rounds in which it receives a flagged estimate:

Proposition 10. *Let $r > 0$ and $p \in \Pi$. Let $flagProcSet_p^r$ be the set of processes in $Active_p^r[r]$ with $sFlag = true$. Assume $flagProcSet_p^r$ is non-empty, and let q be a process such that, $\forall s \in flagProcSet_p^r, est_q^{r-1} \leq est_s^{r-1}$, also $q \notin Failed_s^{r-1}[r-1]$ and p receives a message from q in round r. Then $est_p^r \leq est_q^{r-1}$.*

Safety. We now prove the key lemma which shows that if some process decides, then every other non-failed process has adopted the same estimate. The first part of the proof uses Propositions 5 and 7 to determine precisely the set of processes that remain active just prior to the decision, relying on the fact that there must be one new failure per round. The remainder of the proof carefully examines the behavior in the final two rounds prior to the decision; we show that in these rounds, every process must adopt the same estimate. This analysis depends critically on the mechanism for prioritizing estimates, and thus relies on Proposition 10.

Lemma 2 (Safety). *Let r_d be the first round in which a decision occurs. If process p_d decides on value v in round r_d, then every non-failed process adopts v at the end of round r_d.*

Proof (sketch). Assume for the sake of contradiction that there exists a process q such that $est_q^{r_d} = u \neq v$. Fix f to be the number of failures observed by process p_d and fix round $r_0 > 0$ such that $r_d = r_0 + f + 2$. The case where $f \in \{0, 1\}$ needs to be handled separately; in the following, we assume that $f > 1$.

Since p_d decides at the end of r_0+f+2, Proposition 8 implies that there exists a support set S of at least $n-f$ processes such that p_d receives a message in round $r_0 + f + 2$ from all processes in S, and $\forall p \in S, Active_p^{r_0+f+1}[r_0 + f + 1] = S$. Furthermore, processes in S have $sCount \geq f + 1$ and $est = v$ at the end of $r_0 + f + 1$. Since process q receives at least $n - t$ messages in round $r_0 + f + 2$, it necessarily receives a message from a process in S. Denote this process by p_i. We make the following claim:

Claim. Process q receives a message from some process p_j in round $r_0 + f + 1$ such that $est_j = u$, $p_j \notin S$, $sFlag_j = true$ and $sCount_j \geq f + 1$.

The claim follows from the observation that q cannot discard p_i's flag (as per Proposition 9), therefore there has to exist a process p_j with estimate u and flag set with priority at least as high as p_i's. Hence, at the end of round r_0+f+1 we have two processes p_i and p_j that see rounds $r_0+1, \ldots, r_0+f+1$ as synchronous and adopt distinct estimates. This leads to the following claim:

Claim. For every process $p \in S \cup \{p_j\}$, $Active_p^{r_0+f+1}[r_0 + f] = S \cup \{p_j\}$.

In particular, Proposition 7 implies that p_j sees one failure per round, and hence $|Active_j^{r_0+f+1}[r_0 + f]| \leq n - f + 1$. Since $Active_i^{r_0+f+1}[r_0 + f + 1] = S$, Proposition 5 implies that $S \cup \{p_j\} \subseteq Active_j^{r_0+f+1}[r_0 + f]$. Since $p_j \notin S$, we conclude that $S \cup \{p_j\} = Active_j^{r_0+f+1}[r_0 + f]$. A similar argument yields that, for all $p \in S, Active_p^{r_0+f+1}[r_0 + f] = S \cup \{p_j\}$.

In the remaining portion of the proof, we show that no process in $S \cup \{p_j\}$ adopts estimate $max(u, v)$ at the end of r_0+f+1, which leads to a contradiction. Let $m := min(u, v)$ and $M := max(u, v)$. Proposition 4 ensures that there exist processes $p_m, p_M \in S \cup \{p_j\}$ such that $est_m^{r_0+f-1} = m$ and $est_M^{r_0+f-1} = M$. Let $f_j = |Failed_j^{r_0+f+1}[r_0 + f + 1]|$. We can then conclude:

Claim. There exists a set S' of at least $n - f_j - 1$ processes in S such that every process in $S \cup \{p_j\}$ receives messages from S' in round $r_0 + f + 1$ and processes in S' have $est^{r_0+f} \leq min(u, v)$.

To see this, notice that process p_j receives exactly $n - f_j$ messages in round $r_0 + f + 1$; one of these messages must have been sent by p_j itself, while the remaining $n - f_j - 1$ of these messages were sent by processes in S. We denote these processes by S'. Notice that the processes in S' are not considered failed by other processes in S in round $r_0 + f + 1$ since they support p_d's decision in round $r_0 + f + 2$. It follows that the processes in S' have received messages from every process in $S \cup \{p_j\}$ in round $r_0 + f$. With some careful analysis, we can apply Proposition 10 to conclude that for all $s \in S'$, $est_s^{r_0+f} \leq m$, from which the claim follows. Finally, we show that, because of S', no process in $S \cup \{p_j\}$ can adopt M at the end of $r_0 + f + 1$, which contradicts the existence of either p_i or p_j, concluding the proof.

Claim. For every process p in $S \cup \{p_j\}$, $est_p^{r_0+f+1} \leq m$.

This follows because every process p in S receives a message from a process $s \in S'$ in round $r_0 + f + 1$, and no other process in S could have failed s in $r_0 + f$; thus we can again apply Proposition 10 to conclude that $est_p^{r_0+f+1} \leq est_s^{r_0+f} \leq m$, and the claim follows, which concludes the proof of Lemma 2.

We can now complete the proof of agreement:

Theorem 2 (Agreement). *No two processes decide on different estimates.*

Proof (sketch). Let r_d be the first round in which a decision occurs. Since majority support is needed for a decision (see Proposition 8), all processes deciding in r_d decide on the same value. Lemma 2 shows that all processes adopt the same estimate at the end of r_d, and by Proposition 2, no other value is later decided.

6 Conclusions and Future Work

We have demonstrated an optimally-resilient consensus protocol for the eventually synchronous model that decides *as soon as possible*, i.e., within $f+2$ rounds of GST in every execution with at most f failures. It remains an interesting question for future work as to whether these techniques can be extended to k-set agreement and Byzantine agreement. In particular, it seems possible that the mechanism for assigning priorities to estimates based on what a process can *prove* about the system may be useful in both of these contexts. Indeed, there may be interesting connections between this technique and the knowledge-based approach (see, e.g., [15]).

References

1. Boichat, R., Dutta, P., Frolund, S., Guerraoui, R.: Deconstructing paxos. SIGACT News 34(1), 47–67 (2003)
2. Boichat, R., Dutta, P., Frolund, S., Guerraoui, R.: Reconstructing paxos. SIGACT News 34(2), 42–57 (2003)

3. Chandra, T., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
4. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
5. Dolev, D., Reischuk, R., Strong, H.R.: Early stopping in byzantine agreement. J. ACM 37(4), 720–741 (1990)
6. Dutta, P., Guerraoui, R.: The inherent price of indulgence. In: PODC, pp. 88–97 (2002)
7. Dutta, P., Guerraoui, R.: The inherent price of indulgence. Distributed Computing 18(1), 85–98 (2005)
8. Dutta, P., Guerraoui, R., Keidar, I.: The overhead of consensus failure recovery. Distributed Computing 19(5-6), 373–386 (2007)
9. Dutta, P., Guerraoui, R., Lamport, L.: How fast can eventual synchrony lead to consensus? In: DSN, pp. 22–27 (2005)
10. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988)
11. Fisher, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
12. Gafni, E.: Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In: PODC, pp. 143–152 (1998)
13. Guerraoui, R.: Indulgent algorithms (preliminary version). In: PODC, pp. 289–297 (2000)
14. Guerraoui, R., Raynal, M.: The information structure of indulgent consensus. IEEE Transactions on Computers 53(4), 453–466 (2004)
15. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM 37(3), 549–587 (1990)
16. Keidar, I., Rajsbaum, S.: On the cost of fault-tolerant consensus when there are no faults (preliminary version). SIGACT News 32(2), 45–63 (2001)
17. Keidar, I., Shraer, A.: Timeliness, failure-detectors, and consensus performance. In: PODC, pp. 169–178 (2006)
18. Keidar, I., Shraer, A.: How to choose a timing model? In: DSN, pp. 389–398 (2007)
19. Lamport, L.: Generalized consensus and paxos. Microsoft Research Technical Report MSR-TR-2005-33 (March 2005)
20. Lamport, L.: Lower bounds for asynchronous consensus. Distributed Computing 19(2), 104–125 (2006)
21. Lamport, L., Fisher, M.: Byzantine generals and transaction commit protocols (unpublished) (April 1982)
22. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
23. Lamport, L.: Fast paxos. Distributed Computing 19(2), 79–103 (2006)
24. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
25. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)

Local Terminations and Distributed Computability in Anonymous Networks*

Jérémie Chalopin[1], Emmanuel Godard[1], and Yves Métivier[2]

[1] Laboratoire d'Informatique Fondamentale de Marseille
CNRS & Aix-Marseille Université
{jeremie.chalopin,emmanuel.godard}@lif.univ-mrs.fr
[2] LaBRI, Université de Bordeaux
metivier@labri.fr

Abstract. We investigate the computability of distributed tasks in reliable anonymous networks with arbitrary knowledge. More precisely, we consider tasks computable with *local termination*, i.e., a node knows when to stop to participate in a distributed algorithm, even though the algorithm is not necessarily terminated elsewhere. We also study *weak local termination*, that is when a node knows its final value but continues to execute the distributed algorithm, usually in order to provide information to other nodes.

We give the first characterization of distributed tasks that can be computed with weak local termination and we present a new characterization of tasks computed with local termination. For both terminations, we also characterize tasks computable by polynomial algorithms.

1 Introduction

We investigate the computability of distributed tasks in reliable anonymous networks with arbitrary knowledge. Impossibility results in anonymous networks have been investigated for a long time [Ang80]. Among the notable results are the ones of Boldi and Vigna [BV99, BV01], following works of Angluin [Ang80] and of Yamashita and Kameda [YK96a, YK96b]. In [BV99], a characterization of what is computable with arbitrary knowledge is presented. In a following paper [BV01], another characterization is presented but the processes have to know a bound on the number of nodes in the network. To quote the introduction of [BV99], "in a sense the whole issue becomes trivial, as one of the main problems – termination – is factored out *a priori*". That's why we focus in this paper not only on the way to solve a distributed task, but also on what is exactly at stake when one talks about termination in a distributed context.

About Terminations of Distributed Algorithms. Contrary to sequential algorithms, what is the termination of a distributed algorithm is not so intuitively obvious. If we take a global perspective, termination occurs when there is not

* Partially supported by grant No ANR-06-SETI-015-03 awarded by A.N.R.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 47–62, 2008.

anything left to do in the network: no message is in transit and no process can modify its state. But if we are interested in the local point of view of a node executing a distributed algorithm, it is generally not obvious to detect when it can stop waiting for incoming messages. And as usual in the local-global relationship, this is not always possible, or it involves more computation.

Moreover, if we look carefully at what the distributed algorithm is aimed at, we have to begin to distinguish between the termination of the task we want to achieve (the associated computed values) and the termination of our tool, the distributed algorithm. Indeed, a node does not necessarily need to detect that the algorithm has *globally* terminated, but it is interesting if it can detect it has computed its final value (*local termination*). For example, in the case of composition of algorithms, or for local garbage collecting purpose, there is, a priori, no special need to wait that everyone in the network has computed its final value. One can define a hierarchy of termination for distributed tasks:

- *implicit termination*: The algorithm is globally terminated but no node is, or can be aware of this termination;
- *weak local termination*: Every node knows when it has its final value, but does not immediately halt in order to transmit information needed by some other nodes;
- *local termination*: Every node knows when it has its final value and stops participating in the algorithm;
- *global termination detection*: At least one node knows when every other node has computed its final value.

Related Works. In the seminal work of Angluin [Ang80], the first impossibility results for distributed computability in anonymous networks were established. Using the notion of coverings we also use in this paper, she prove that it is impossible to elect a leader in a wide class of "symmetric" networks. She also shows that it is impossible to have a universal termination detection algorithm for any class of graphs that strictly contains the family of all trees.

Distributed computability on asynchronous anonymous rings have been first investigated by Attiya, Snir and Warmuth [ASW88]. They show that any function can be computed with a quadratic number of messages. Some results have been presented when processes initially have ids (or inputs) but they are not assumed to be unique. In this setting, Flocchini *et al.* [FKK$^+$04] consider Election and Multiset Sorting. Mavronicolas, Michael and Spirakis [MMS06] present efficient algorithms for computing functions on some special classes of rings. In all these works, it is assumed that processes initially know the size of the ring. In [DP04], Dobrev and Pelc consider Leader Election on a ring assuming the processes initially know a lower and an upper bound on its size.

Yamashita and Kameda have investigated computability on anonymous arbitrary graphs in [YK96a]. They assume either that the topology of the network or a bound on the size is initially known by the processes. They use the notion of views to characterize computable functions. In [BV02b], Boldi and Vigna characterize what can be computed in a self-stabilizing way in a synchronous setting.

This result enables them to characterize what can be computed in anonymous networks with an implicit termination. This characterization is based on fibrations and coverings, that are some tools we use in this paper. In [BV01], Boldi and Vigna characterize what can be computed on an anonymous networks with local termination provided the processes initially know a bound on the size of the network. This characterization is the same as for implicit termination.

In [BV99], Boldi and Vigna consider tasks computable with local termination with arbitrary knowledge. Their characterization is based on partial views and is really different from the one given in [BV01]. As explained by Boldi and Vigna in [BV99], in all these works (except [BV99]), the processes initially know at least a bound on the size of the network. In this case, all kinds of terminations are equivalent: what can be computed with implicit termination can be computed with global termination detection. In the literature, one can found different algorithms to detect global termination provided that there exists a leader [DS80], that processes have unique ids [Mat87], or that processes know a bound on the diameter of the network [SSP85]. A characterization of tasks computable with global termination detection is presented in [CGMT07].

Our Results. In this regard where termination appears as a natural and key parameter for unification of distributed computability results – the link made by Boldi and Vigna in [BV02b] between computability with implicit termination on anonymous network and self-stabilization is very enlightening –, we present here two characterizations of computability with local and weak local terminations on asynchronous message passing networks where there is *no failure* in the communication system. By considering arbitrary families of labelled graphs, one can model arbitrary initial knowledge and arbitrary level of anonymity (from completely anonymous to unique ids).

We characterize the tasks that are computable with *weak local termination* (Theorem 5.2). Such tasks are interesting, because weak local termination is a good notion to compose distributed algorithms. Indeed, it is not necessary to ensure that all processes have terminated executing the first algorithm before starting the second one. We show that the following intuitive idea is necessary and sufficient for computability with weak local termination: if the k-neighbourhoods of two processes v, w cannot be distinguished locally, then if v computes its final value in k steps, w computes the same final value in k steps.

Then, we present a new characterization of the tasks that are computable with local termination (Theorem 8.3). Our characterization is built upon the one for weak local termination. When we deal with local termination, one has also to take into account that the subgraph induced by the processes that have not terminated may become disconnected during the execution. In some cases, it is impossible to avoid such a situation to occur (see Section 3 for examples).

With the results from [BV02b, CGMT07], we now get characterizations of computability for each kind of termination we discussed. What is interesting is that all of them can be expressed using the same combinatorial tools.

Moreover, the complexity of our universal algorithms is better than the view-based algorithms of Boldi and Vigna and of Yamashita and Kameda that necessitate exchanges of messages of exponential size. It enables us to characterize tasks that are computable with (weak) local termination by polynomial algorithms, i.e., algorithms where for each execution, the number of rounds, the number and the size of the messages are polynomial in the size of the network.

2 Definitions

The Model. Our model corresponds to the usual asynchronous message passing model [Tel00, AW04]. A network is represented by a simple connected graph G where vertices correspond to processes and edges to direct communication links. The state of each process is represented by a label $\lambda(v)$ associated to the corresponding vertex $v \in V(G)$; we denote by $\mathbf{G} = (G, \lambda)$ such a labelled graph. We assume that each process can distinguish the different edges that are incident to it, i.e., for each $u \in V(G)$ there exists a bijection δ_u between the neighbours of u in G and $[1, \deg_G(u)]$ (thus, u knows $\deg_G(u)$). We denote by δ the set of functions $\{\delta_u \mid u \in V(G)\}$. The numbers associated by each vertex to its neigbours are called *port-numbers* and δ is called a *port-numbering* of G. A *network* is a labelled graph $\mathbf{G}$ with a port-numbering δ and is denoted by $(\mathbf{G}, \delta)$.

Each processor v in the network represents an entity that is capable of performing computation steps, sending messages via some port and receiving any message via some port that was sent by the corresponding neighbour. We consider asynchronous systems, i.e., each of the steps of execution may take an unpredictable (but finite) amount of time. Note that we consider only reliable systems: no fault can occur on processes or communication links. We also assume that the channels are FIFO, i.e., for each channel, the messages are delivered in the order they have been sent. In this model, a distributed algorithm is given by a local algorithm that all processes should execute (note that all the processes have the same algorithm). A local algorithm consists of a sequence of computation steps interspersed with instructions to send and to receive messages.

In the paper, we sometimes refer to the *synchronous* execution of an algorithm. Such an execution is a particular execution of the algorithm that can be divided in *rounds*. In each round, each process receives all the messages that have been sent to it by its neighbours in the previous round; then according to the information it gets, it can modify its state and send messages to its neighbours before entering the next round. Note that the synchronous execution of an algorithm is just a special execution of the algorithm and thus it belongs to the set of asynchronous executions of this algorithm.

Distributed Tasks and Terminations. As mentioned earlier, when we are interested in computing a task in a distributed way, we have to distinguish what kind of termination we want to compute the task with. Given a family $\mathcal{F}$ of networks, a network $(\mathbf{G}, \delta) \in \mathcal{F}$ and a process v in $(\mathbf{G}, \delta)$, we assume that the state of v during the execution of any algorithm is of the form $(\mathtt{mem}(v), \mathtt{out}(v), \mathtt{term}(v))$: $\mathtt{mem}(v)$ is the memory of v, $\mathtt{out}(v)$ is its output value and $\mathtt{term}(v)$ is a flag in $\{\textsc{Term}, \bot\}$ mentioning whether v has computed its final value or not. The initial state of v is $(\mathtt{in}(v), \bot, \bot)$ where the input $\mathtt{in}(v)$ is the label of v in $(\mathbf{G}, \delta)$.

A *distributed task* is a couple $(\mathcal{F}, \mathcal{S})$ where $\mathcal{F}$ is a family of labelled graphs and $\mathcal{S}$ is a vertex-relabelling relation (i.e., if $((G, \lambda), \delta)\ \mathcal{S}\ ((G', \lambda'), \delta')$, then $G = G'$ and $\delta = \delta'$) such that for every $(\mathbf{G}, \delta) \in \mathcal{F}$, there exists $(\mathbf{G}', \delta)$ such that $(\mathbf{G}, \delta)\ \mathcal{S}\ (\mathbf{G}', \delta)$. The set $\mathcal{F}$ is the *domain* of the task, $\mathcal{S}$ is the *specification* of the task. The classical leader election problem on some family $\mathcal{F}$ of networks is described in our settings by a task $(\mathcal{F}, \mathcal{S})$ where for each $(\mathbf{G}, \delta) \in \mathcal{F}$, $(\mathbf{G}, \delta)\ \mathcal{S}\ (\mathbf{G}', \delta)$ for any $\mathbf{G}' = (G, \lambda')$ such that there is a unique $v \in V(G)$ with $\lambda'(v) = \texttt{leader}$. Considering arbitrary families of labelled graphs enables to represent any initial knowledge: e.g. if the processes initially know the size of the network, then in the corresponding family $\mathcal{F}$, for each $(\mathbf{G}, \delta) \in \mathcal{F}$ and each $v \in V(G)$, $|V(G)|$ is a component of the initial label of v.

We say that an algorithm $\mathcal{A}$ has an *implicit termination* on $\mathcal{F}$ if for any execution of $\mathcal{A}$ on any graph $(\mathbf{G}, \delta) \in \mathcal{F}$, the network reaches a global state where no messages are in transit and the states of the processes are not modified any more. Such a global final state is called the *final configuration* of the execution.

Given a task $(\mathcal{F}, \mathcal{S})$, an algorithm $\mathcal{A}$ is *normalized* for $(\mathcal{F}, \mathcal{S})$ if $\mathcal{A}$ has an implicit termination on $\mathcal{F}$ and in the final configuration of any execution of $\mathcal{A}$ on some $(\mathbf{G}, \delta) \in \mathcal{F}$, for each $v \in V(G)$, $\texttt{term}(v) = \textsc{Term}$, $\texttt{out}(v)$ is defined and $((G, \texttt{in}), \delta)\ \mathcal{S}\ ((G, \texttt{out}), \delta)$ (i.e., the output of the algorithm solves the task $\mathcal{S}$).

A task $(\mathcal{F}, \mathcal{S})$ is computable with *local termination* (Lt) if there exists a normalized algorithm $\mathcal{A}$ for $(\mathcal{F}, \mathcal{S})$ such that for each $v \in V(G)$, once $\texttt{term}(v) = \textsc{Term}$, $(\texttt{mem}(v), \texttt{out}(v), \texttt{term}(v))$ is not modified any more. A task $(\mathcal{F}, \mathcal{S})$ is computable with *weak local termination* (wLt) if there exists a normalized algorithm $\mathcal{A}$ for $(\mathcal{F}, \mathcal{S})$ such that for each $v \in V(G)$, once $\texttt{term}(v) = \textsc{Term}$, $(\texttt{out}(v), \texttt{term}(v))$ is not modified any more.

For both terminations, one can show that we can restrict ourselves to tasks where $\mathcal{F}$ is recursively enumerable. A vertex v is *active* if it has not stopped the execution of the algorithm, i.e., v can still modify the value of $\texttt{mem}(v)$. When we consider weak local termination, all vertices always remain active, whereas when we consider local termination, a vertex is active if and only if $\texttt{term}(v) \neq \textsc{Term}$. If a vertex is not active anymore, we say that it is *inactive*.

3 Examples of Tasks with Different Kinds of Terminations

We present here three simple examples that demonstrate the hierarchy of terminations. We consider the family $\mathcal{F}$ containing all networks $((G, \texttt{in}), \delta)$ where for each vertex v, its input value has the form $\texttt{in}(v) = (val(v), d(v))$ where $val(v) \in \mathbb{N}$ and $d(v) \in \mathbb{N} \cup \{\infty\}$. The specification we are interested in is the following: in the final configuration, for each $(\mathbf{G}, \delta)$ and for each vertex $v \in V(G)$, $\texttt{out}(v) = \max\{val(u) \mid \mathrm{dist}_G(u, v) \leq d(v)\}$[1].

We add some restrictions on the possible initial value for $d(v)$ in order to define three different tasks. In the general case (no restriction on the values of $d(v)$), the task we just described is called the Maximum Problem. If we consider the same task on the family $\mathcal{F}'$ containing all networks such that for each $v \in V(G)$,

[1] When $d(v) = \infty$, $\texttt{out}(v)$ is the maximum value $val(u)$ on the entire graph.

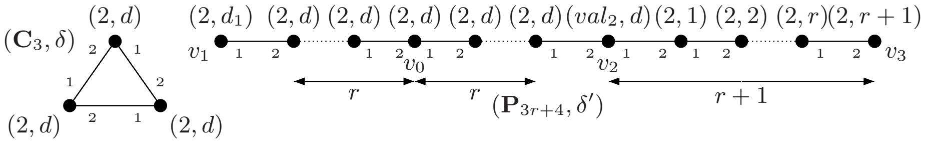


Fig. 1. Networks highlighting differences between the different kinds of termination

$d(v) \neq \infty$, the corresponding task is called the Local Maximum Problem. If we consider the same task on the family $\mathcal{F}''$ containing all networks such that for each edge $\{v, w\}$, $|d(v) - d(w)| \leq 1$, then we obtain a different problem that is called the Locally Bounded Maximum Problem.

The Maximum Problem can be solved with implicit termination by a flooding algorithm. Suppose now that there exists an algorithm $\mathcal{A}$ that can solve the Maximum Problem with wLt. Consider the synchronous execution of $\mathcal{A}$ over the graph $(\mathbf{C}_3, \delta)$ of Figure 1 where $d = \infty$ and let r be the number of rounds of this execution; after r rounds, $\texttt{term}(v) = \text{Term}$ and $\texttt{out}(v) = 2$, for each $v \in V(C_3)$. Consider now the path $(\mathbf{P}_{3r+4}, \delta')$ on $3r + 4$ vertices of Figure 1 where $d = d_1 = \infty$ and $val_2 > 2$. After r synchronous rounds over $(\mathbf{P}_{3r+4}, \delta')$, v_0 gets exactly the same information as any $v \in V(C_3)$ and thus after r rounds, $\texttt{term}(v_0) = \text{Term}$ and $\texttt{out}(v_0) = 2$ whereas the correct output is $\texttt{out}(v_0) = val_2$. Thus $\mathcal{A}$ does not solve the Maximum Problem with wLt.

The Local Maximum Problem can be solved with weak local termination by a flooding algorithm running in waves. Suppose now that there exists an algorithm $\mathcal{A}$ that can solve the Local Maximum Problem with Lt. Consider the synchronous execution of $\mathcal{A}$ over the graph $(\mathbf{C}_3, \delta)$ of Figure 1 where $d = 1$ and let r be the number of rounds of this execution. Consider now the path $(\mathbf{P}_{3r+4}, \delta')$ of Figure 1 where $d = 1$, $val_2 > 2$ and $d_1 \geq 2r + 2$. After r synchronous rounds on P_{3r+4}, for the same reasons as before, $\texttt{term}(v_0) = \text{Term}$. Since v_0 has stopped before it knows val_2 and since after v_0 has stopped, no information can be transmitted through v_0, $\texttt{out}(v_1)$ cannot possibly be val_2, but the correct output is $\texttt{out}(v_1) = val_2$. Thus $\mathcal{A}$ does not solve the Local Maximum Problem with wLt.

The Locally Bounded Maximum Problem can be solved with local termination by the previous wave-based flooding algorithm. Suppose now that there exists an algorithm $\mathcal{A}$ that can solve the Locally Bounded Maximum Problem with global termination detection. Consider the synchronous execution of $\mathcal{A}$ over the graph $(\mathbf{C}_3, \delta)$ of Figure 1 where $d = 1$ and let r be the number of rounds of this execution; after r rounds, a vertex $v \in V(C_3)$ is in a state S indicating that all processes have computed their final values. Consider now the path $(\mathbf{P}_{3r+4}, \delta')$ of Figure 1 where $d_1 = d = 1$, $val_2 > 2$ and on the path $v_2 = w_0, w_1, \ldots, w_{r+1} = v_3$ between v_2 and v_3, for each $i \in [1, r]$, $d(w_i) = i$. After r synchronous rounds on P_{3r+4}, for the same reasons as before v is in the state S indicating that all processes have computed their final values. However, since $\mathrm{dist}(v_2, v_3) = r + 1$, after r rounds, the vertex v_3 does not know the value of val_2 and thus $\texttt{out}(v_3)$ cannot possibly be val_2. Thus $\mathcal{A}$ does not solve the Locally Bounded Maximum Problem with global termination detection.

4 Digraphs and Coverings

Labelled Digraphs. In the following, we will consider directed graphs (digraphs) with multiple arcs and self-loops. A *digraph* $D = (V(D), A(D), s, t)$ is defined by a set $V(D)$ of vertices, a set $A(D)$ of arcs and by two maps s and t that assign to each arc two elements of $V(D)$: a source and a target. If a is an arc, we say that a is incident to $s(a)$ and $t(a)$. A *symmetric* digraph D is a digraph endowed with a symmetry, that is, an involution $Sym : A(D) \to A(D)$ such that for every $a \in A(D), s(a) = t(Sym(a))$. In a symmetric digraph D, the degree of a vertex v is $\deg_D(v) = |\{a \mid s(a) = v\}| = |\{a \mid t(a) = v\}|$ and we denote by $N_D(v)$ the set of neighbours of v. Given two vertices $u, v \in V(D)$, a *path* π of length p from u to v in D is a sequence of arcs $a_1, a_2, \dots a_p$ such that $s(a_1) = u, \forall i \in [1, p-1], t(a_i) = s(a_{i+1})$ and $t(a_p) = v$. If for each $i \in [1, p-1]$, $a_{i+1} \neq Sym(a_i)$, π is *non-stuttering*. A digraph D is *strongly connected* if for all vertices $u, v \in V(D)$, there exists a path from u to v in D. In a symmetric digraph D, the *distance* between two vertices u and v, denoted $\mathrm{dist}_D(u, v)$ is the length of the shortest path from u to v in D.

A *homomorphism* γ between the digraph D and the digraph D' is a mapping $\gamma\colon V(D) \cup A(D) \to V(D') \cup A(D')$ such that for each arc $a \in A(D)$, $\gamma(s(a)) = s(\gamma(a))$ and $\gamma(t(a)) = t(\gamma(a))$. An homomorphism $\gamma : D \to D'$ is an *isomorphism* if γ is bijective.

Throughout the paper we will consider digraphs where the vertices and the arcs are labelled with labels from a recursive label set L. A digraph D labelled over L will be denoted by (D, λ), where $\lambda\colon V(D) \cup A(D) \to L$ is the labelling function. A mapping $\gamma\colon V(D) \cup A(D) \to V(D') \cup A(D')$ is a homomorphism from (D, λ) to (D', λ') if γ is a digraph homomorphism from D to D' which preserves the labelling, i.e., such that $\lambda'(\gamma(x)) = \lambda(x)$ for every $x \in V(D) \cup A(D)$. Labelled digraphs will be designated by bold letters like $\mathbf{D}, \mathbf{G}, \dots$

In a symmetric digraph $\mathbf{D}$, we denote by $\mathbf{B_D}(v_0, r)$, the labelled ball of center $v_0 \in V(D)$ and of radius r that contains all vertices at distance at most r of v_0 and all arcs whose source or target is at distance at most $r-1$ of v_0.

Given a simple connected labelled graph $\mathbf{G} = (G, \lambda)$ with a port-numbering δ, we will denote by $(\mathrm{Dir}(\mathbf{G}), \delta)$ the labelled digraph $(\mathrm{Dir}(G), (\lambda, \delta))$ constructed in the following way. The vertices of $\mathrm{Dir}(G)$ are the vertices of G and they have the same labels as in $\mathbf{G}$. Each edge $\{u, v\}$ is replaced by two arcs $a_{(u,v)}, a_{(v,u)} \in A(\mathrm{Dir}(G))$ such that $s(a_{(u,v)}) = t(a_{(v,u)}) = u$, $t(a_{(u,v)}) = s(a_{(v,u)}) = v$, $\delta(a_{(u,v)}) = (\delta_u(v), \delta_v(u))$, $\delta(a_{(v,u)}) = (\delta_v(u), \delta_u(v))$ and $Sym(a_{(u,v)}) = a_{(v,u)}$. This construction encodes that a process can answer to a neighbour, i.e., "pong" any message.

Given a set L, we denote by $\mathcal{D}_L$ the set of all symmetric digraphs $\mathbf{D} = (D, \lambda)$ where for each $a \in A(D)$, there exist $p, q \in \mathbb{N}$ such that $\lambda(a) = (p, q)$ and $\lambda(Sym(a)) = (q, p)$ and for each $v \in V(D)$, $\lambda(v) \in L$ and $\{p \mid \exists a, \lambda(a) = (p, q) \text{ and } s(a) = v\} = [1, \deg_D(v)]$. In other words, $\mathcal{D}_L$ is the set of digraphs that locally look like some digraph obtained from a simple labelled graph $\mathbf{G}$.

Symmetric Coverings, Quasi-Coverings. The notion of symmetric coverings is fundamental in this work; definitions and main properties are presented in [BV02a]. This notion enables to express "similarity" between two digraphs.

A (labelled) digraph $\mathbf{D}$ is a *covering* of a digraph $\mathbf{D}'$ via φ if φ is a homomorphism from $\mathbf{D}$ to $\mathbf{D}'$ such that each arc $a' \in A(D')$ and for each vertex $v \in \varphi^{-1}(t(a'))$ (resp. $v \in \varphi^{-1}(s(a'))$, there exists a unique arc $a \in A(D)$ such that $t(a) = v$ (resp. $s(a) = v$) and $\varphi(a) = a'$. A symmetric digraph $\mathbf{D}$ is a *symmetric covering* of a symmetric digraph $\mathbf{D}'$ via φ if $\mathbf{D}$ is a covering of $\mathbf{D}'$ via φ and if for each arc $a \in A(D)$, $\varphi(Sym(a)) = Sym(\varphi(a))$.

The following lemma shows the importance of symmetric coverings when we deal with anonymous networks. This is the counterpart of the lifting lemma that Angluin gives for coverings of simple graphs [Ang80] and the proof can be found in [BCG$^+$96, CM07].

Lemma 4.1 (Lifting Lemma [BCG$^+$96]). *If $\mathbf{D}$ is a symmetric covering of $\mathbf{D}'$ via φ, then any execution of an algorithm $\mathcal{A}$ on $\mathbf{D}'$ can be lifted up to an execution on $\mathbf{D}$, such that at the end of the execution, for any $v \in V(D)$, v is in the same state as $\varphi(v)$.*

In the following, one also needs to express similarity between two digraphs up to a certain distance. The notion of quasi-coverings was introduced in [MMW97] for this purpose. The next definition is an adaptation of this tool to digraphs.

Definition 4.2. *Given two symmetric labelled digraphs $\mathbf{D}_0, \mathbf{D}_1$, an integer r, a vertex $v_1 \in V(D_1)$ and a homomorphism γ from $\mathbf{B}_{\mathbf{D}_1}(v_1, r)$ to $\mathbf{D}_0$, the digraph $\mathbf{D}_1$ is a* quasi-covering *of $\mathbf{D}_0$ of center v_1 and of radius r via γ if there exists a finite or infinite symmetric labelled digraph $\mathbf{D}_2$ that is a symmetric covering of $\mathbf{D}_0$ via a homomorphism φ and if there exist $v_2 \in V(D_2)$ and an isomorphism δ from $\mathbf{B}_{\mathbf{D}_1}(v_1, r)$ to $\mathbf{B}_{\mathbf{D}_2}(v_2, r)$ such that for any $x \in V(B_{D_1}(v_1, r)) \cup A(B_{D_1}(v_1, r))$, $\gamma(x) = \varphi(\delta(x))$.*

If a digraph $\mathbf{D}_1$ is a symmetric covering of $\mathbf{D}_0$, then for any $v \in V(D_1)$ and for any $r \in \mathbb{N}$, $\mathbf{D}_1$ is a quasi-covering of $\mathbf{D}_0$, of center v and of radius r. Reversely, if $\mathbf{D}_1$ is a quasi-covering of $\mathbf{D}_0$ of radius r strictly greater than the diameter of $\mathbf{D}_1$, then $\mathbf{D}_1$ is a covering of $\mathbf{D}_0$. The following lemma is the counterpart of the lifting lemma for quasi-coverings.

Lemma 4.3 (Quasi-Lifting Lemma). *Consider a digraph $\mathbf{D}_1$ that is a quasi-covering of $\mathbf{D}_0$ of center v_1 and of radius r via γ. For any algorithm $\mathcal{A}$, after r rounds of the synchronous execution of an algorithm $\mathcal{A}$ on $\mathbf{D}_1$, v_1 is in the same state as $\gamma(v_1)$ after r rounds of the synchronous execution of $\mathcal{A}$ on $\mathbf{D}'$.*

5 Characterization for Weak Local Termination

We note $\mathcal{V}$ the set $\{(\mathbf{D}, v) \mid \mathbf{D} \in \mathcal{D}_L, v \in V(D)\}$. In other words, the set $\mathcal{V}$ is the disjoint union of all symmetric labelled digraphs in $\mathcal{D}_L$. Given a family of networks $\mathcal{F}$, we denote by $\mathcal{V}_{\mathcal{F}}$ the set $\{((\mathrm{Dir}(\mathbf{G}), \delta), v) \mid (\mathbf{G}, \delta) \in \mathcal{F}, v \in V(G)\}$.

A function $f : \mathcal{V} \longrightarrow L \cup \{\bot\}$ is an *output* function for a task $(\mathcal{F}, \mathcal{S})$ if for each network $(\mathbf{G}, \delta) \in \mathcal{F}$, the labelling obtained by applying f on each $v \in V(G)$ satisfies the specification $\mathcal{S}$, i.e., $(\mathbf{G}, \delta)\ \mathcal{S}\ (\mathbf{G}', \delta)$ where $\mathbf{G}' = (G, \lambda')$ and $\lambda'(v) = f((\mathrm{Dir}(\mathbf{G}), \delta), v)$ for all $v \in V(G)$.

In order to give our characterization, we need to formalize the following idea. When the neighbourhood at distance k of two processes v, v' in two digraphs $\mathbf{D}, \mathbf{D}'$ cannot be distinguished (this is captured by the notion of quasi-coverings and Lemma 4.3), and if v computes its final value in less than k rounds, then v' computes the same final value in the same number of rounds. In the following definition, the value of $r(\mathbf{D}, v)$ can be understood as the number of rounds needed by v to compute in a synchronous execution its final value in $\mathbf{D}$.

Definition 5.1. *Given a function $r : \mathcal{V} \longrightarrow \mathbb{N} \cup \{\infty\}$ and a function $f : \mathcal{V} \longrightarrow L'$ for some set L', the function f is* r-lifting closed *if for all $\mathbf{D}, \mathbf{D}' \in \mathcal{D}_L$ such that $\mathbf{D}$ is a quasi-covering of $\mathbf{D}'$, of center $v_0 \in V(G)$ and of radius R via γ with $R \geq \min\{r(\mathbf{D}, v_0), r(\mathbf{D}', \gamma(v_0))\}$, then $f(\mathbf{D}, v_0) = f(\mathbf{D}', \gamma(v_0))$.*

Using the previous definition, we now give the characterization of tasks computable with WLT. We also characterize distributed tasks computable with WLT by polynomial algorithms (using a polynomial number of messages of polynomial size). We denote by $|\mathbf{G}|$ the size of $V(G)$ plus the maximum over the sizes (in bits) of the initial labels that appear on $\mathbf{G}$.

Theorem 5.2. *A task $(\mathcal{F}, \mathcal{S})$ where $\mathcal{F}$ is recursively enumerable is computable with* WLT *if and only if there exist a function $r : \mathcal{V} \longrightarrow \mathbb{N} \cup \{\infty\}$ and an output function $f : \mathcal{V} \longrightarrow L \cup \{\bot\}$ for $(\mathcal{F}, \mathcal{S})$ such that*

(i) for all $\mathbf{D} \in \mathcal{D}_L$, for all $v \in V(D)$, $r(\mathbf{D}, v) \neq \infty$ if and only if $f(\mathbf{D}, v) \neq \bot$,
(ii) $f_{|\mathcal{V}_{\mathcal{F}}}$ and $r_{|\mathcal{V}_{\mathcal{F}}}$ are recursive functions,
(iii) f and r are r-lifting-closed.

The task $(\mathcal{F}, \mathcal{S})$ is computable by a polynomial algorithm with WLT *if and only if there exist such f and r and a polynomial p such that for each $(\mathbf{G}, \delta) \in \mathcal{F}$ and each $v \in V(G)$, $r((\mathrm{Dir}(\mathbf{G}), \delta), v) \leq p(|\mathbf{G}|)$.*

Proof (of the necessary condition). Assume $\mathcal{A}$ is a distributed algorithm that computes the task $(\mathcal{F}, \mathcal{S})$ with weak local termination. We define r and f by considering the synchronous execution of $\mathcal{A}$ on any digraph $\mathbf{D} \in \mathcal{D}_L$. For any $v \in V(D)$, if $\mathtt{term}(v) = \bot$ during the whole execution, then $f(\mathbf{D}, v) = \bot$ and $r(\mathbf{D}, v) = \infty$. Otherwise, let r_v be the first round after which $\mathtt{term}(v) = \textsc{Term}$; in this case, $f(\mathbf{D}, v) = \mathtt{out}(v)$ and $r(\mathbf{D}, v) = r_v$. Since $\mathcal{A}$ computes $(\mathcal{F}, \mathcal{S})$, it is easy to see that f is an output function and that f and r satisfy (i) and (ii).

Consider two digraphs $\mathbf{D}, \mathbf{D}' \in \mathcal{D}_L$ such that $\mathbf{D}$ is a quasi-covering of $\mathbf{D}'$, of center $v_0 \in V(G)$ and of radius R via γ with $R \geq r_0 = \min\{r(\mathbf{D}, v_0), r(\mathbf{D}', \gamma(v_0))\}$. If $r_0 = \infty$, then $r(\mathbf{D}, v_0) = r(\mathbf{D}', \gamma(v_0)) = \infty$ and $f(\mathbf{D}, v_0) = f(\mathbf{D}', \gamma(v_0)) = \bot$. Otherwise, from Lemma 4.3, we know that after r_0 rounds, $\mathtt{out}(v_0) = \mathtt{out}(\gamma(v_0))$ and $\mathtt{term}(v_0) = \mathtt{term}(\gamma(v_0)) = \textsc{Term}$. Thus $r_0 = r(\mathbf{D}, v_0) = r(\mathbf{D}', \gamma(v_0))$ and $f(\mathbf{D}, v_0) = f(\mathbf{D}', \gamma(v_0))$. Consequently, f and r are r-lifting closed. □

The sufficient condition is proved in Section 7 and relies on a general algorithm described in Section 6. Using this theorem, one can show that there is no universal election algorithm for the family of networks with non-unique ids where at least one id is unique, but that there exists such an algorithm for such a family where a bound on the multiplicity of each id in any network is known.

6 A General Algorithm

In this section, we present a general algorithm that we parameterize by the task and the termination we are interested in, in order to obtain our sufficient conditions. This algorithm is a combination of an election algorithm for symmetric minimal graphs presented in [CM07] and a generalization of an algorithm of Szymanski, Shy and Prywes (the SSP algorithm for short) [SSP85]. The algorithm described in [CM07] is based on an enumeration algorithm presented by Mazurkiewicz in a different model [Maz97] where each computation step involves some synchronization between adjacent processes. The SSP algorithm enables to detect the global termination of an algorithm with local termination provided the processes know a bound on the diameter of the graph. The Mazurkiewicz-like algorithm always terminates (implicitly) on any network $(\mathbf{G}, \delta)$ and during its execution, each process v can reconstruct at some computation step i a digraph $\mathbf{D}_i(v)$ such that $(\mathrm{Dir}(\mathbf{G}), \delta)$ is a quasi-covering of $\mathbf{D}_i(v)$. However, this algorithm does not enable v to compute the radius of this quasi-covering. We use a generalization of the SSP algorithm to compute a lower bound on this radius, as it has already be done in Mazurkiewicz's model [GMT06].

We consider a network $(\mathbf{G}, \delta)$ where $\mathbf{G} = (G, \lambda)$ is a simple labelled graph and where δ is a port-numbering of $\mathbf{G}$. The function $\lambda : V(G) \to L$ is the initial labelling. We assume there exists a total order $<_L$ on L and we assume that if the label $\lambda(v)$ is modified during the execution, then it can only increase for $<_L$.

The state of each v is a tuple $(\lambda(v), n(v), N(v), M(v), a(v), A(v))$ where:

- $\lambda(v) \in L$ is the initial label of v and if it is modified during the execution, it will necessarily increase for $<_L$.
- $n(v) \in \mathbb{N}$ is the *number* of v computed by the algorithm; initially $n(v) = 0$.
- $N(v) \in \mathcal{P}_{\text{fin}}(\mathbb{N} \times L \times \mathbb{N}^2)$[2] is the *local view* of v. At the end of the execution, if $(m, \ell, p, q) \in N(v)$, then v has a neighbour u whose number is m, whose label is ℓ and the arc from u to v is labelled (p, q). Initially $N(v) = \{(0, \perp, 0, q) \mid q \in [1, \deg_G(v)]\}$.
- $M(v) \subseteq \mathbb{N} \times L \times \mathcal{P}_{\text{fin}}(\mathbb{N} \times L \times \mathbb{N}^2)$ is the *mailbox* of v; initially $M(v) = \emptyset$. It contains all information received by v during the execution of the algorithm. If $(m, \ell, N) \in M(v)$, it means that at some previous step of the execution, there was a vertex u such that $n(u) = m$, $\lambda(u) = \ell$ and $N(u) = N$.
- $a(v) \in \mathbb{Z} \cup \{\infty\}$ is a counter and initially $a(v) = -1$. In some sense, $a(v)$ represent the distance up to which all vertices have the same mailbox as v. If $a(v) = \infty$, it means that v has terminated the algorithm (local termination).

[2] For any set S, $\mathcal{P}_{\text{fin}}(S)$ denotes the set of finite subsets of S.

- $A(v) \in \mathcal{P}_{\text{fin}}(\mathbb{N} \times (\mathbb{Z} \cup \{\infty\}))$ encodes the information v has about the values of $a(u)$ for each neighbour u. Initially, $A(v) = \{(q, -1) \mid q \in [1, \deg_G(v)]\}$.

In our algorithm, processes exchange messages of the form $< (n, \ell, M, a), p >$. If a vertex u sends a message $< (n, \ell, M, a), p >$ to one of its neighbour v, then the message contains following information: n is the current number $n(u)$ of u, ℓ is the label $\lambda(u)$ of u, M is the mailbox of u, a is the value of $a(u)$ and $p = \delta_u(v)$.

As in Mazurkiewicz's algorithm [Maz97], the nice properties of the algorithm rely on a total order on local views, i.e., on finite subsets of $\mathbb{N}^3 \times L$. Given two distinct sets $N_1, N_2 \in \mathcal{P}_{\text{fin}}(\mathbb{N} \times L \times \mathbb{N}^2)$, we define $N_1 \prec N_2$ if the maximum of the symmetric difference $N_1 \triangle N_2 = (N_1 \setminus N_2) \cup (N_2 \setminus N_1)$ for the lexicographic order belongs to N_2. One also says that $(\ell, N) \prec (\ell', N')$ if either $\ell <_L \ell'$, or $\ell = \ell'$ and $N \prec N'$. We denote by $\preceq$ the reflexive closure of $\prec$.

Our algorithm $\mathcal{A}_{gen}(\varphi)$ is described in Algorithm 1. The algorithm for the vertex v_0 is expressed in an event-driven description. The first rule **I** can be applied by a process v on wake-up only if it has not received any message: it takes the number 1, updates its mailbox and informs its neighbours. The second rule **R** describes the instructions a process v has to follow when it receives a message m from a neighbour. It updates its mailbox $M(v)$ and its local view $N(v)$ according to m. Then, if it discovers the existence of another vertex with the same number and a stronger local view, it takes a new number. Then, if its mailbox has not changed, it updates $A(v)$ and increases $a(v)$ if possible (according to a function φ). Finally, if $M(v)$ or $a(v)$ has been modified, it informs its neighbours.

Later, we will add rules that enable a process to compute its final value and we will define the function $\varphi(v)$ depending on the termination we are interested in. Using the information stored in its mailbox, each v will be able to reconstruct a digraph $\mathbf{D}$ such that $(\mathrm{Dir}(\mathbf{G}), \delta)$ locally looks like $\mathbf{D}$ up to distance $a(v)$.

Properties of the Algorithm. We consider a graph $\mathbf{G}$ with a port numbering δ and an execution of Algorithm 1 on $(\mathbf{G}, \delta)$. For each vertex $v \in V(G)$, we note $(\lambda_i(v), n_i(v), N_i(v), M_i(v), a_i(v), A_i(v))$ the state of v after the ith computation step. The following proposition summarizes some nice properties that are satisfied during any execution of Algorithm 1 on $(\mathbf{G}, \delta)$.

Proposition 6.1 ([CM07, Cha06]). *Consider a vertex v and a step $i \geq 1$. Then, $\lambda_{i-1}(v) \leq_L \lambda_i(v)$, $n_{i-1}(v) \leq n_i(v)$, $N_{i-1}(v) \preceq N_i(v)$, $M_{i-1}(v) \subseteq M_i(v)$.*

If $M_{i-1}(v) = M_i(v)$ and if v is active at step i, then $a_{i-1}(v) \leq a_i(v) \leq a_{i-1}(v) + 1$ and $a_i(v) \geq \min\{a \mid \exists (q, a) \in A_i(v)\}$ if $\exists (q, a) \in A_i(v)$ with $a \neq \infty$.

For each $(m, \ell, N) \in M_i(v)$ and each $m' \in [1, m]$, $\exists (m', \ell', N') \in M_i(v), \exists v' \in V(G)$ such that $n_i(v') = m'$. If $m = n_i(v)$, $(\ell, N) \leq (\lambda_i(v), N_i(v))$.

If $a_i(v) \geq 1$, for each $w \in N_G(v)$, there exists a step $j \leq i - 1$ such that w is inactive at step j, or $a_j(w) \geq a_i(v) - 1$ and $M_j(w) = M_i(v)$.

An interesting corollary of Proposition 6.1 is that if the label $\lambda(v)$ of each v is modified only finitely many times, then there exists a step i_0 after which for any v, the value of $(\lambda(v), n(v), N(v), M(v))$ is not modified any more.

Algorithm 1. The general algorithm $\mathcal{A}_{gen}(\varphi)$.

```
I : {$n(v_0) = 0$ and no message has arrived at $v_0$}
begin
    $n(v_0) := 1$ ;
    $M(v_0) := \{(n(v_0), \lambda(v_0), \emptyset)\}$ ;
    $a(v_0) := 0$ ;
    for $i := 1$ to $\deg(v_0)$ do
        send $< (n(v_0), \lambda(v), M(v_0), a(v_0)), i >$ through $i$ ;
end

R : {A message $< (n_1, \ell_1, M_1, a_1), p_1 >$ has arrived at $v_0$ through port $q_1$}
begin
    $M_{old} := M(v_0)$ ;
    $a_{old} := a(v_0)$ ;
    $M(v_0) := M(v_0) \cup M_1$ ;
    if $n(v_0) = 0$ or $\exists (n(v_0), \ell', N') \in M(v_0)$ such that $(\lambda(v_0), N(v_0)) \prec (\ell', N')$
    then
        $n(v_0) := 1 + \max\{n' \mid \exists (n', \ell', N') \in M(v_0)\}$ ;
    $N(v_0) := N(v_0) \setminus \{(n', \ell', p', q_1) \mid \exists (n', \ell', p', q_1) \in N(v_0)\} \cup \{(n_1, \ell_1, p_1, q_1)\}$ ;
    $M(v_0) := M(v_0) \cup \{(n(v_0), \lambda(v_0), N(v_0))\}$ ;
    if $M(v_0) \neq M_{old}$ then
        $a(v_0) := -1$ ;
        $A(v_0) := \{(q', -1) \mid \exists (q', a') \in A(v_0)$ with $a' \neq \infty\}$ ;
    if $M(v_0) = M_1$ then
        $A(v_0) := A(v_0) \setminus \{(q_1, a') \mid \exists (q_1, a') \in A(v_0)\} \cup \{(q_1, a_1)\}$ ;
    if $\forall (q', a') \in A(v_0), a(v_0) \leq a'$ and $(\varphi(v) = \text{TRUE}$ or $\exists (q', a') \in A(v)$ such that
    $a(v_0) < a'$ and $a' \neq \infty)$ then $a(v_0) := a(v_0) + 1$ ;
    if $M(v_0) \neq M_{old}$ or $a(v_0) \neq a_{old}$ then
        for $q := 1$ to $\deg(v_0)$ do
            if $(q, \infty) \notin A(v)$ then
                send $< (n(v_0), \lambda(v), M(v_0), a(v_0)), q >$ through port $q$ ;
end
```

7 Tasks Computable with Weak Local Termination

In order to show that the conditions of Theorem 5.2 are sufficient, we use the general algorithm presented in Section 6 parameterized by the functions f and r. In the following, we consider a function $\texttt{enum}_{\mathcal{F}}$ that enumerates the elements of $\mathcal{F}$. During the execution of this algorithm on any graph $(\mathbf{G}, \delta) \in \mathcal{F}$, for any $v \in V(G)$, the value of $\lambda(v) = \texttt{in}(v)$ is not modified.

Consider the mailbox $M = M(v)$ of a vertex v during the execution of the algorithm $\mathcal{A}_{gen}$ on a graph $(\mathbf{G}, \delta) \in \mathcal{F}$. We say that an element $(n, \ell, N) \in M$ is *maximal* in M if there does not exists $(n, \ell', N') \in M$ such that $(\ell, N) \prec (\ell', N')$. We denote by $S(M)$ be the set of maximal elements of M. From Proposition 6.1, after each step of Algorithm 1, $(n(v), \lambda(v), N(v))$ is maximal in $M(v)$. The set $S(M)$ is said *stable* if it is non-empty and if for all $(n_1, \ell_1, N_1) \in S(M)$, for all

$(n_2, \ell_2, p, q) \in N_1$, $p \neq 0$, $n_2 \neq 0$ and $\ell_2 \neq \bot$ and for all $(n_2', \ell_2', N_2') \in S(M)$, there exists $(n_2', \ell_2'', p', q') \in N_1$ if and only if $\ell_2' = \ell_2''$ and $(n_1, \ell_1, q', p') \in N_2'$. From [CM07], we know that once the values of $n(v), N(v), M(v)$ are final, then $S(M(v))$ is stable. Thus, if $S(M(v))$ is not stable, $M(v)$ will be modified.

If the set $S(M)$ is stable, one can construct a labelled symmetric digraph $\mathbf{D}_M = (D_M, \lambda_M)$ as follows. The set of vertices $V(D_M)$ is the set $\{n \mid \exists (n, \ell, N) \in S(M)\}$. For any $(n, \ell, N) \in S(M)$ and any $(n', \ell', p, q) \in N$, there exists an arc $a_{n,n',p,q} \in A(D_M)$ such that $t(a) = n, s(a) = n'$, $\lambda_M(a) = (p, q)$. Since $S(M)$ is stable, we can define Sym by $Sym(a_{n,n',p,q}) = a_{n',n,q,p}$.

Proposition 7.1. *If $S(M(v))$ is stable, $(\mathrm{Dir}(\mathbf{G}), \delta)$ is a quasi-covering of $\mathbf{D}_{M(v)}$ of radius $a(v)$ of center v via a mapping γ where $\gamma(v) = n(v)$.*

Thus, once v has computed $\mathbf{D}_{M(v)}$, it can enumerate networks $(\mathbf{K}', \delta_K') \in \mathcal{F}$ and vertices $w' \in V(K')$ until it finds a $(\mathbf{K}', \delta_K')$ such that $\mathbf{K}(v) = (\mathrm{Dir}(\mathbf{K}'), \delta_{K'})$ is a quasi-covering of $\mathbf{D}_{M(v)}$ of center $w(v) \in V(K)$ and of radius $a(v)$ via some homomorphism γ such that $\gamma(w(v)) = n(v)$ (this enumeration terminates by Proposition 7.1). We add a rule to the algorithm, called WLT$(\mathsf{enum}_{\mathcal{F}}, f, r)$, that a process v can apply to computes its final value, once it has computed $\mathbf{K}(v)$ and $w(v)$. We also add priorities between rules such that a vertex that can apply the rule WLT(f, r) cannot apply the rule $\mathbf{R}$ of algorithm $\mathcal{A}_{gen}(\varphi)$.

Procedure WLT(*enum*$_{\mathcal{F}}$,*f*,*r*): The rule added to the algorithm for WLT

```
if $\mathtt{term}(v_0) \neq$ TERM and $a(v_0) \geq r(\mathbf{K}(v_0), w(v_0))$ then
    $\mathtt{out}(v_0) := f(\mathbf{K}(v_0), w(v_0))$ ;
    $\mathtt{term}(v_0) :=$ TERM ;
```

We now define the function φ that enables a vertex v to increase $a(v)$. The function φ is true for v only if $\mathtt{term}(v) \neq$ TERM and $S(M(v))$ is stable (otherwise, v knows that its mailbox will be modified in the future) and $r(\mathbf{K}(v), w(v)) > a(v)$ (otherwise, v can compute its final value).

Correction of the Algorithm. We denote by $\mathcal{A}_{\text{WLT}}(\mathsf{enum}_{\mathcal{F}}, f, r)$ the algorithm defined by $\mathcal{A}_{gen}(\varphi)$ and by WLT$(\mathsf{enum}_{\mathcal{F}}, f, r)$. We consider a network $(\mathbf{G}, \delta) \in \mathcal{F}$ and an execution of $\mathcal{A}_{\text{WLT}}(\mathsf{enum}_{\mathcal{F}}, f, r)$ on $(\mathbf{G}, \delta)$. Let $\mathbf{G}' = (\mathrm{Dir}(\mathbf{G}), \delta)$.

Using Propositions 6.1 and 7.1, one can show that the execution terminates (implicitly) and that in the final configuration, for any $v \in V(G)$, $\mathtt{term}(v) =$ TERM. Since f is an output function for $(\mathcal{F}, \mathcal{S})$, the next proposition shows that $\mathcal{A}_{\text{WLT}}(\mathsf{enum}_{\mathcal{F}}, f, r)$ computes the task $(\mathcal{F}, \mathcal{S})$ with weak local termination.

Proposition 7.2. *For any $v \in V(G)$, if $\mathtt{term}(v) =$ TERM, $\mathtt{out}(v) = f(\mathbf{G}', v)$.*

Proof. Consider a process v just after it has applied Procedure WLT$(\mathsf{enum}_{\mathcal{F}}, f, r)$: $S(M(v))$ is stable, $r(\mathbf{K}(v), w(v)) \leq a(v)$ and $\mathtt{out}(v) = f(\mathbf{K}(v), w(v))$.

Since $\mathbf{K}(v)$ is a quasi-covering of $\mathbf{D}_{M(v)}$ of radius $a(v) \geq r(\mathbf{K}(v), w(v))$ and of center $w(v)$ via a mapping γ such that $\gamma(w(v)) = n(v)$ and since f and r are

r-lifting closed, $\mathtt{out}(v) = f(\mathbf{K}(v), w(v)) = f(\mathbf{D}_{M(v)}, n(v))$ and $r(\mathbf{K}(v), w(v)) = r(\mathbf{D}_{M(v)}, n(v))$. From Proposition 7.1, since $a(v) \geq r(\mathbf{D}_{M(v)}, n(v))$ and since f is r-lifting closed, $\mathtt{out}(v) = f(\mathbf{D}_{M(v)}, n(v)) = f(\mathbf{G}', v)$. □

8 Tasks Computable with Local Termination

When we consider local termination, one needs to consider the case where some vertices that terminate quickly disconnect the graph induced by active vertices.

We extend to symmetric digraphs the notion of views that have been introduced to study leader election by Yamashita and Kameda [YK96b] for simple graphs and by Boldi *et al.* [BCG$^+$96] for digraphs.

Definition 8.1. *Consider a symmetric digraph* $\mathbf{D} = (D, \lambda) \in \mathcal{D}_L$ *and a vertex* $v \in V(D)$*. The* view *of* v *in* $\mathbf{D}$ *is an infinite rooted tree denoted by* $\mathbf{T_D}(v) = (T_D(v), \lambda')$ *and defined as follows:*

- $V(T_D(v))$ *is the set of non-stuttering paths* $\pi = a_1, \dots, a_p$ *in* $\mathbf{D}$ *with* $s(a_1) = v$*. For each path* $\pi = a_1, \dots, a_p$*,* $\lambda'(\pi) = \lambda(t(a_p))$*.*
- *for each* $\pi, \pi' \in V(T_D(v))$*, there are two arcs* $a_{\pi,\pi'}, a_{\pi',\pi} \in A(T_D(v))$ *such that* $Sym(a_{\pi,\pi'}) = a_{\pi',\pi}$ *if and only if* $\pi' = \pi, a$*. In this case,* $\lambda'(a_{\pi,\pi'}) = \lambda(a)$ *and* $\lambda'(a_{\pi',\pi}) = \lambda(Sym(a))$*.*
- *the root of* $T_D(v)$ *is the vertex corresponding to the empty path and its label is* $\lambda(v)$*.*

Consider the view $\mathbf{T_D}(v)$ of a vertex v in a digraph $\mathbf{D} \in \mathcal{D}_L$ and an arc a such that $s(a) = v$. We define $\mathbf{T}_{\mathbf{D}-a}(v)$ be the infinite tree obtained from $\mathbf{T_D}(v)$ by removing the subtree rooted in the vertex corresponding to the path a. Given $n \in \mathbb{N}$ and an infinite tree $\mathbf{T}$, we note $\mathbf{T}\lceil^n$ the truncation of the tree at depth n. Thus the truncation of the view at depth n of a vertex v in a symmetric digraph $\mathbf{D}$ is denoted by $\mathbf{T_D}(v)\lceil^n$. It is easy to see that for any $\mathbf{D} \in \mathcal{D}_L$, any $v \in V(D)$ and any integer $n \in \mathbb{N}$, $\mathbf{T_D}(v)\lceil^n$ is a quasi-covering of $\mathbf{D}$ of center v' and of radius n where v' is the root of $\mathbf{T_D}(v)\lceil^n$.

Given a digraph $\mathbf{D}$ and a process $v_0 \in V(D)$ that stops its computation after n steps, the only information any other process v can get from v_0 during the execution is contained in the n-neighbourhood of v_0. In order to take this property into account, we define an operator split. In $\mathsf{split}(\mathbf{D}, v_0, n)$, we remove v_0 from $V(D)$ and for each neighbour v of v_0, we add a new neighbour v_0' to v that has a n-neighbourhood indistinguishable from the one of v_0. Thus, for any process $v \neq v_0$, in a synchronous execution, both v_0 and the vertices we have just added stop in the same state after n rounds and consequently v should behave in the same way in the two networks and stop with the same final value after the same number of rounds. This idea is formalized in Definition 8.2.

Given a digraph $\mathbf{D} = (D, \lambda) \in \mathcal{D}_L$, a vertex $v_0 \in V(D)$ and an integer $n \in \mathbb{N}$, $\mathsf{split}(\mathbf{D}, v_0, n) = (D', \lambda')$ is defined as follows. First, we remove v_0 and all its incident arcs from $\mathbf{D}$. Then for each arc $a \in A(D)$ such that $s(a) = v_0$, we add a copy of $\mathbf{T}_{\mathbf{D}-a}(v_0)\lceil^n$ to the graph. We denote by $v(a)$ the root of this tree and we add two arcs a_0, a_1 to the graph such that $Sym(a_0) = a_1$, $s(a_0) = t(a_1) = v(a)$, $s(a_1) = t(a_2) = t(a)$, $\lambda'(a_0) = \lambda(a)$ and $\lambda'(a_1) = \lambda(Sym(a))$. Note that for any vertex $v \neq v_0 \in V(D)$, v can be seen as a vertex of $\mathsf{split}(\mathbf{D}, v_0, n)$.

Definition 8.2. *Given a function $r : \mathcal{V} \longrightarrow \mathbb{N} \cup \{\infty\}$, a function $f : \mathcal{V} \longrightarrow L'$ is r-*splitting closed *if for any $\mathbf{D} \in \mathcal{D}_L$, for any vertex $v_0 \in V(D)$ and any vertex $v \neq v_0 \in V(D)$, $f(\mathbf{D}, v) = f(\mathit{split}(\mathbf{D}, v_0, n), v)$ where $n = r(\mathbf{D}, v_0)$.*

We now give the characterization of tasks computable with Lt. The proof is postponed to the journal version. For the necessary condition, we uses the same ideas as for the necessary condition of Theorem 5.2 and the proof of the sufficient condition also relies on the general algorithm presented in Section 6.

Theorem 8.3. *A task $(\mathcal{F}, S)$ where $\mathcal{F}$ is recursively enumerable is computable with* Lt *if and only if there exist some functions f and r satisfying the conditions of Theorem 5.2 and such that f and r are r-splitting closed.*

The task $(\mathcal{F}, \mathcal{S})$ is computable by a polynomial algorithm with Lt *if and only if there exist such f and r and a polynomial p such that for each $(\mathbf{G}, \delta) \in \mathcal{F}$ and each $v \in V(G)$, $r((\mathrm{Dir}(\mathbf{G}), \delta), v) \leq p(|\mathbf{G}|)$.*

References

[Ang80] Angluin, D.: Local and global properties in networks of processors. In: Proc. of STOC 1980, pp. 82–93 (1980)

[ASW88] Attiya, H., Snir, M., Warmuth, M.: Computing on an anonymous ring. J. ACM 35(4), 845–875 (1988)

[AW04] Attiya, H., Welch, J.: Distributed computing: fundamentals, simulations, and advanced topics. John Wiley and Sons, Chichester (2004)

[BCG$^+$96] Boldi, P., Codenotti, B., Gemmell, P., Shammah, S., Simon, J., Vigna, S.: Symmetry breaking in anonymous networks: characterizations. In: Proc. of ISTCS 1996, pp. 16–26. IEEE Press, Los Alamitos (1996)

[BV99] Boldi, P., Vigna, S.: Computing anonymously with arbitrary knowledge. In: Proc. of PODC 1999, pp. 181–188. ACM Press, New York (1999)

[BV01] Boldi, P., Vigna, S.: An effective characterization of computability in anonymous networks. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 33–47. Springer, Heidelberg (2001)

[BV02a] Boldi, P., Vigna, S.: Fibrations of graphs. Discrete Mathematics 243(1-3), 21–66 (2002)

[BV02b] Boldi, P., Vigna, S.: Universal dynamic synchronous self-stabilization. Distributed Computing 15(3), 137–153 (2002)

[CGMT07] Chalopin, J., Godard, E., Métivier, Y., Tel, G.: About the termination detection in the asynchronous message passing model. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 200–211. Springer, Heidelberg (2007)

[Cha06] Chalopin, J.: Algorithmique distribuée, calculs locaux et homomorphismes de graphes. PhD thesis, Université Bordeaux 1 (2006)

[CM07] Chalopin, J., Métivier, Y.: An efficient message passing election algorithm based on Mazurkiewicz's algorithm. Fundamenta Informaticae 80(1–3), 221–246 (2007)

[DP04] Dobrev, S., Pelc, A.: Leader election in rings with nonunique labels. Fundamenta Informaticae 59(4), 333–347 (2004)

[DS80] Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computation. Information Processing Letters 11(1), 1–4 (1980)

[FKK$^+$04] Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F., Santoro, N.: Sorting and election in anonymous asynchronous rings. J. Parallel Distrib. Comput. 64(2), 254–265 (2004)

[GMT06] Godard, E., Métivier, Y., Tel, G.: Termination detection of distributed tasks. Technical Report 1418–06, LaBRI (2006)

[Mat87] Mattern, F.: Algorithms for distributed termination detection. Distributed computing 2(3), 161–175 (1987)

[Maz97] Mazurkiewicz, A.: Distributed enumeration. Information Processing Letters 61(5), 233–239 (1997)

[MMS06] Mavronicolas, M., Michael, L., Spirakis, P.: Computing on a partially eponymous ring. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 380–394. Springer, Heidelberg (2006)

[MMW97] Métivier, Y., Muscholl, A., Wacrenier, P.-A.: About the local detection of termination of local computations in graphs. In: Proc. of SIROCCO 1997, pp. 188–200. Carleton Scientific (1997)

[SSP85] Szymanski, B., Shy, Y., Prywes, N.: Synchronized distributed termination. IEEE Transactions on software engineering 11(10), 1136–1140 (1985)

[Tel00] Tel, G.: Introduction to distributed algorithms. Cambridge University Press, Cambridge (2000)

[YK96a] Yamashita, M., Kameda, T.: Computing functions on asynchronous anonymous networks. Math. Systems Theory 29(4), 331–356 (1996)

[YK96b] Yamashita, M., Kameda, T.: Computing on anonymous networks: Part I - characterizing the solvable cases. IEEE Transactions on parallel and distributed systems 7(1), 69–89 (1996)

A Self-stabilizing Algorithm with Tight Bounds for Mutual Exclusion on a Ring
(Extended Abstract)

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

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

Abstract. In [Dij74] Dijkstra introduced the notion of self-stabilizing algorithms and presented, among others, an algorithm with three states for the problem of mutual exclusion on a ring of processors. In this work we present a new three state self-stabilizing algorithm for mutual exclusion, with a tight bound of $\frac{5}{6}n^2 + O(n)$ for the worst case complexity, which is the number of moves of the algorithm until it stabilizes. This bound is better than lower bounds of other algorithms, including Dijkstra's. Using similar techniques we improve the analysis of the upper bound for Dijkstra's algorithm and show a bound of $3\frac{13}{18}n^2 + O(n)$.

1 Introduction

The notion of self stabilization was introduced by Dijkstra in [Dij74]. He considers a system, consisting of a set of processors, and each running a program of the form: **if** *condition* **then** *statement*. A processor is termed *privileged* if its condition is satisfied. A *scheduler* chooses any privileged processor, which then executes its statement (*i.e.,* makes a move); if there are several privileged processors, the scheduler chooses any of them. Such a scheduler is termed *centralized*. A scheduler that chooses any subset of the privileged processors, which are then making their moves simultaneously, is termed *distributed*. Thus, starting from any initial configuration, we get a sequence of moves (termed an *execution*). The scheduler thus determines all possible executions of the system. A specific subset of the configurations is termed *legitimate*. The system is *self-stabilizing* if any possible execution will eventually get — that is, after a finite number of moves — only to legitimate configurations. The number of moves from any initial configuration until the system stabilizes is often referred to as *stabilization time* (see, *e.g.,* [BJM06, CG02, NKM06, TTK00]).

Dijkstra studied in [Dij74] the fundamental problem of mutual exclusion, for which the subset of legitimate configurations includes the configurations in which exactly one processor is privileged. In [Dij74] the processors are arranged in a

[⋆] This research was supported in part by the Technion V.P.R. fund.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 63–77, 2008.

ring, so that each processor can communicate with its two neighbors using shared memory, and where not all processors use the same program. Three algorithms were presented — without proofs for either correctness or complexity — in which each processor could be in one of $k > n$, four and three states, respectively (n being the number of processors). A centralized scheduler was assumed. The analysis — correctness and complexity — of Dijkstra's first algorithm is rather straightforward; its correctness under a centralized scheduler is for any $k \geq n-1$, and under a distributed scheduler for any $k \geq n$. The stabilization time under a centralized scheduler is $\Theta(n^2)$ (following [CGR87] this is also the expected number of moves). There is little in the literature regarding the second algorithm, probably since it was extended in [Kru79] to general trees, or since more attention was devoted to the third algorithm, which is rather non-intuitive. For this latter algorithm Dijkstra presented in [Dij86] a proof of correctness (another proof was given in [Kes88], and a proof of correctness under a distributed scheduler was presented in [BGM89]). Though while dealing with proofs of correctness one can sometimes get also complexity results, this was not the case with this proof of [Dij86]. In [CSZ07] we provide an upper bound of $5\frac{3}{4}n^2$ on the stabilization time of Dijkstra's third algorithm. In [BD95] a similar three state algorithm with an upper bound of $5\frac{3}{4}n^2$ is presented. In [CSZ08a] this upper bound was improved to $1\frac{1}{2}n^2$, and a lower bound of n^2 was shown.

2 Our Contribution

In this work we present a new three state self-stabilizing algorithm for mutual exclusion for a ring of processors, and show a tight bound of $\frac{5}{6}n^2 + O(n)$ for its worst case time complexity. For the lower bound we provide an example that requires $\frac{5}{6}n^2 - O(n)$ moves until stabilization. For the upper bound we proceed in two ways. The first one is using the more conventional tool of potential functions, that is used in the literature of self-stabilizing algorithms to deal mainly with the issue of correctness (see, e.g., [Dol00]). In our case the use of this tool is not straightforward, since the potential function can also increase by some of the moves (see Section 4.5). We use this tool to achieve a complexity result; namely, an upper bound of $1\frac{1}{12}n^2 + O(n)$. The second one is using amortized analysis. This more refined technique enables us to achieve a tight bound of $\frac{5}{6}n^2 + O(n)$.

We use both techniques to improve the analysis for Dijkstra's algorithm that results in an improved upper bound of $3\frac{13}{18}n^2 + O(n)$ for its worst case time complexity. We also show a lower bound of $1\frac{5}{6}n^2 - O(n)$ of Dijkstra's algorithm. Therefore in the worst case our algorithm has better performance than Dijkstra's and than the one of [BD95].

In Section 3 we present Dijkstra's algorithm and outline the details of the proof of [Dij86] needed for our discussion. In Section 4 we present our new self-stabilizing algorithm and its analysis. In Section 5 we present the above-mentioned lower and upper bounds for Dijkstra's algorithm. We summarize our results in Section 6. Most proofs are sketched only or omitted in this Extended Abstract (for more details see [CSZ08b]).

3 Dijkstra's Algorithm

In this section we present Dijkstra's third algorithm of [Dij74] (to which we refer throughout this paper as *Dijkstra's algorithm*, and its proof of correctness of [Dij86]). Our discussion assumes a centralized scheduler. In our system there are n processors $p_0, p_1, \ldots, p_{n-1}$ arranged in a ring; that is for every $0 \leq i \leq n-1$, the processors adjacent to p_i are $p_{(i-1) \bmod n}$ and $p_{(i+1) \bmod n}$. Processor p_i has a local state $x_i \in \{0, 1, 2\}$. Two processors — namely, p_0 and p_{n-1} — run special programs, while all intermediate processors p_i, $1 \leq i \leq n-2$, run the same program. The programs of the processors are as follows:

```
Program for processor $p_0$:
IF  $x_0 + 1 = x_1$  THEN
    $x_0 := x_0 + 2$
END.

Program for processor $p_i$,  $1 \leq i \leq n-2$:
IF  $(x_{i-1} - 1 = x_i)$  OR  $(x_i = x_{i+1} - 1)$  THEN
    $x_i := x_i + 1$
END.

Program for processor $p_{n-1}$:
IF  $(x_{n-2} = x_{n-1} = x_0)$  OR  $(x_{n-2} = x_{n-1} + 1 = x_0)$  THEN
    $x_{n-1} := x_{n-2} + 1$
END.
```

The legitimate configurations for this problem are those in which exactly one processor is privileged. The configurations $x_0 = \cdots = x_{n-1}$ and $x_0 = \cdots = x_i \neq x_{i+1} = \cdots = x_{n-1}$ are legitimate.

This algorithm self stabilizes; namely, starting from any initial configuration the system achieves mutual exclusion. Given an initial configuration $x_0, x_1, \ldots,$ x_{n-1}, and placing the processors on a line, consider each pair of neighbors p_{i-1} and p_i, for $i = 1, \ldots, n-1$ (note p_{n-1} and p_0 are not considered here to be neighbors). In this work we denote *left arrow* and *right arrow*, introduced in [Dij86], by '<' and '>'. Notation $x_{i-1} < x_i$ means $x_i = (x_{i-1} + 1) \bmod 3$ and $x_{i-1} > x_i$ means $x_i = (x_{i-1} - 1) \bmod 3$. Thus, for each two neighboring processors with states x_{i-1} and x_i, either $x_{i-1} = x_i$, or $x_{i-1} < x_i$, or $x_{i-1} > x_i$. For a given configuration $C = x_0, x_1, \ldots, x_{n-1}$, Dijkstra introduces the function

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

Example 1. For $n = 7$, a possible configuration C is $x_0 = 1, x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 2, x_5 = 2, x_6 = 0$. This configuration is denoted as $[1 = 1 > 0 < 1 < 2 = 2 < 0]$. For this configuration we have $f(C) = 3 + 2 \times 1 = 5$.

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

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

Table 1. Dijkstra's algorithm

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

There are eight possible types of moves of the system: one possible move for processor p_0, five moves for any intermediate processor p_i, $0 < i < n-1$, and two moves for p_{n-1}. These possibilities are summarized in Table 1. In this table C_1 and C_2 denote the configurations before and after the move, respectively, and $\Delta f = f(C_2) - f(C_1)$. In the table we show only the local parts of these configurations. As an example, consider the type 0 move in which p_0 is privileged. In this case C_1 and C_2 are the local configurations $x_0 < x_1$ and $x_0 > x_1$, correspondingly. Since one left arrow is replaced by a right arrow, we have $\Delta f = f(C_2) - f(C_1) = 1$.

It is proved that each execution is infinite (that is, there is always at least one privileged processor). Then it is shown that p_0 makes infinite number of moves. Then the execution is partitioned into *phases*; each phase starts with a move of p_0 and ends just before its next move. It is argued that the function f decreases at least by 1 after each phase. By (2) it follows that Dijkstra's algorithm terminates after at most $2(n-1)$ phases.

4 New Self-stabilizing Algorithm for Mutual Exclusion

In this section we present a new algorithm $\mathcal{A}$ for self-stabilization. Our discussion includes the following steps. We first describe the new algorithm; we note the issues in which it differs from Dijkstra's algorithm, discuss its lower bound and prove its correctness. We then introduce a new function h, with which we achieve an upper bound, and finally provide a proof for the tight upper bound using amortized analysis.

4.1 Algorithm $\mathcal{A}$

Algorithm $\mathcal{A}$ is similar to Dijkstra's algorithm with the following changes: moves of types 4 and 5 are not allowed, and moves of type 6 do not depend on processor p_0. Informally Algorithm $\mathcal{A}$ allows the arrows to move and bounce (change direction at p_0 and p_{n-1}) until they are destroyed by moves of type 3. New arrows may be created by a move of type 7.

Table 2. Algorithm $\mathcal{A}$

Type	**Proc.**	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{\Delta\hat{f}}$	$\mathbf{\Delta f}$	$\mathbf{\Delta h}$
0	p_0	$x_0 < x_1$	$x_0 > x_1$	$+1$	$+1$	$n-2$
1	p_i	$x_{i-1} > x_i = x_{i+1}$	$x_{i-1} = x_i > x_{i+1}$	0	0	-1
2	p_i	$x_{i-1} = x_i < x_{i+1}$	$x_{i-1} < x_i = x_{i+1}$	0	0	-1
3	p_i	$x_{i-1} > x_i < x_{i+1}$	$x_{i-1} = x_i = x_{i+1}$	0	-3	$-(n+1)$
4						
5						
6	p_{n-1}	$x_{n-2} > x_{n-1}$	$x_{n-2} < x_{n-1}$	-1	-1	$n-2$
7	p_{n-1}	$x_{n-2} = x_{n-1},\ \hat{f} = 0$	$x_{n-2} < x_{n-1}$	$+1$	$+1$	$n-1$

```
Program for processor $p_0$:
IF $x_0 + 1 = x_1$ THEN
    $x_0 := x_0 + 2$
END.
```

```
Program for processor $p_i,\ 1 \le i \le n-2$:
IF $(x_{i-1} - 1 = x_i = x_{i+1})$ OR $(x_{i-1} = x_i = x_{i+1} - 1)$ OR $(x_{i-1} = x_i + 1 = x_{i+1})$
THEN
    $x_i := x_i + 1$
END.
```

```
Program for processor $p_{n-1}$:
IF $(x_{n-2} = x_{n-1} = x_0)$ OR $(x_{n-2} = x_{n-1} + 1)$ THEN
    $x_{n-1} := x_{n-2} + 1$
END.
```

We define the function $\hat{f}$ for any configuration C as follows:

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

Recalling (1) we get: $\hat{f}(C) \equiv f(C) \pmod 3$. Since '<' (resp. '>') is a shortcut for $x_i - x_{i-1} \equiv 1 \pmod 3$ (resp. $x_i - x_{i-1} \equiv -1 \pmod 3$), we have that $\hat{f}(C) \equiv (x_{n-1} - x_0) \pmod 3$. In particular $\hat{f}(C) = 0$ **iff** $x_{n-1} = x_0$.

We summarize the moves of algorithm $\mathcal{A}$ in Table 2. In this table we also include the changes in the function $\hat{f}$ and the function h (that will be introduced in Section 4.5) implied by each move. We include the rows for moves 4 and 5 (that do not exist) to simplify the analogy to Dijkstra's algorithm.

4.2 Lower Bound

We denote configurations by regular expressions over $\{<,>,=\}$. For example, $\left[<^3{=}{=}{<}{>}{>}\right]$ and $\left[<^3{=}^2{<}{>}^2\right]$ are possible notations for the configuration $x_0 < x_1 < x_2 < x_3 = x_4 = x_5 < x_6 > x_7 > x_8$. Note that this notation does not lose relevant information, since the behavior of the algorithm is dictated by the arrows (see Table 2).

Theorem 1. *The worst case stabilization time of algorithm $\mathcal{A}$ is at least $\frac{5}{6}n^2 - O(n)$.*

Proof (sketch). Assume $n = 3k + 3$. For any $0 \leq i \leq k$, let $C_i := \left[=^{3i}<^{3k-3i+2}\right]$. In particular, C_0 is $\left[<^{3k+2}\right]$ and C_k is $\left[=^{3k}<<\right]$. One can show (see [CSZ08b]) an execution with $3k+9i+7$ moves, starting from C_i and ending at C_{i+1}. Then, starting from C_0 the execution reaches C_k in $\sum_{i=0}^{k-1}(3k + 9i + 7) = \frac{15}{2}k^2 + O(k)$ moves. Substituting $k = \frac{1}{3}n - 1$ we get $\frac{5}{6}n^2 - O(n)$. □

4.3 Correctness

In this section we prove the correctness of algorithm $\mathcal{A}$. It is similar to that of [Dij86], but simpler, mainly since there are fewer cases to consider.

Lemma 1 (no deadlock). *In any configuration at least one processor is privileged.*

Proof. Assume, by contradiction, that there is deadlock in some configuration $C = [x_0, x_1, \ldots, x_{n-1}]$. If $x_i > x_{i+1}$ for some i, then $x_{i+1} > x_{i+2}$ and similarly this will imply $x_j > x_{j+1}$, for every $j \geq i$; thus p_{n-1} is privileged, a contradiction. Hence we can assume there are no '>' arrows. Similarly there are no '<' arrows. Therefore C is $\left[=^{n-1}\right]$. In this case p_{n-1} is privileged, a contradiction. □

Lemma 2. *In any infinite execution, p_0 makes an infinite number of moves.*

Proof (sketch). Consider an infinite execution $C_0 \to C_1 \to \ldots \to C_i \to \ldots$, in which starting from some configuration C_{k_0}, p_0 doesn't move. Then p_1 is allowed to make at most two moves, hence starting from some future configuration C_{k_1}, p_1 and p_0 do not move. By induction, for any $0 \leq i \leq n-2$, there is a configuration C_{k_i}, starting from which neither of $p_0, p_1, \ldots, p_i$ move. Now, starting from configuration $C_{k_{n-2}}$, p_{n-1} is allowed to make at most one move. Since no other processor makes a move, we are deadlocked — a contradiction. □

Given a segment e of an execution of algorithm $\mathcal{A}$, $t_i(e)$ denotes the number of type i moves in e. Our analysis shows that each execution eventually stabilizes, and we are thus interested in its prefix until it reaches stabilization. For this reason, through the paper we will denote such a prefix by E. We will use t_i as a shortcut for $t_i(E)$. In the discussion we will consider segments of E *delimited by* two successive moves of type i; this will mean that each such segment starts with the first type i move and ends just before the second type i move. For example, a phase is a segment of E delimited by two successive moves of type 0.

Lemma 3. *Assume $e \subseteq E$ is a segment delimited by any two successive moves of type 7. Then $t_0(e) + 2t_6(e) \equiv 2 \pmod 3$.*

Proof. After the first type 7 move $\hat{f} = 1$, before the second move $\hat{f} = 0$. The only moves that change $\hat{f}$ in e are the moves of type 0 (resp. 6), which increases (resp. decreases) it by 1. Therefore $1 + t_0(e) - t_6(e) \equiv 0 \pmod 3$. □

Lemma 4. *Assume $e \subseteq E$ is a phase. Then*

1. $t_3(e) \geq 1$.
2. $t_6(e) \geq t_7(e) - 1$.

Proof (sketch).

1. The arrow '>' that is created in the first type 0 move must disappear, in order to allow to an arrow '<' to reach the left end and to initiate the next type 0 move.
2. If $t_7(e) \leq 1$, the claim holds trivially. Otherwise $t_7(e) \geq 2$, and there are $t_7(e) - 1$ segments in e each of which is delimited by two successive type 7 moves. For each such segment e', $t_0(e') = 0$. Applying Lemma 3, we get $t_6(e') \geq 1$. Therefore $t_6(e) \geq t_7(e) - 1$. □

Lemma 5. *Assume $e \subseteq E$ is a phase. Then the function f decreases at least by 1 during e.*

Proof. Using Lemma 4, we get that the function f decreases by $(-1) \cdot t_0(e) + (+3) \cdot t_3(e) + (-1) \cdot t_7(e) + (+1) \cdot t_6(e) \geq -1 \cdot 1 + 3 \cdot 1 - 1 \cdot t_7(e) + 1 \cdot (t_7(e) - 1) = 1$. □

The above lemmas prove the following theorem:

Theorem 2 (correctness). *Algorithm $\mathcal{A}$ self-stabilizes.*

4.4 Basic Properties

Let a be the number of arrows in the initial configuration of E. Since the number of arrows is always non-negative, we have $a + t_7 - 2t_3 \geq 0$. Lemma 3 implies $2t_7 \leq t_0 + 2t_6$. Starting with these basic inequalities, and exploring more properties of the execution we derive a system of inequalities that allows us to get the following bounds.

We start by informally introducing the term *life-cycle* of an arrow. We say that a move of type 7 creates an arrow and in this way starts its life-cycle. A move of type 3 destroys both arrows, ending their life-cycles. The life-cycle of an arrow appearing in the initial configuration starts from that configuration.

Next we introduce the term *mark*. If an arrow is created by a move 7 it is marked by '7'. If an arrow makes a move of types 0 (resp. 6) it gets an additional mark 0 (resp. 6). That allows us to introduce *types of arrows* — according to marks collected during the execution.

Example 2. Arrow of type $>_{60}$ starts its life-cycle from the initial configuration, then it reaches processor p_{n-1} and makes a move of type 6. Afterwards it makes $n - 2$ moves of type 2, reaches processor p_0 and makes a move of type 0. After making some, possibly 0, moves of type 1 it is destroyed. Only one arrow of this type can be in the given execution and such an arrow can make at most $3n$ moves during its life-cycle.

Example 3. Arrow of type $>_{70}$ starts its life-cycle by a move of type 7. Then it reaches processor p_0 and makes a move of type 0. Afterwards, it possibly makes some moves of type 1. Such an arrow can make at most $2n$ moves.

Lemma 6. *The execution E may contain arrows of the following types only:* $>$, $>_0$, $>_{70}$, $>_{60}$, $<$, $<_7$, $<_6$.

Proof. Omitted.

The next step is to introduce the *type of collision*. Consider a move of type 3, for short *collision*. The types of two arrows destroyed in the collision define *the type of the collision*.

Example 4. Collision of type $>_{60}<_7$ is a move of type 3 that destroys arrows $>_{60}$ and $<_7$. Clearly, only one collision of such a type can occur during the execution.

Lemma 7. *The execution E may contain collisions of the following types only:* $><$, $><_7$, $><_6$, $>_0<$, $>_0<_7$, $>_0<_6$, $>_{70}<_7$, $>_{60}<_7$.

Proof. Omitted.

The following theorem presents the main property of algorithm $\mathcal{A}$, and is the basis of the subsequent analysis.

Theorem 3. $t_7 \leq \frac{1}{3}a$.

Proof (sketch). We consider two cases:

1. Case 1: $t_6 = 0$. Then by Lemma 3: $2t_7 \leq t_0$ and according to Lemma 4: $t_0 \leq t_3$. Hence, $4t_7 \leq 2t_0 \leq 2t_3 \leq a + t_7$ or $3t_7 \leq a$.
2. Case 2: $t_6 > 0$. The last move of type 6 divides the execution into two parts. The first part ends with the last move of type 6. The second part starts after the move and ends until stabilization. Clearly, this part is also not empty.
   The first part: $\ldots 7 \ldots 0 \ldots 0 \ldots 7 \ldots 6 \ldots 7 \ldots 6 \ldots 6 \ldots 0 \ldots 7 \ldots 6$.
   The second part: $\ldots 0 \ldots 7 \ldots 0 \ldots 0 \ldots 7 \ldots 0 \ldots 0 \ldots 0 \ldots 0 \ldots 7 \ldots 0 \ldots$.
   Let's denote by t_{01}, t_{02}, t_{31}, t_{32}, t_{71}, t_{72} the number of moves of types 0, 3 and 7 in the first and second parts, respectively. Clearly, $t_0 = t_{01} + t_{02}$, $t_3 = t_{31} + t_{32}$, $t_7 = t_{71} + t_{72}$.
   Then, the following holds:
   $2t_{72} \leq t_{02} \leq t_{32}$ — see the case 1.
   $2t_{71} \leq t_{01} + 2t_6$ — according to Lemma 4.
   $t_{71} + t_{01} + t_6 \leq t_{31}$ — any move of type 0, 6, 7 corresponds to a specific move of type 3 (the only collisions that can occur in the first part of the execution are $><$, $><_7$, $><_6$, $>_0<$). By the above inequalities, using LP techniques (details are omitted) we get: $t_7 \leq \frac{a}{3}$.

□

The following inequalities follow from the above:

Lemma 8

1. $t_3 \leq \frac{2}{3}a$.
2. $t_0 + t_6 + t_7 - t_3 \leq \frac{1}{3}a \leq \frac{1}{3}n$.
3. $t_0 + t_6 + t_7 + t_3 = O(n)$.

Proof. Omitted.

Using the inequalities of Lemma 8, we now proceed in two ways — as detailed in Section 2 — to derive the upper bound. We first use a potential function (Section 4.5), and then amortized analysis, which enables us to track the route made by each arrow, and thus achieve a tight bound (Section 4.6).

4.5 Upper Bound Using a Potential Function

We now introduce the function h. This function decreases by 1 during each move of types 1 or 2 and decreases by $(n+1)$ during each move of type 3. Unfortunately, moves of other types increase the function. This exhibits the main technical difficulty in applying this technique for bounding the time complexity. However, by combining results of the previous section and the properties of h we manage to derive the upper bound on the number of moves to reach stabilization.

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

$$h(C) \;=\; \sum_{\substack{1 \leq i \leq n-1 \\ x_{i-1} < x_i}} i \;+\; \sum_{\substack{1 \leq i \leq n-1 \\ x_{i-1} > x_i}} (n-i) \;\;. \tag{4}$$

The changes of the function h in each of the six possible types of moves are summarized in Table 2. These changes can be obtained directly from the definition of h. For example, for a move of type 0 we get that $\Delta h = (n-1) - (1) = n-2$, and for a move of type 3 $\Delta h = (0) - ((i+1) + (n-i)) = -(n+1)$.

Lemma 9. *For any configuration C, $0 \leq h(C) \leq \frac{3}{4}n^2$.*

Proof. Omitted.

Theorem 4. *The stabilization time of algorithm $\mathcal{A}$ is bounded by $1\frac{1}{12}n^2 + O(n)$.*

Proof. We denote by Δh_i the changes of h in a move of type i. Let C be the initial configuration of E. Considering the last (may be legitimate) configuration C' of E, we get $h(C') = h(C) + \sum_i t_i \cdot \Delta h_i$. By Lemma 9, $h(C') \geq 0$. Therefore $t_1 + t_2 \leq h(C) + \sum_{i \neq 1,2} t_i \cdot \Delta h_i \leq h(C) + (t_0 + t_6 + t_7 - t_3)(n-1)$. Applying Lemmas 9 and 8 (part 2), we get: $t_1 + t_2 \leq \frac{3}{4}n^2 + \frac{1}{3}n \cdot n = 1\frac{1}{12}n^2$. By Lemma 8 (part 3), the number of moves of other types is $O(n)$. □

4.6 Upper Bound Using Amortized Analysis

We define the weight of an arrow to be the number of moves of types 1 and 2 made by the arrow during its life-cycle. We also define the weight of a collision to be the sum of weights of the two arrows participating in the collision.

Clearly, the number of moves of both types 1 and 2 made by all arrows until stabilization equals the sum of weights of all collisions occurred in the execution.

Consider the i^{th} collision, for some $i \geq 1$. Denote by $a(i)$ the number of arrows in the configuration in which the collision i occurs. Denote by $t_7(i)$ the number of moves of type 7 made before the collision i. Clearly, $a - 2i + t_7(i) = a(i)$. The following key lemma is the main tool for estimating the weight of a collision.

Lemma 10. *Consider the i^{th} collision in the execution E. If the collision is of any type except $>_{60}<_7$, its weight is bounded by* $\max\{2n, 2(n-a+2i)\}$. *If the collision is of type $>_{60}<_7$, its weight is bounded by* $\max\{3n, 3(n-a+2i)\}$.

Proof (sketch). The initial configuration has a arrows and $n-a$ empty places where arrows are allowed to move. i collisions destroy $2i$ arrows. Assume first that $t_7(i) = 0$, then the number of empty places where the arrows participating in the collision are allowed to move is bounded by $n - a(i) = n - a + 2i$. And hence, the weight of the collision is bounded by $2(n - a(i)) = 2(n - a + 2i)$. Now assume $t_7(i) > 0$. The number of empty places is now $n - a(i) = n - a + 2i - t_7(i)$. But the weight of collision is bounded by $2(n - a(i) + t_7(i)) = 2(n - a + 2i)$. □

Using the last lemma we compute the tight bound on the number of moves until stabilization.

Theorem 5. *The stabilization time of algorithm $\mathcal{A}$ is bounded by $\frac{5}{6}n^2 + O(n)$.*

Proof (sketch). Note that Lemma 8 (part 3) bounds by $O(n)$ the number of moves of all types except for types 1 and 2. The number of these moves is bounded as follows. By Lemma 10 we get

$$t_1 + t_2 \leq \sum_{i=1}^{t_3} \min\{2(n-a+2i), 2n\} + 3n \ .$$

Using Lemma 8 (part 1), we then derive

$$t_1 + t_2 \leq \sum_{i=1}^{\frac{1}{2}a} 2(n-a+2i) + \sum_{i=\frac{1}{2}a}^{\frac{2}{3}a} 2n + 3n = \frac{4}{3}an - \frac{1}{2}a^2 + O(n) \ .$$

Since $0 < a < n$ it follows that $t_1 + t_2 \leq \frac{5}{6}n^2 + O(n)$. □

5 Analysis of Dijkstra's Algorithm

In this section we present an improved analysis for the upper bound of Dijkstra's algorithm. Our discussion includes three steps. We first discuss its lower bound, next we improve its upper bound by using the function h and finally provide a proof for a better upper bound using amortized analysis. We use the definitions and notations in earlier sections, in particular E is the prefix until stabilization of any given execution of Dijkstra's algorithm.

5.1 Lower Bound

Theorem 6. *The worst case stabilization time of Dijkstra's algorithm is at least* $1\frac{5}{6}n^2 - O(n)$*.*

Proof (sketch). Assume $n = 3k$. For any $0 \leq i \leq k-1$, we define configuration $C_i := \left[=^{3i}<^{3k-3i-1}\right]$. For example C_0 is $\left[<^{3k-1}\right]$ and C_{k-1} is $\left[=^{3(k-1)}<<\right]$. One can show (see [CSZ08b]) an execution that takes $3k + 27i + 13$ moves to go from C_i to C_{i+1}. Then the execution starting from C_0 takes: $\sum_{i=0}^{k-2}(3k + 27i + 13) = \frac{33}{2}k^2 - O(k)$ moves to reach C_{k-1}. Substituting $k = \frac{1}{3}n$ we get $1\frac{5}{6}n^2 - O(n)$. □

5.2 Extended Properties of Dijkstra's Algorithm

In this section we derive some properties of Dijkstra's algorithm. They refine the ones in [CSZ07], and enable us to improve the upper bound, as presented in the next section.

The following is easily observed by inspection of Table 3 describing Dijkstra's algorithm.

Observation 1 ([CSZ07]). *For any configuration C:*

1. *Any move of processor p_i, $1 \leq i \leq n-2$, does not change the function $\hat{f}$, i.e., $\Delta\hat{f} = 0$.*
2. *p_{n-1} is privileged according to case 7* **iff** *$\hat{f}(C) = 0$ and $x_{n-2} = x_{n-1}$.*
3. *p_{n-1} is privileged according to case 6* **iff** *$\hat{f}(C) = 2$ and $x_{n-2} > x_{n-1}$.*
4. *After processor p_{n-1} makes a move (case 6 or 7), we reach a configuration C such that $\hat{f}(C) = 1$.*

We summarize the changes of the functions $\hat{f}$ implied by each move in Table 3. In this table we also include the changes of function h (to which we will return back later).

For any execution E, we denote by $|E|$ the number of moves in E. Let a_r (resp. a_l) be the number of '>' (resp. '<') arrows in the initial configuration of given execution. Let also $a = a_l + a_r$. Moves of types 3, 4 and 5 are termed collisions. Intuitively, an execution with maximal number of moves must contain no moves of type 3, since such collisions destroy two arrows while other collisions destroy only one arrow. The following key lemma allows to focus on executions E with $t_3(E) = 0$, and is the basis for the amortized analysis in Section 5.3.

Lemma 11. *For every execution E, there is an execution E' containing no moves of type 3, such that $|E'| \geq |E| - O(n)$.*

Proof. See [CSZ08b].

In particular for any worst case execution E there is an execution E' ($t_3(E') = 0$) with the same number of moves up to a term of $O(n)$. As we are interested in $O(n^2)$ bounds, we will ignore this term and assume without loss of generality that a worst case execution does not contain type 3 moves.

From now on we consider only executions, such that $t_3 = t_3(E) = 0$.

Table 3. Dijkstra's algorithm

Type	**Proc.**	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{\Delta\hat{f}}$	$\mathbf{\Delta f}$	$\mathbf{\Delta h}$
0	p_0	$x_0 < x_1$	$x_0 > x_1$	$+1$	$+1$	$n-2$
1	p_i	$x_{i-1} > x_i = x_{i+1}$	$x_{i-1} = x_i > x_{i+1}$	0	0	-1
2	p_i	$x_{i-1} = x_i < x_{i+1}$	$x_{i-1} < x_i = x_{i+1}$	0	0	-1
3	p_i	$x_{i-1} > x_i < x_{i+1}$	$x_{i-1} = x_i = x_{i+1}$	0	-3	$-(n+1)$
4	p_i	$x_{i-1} > x_i > x_{i+1}$	$x_{i-1} = x_i < x_{i+1}$	0	-3	$3i-2n+2 \le n-4$
5	p_i	$x_{i-1} < x_i < x_{i+1}$	$x_{i-1} > x_i = x_{i+1}$	0	0	$n-3i-1 \le n-4$
6	p_{n-1}	$x_{n-2} > x_{n-1},\ \hat{f} = 2$	$x_{n-2} < x_{n-1}$	-1	-1	$n-2$
7	p_{n-1}	$x_{n-2} = x_{n-1},\ \hat{f} = 0$	$x_{n-2} < x_{n-1}$	$+1$	$+1$	$n-1$

Lemma 12 ([CSZ07])

1. *Assume $e \subseteq E$ is a segment delimited by any two successive moves of processor p_{n-1} where the second move is of type 6. Then $t_0(e) \ge 1$.*
2. *Assume $e \subseteq E$ is a segment delimited by any two successive moves of processor p_{n-1} where the second move is of type 7. Then $t_0(e) \ge 2$.*
3. *Assume $e \subseteq E$ is a phase. Then $t_4(e) \ge 1$.*

Lemma 13

1. *Assume $e \subseteq E$ is a segment delimited by any two successive moves of processor p_{n-1} where the second move is of type 6. Then $t_5(e) \ge 1$.*
2. $a_r - 2t_4 + t_5 + t_0 - t_6 \ge 0$.
3. $a_l - 2t_5 + t_4 - t_0 + t_6 + t_7 \ge 0$.

Proof (sketch). The proof of 1 is omitted. The proofs of 2 and 3 follow from counting the number of left and right arrows in any configuration. □

We summarize all constraints of Lemmas 12 and 13 in the following system:

$$\begin{cases} t_6 + 2t_7 \le t_0 \\ t_0 \le t_4 \\ t_6 \le t_5 \\ 0 \le a_r - 2t_4 + t_5 + t_0 - t_6 \\ 0 \le a_l - 2t_5 + t_4 - t_0 + t_6 + t_7 \\ a_l + a_r \le a \end{cases}$$

Using LP techniques (whose details are omitted here) it can be shown that:

Lemma 14

1. $t_7 \le \frac{2}{3}a$.
2. $t_0 \le \frac{4}{3}a$.
3. $t_4 + t_5 \le \frac{5}{3}a$.
4. $t_0 + t_4 + t_5 + t_6 + t_7 \le 3\frac{2}{3}a \le 3\frac{2}{3}n$.

5.3 Upper Bound Analysis

We now present the upper bound analysis for Dijkstra's algorithm. We start by using the tool of potential functions. It turns out that we can use the same function h above (see (4)). Clearly, all the properties of this function, including Lemma 9, are applied here too, and we can get:

Theorem 7. *The stabilization time of Dijkstra's algorithm is bounded by* $4\frac{5}{12}n^2 + O(n)$.

Proof (sketch). Since $t_3 = 0$: $t_1 + t_2 \leq h(C) + (t_0 + t_4 + t_5 + t_6 + t_7)(n-1)$. Applying Lemmas 9 and 14 (part 4), we get $t_1 + t_2 \leq \frac{3}{4}n^2 + 3\frac{2}{3}n \cdot n = 4\frac{5}{12}n^2$. □

As in the case of algorithm $\mathcal{A}$, the amortized analysis tool enables us to derive a better bound. We extend the notation from Section 4.6: a move of type 4 (resp. 5) destroys two arrows and creates a new arrow of type '$<_4$' (resp. '$>_5$').

Example 5. An arrow of type $<_{56}$ starts its life-cycle by being created by a move of type 5, then it reaches processor p_{n-1} and makes a move of type 6. Afterwards, it possibly makes some moves of type 2. Clearly, such an arrow may make at most $2n$ moves during its life-cycle.

Example 6. An arrow of type $>_{60}$ is possible in the execution. Assume an arrow makes a move of type 6, as the execution is before stabilization, the configuration contains other arrows (necessarily at the left side of the arrow). All these arrows must be destroyed to allow the arrow to reach processor p_0. At the same time new arrows may be created by type 7 moves, so that the system remains unstabilized.

Lemma 15. *The execution E may contain arrows of the following types only:* $>, >_5, >_0, >_{40}, >_{70}, <, <_4, <_7, <_6, <_{56}$.

Proof. Omitted.

Example 7. A collision of type $<_{56}<_{56}$ is a collision of two arrows having the same type $<_{56}$. Clearly, the weight of the collision is bounded by $4n$.

Lemma 16. *The execution E contains at most one collision of type $<_6<_{56}$, and at most $\frac{1}{6}t_0$ collisions of type $<_{56}<_{56}$.*

Proof (sketch). We omit the proof of the first part of the lemma. The second part is implied by the the fact that between two such collisions there are two moves of type 7 and two moves of type 6, and hence between every two collisions of that type there are at least 6 moves of type 0. □

Our purpose is to find an estimation for the sum of weights of all collisions occurred in the execution. Let's consider i^{th} collision ($i \geq 1$). We denote by $a(i)$ the number of arrows in the configuration in which the collision i occurs. Recall that a denotes the number of arrows in the initial configuration. Let's denote by $t_7(i)$ the number of moves of type 7 made before the collision i occurs. Since $t_3 = 0$, $a - i + t_7(i) = a(i)$ holds. The following key lemma allows tighter estimate of the weight of collisions (its proof is similar to that of Lemma 10):

Lemma 17. *Consider the i^{th} collision in the execution E. If the collision is of any type except $<_{56}<_{56}$ and $<_6<_{56}$, its weight is bounded by* $\min\{3n, 3(n-a+i)\}$. *If the collision is of type $<_{56}<_{56}$ or $<_6<_{56}$, its weight is bounded by* $\min\{4n, 4(n-a+i)\}$.

Using the last lemma we compute a tighter bound on the number of moves until stabilization.

Theorem 8. *The stabilization time of Dijkstra's algorithm is bounded by $3\frac{13}{18}n^2 + O(n)$.*

Proof (sketch). Note that Lemma 14 bounds by $O(n)$ the number of moves of all types except for types 1 and 2. In order to estimate the number $t_1 + t_2$ of these moves, we consider two cases:

1. The execution does not contain collisions of types $<_{56}<_{56}$ or $<_6<_{56}$. In this case

$$t_1+t_2 \leq \sum_{i=1}^{t_4+t_5} \min\{3(n-a+i), 3n\} = \sum_{i=1}^{a} 3(n-a+i) + \sum_{i=a}^{\frac{5}{3}a} 3n \leq 3\frac{1}{2}n^2 + O(n) \ .$$

2. The execution contains collisions of types $<_{56}<_{56}$ or $<_6<_{56}$. According to Lemma 16, the number of $<_{56}<_{56}$ collisions is bounded by $\frac{1}{6}t_0 \leq \frac{2}{9}a$, and the number of $<_6<_{56}$ collisions is bounded by one. In this case we estimate the total weight of all collisions by giving $4n$ weight to the last $\frac{2}{9}a + 1$ collisions:

$$t_1 + t_2 \leq \sum_{i=1}^{a} 3(n-a+i) + \sum_{i=a}^{\frac{5}{3}a-\frac{2}{9}a-1} 3n + \sum_{i=\frac{5}{3}a-\frac{2}{9}a-1}^{\frac{5}{3}a} 4n \leq 3\frac{13}{18}n^2 + O(n) \ .$$

□

6 Conclusion

In this work we presented a new three state self-stabilizing algorithm for mutual exclusion for a ring of processors, and showed a tight bound of $\frac{5}{6}n^2 + O(n)$ for its time complexity. For the upper bound we used two techniques: potential functions and amortized analysis; the first technique is simpler, but the second one leads to the tight bound. Our algorithm has a better worst case performance than two known three-state algorithms; namely, Dijkstra's algorithm and the one in [BD95]. We also improved the analysis of Dijkstra's algorithm and showed an upper bound of $3\frac{13}{18}n^2 + O(n)$ and a lower bound of $1\frac{5}{6}n^2 - O(n)$.

References

[BD95] Beauquier, J., Debas, O.: An optimal self-stabilizing algorithm for mutual exclusion on bidirectional non uniform rings. Proceedings of the Second Workshop on Self-Stabilizing Systems 13, 17.1–17.13 (1995)

[BGM89] Burns, J.E., Gouda, M.G., Miller, R.E.: On relaxing interleaving assumptions. In: Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89 (1989)

[BJM06] Beauquier, J., Johnen, C., Messika, S.: Brief announcement: Computing automatically the stabilization time against the worst and the best schedules. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 543–547. Springer, Heidelberg (2006)

[CG02] Cobb, J.A., Gouda, M.G.: Stabilization of general loop-free routing. Journal of Parallel and Distributed Computing 62(5), 922–944 (2002)

[CGR87] Chang, E.J.H., Gonnet, G.H., Rotem, D.: On the costs of self-stabilization. Information Processing Letters 24, 311–316 (1987)

[CSZ07] Chernoy, V., Shalom, M., Zaks, S.: On the performance of Dijkstra's third self-stabilizing algorithm for mutual exclusion. In: 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Paris, November 2007, pp. 114–123 (2007)

[CSZ08a] Chernoy, V., Shalom, M., Zaks, S.: On the performance of Beauquier and Debas' self-stabilizing algorithm for mutual exclusion. In: Shvartsman, M.M.A.A., Felber, P. (eds.) SIROCCO 2008. LNCS, vol. 5058, pp. 221–233. Springer, Heidelberg (2008)

[CSZ08b] Chernoy, V., Shalom, M., Zaks, S.: A self-stabilizing algorithm with tight bounds for mutual exclusion on a ring. Technical Report CS-2008-09, Department of Computer Science, Technion, Haifa, Israel (July 2008)

[Dij74] Dijkstra, E.W.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery 17(11), 643–644 (1974)

[Dij86] Dijkstra, E.W.: A belated proof of self-stabilization. Distributed Computing 1, 5–6 (1986)

[Dol00] Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)

[Kes88] Kessels, J.L.W.: An exercise in proving self-stabilization with a variant function. Information Processing Letters 29, 39–42 (1988)

[Kru79] Kruijer, H.S.M.: Self-stabilization (in spite of distributed control) in tree-structured systems. Information Processing Letters 8, 91–95 (1979)

[NKM06] Nakaminami, Y., Kakugawa, H., Masuzawa, T.: An advanced performance analysis of self-stabilizing protocols: stabilization time with transient faults during convergence. In: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Rhodes Island, Greece, April 25-29 (2006)

[TTK00] Tsuchiya, T., Tokuda, Y., Kikuno, T.: Computing the stabilization times of self-stabilizing systems. IEICE Transactions on Fundamentals of Electronic Communications and Computer Sciences E83A(11), 2245–2252 (2000)

Fast Distributed Approximations in Planar Graphs

Andrzej Czygrinow[1], Michal Hańćkowiak[2,⋆], and Wojciech Wawrzyniak[3,⋆⋆]

[1] Department of Mathematics and Statistics
Arizona State University
Tempe, AZ 85287-1804, USA
andrzej@math.la.asu.edu

[2] Faculty of Mathematics and Computer Science
Adam Mickiewicz University, Poznań, Poland
mhanckow@amu.edu.pl

[3] Faculty of Mathematics and Computer Science
Adam Mickiewicz University, Poznań, Poland
wwawrzy@amu.edu.pl

Abstract. We give deterministic distributed algorithms that given $\delta > 0$ find in a planar graph G, $(1 \pm \delta)$-approximations of a maximum independent set, a maximum matching, and a minimum dominating set. The algorithms run in $O(\log^* |G|)$ rounds. In addition, we prove that no faster deterministic approximation is possible and show that if randomization is allowed it is possible to beat the lower bound for deterministic algorithms.

1 Introduction

In recent years, there has been a growing interest in designing distributed approximation algorithms for special families of networks. In particular, efficient (in the model described below) distributed algorithms for some problems in constant-degree graphs, unit-disc graphs, or planar networks have been recently proposed. In contrast, for general networks most of the problems that admit easy solutions in special classes of graphs seem either unapproachable or are provably intractable (see for example [KMW04]). In this paper, we give deterministic distributed approximation algorithms for the maximum independent set, the maximum matching, and the minimum dominating set problems in planar graphs. The algorithms run $O(\log^* |G|)$ rounds and find a $(1 \pm \delta)$-approximation in a planar graph G. In addition, we prove lower bounds for the time complexity of deterministic approximation algorithms and show that if randomization is allowed then it is possible to beat the lower bound for deterministic procedures and give faster solutions.

[⋆] This work was supported by grant N206 017 32/2452 for years 2007-2010.

[⋆⋆] This work was supported by grant N206 017 32/2452 for years 2007-2010.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 78–92, 2008.

1.1 Model of Computations and Notation

We will work in a synchronous, message-passing model of computations (model *LOCAL* in [Pe00]). In this model a graph is used to represent an underlying network. Vertices of the graph correspond to computational units, and edges represent communication links. The network is synchronized and in one round a vertex can send, receive messages from its neighbors, and can perform some local computations. Neither the amount of local computations nor the size of messages sent is restricted in any way. Consequently, in this model in a graph of diameter t, any graph-theoretic function can be computed in $O(t)$ rounds. In addition, we assume that vertices have unique identifiers from $\{1, \ldots, |G|\}$ where $|G|$ is the order of the underlying graph G. In some applications graphs will have additional weights. The interpretation of the weights depends on specific applications and they do not impact the communication in any way. We will mostly follow [D05] in graph-theoretic terminology. In particular, we will use $|G|$ to denote the order of graph G and $||G||$ to denote the size of G.

1.2 Related Work

Theory of distributed approximation algorithms has attracted some attention recently. For a nice overview of important results in this area the reader is referred to the survey by Elkin [E04]. Although there are very few deterministic distributed approximation algorithms that run in $o(|G|)$ rounds in a general graph G, in the case when the underlying network has additional properties, algorithms that give a non-trivial approximation are much easier to design. The most eminent example is the classical algorithm of Cole and Vishkin from [CV86]. The algorithm finds in $O(\log^* |G|)$ rounds a maximal independent set in a constant-degree graph G and provides therefore a constant approximation for the maximum independent set problem in this type of a network. Results of Linial ([L92]) and Naor ([N91]) give matching $\Omega(\log^* |G|)$ lower bounds for the running time of deterministic and randomized distributed algorithm that find a maximal independent set in a cycle and show that the $\log^* |G|$ running time cannot be beaten if one expects exact, non-approximate, solutions. Similarly, in the case of unit-disk graphs, it is possible (see [KMNW05a], [KMNW05b], [CH06b], or [SW08]) to give fast approximation algorithms for many graph-theoretic problems that seem intractable in general networks.

In planar graphs, approximations that run in the poly-logarithmic number of rounds and give the approximation error of $(1 \pm O(1/\log^k |G|)$ are known for all problems discussed in this paper ([CHS06], [CH06a]). However if one is willing to accept a larger approximation error (for example $(1 \pm \delta)$) but in a much faster fashion (say in $O(\log^* |G|)$ or $O(1)$ rounds) then the methods from [CHS06] or [CH06a] do not give any indication if such algorithms are possible to obtain. In addition, it has not been clear if approximation problems are significantly easier than the original Maximal Independent Set problem and if it is possible to beat the $\log^* |G|$ bound and give $O(1)$-running time algorithms that find approximate solutions. In this direction, very recently and independently of the work in this paper, Lenzen, Oswald, and Wattenhofer ([LOW08]) gave a 71-approximation for the minimum dominating set problem that runs in $O(1)$ rounds and Lenzen, Wattenhofer proved in [LW08] a lower bound which is identical with our result from Section 4.1. The arguments from [LW08] and Section 4.1 are completely different.

1.3 Results

We will prove a few results on distributed approximations in planar graphs. The main result is a collection of deterministic distributed algorithms which approximate a maximum independent set, a maximum matching, and a minimum dominating set in planar graphs. The algorithms run in $O(\log^* |G|)$ rounds. In addition, we prove that no deterministic algorithm can run faster and give an approximation error achieved by our algorithms. At the same time, we note that an easy randomized procedure beats the lower bound for deterministic algorithms and with high probability finds a $(1 \pm \delta)$-approximation and runs in $O(1)$ rounds.

More formally, we show that there is a deterministic distributed algorithm which for a given $\delta > 0$ finds in a planar graph G an independent set I of size which is at least $(1 - \delta)\alpha(G)$ where $\alpha(G)$ is the independence number of G. The algorithm runs in $O(\log^* |G|)$ rounds (Theorem 4). Algorithms with a similar performance can be obtained for the maximum matching problem (Theorem 5), and for the minimum dominating set problem (Theorem 6). In addition, in the case of the maximum independent set problem, the weighted version of the problem can be approximated in $O(\log^* |G|)$ rounds (Theorem 4). These results are complemented by some lower bounds. We prove that for any $c > 0$, no deterministic distributed algorithm can find a c-approximation of the maximum independent set in a planar graph G in $o(\log^* |G|)$ rounds, nor there is a deterministic distributed algorithm that finds a c-approximation of a maximum matching in a planar graph G in $o(\log^* |G|)$ rounds (Section 4.1). In the case of the dominating set problem the situation is different and it is possible to find a $O(1)$-approximation in $O(1)$ rounds (see [LOW08] or Section 3.3) but for any $\delta > 0$ there is no deterministic algorithm which in $o(\log^* |G|)$ rounds finds a $(5 - \delta)$-approximation of a minimum dominating set in a planar graph G (Section 4.1). Finally, we note that an easy randomized procedure can find with high probability a $(1-\delta)$-approximation of the maximum independent set in a planar graph in $O(1)$ rounds (Section 4.2) and so in the case randomness is allowed, approximation is significantly easier than solving the Maximum Independent Set problem.

1.4 Organization

In the next section, we describe a partitioning algorithm which is used in Section 3 to obtain deterministic approximations for the maximum independent set, the maximum matching, and the minimum dominating set problems. In Section 4 we give lower bounds and discuss randomized algorithms.

2 Clustering Algorithm

In this section, we give a deterministic algorithm which in $O(\log^* |G|)$ rounds finds a partition of a weighted planar graph with the property that the weight of the edges between different partition classes is significantly smaller than the total weight of the graph. The procedure is invoked in the next section to give approximation algorithms. We will consider weighted graphs with weights defined on vertices as well as weights defined on edges. For a graph $G = (V, E)$ we will use $\omega : V \to R^+$ to denote a vertex-weight function and $\bar{\omega} : E \to R^+$ to denote a edge-weight function. In addition, we will sometimes slightly abuse the notation and if F is a subgraph of $G = (V, E)$ with $\bar{\omega} : E \to R^+$ (or $\omega : V \to R^+$) then $\bar{\omega}(F)$ (or $\omega(F)$) will be used to denote $\bar{\omega}(E(F))$ (or $\omega(V(F))$). If P is a path then the length of P will be the number of edges in P (i.e $|P| - 1$).

In the course of computations we will be contracting connected subgraphs of a planar graph and recomputing the weights. Specifically, if $G = (V, E)$ is a planar graph, $\bar{\omega} : E \to R^+$, and $U_1, U_2, \ldots, U_l$ are pair-wise disjoint subsets of V such that $G[U_i]$ is connected then we define $\tilde{G}$ to be the weighted graph in which every U_i is contracted to a vertex and for $u, u' \in V(\tilde{G})$ with $u \neq u'$ we set

$$\bar{\omega}_{\tilde{G}}(u, u') = \sum_{x \in U, y \in U'} \bar{\omega}(x, y) \tag{1}$$

where $U = U_i$ if u is obtained by contracting U_i and $U = \{u\}$ otherwise (the same for U'). We proceed with a few auxiliary definitions and facts.

Definition 1. *A directed graph $\overrightarrow{F}$ such that the maximum out-degree in F is one is called a pseudo-forest.*

If $\overrightarrow{F}$ is a directed graph then we use F to denote the graph obtained from $\overrightarrow{F}$ by ignoring the orientation of edges.

Fact 1. *Let $G = (V, E)$ be a planar graph with edge-weight function $\bar{\omega} : E \to R^+$. There is a distributed procedure which in two steps finds a pseudo-forest $\overrightarrow{F}$ such that F is a subgraph of G and $\bar{\omega}(F) \geq \bar{\omega}(G)/6$.*

Proof. First we show that there exists a pseudo-forest $\overrightarrow{F}$ such that F is a subgraph of G and $\bar{\omega}(F) \geq \bar{\omega}(G)/3$. Indeed, since G is planar its edge set is the union of at most 3 forests by the theorem of Nash-Williams. Now, select the heaviest forest and root the trees to obtain the desired graph F with $\bar{\omega}(F) \geq \bar{\omega}(G)/3$. Next we prove that a desired pseudo-forest can be obtained by a distributed procedure. Consider the following simple algorithm. Every vertex v selects the heaviest edge $\{u, v\}$ incident to v and puts the orientation from v to u. If an edge has been assigned the orientation in both directions then one of them is selected arbitrarily. Clearly the obtained graph is a pseudo-forest. Since G is the union of three forests F_1, F_2, F_3 and $2\bar{\omega}(F) \geq \max\{\bar{\omega}(F_i)\}$, we have $\bar{\omega}(F) \geq \bar{\omega}(G)/6$. □

In addition we have the following simple fact.

Fact 2. *If $\overrightarrow{F}$ is a connected pseudo-forest such that the diameter of F is d then $\overrightarrow{F}$ contains a directed path of length at least $d/2$.*

We develop some more notation which will be used in the next procedure. Let $\overrightarrow{F}$ be a pseudo-forest vertices of which are properly colored with colors from some set S. For a vertex v and for a set of colors $C \subset S$, let $in(v, C)$ be the set of arcs (u, v) (from u to v) such that the color of u is in C and let $out(v, C)$ be defined analogously. If C is empty then $in(v, C)$ and $out(v, C)$ are empty and their weights are equal to zero.

```
HeavyStars
1. Find a pseudo-forset $\overrightarrow{F}$ in $G$ using the procedure from Fact 1.
2. Use Cole-Vishkin algorithm [CV86] to properly color the vertices of $\overrightarrow{F}$ using colors from $\{1, 2, 3\}$.
3. For every non-isolated vertex $v$ in parallel:
   (a) If color of $v$ is 1 then $v$ marks all edges from $in(v,\{2,3\})$ if $\bar{\omega}(in(v,\{2,3\})) > \bar{\omega}(out(v,\{2,3\}))$ and $v$ marks the edge from $out(v,\{2,3\})$ otherwise.
   (b) If color of $v$ is 2 then $v$ marks all edges from $in(v,\{3\})$ if $\bar{\omega}(in(v,\{3\})) > \bar{\omega}(out(v,\{3\}))$ and $v$ marks the edge from $out(v,\{3\})$ otherwise.
4. Let $Q_i$'s denote connected components of the graph induced by marked edges. In parallel, find in each $Q_i$ vertex-disjoint stars with weight of at least $\bar{\omega}(Q_i)/2$ and return them. (This is easily accomplished by rooting $Q_i$'s and considering odd and even levels.)
```

Lemma 1

$$diam(Q_i) < 10.$$

Proof. Suppose $diam(Q_i) \geq 10$. Then from Fact 2, there is a directed path of length at least 5. If there is an internal vertex v in the path of color 1 then either the edge coming to v or coming out of v (but not both) is marked by 3(a). Otherwise, since the length is at least 5, there must be an internal vertex of color 2 with both of its neighbors of color 3 and by 3(b) only one of these edges can be marked. □

Lemma 2. HeavyStars *returns stars of weight at least $\bar{\omega}(G)/24$ and runs in $O(\log^* |G|)$ rounds.*

Proof. From Fact 1, we have $\bar{\omega}(F) \geq \bar{\omega}(G)/6$. Every edge of F has either one endpoint in color 1 and the second from $\{2, 3\}$ or one endpoint in color 2 another in color 3. Consequently the edges considered in steps 3(a) and 3(b) form a partition of $E(F)$ and so the weight of the union of Q_i's is at least half of the the weight of F. Finally, the stars will have at least half of the weight of Q_i's and so the weight of them is at least $\bar{\omega}(G)/24$. The first, third, and the fourth step require $O(1)$ rounds and the coloring from step two can be found in $O(\log^* |G|)$ rounds. □

We will now describe the clustering algorithm. The procedure takes $0 < \epsilon < 1$ as an input.

```
Clustering
1. $H := G$
2. Iterate $\lceil \left(\log\left(\frac{1}{\epsilon}\right) / \log\left(\frac{24}{23}\right)\right) \rceil$ times:
   (a) Call HeavyStars to find vertex-disjoint stars in $H$.
   (b) Modify $H$ by contracting each star to a vertex and computing the weights
       as in (1).
3. Let $W$ denote the set of vertices contracted to $w$. Return $\{W | w \in V(H)\}$.
```

Lemma 3. *Given* $0 < \epsilon < 1$, Clustering *finds a partition* $(V_1, \ldots, V_l)$ *of* $V(G)$ *such that if* $\tilde{G}$ *is obtained by contracting each of* V_i*'s and recomputing the weights as in* (1) *then*

$$\bar{\omega}(\tilde{G}) \leq \epsilon \bar{\omega}(G).$$

The algorithm runs in $O(\log^* |G|)$ *rounds.*

Proof. From Lemma 2 in each iteration the algorithm contracts the stars of weight which is at least $1/24$ of the total weight of the graph. Consequently after l iterations the weight of graph H is at most $\left(\frac{23}{24}\right)^l \bar{\omega}(G) \leq \epsilon \bar{\omega}(G)$ if $l = \lceil \left(\log\left(\frac{1}{\epsilon}\right) / \log\left(\frac{24}{23}\right)\right) \rceil$. The running time is $O(\log^* |G|)$ from Lemma 2. □

3 Approximating Algorithms

In this section, we will use the clustering procedure from the previous section to give deterministic distributed approximations.

3.1 Maximum-Weight Independent Set

We will start with the maximum-weight independent set problem. Let $G = (V, E)$ be a planar graph with $\omega : V \rightarrow R^+$. For an edge $\{u, v\} \in E$, define

$$\bar{\omega}(\{u, v\}) = \min\{\omega(u), \omega(v)\}. \tag{2}$$

We have the following fact.

Fact 3

$$\bar{\omega}(G) \leq 3\omega(G).$$

Proof. From Nash-Williams theorem G has an orientation such the out-degree is at most three. For an oriented edge (u, v) (from u to v) we have the weight of the edge to be at most $\omega(u)$. Since the out-degree is at most three, a vertex can be the starting point of at most three edges. □

To approximate a maximum-weight independent set we will invoke a modified procedure from [CH06c]. This algorithm proceeds as follows. First, consider the edge-weighted graph with weights from (2) and invoke CLUSTERING to find a partition $(V_1, \dots, V_l)$ of $V(G)$. Then find optimal independent sets I_i in each of the $G[V_i]$. (Note that the diameter of $G[V_i]$ is $O(1)$.) Finally correct the solution obtained in the previous step by deleting u from I_i if $\{u, v\}$ is the edge with $v \in I_j$ and $\omega(u) < \omega(v)$ (In the case the weights are equal use the identifiers to break the symmetry). Using a similar argument to the one from [CH06c] we obtain:

Theorem 4. *There is a deterministic distributed algorithm which for given* $0 < \delta < 1$ *finds in a planar weighted graph* $G = (V, E)$ *with* $\omega : V \to R^+$ *an independent set* I *of weight which is at least* $(1 - \delta)OPT$ *where* OPT *denotes the weight of a max-weight independent set. The algorithm runs in* $O(\log^* |G|)$ *rounds.*

Proof Outline. Since G is planar, $OPT \geq \omega(G)/4$ as $\chi(G) \leq 4$. Let I be the union of I_i's before the correction step. Then $\omega(I) \geq OPT$ as the procedure finds the optimal solution in each of $G[V_i]$'s and the restriction of any independent set to V_i is an independent set. In the correction step, for every edge between two different clusters we delete one of its endpoints, weight of which is at most the weight of the edge by (2). Consequently, the total weight of deleted vertices is at most the weight of edges between different clusters. By Fact 3, the weight of deleted vertices is at most

$$\epsilon\bar{\omega}(G) \leq 3\epsilon\omega(G) \leq 12\epsilon OPT = \delta OPT$$

provided $\epsilon = \delta/12$. □

Unlike the maximum independent set problem, in the case of matchings and dominating sets we can only give algorithms for the un-weighted versions of these problems. The reason is very simple, since G is planar, we know that the optimal solution to the maximum-weight independent set problem is of size proportional to $\omega(G)$. This however is not the case for the weight of an optimal solution to the max-wight matching or the min-weight dominating set problem. In fact, even when the weights are all equal to one, an optimal solution to the above two problems can be substantially smaller than the order of the graph. However, in the case a graph G is un-weighted, one can use simple preprocessing to reduce G to a graph where optimal solution is of size which is proportional to its order.

3.2 Maximum Matching

In the case of the maximum matching problem we can adopt the ideas from [CHS06] to show:

Theorem 5. *There is a deterministic distributed algorithm which for given* $0 < \delta < 1$ *finds in a planar graph* $G = (V, E)$ *in* $O(\log^* |G|)$ *rounds a matching* M *such that*

$$|M| \geq (1 - \delta)\beta(G)$$

where $\beta(G)$ *is the size of a maximum matching in* G*.*

Proof Outline. The algorithm is similar to the one from [CHS06]. First, from [CHS06] (Lemma 6), we know that if induced stars of size at least two and induced double stars of size at least three are eliminated from G so that only one edge from a star and two paths of length two from each double star are left (which can be done in $O(1)$ rounds) then the matching in the new graph G' will be of size $\Omega(|G'|)$. Since at most one edge from a single star and at most two edges from a double star can be in a matching in G, we have $\beta(G) = \beta(G')$. Second, in G' we invoke the procedure Clustering and find a partition of G' (the edge weights are initially equal to one). Finally, in each subgraph induced by partition classes we find an optimal solution, which can be done in $O(1)$ rounds, and return the union. The fact that the error of approximation is as claimed can be verified in the same way as in the argument for Theorem 4. □

3.3 Minimum Dominating Set

In the case of the dominating set, we must do some more preprocessing. Specifically, the algorithm first finds a constant approximation of the minimum dominating set in $O(1)$ rounds and then proceeds to improve this approximation and finds a dominating set of size $(1 - \delta)\gamma(G)$ where $\gamma(G)$ is the size of a minimum dominating set in G. To complete the second step $O(\log^* |G|)$ rounds are needed. As noted in the introduction, recently in [LOW08], a $O(1)$-rounds 71-approximation of the minimum dominating set for planar graphs is given. Although the constant approximation is not the focus of this paper we briefly describe an alternative way for finding a $O(1)$-approximation in Section A. Using a similar argument as in the case of matchings we have:

Theorem 6. *There is a deterministic distributed algorithm which for given* $0 < \delta < 1$ *finds in a planar graph* $G = (V, E)$ *in* $O(\log^* |G|)$ *rounds a dominating set* D *such that*

$$|D| \leq (1 + \delta)\gamma(G)$$

where $\gamma(G)$ *is the size of a minimum dominating set in* G*.*

Proof. The algorithm is similar to the one from [CH06a]. After a constant approximation is obtained using the procedure from Lemma 9 from Section A or the algorithm from [LOW08], we proceed in the following fashion. Let $D = \{w_1, \ldots, w_k\}$ denote the dominating set obtained from the preprocessing phase. Then

$$|D| \leq c\gamma(G) \tag{3}$$

for some constant c. A partition $(W_1, \ldots, W_k)$ of $V(G)$ is obtained by every vertex of G joining the group of exactly one of the vertices from D that dominates it. Then $k = |D|$ and each V_i induces a graph of diameter at most two in G. We contract V_i's to obtain a planar graph H and set the weights of the edges to be equal to one. Note that $||H|| < 3k$. Set $\epsilon = \delta/(6 \cdot c)$ and use Clustering to find a partition $(V_1, \ldots, V_l)$ of $V(H)$ which by Lemma 3 is such that the weight of the edges between different clusters is at most $\epsilon||H||$. Un-contract V_i's and W_i's to obtain a partition $(U_1, \ldots, U_l)$ of $V(G)$. Finally in each of $G[U_i]$'s (which has a constant diameter) find an optimal dominating set D_i and return the union of these dominating sets. The running time of the algorithm is $O(\log^* |G|)$. To prove the approximation error, let D^* be a minimum dominating set in G and let $\bar{D}$ be obtained from D^* by adding all vertices w_i from D with the property that a vertex in $W_i \subset U_j$ has a neighbor in $V \setminus U_j$. We have

$$|\bar{D}| \leq |D^*| + 2\epsilon||H|| < \gamma(G) + 6\epsilon|D| \leq \gamma(G)(1+\delta)$$

from (3). In addition

$$|\bigcup D_i| \leq |\bar{D}|$$

as $\bar{D} \cap U_i$ is a dominating set in $G[U_i]$ and D_i is an optimal solution in $G[U_i]$. Therefore, $|\bigcup D_i| \leq (1+\delta)\gamma(G)$. □

4 Lower Bounds and Randomization

In this section, we will establish lower bounds for deterministic approximation algorithms and discuss randomized procedures.

4.1 Lower Bounds

Our lower bounds will be based on the general Ramsey's theorem (see for example [We01]). It is known (see [Pa99]) that Ramsey's theorem can be used to argue that no deterministic distributed algorithm can properly color a cycle C with $O(1)$ colors and run in $o(\log^* |C|)$ rounds. We will first use this method to establish similar results for approximation algorithms. Let $R(2, m; l)$ denote the least number of vertices n such that in any 2-coloring of the edges of the complete l-uniform hypergraph on n vertices there is a monochromatic complete sub-hypergraph on m vertices. The general Ramsey's theorem shows that $R(2, m; l)$ is finite and a proof of the theorem provides a tower-type upper bound (height of tower is proportional to l) for $R(2, m; l)$.

Lemma 4. *For any positive integer T there is no deterministic distributed algorithm that finds in a cycle on n vertices an independent set of size $\Theta(n/\log^{(2T)} n)$ in T rounds. There is no deterministic distributed algorithm that finds an independent set of size $\Theta(n/\log^* n)$ in a cycle on n vertices in $o(\log^* n)$ rounds.*

Proof. For notational convenience we will assume that if $S = \{i_1, \ldots, i_l\}$ is a subset of $[n] := \{1, \ldots, n\}$ with l elements then the elements are indexed so that $i_k < i_l$ when $k < l$. Let C be a cycle with $V(C) = [n]$. For a distributed algorithm A that finds in T rounds an independent set in C, let $F_A : \binom{[n]}{2T+1} \to \{0, 1\}$ be defined by $F_A(\{i_1, \ldots, i_{2T+1}\}) = 1$ if and only if i_{T+1} is selected by A to the independent set provided $i_1, i_2, \ldots i_{2T+1}$ is a path in C. Then F_A is a 2-coloring of the edges of the complete $(2T+1)$-uniform hypergraph H with $V(H) = [n]$. Let m be such that $n \geq R(2, m; 2T+1)$. Then, from the Ramsey's Theorem, H contains a monochromatic complete hypergraph K on m vertices. Observe that if $m > 2T+1$ then the color of the edges in K cannot be one. Indeed, if $\{i_1, \dots, i_{2T+2}\}$ is a subset of $V(K)$ then by definition of F_A, i_{T+1} and i_{T+2} are selected by A to the independent set if the path $i_1, \dots, i_{2T+2}$ is a subgraph of C. Consequently every edge in K has color zero and if $K = \{v_1, \dots, v_m\}$ then none of $v_{T+1}, \dots, v_{m-T}$ is in the independent set returned by A provided $P := v_1, \dots, v_m$ is the subgraph of C. Therefore, out of m vertices in K, $m - 2T$ are not in the independent set returned by A. If $n - m \geq R(2, m; 2T + 1)$ then we can repeat the above argument to the hypergraph induced by $[n] \setminus V(K)$. Let p denote the number of times we will repeat the above reasoning. Then the size of the independent set returned by the algorithm A when the ordering of vertices in C is determined by the repeated application of the Ramsey's theorem is at most $2Tp + R(2, m; 2T + 1)$ which is at most $2nT/m + R(2, m; 2T + 1)$ as $pm \leq n$. It is known (see [N95]) that for some constant c,

$$R(2, m; 2T + 1) \leq 2^{2^{\cdots^{2^{cm}}}}$$

where the number of 2's in the tower is $2T$. Thus for any constant T if $m = \Theta(\log^{(2T)} n)$ then the size of the independent set is at most $O(n/\log^{(2T)} n)$. In addition, very similar computations give that for the size of an independent set to be $\Omega(n/\log^* n)$, T must be $\Omega(\log^* n)$. Indeed, let $\epsilon > 0$ and for n large enough let m and T be two integers with $(\log^* n)^2/(2c) \leq m \leq (\log^* n)^2/c$ and $\epsilon \log^* n/(8 \cdot c) \leq 2T + 1 < \epsilon \cdot \log^* n/(4 \cdot c)$. Then

$$2nT/m < \epsilon n/(2 \log^* n).$$

At the same time, for n large enough,

$$\log^{(2T+1)} R(2, m; 2T + 1) \leq 2 \log \log^* n$$

and $2 \log \log^* n < 0.5 \log^{(2T+1)} n < \log^{(2T+1)} (\epsilon n/(2 \log^* n))$. Consequently, the size of the independent set returned by the algorithm is smaller than $\epsilon n/\log^* n$. □

A very similar lower bound can be obtained for matchings. In the case of the dominating set the situation is different and it is possible to find a $O(1)$-approximation in zero rounds. On the other hand, one can show that no deterministic α-approximation with $\alpha < 2$ can run in $o(\log^* |G|)$ rounds. We will prove something slightly stronger for planar graphs.

Fact 7. *There is no deterministic distributed algorithm which for every* $\delta > 0$ *finds in* $o(\log^* |G|)$ *rounds a dominating set of size which is at most* $(5 - \delta)\gamma(G)$ *in a planar graph* G.

Proof. Let $\delta > 0$ be fixed and suppose that there is a deterministic distributed algorithm that finds a dominating set in any planar graph G of size at most $(5 - \delta)\gamma(G)$. From a cycle C on n vertices (with $10|n$), we create a graph G in the following way. Let $G = (V, E)$, where $V = V(C)$ and $E = E(C) \cup \{v, u \in V, d_C(v, u) = 2\}$. Note that this virtual graph can be obtained from C by a distributed algorithm, G is 4-regular, $\gamma(G) = n/5$, and G is planar. From a dominating set D in G returned by the algorithm we can obtain an independent set I in C as follows. Consider $C[D]$ and add to I all isolated vertices. For every vertex $v \in D$ with $deg_C(v) = 1$, if the neighbor of v, w, has $deg_C(w) = 1$ then add the vertex with a smaller identifier to I and if $deg_C(w) = 2$ then add v to I. Let D_i be the subset of D with vertices of degree i in $C[D]$. Note that if $v \in D_2$, that is v has degree two in $C[D]$, then every vertex from $V \setminus D$ which is dominated by v in G is also dominated by a vertex from D_1. Therefore, we have $|I| \geq |D_0| + |D_1|/2$ and $4|D_0| + 3|D_1| \geq n - |D|$. Since $|D| \leq (5-\delta)\gamma(C) = (1-\delta/5)n$, $4|D_0| + 3|D_1| \geq \delta n/5$ and so $|I| \geq |D_0| + |D_1|/2 \geq \delta n/30$. By Lemma 4 this is not possible. □

4.2 Randomized Algorithms

In this section we will briefly discuss randomized algorithms. We focus on the maximum independent set problem as algorithms for other problems can be obtained using similar consideration.

Lemma 5. *Let G be a graph with maximum degree at most Δ. Then there is distributed randomized algorithm which in two rounds finds an independent set I in G such that with probability larger than $1 - \exp\{-|G|/(2^{\Delta+4}(\Delta^2+1))\}$ the size of I is at least $|G|/(2^{\Delta+2}(\Delta^2+1))$.*

Proof. The algorithm is trivial and we will only outline it. It proceeds in two rounds and in the first round every vertex marks itself with probability 1/2, choices made independently. In the second round a marked vertex un-marks itself if at least one of its neighbors is marked. Let S be a maximal set of vertices at distance at least three in G. Note that $|S| \geq |G|/(\Delta^2+1)$ and the events that two vertices from S are marked after the second round are independent. The expected number of vertices selected from S is at least $|G|/(2^{\Delta+1}(\Delta^2+1))$ and by Chernoff's bound the probability that it is smaller than $|G|/(2^{\Delta+2}(\Delta^2+1))$ is less than $\exp\{-|G|/(2^{\Delta+4}(\Delta^2+1))\}$. □

Lemma 6. *Let K be a positive integer and let $G = (V, E)$ be a weighted planar graph with $\bar{\omega} : E \to \{1, \dots K\}$, the maximum degree Δ, and with no isolated vertices. Then there exist pair-wise disjoint subsets $V_1, \dots, V_l$ of $V(G)$ such that*

- *$diam(G[V_i]) \leq 2(M-1)$;*
- *$\Delta(\tilde{G}) \leq \Delta^M$, $\max\{\bar{\omega}(e) | e \in E(\tilde{G})\} \leq K\Delta^M$;*
- *$\bar{\omega}(\tilde{G}) \leq 9\bar{\omega}(G)/10$*

where $M := \lceil \log_{(1-1/(5\cdot 82\cdot 2^{10}))} \left(\frac{1}{20K\Delta}\right) \rceil$ and $\tilde{G}$ is obtained from G by contracting V_i's and recomputing the weights as in (1).

Proof. Let $\epsilon := 1/(20K\Delta)$ and let M be as above. Let $I = \{v \in V(G) | deg_G(v) \leq 9\}$. Clearly $|I| \geq 2|G|/5$. Use the algorithm from Lemma 5 to find an independent set $I_1 \subseteq I$ with $|I_1| \geq |G|/(5 \cdot 82 \cdot 2^{10})$. Repeat the process in $V \setminus I_1$ to find I_2 of size $(|G| - |I_1|)/(5 \cdot 82 \cdot 2^{10})$ and continue M times. Let $I_1, \dots, I_M$ be independent sets obtained by repeating the above procedure M times. Then with parameters as above every vertex $v \in I_j$ has at most nine neighbors in $V \setminus (I_1 \cup \dots \cup I_j)$ and with high probability, $\sum |I_j| \geq (1-\epsilon)|G|$. For every j and every $v \in I_j$, v selects one edge from v to the set $V \setminus (I_1 \cup \dots \cup I_j)$ of the largest weight if it has at least one neighbor in $V \setminus (I_1 \cup \dots \cup I_j)$. The subgraph obtained in this way is a forest with trees $T_1, \dots, T_l$ each of diameter at most $2(M-1)$. Let $V_i := V(T_i)$. Note that the total weight of edges incident to $V \setminus (I_1 \cup \dots \cup I_M)$ is at most $K \cdot \Delta \cdot \epsilon|G|$ provided $\sum |I_j| \geq (1-\epsilon)|G|$. Therefore,

$$\sum \bar{\omega}(T_i) \geq (\bar{\omega}(G) - K \cdot \Delta \cdot \epsilon|G|)/9 \geq \bar{\omega}(G)/10$$

as $\epsilon = 1/(20K\Delta)$ and $\bar{\omega}(G) \geq |G|/2$. In addition, $\Delta(\tilde{G}) \leq \Delta^M$ as there are at most $(\Delta^M - 1)/(\Delta - 1)$ vertices in T_i and consequently $\max\{\bar{\omega}(e)|e \in \tilde{G}\} \leq K\Delta^M$. Finally, note that the probability that the above procedure fails is at most $O(M \exp\{-O(\epsilon|G|)\}$ by Lemma 5 as if in the course of computations $|G| - (|I_1| + \dots + |I_q|) \leq \epsilon|G|$ for some $q < M$ then we have $\sum |I_j| \geq (1-\epsilon)|G|$ with probability one. □

Corollary 8. *There exists a randomized distributed algorithm which for a given $\delta > 0$ finds in the constant number of rounds (the constant depends on δ only) an independent set I in a planar graph G the size of which is with high probability larger than or equal to $(1-\delta)\alpha(G)$.*

Proof. Let $R := 48/\delta$ and let $\bar{V} = \{v \in V(G)|deg_G(v) > R\}$. Then $|\bar{V}| \leq \delta\alpha(G)/2$ as $\alpha(G) \geq |G|/4$. Let G' be the subgraph of G induced by $V \setminus \bar{V}$ and define $\bar{\omega}(e) := 1$ when $e \in E(G')$. We apply Lemma 6 L times with L such that $(0.9)^L \leq \delta/16$ contracting subsets repeatedly and putting aside isolated vertices if any arise. Now we consider partition of V into $V_1, \dots, V_l$ obtained by un-contracting the vertices (some of the V_i's can be singletons). We have $diam(G[V_i]) \leq C$ where C depends on δ only and we call a vertex $v \in V_i$ *border* if it has a neighbor in $V(G) \setminus V_i$. Disregard all the border vertices to obtain $\bar{V}_i$'s and find optimal independent sets I_i's in $G[\bar{V}_i]$'s. Finally return the union of I_i's. The solution returned by the procedure has size of at least $\alpha(G) - |B|$ where B is the set of border vertices and $|B| \leq \delta\alpha(G)/2 + 2(0.9)^L|G| \leq \delta\alpha(G)$. □

Acknowledgment

The authors are grateful to referees for many useful comments and suggestions.

References

[AGLP89] Awerbuch, B., Goldberg, A.V., Luby, M., Plotkin, S.A.: Network Decomposition and Locality in Distributed Computation. In: Proc. 30th IEEE Symp. on Foundations of Computer Science, pp. 364–369 (1989)

[CV86] Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control 70, 32–53 (1986)

[CH06a] Czygrinow, A., Hańćkowiak, M.: Distributed almost exact approximations for minor-closed families. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 244–255. Springer, Heidelberg (2006)

[CH06b] Czygrinow, A., Hańćkowiak, M.: Distributed approximation algorithms in unit disc graphs. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 385–398. Springer, Heidelberg (2006)

[CH06c] Czygrinow, A., Hańćkowiak, M.: Distributed algorithms for weighted problems in sparse graphs. Journal of Discrete Algorithms 4(4), 588–607 (2006)

[CHS06] Czygrinow, A., Hańćkowiak, M., Szymanska, E.: Distributed approximation approximations for planar graphs. In: Calamoneri, T., Finocchi, I., Italiano, G.F. (eds.) CIAC 2006. LNCS, vol. 3998, pp. 296–307. Springer, Heidelberg (2006)

[D05] Diestel, R.: Graph Theory, 3rd edn. Springer, Heidelberg (2005)

[E04] Elkin, M.: An Overview of Distributed Approximation. ACM SIGACT News Distributed Computing Column 35(4,132), 40–57 (2004)

[KMW04] Kuhn, F., Moscibroda, T., Wattenhofer, R.: What Cannot Be Computed Locally! In: Proceedings of 23rd ACM Symposium on the Principles of Distributed Computing (PODC), pp. 300–309 (2004)

[KMNW05a] Kuhn, F., Moscibroda, T., Nieberg, T., Wattenhofer, R.: Fast Deterministic Distributed Maximal Independent Set Computation on Growth-Bounded Graphs. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 273–287. Springer, Heidelberg (2005)

[KMNW05b] Kuhn, F., Moscibroda, T., Nieberg, T., Wattenhofer, R.: Local Approximation Schemes for Ad Hoc and Sensor Networks. In: 3rd ACM Joint Workshop on Foundations of Mobile Computing (DIALM-POMC), Cologne, Germany, pp. 97–103 (2005)

[LOW08] Lenzen, C., Oswald, Y.A., Wattenhofer, R.: What Can Be Approximated Locally? In: 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2008), Munich, Germany, pp. 46–54 (2008)

[LW08] Lenzen, C., Wattenhofer, R.: Leveraging Linial's Locality Limit. In: 22nd International Symposium on Distributed Computing (DISC 2008), Arcachon, France (to appear, 2008)

[L92] Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)

[N91] Naor, M.: A lower bound on probabilistic algorithms for distributive ring coloring. SIAM J. on Discrete Mathematics 4(3), 409–412 (1991)

[N95] Nesetril, J.: Ramsey Theory. In: Graham, R.L., Groschel, M., Lovász, L. (eds.) Handbook of Combinatorics, vol. II, pp. 1331–1403. Elsevier, Amsterdam (1995)

[Pa99] Panconesi, A.: Distributed Algorithms Notes (manuscript)

[Pe00] Peleg, D.: Distributed Algorithms. In: A Locality-Sensitive Approach, SIAM Press, Philadelphia (2000)

[SW08] Schneider, J., Wattenhofer, R.: A log* Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In: PODC 2008 (to appear, 2008)

[We01] West, D.: Introduction to graph theory, 2nd edn. Penitence-Hall Inc. (2001)

A Appendix

A.1 Constant Approximation of the Minimum-Dominating Set

Our algorithm follows from the next few observation. Let K be a positive integer which is large enough (the precise bound for K follows from computations and in the current argument is significantly larger than 71, the constant from [LOW08]) and let $G = (V, E)$ be a planar graph. We define $B := \{v | deg(v) \geq K\}$, $S := V \setminus B$, and $B' := \{v \in B || N(v) \cap S| \geq 20^2\}$.

Definition 2. *Let $u, v \in B'$. Vertex u is called v-redundant if the following conditions are satisfied.*

(a) $|N(u) \cap N(v) \cap S| \geq |N(u) \cap S|/10$.
(b) Either $|N(v) \cap S| > |N(u) \cap S|$ *or* $|N(v) \cap S| = |N(u) \cap S|$ *and* $ID(v) > ID(u)$.

Note that the constant 10 in (a) is almost arbitrary and the reason for part (b) is to break symmetry. In particular, from part (b), a vertex u is never u-redundant. We first observe a few simple facts about redundant vertices.

Fact 9. *Let $u, v \in B'$ be such that $|N(u) \cap N(v) \cap S| \geq |N(u) \cap S|/10$. Then either u is v-redundant or v is u-redundant.*

Fact 10. *If u is v-redundant then v is not u-redundant.*

Finally we note that u cannot be v-redundant with too many v's.

Fact 11. *Let $G = (V, E)$ be a planar graph. Then for $u \in B'$ there are at most 19 vertices $v \in B'$ such that u is v-redundant.*

Proof. Since G is planar, we have $|N(u) \cap N(v) \cap N(w)| \leq 2$ for any three distinct vertices u, v, w. Assume u is v-redundant for every $v \in \{v_1, \ldots, v_k\}$. Then

$$|N(u) \cap S| \geq \sum_{i=1}^{k} |N(u) \cap N(v_i) \cap S| - 2\binom{k}{2} \geq \frac{k|N(u) \cap S|}{10} - k^2 > |N(u) \cap S|$$

when $k = 20$. □

Lemma 7

$$\gamma(G') \leq K\gamma(G).$$

Proof. First notice that to obtain G' only edges between S and B' can be deleted from G. Let D be a dominating set in G. We will add some vertices to D to obtain a dominating set D' in G'. If $u \in D \cap B'$ and $deg_G(u) \neq deg_{G'}(u)$ then add to D all vertices v such that u is v-redundant. By Fact 11 there are 19 such v's. If $u \in D \cap S$ and $deg_G(u) \neq deg_{G'}(u)$ then $deg_G(u) \leq K - 1$ and add all neighbors of u in G to D. Then $|D'| \leq K|D|$ as $K \geq 20$ and D' is a dominating set in G'. Indeed, any vertex v which is dominated by a vertex $u \in D$ with $uv \in E \setminus E'$ is in D'. □

Lemma 8. *Let $H = (V, E)$ be a planar graph with no redundant pairs, let K be a positive integer which is large enough, and let $B = \{v \in H | deg_H(v) \geq K\}$. Then*

$$\gamma(H) \geq \min \frac{|B|}{2}, \frac{|H|}{K^2}.$$

Proof. Let D be a dominating set in H. Consider $D_1 = D \cap B$ and $D_2 = D \setminus D_1$. Every vertex from D_2 has the degree of at most $K - 1$ and so there are at most $K|D_2|$ vertices dominated by D_2. Let W be the set of vertices dominated by D_2. If $|W| \geq |H|/K$ then $|D_2| \geq |H|/K^2$ and we are done. Assume therefore that $|W| < |H|/K$ and let $A = V \setminus (W \cup B)$. Since H is planar, $|B| \leq 6|H|/K$ and so $|A| \geq (1 - 7/K)|H|$ with every vertex from A dominated by a vertex from $D_1 \subseteq B$. Consider the bipartite subgraph H' of H with bipartition (B, A) and all edges of H with one endpoint in B, another in A and let $d_1, \ldots, d_k$ be vertices from D_1 that dominate A. Every vertex a from A chooses one $i \in \{1, \ldots, k\}$ such that d_i dominates a and joins the group of d_i. In this way we obtain stars $S_1, \ldots, S_k$ with centers in $d_1, \ldots, d_k$ and every $a \in A$ belonging to exactly one star. Let $\bar{B} = \{b \in B | deg_{H'}(b) \geq deg_H(b)/2\}$. We note that

$$|\bar{B}| \geq \left(1 - \frac{12}{K}\right)|B| \tag{4}$$

as otherwise $2||H[B]|| > \frac{12}{K}|B|K/2 \geq 6|B|$ but $H[B]$ is planar. Note that if $b \in \bar{B}$ then $deg'_H(b) \geq K/2 \geq 20^2$ and so $b \in B'$. Since there are no redundant pairs in H, for every $b, d_i \in B'$ we have $|N_{H'}(b) \cap N_{H'}(d_i)| < \frac{|N_{H'}(d_i) \cap S|}{10}$. Now consider the planar graph H'' obtained from H by contracting each of the stars to a single vertex. Every vertex from $\bar{B}$ has degree of at least 10 in H'' and H'' is planar. Thus,

$$5|\bar{B}| \leq ||H''|| < 3(|B| + k)$$

which gives

$$|D_1| \geq k \geq \frac{(2 - 12/K)|B|}{3} \geq |B|/2$$

if $K \geq 24$. □

Lemma 9. *There is a distributed algorithm that finds a $\Omega(1)$-approximation of a minimum dominating set in a planar graph in $O(1)$ rounds.*

Proof. After fixing K in the argument above, graph G' is obtained by deleting edges as described in Lemma 7. Then in H all vertices from $B = \{v \in H | deg_{G'}(v) \geq K\}$ are added to the dominating set and finally all vertices in G' which are not dominated in G' by B are added to the dominating set. The dominating set has size $\Omega(\gamma(G))$. Indeed, from Lemma 7, $\gamma(G) = \Omega(\gamma(G'))$ and from Lemma 8, $\gamma(G') = \Omega(|B|) + \Omega(|C|)$ where C is the set of vertices not dominated by B in G' (Clearly these vertices can only be dominated by vertices of degree at most $K - 1$ in G'). □

Closing the Complexity Gap between FCFS Mutual Exclusion and Mutual Exclusion*

Robert Danek and Wojciech Golab

Department of Computer Science
University of Toronto
{rdanek,wgolab}@cs.toronto.edu

Abstract. First-Come-First-Served (FCFS) mutual exclusion (ME) is the problem of ensuring that processes attempting to concurrently access a shared resource do so one by one, in a fair order. In this paper, we close the complexity gap between FCFS ME and ME in the asynchronous shared memory model where processes communicate using atomic reads and writes only, and do not fail. Our main result is the first known FCFS ME algorithm that makes $O(\log N)$ remote memory references (RMRs) per passage and uses only atomic reads and writes. Our algorithm is also adaptive to point contention. More precisely, the number of RMRs a process makes per passage in our algorithm is $\Theta(\min(k, \log N))$, where k is the point contention. Our algorithm matches known RMR complexity lower bounds for the class of ME algorithms that use reads and writes only, and beats the RMR complexity of prior algorithms in this class that have the FCFS property.

1 Introduction

Coordinating access to shared resources is a key problem in programming multiprocessors. Mutual exclusion [1], also known as locking, is the approach most popular in practice for allowing multiple processes to access a shared resource safely. We consider this problem under the customary assumptions that processes are asynchronous (i.e., execute at arbitrary speeds) but do not fail. A mutual exclusion algorithm for a shared memory multiprocessor consists of a *trying protocol* (TP) and an *exit protocol* (EP) that surround the critical section (CS). The latter contains code that actually accesses the shared resource. A single execution of the TP, CS, and EP is called a *passage*. When a process is not inside the TP, EP, or CS, we say that it is in the non-critical section (NCS).

The trying and exit protocols ensure that at most one process can be in the critical section at a time, while also guaranteeing that processes wanting to access the resource can eventually do so. We can state the correctness properties of a mutual exclusion algorithm more precisely as follows:

Mutual Exclusion (ME): If a process p is in the CS, then no process $q \neq p$ is in the CS concurrently with p.

* Research supported in part by the Natural Sciences and Engineering Research Council of Canada.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 93–108, 2008.

Lockout Freedom (LF): If a process p enters the trying protocol, then p eventually enters the CS.

Bounded Exit (BE): If a process enters the exit protocol, then the process returns to the NCS in a bounded number of its own steps.

Note that to satisfy lockout freedom, we must make the (standard) assumption that every process is *live*, meaning that as long as it is outside the NCS, it continues to take steps until it returns to the NCS.

The above properties do not preclude situations where a process waits inside the trying protocol for a long time while other processes are repeatedly granted entry to the critical section. This may be undesirable, and a mutual exclusion algorithm that grants processes entry into the critical section in an order that is more fair may be preferred. One form of fairness is captured by the First-Come-First-Served (FCFS) property [2], which informally requires that processes are granted entry into the critical section in the order in which they execute the beginning of the trying protocol. To define this more precisely, we split the trying protocol into two parts: the first part, the doorway (DWY), which a process completes in a bounded number of its own steps; and a second part, the waiting room (WRM). We can now define FCFS as follows:

First-Come-First-Served (FCFS): If a process p finishes the doorway before a process $q \neq p$ starts the doorway, then q does not enter the CS before p does in the corresponding passages.

A natural way to measure the time complexity of a mutual exclusion algorithm is to count the number of memory accesses performed during a passage. This is problematic in the asynchronous model as a process may execute an unbounded number of memory accesses while busy-waiting for another process to clear the critical section. Instead, we measure time complexity by counting only the *remote memory references* (RMRs) performed during a passage, where an RMR is a memory access that traverses the processor-to-memory interconnect. We refer to this measure as an algorithm's RMR complexity.

To classify memory accesses as local or remote, we consider two shared memory architectures: the Distributed Shared Memory (DSM) model, and the Cache-Coherent (CC) model [3]. In the DSM model, each processor is associated with a memory module that it can access locally, and that others may access only remotely. In the CC model, on the other hand, any memory location can be made locally accessible by storing its contents in a local cache, which is kept up to date (by either updating or invalidating stale entries) by a *coherence protocol*. Different varieties of the CC model exist, all satisfying the following property under ideal conditions: once a variable is loaded into a cache, it remains cached at least until it is overwritten by another process.

Algorithms that perform all busy-waiting using local memory references (e.g., repeatedly testing the value of a cached variable) are known as *local-spin*; they have bounded RMR complexity and offer practical performance benefits [4]. The RMR complexity of an ME algorithm may depend on the number of processes contending for access to the CS. *Point contention* describes this quantity precisely; for our purposes it is defined as the maximum number of processes simultaneously outside of the NCS during an execution fragment. An ME algorithm whose RMR complexity grows gradually with point contention is known as *adaptive* (to point contention).

Summary of results. Our main technical contribution is an FCFS ME algorithm based on reads and writes only, which has RMR complexity $O(\min(k, \log N))$ when point contention is k and there are N processes. The complexity of our algorithm is optimal, at least when $k \in O(\log \log N)$ [5] or $k \in \Theta(N)$ [6]. Prior algorithms either require stronger synchronization primitives, lack the FCFS property, or have suboptimal RMR complexity.

Our algorithm uses as building blocks two novel wait-free components: a set-like data structure called *SpecialSet*, and a ticket dispensing mechanism. The *SpecialSet* records a set of process IDs, and has two operations: InsertSelf(), which a process can use to add its ID to the set, and RemoveSelf(), which a process can use to remove itself from the set and also to learn the ID of exactly one other process in the set (if any). Our *SpecialSet* and the ticket dispenser are accessed according to certain restrictions on parallelism, which simplifies their implementation.

As a complexity upper bound, our algorithm has several implications regarding mutual exclusion:

(1) The worst-case RMR complexity of FCFS ME using only reads and writes is no greater than for ordinary (i.e., deadlock-free) ME; both problems are solvable using $O(\log N)$ RMRs per passage, matching the recent lower bound of Attiya, Hendler and Woelfel [6].

(2) FCFS ME can be solved using only reads and writes with RMR complexity adaptive to point contention, matching the linear lower bound of Kim and Anderson for $k \in O(\log \log N)$ [5] in addition to the logarithmic worst-case lower bound [6].

(3) As a consequence of (1) and (2), and the fact that the lower bounds on ME RMR complexity [6,5] hold even if comparison primitives (such as Compare-And-Swap) are available, FCFS ME and adaptive FCFS ME are no more costly to solve (in terms of RMRs) using reads and writes only than using reads, writes, and comparison primitives. This strengthens somewhat a prior result of Golab, Hadzilacos, Hendler and Woelfel [7], which implies the analogous conclusion for ME algorithms that do not have FCFS or bounded exit.

2 Related Work

The mutual exclusion problem was first solved by Dijkstra [1], although his solution did not satisfy lockout freedom. Rather, it satisfied a weaker progress property, called deadlock freedom:

Deadlock Freedom: If some process p is stuck forever in the trying protocol, then some process $q \neq p$ executes through the critical section infinitely often.

FCFS mutual exclusion was first formulated and solved by Lamport in [2], where he presented his famous Bakery algorithm. Lamport [8] was also the first to study *fast* mutual exclusion. Fast mutual exclusion ensures that a process takes a constant number of steps entering the CS when there is no contention, but provides no performance guarantees otherwise. In *adaptive* mutual exclusion the performance of an algorithm instead degrades gradually as the contention for the CS increases. Adaptive mutual exclusion algorithms were presented by Styer [9], Choy and Singh [10], and Attiya and Bortnikov [11]. These algorithms are adaptive to metrics different from RMR complexity, and moreover, the RMR complexity of these algorithms is unbounded.

Yang and Anderson (YA) [12] presented the first mutual exclusion algorithm that uses only reads and writes and has RMR complexity $O(\log N)$. Kim and Anderson [13] (KA) later presented an adaptive mutual exclusion algorithm, also using only reads and writes, that used as building blocks parts of Lamport's fast mutual exclusion algorithm and the YA algorithm. Its RMR complexity is $O(\min(k, \log N))$, where k denotes point contention. This improves upon the adaptive algorithm of Afek, Stupp and Touitou [14].

Several lower bounds exist for the RMR complexity of mutual exclusion and adaptive mutual exclusion. Kim and Anderson [5] showed that the RMR complexity of adaptive ME algorithms based on reads and writes only must grow at least linearly with point contention up to $\Omega(\log \log N)$, which is matched by algorithm KA. Attiya, Hendler and Woelfel [6] later showed that the worst-case RMR complexity for the same class of algorithms is $\Theta(\log N)$, which is matched by algorithm YA. (This builds on prior results by Cypher [15], Anderson and Kim [16], and Fan and Lynch [17].) A related result by Golab, Hadzilacos, Hendler and Woelfel [7] implies that the $\Theta(\log N)$ lower bound is tight also for algorithms that use comparison primitives, such as Compare-And-Swap (CAS), and do not require FCFS or bounded exit.

Jayanti [18] presented the first FCFS adaptive mutual exclusion algorithm. It has RMR complexity $O(\min(k, \log N))$, and makes use of LoadLinked and StoreConditional – a pair of synchronization primitives stronger than reads and writes.

Taubenfeld [19] also presented an FCFS adaptive mutual exclusion algorithm. This algorithm is a modification of Lamport's Bakery algorithm, and uses only reads and writes. Its RMR complexity, however, is $O(k^2)$, which is suboptimal in light of our results.

3 FCFS Algorithm and High-Level Description

Our algorithm (shown below in Figure 1) has the following high-level structure. In the doorway, a process receives a ticket from a wait-free ticket dispenser (line 4) that incurs $O(\min(k, \log N))$ RMRs per invocation. The dispenser is similar to a modular atomic counter, which returns tickets with increasing values from a bounded interval. As the dispenser is not actually atomic, processes that invoke the dispenser concurrently may receive the same ticket. Also, even though

```
shared variables:
    $Set$: $SpecialSet$, $Q$: $PriorityQueue$, $Head$: array$[1..N]$ of Boolean
    (In the DSM model, $Head[p]$ is local to process $p$)
private variables:
    $ticket$: $\{0, .., 7N-1\}$, $tmp\_id$: integer

1  loop
2      NCS
3      $Set$.InsertSelf()                                     // Doorway begins.
4      $ticket$ := ObtainTicket()                               // Doorway ends.
5      LOCK()
6          $Head[p]$ := false
7          $tmp\_id$ := $Set$.RemoveSelf()
8          if $tmp\_id \neq \bot$ then
               // Enqueue process $tmp\_id$ with ``dummy'' ticket.
9              $Q$.Insert(($tmp\_id, -1$))
10         $Q$.Remove(($p, -1$))         // Remove $(p, -1)$ from queue if present.
11         $Q$.Insert(($p, ticket$))         // Reinsert $p$ with ``proper'' ticket.
12         $tmp\_id$ := $Q$.FindMin()     // Get the head process in the queue.
13         $Head[tmp\_id]$ := true            // Notify head process to advance.
14     UNLOCK()
15     await $Head[p]$ = true         // Wait to reach the head of the queue.
16     LOCK()
17         CS                                        // The critical section.
18         $Q$.Remove(($p, ticket$))       // Remove $p$ from the priority queue.
19         DoneWithTicket()
20         $tmp\_id$ := $Q$.FindMin()
21         if $tmp\_id \neq \bot$ then
22             $Head[tmp\_id]$ := true        // Notify next process to advance.
23     UNLOCK()
24 end loop
```

Fig. 1. FCFS Mutual Exclusion Algorithm for process $p \in \{1, ..., N\}$

the dispenser returns tickets from a bounded interval, the interval is large enough to ensure that tickets are not reused too soon. After a process p obtains a ticket, it enters the waiting room (lines 5–16) where it adds itself to a priority queue (Q) ordered by ticket (line 11). To ensure that FCFS is not violated, p waits to reach the front of the queue before entering the CS (line 15). Once p is done with the CS, p removes itself from the queue (line 18), and notifies its successor (lines 20–21).

We use an *auxiliary lock* (lines 5, 14, 16, 23) to serialize operations on Q. This allows us to implement Q with a min-heap, which has time complexity $O(\log k)$. The ME algorithm that we use for the auxiliary lock is Kim and Anderson's algorithm [13], which has RMR complexity $O(\min(k, \log N))$.

The priority queue has standard operations Insert, Remove, and FindMin, and its entries are pairs of the form (process ID, ticket). Insert is idempotent, and Remove has no effect if attempting to remove an item that is not in the queue. FindMin() returns the process ID whose corresponding ticket is minimal (i.e., the head element), or $\perp$ if the queue is empty. What it means for a ticket to be minimal in a collection of tickets, and more generally how tickets are ordered, is explained in detail in Section 5.

Processes use the Boolean array *Head* to notify another process when it becomes the head of the queue. A process can become the head of the queue after another process removes itself from the queue in the exit protocol (line 18), or after the queue is modified in the waiting room (lines 10–11).

Our algorithm contains additional features, not described above, to handle the following race condition: process p finishes the doorway before q starts the doorway, but then q adds itself to Q before p. By the FCFS property, p should enter the CS before q, but until p is added to Q, q cannot tell (from the state of Q alone) whether it should enter the CS before or after p. To prevent q from entering the CS prematurely, we use special "dummy tickets" and a shared object, *Set*, of a set-like type called *SpecialSet*. At the beginning of the doorway, at line 3, a process q adds itself to *Set*. In the waiting room, at line 7, q removes itself from *Set*, and also learns the ID of one other process $p \neq q$ in *Set*, if it exists ($\perp$ otherwise). If p exists, then p must be in the trying protocol at or before the lock at line 5. In that case, q adds p to Q at line 9 with a "dummy" ticket -1, which is smaller than any "proper" ticket returned by the ticket dispenser at line 4. The insertion of p into Q in this way guarantees that p will be ahead of q in Q until p executes the locked code at lines 6–13, where it replaces its dummy ticket in Q with its proper ticket (lines 10–11). This ensures that q cannot advance into the CS prematurely.

The set operations (line 3 and line 7), and the ticket dispenser operations (line 4 and line 19), are explained in more detail in Sections 4 and 5, respectively.

4 *SpecialSet* – A Set-Like Data Structure

In this section, we describe the data type of the shared object *Set* used in our mutual exclusion algorithm. We refer to this type as *SpecialSet*, because its state is represented by a set but it supports only a few set operations, and only in restricted ways.

The sequential specification of *SpecialSet* is as follows. The state of *SpecialSet* is a set of process IDs. Two operations are used to access *SpecialSet*:

- InsertSelf() adds the ID of the calling process to the set, and returns nothing.
- RemoveSelf() removes the caller's ID from the set, and returns the ID of one other process in the set, if it exists, otherwise returns $\perp$.

Processes must access *SpecialSet* according to the following etiquette:

Condition 1

(a) *The calls to* InsertSelf() *and* RemoveSelf() *made by any process occur in an alternating sequence, beginning with* InsertSelf()*, and ending with* RemoveSelf()*; and*
(b) *Operation* RemoveSelf() *is executed in mutual exclusion.*

For our purposes, it suffices to make the implementation of *SpecialSet* for N processes linearizable and wait-free, with step complexity $O(\min(k, \log N))$, where k denotes point contention. (Note that by Condition 1, if a process has completed InsertSelf() but not yet started its subsequent call to RemoveSelf(), then it is enabled to execute another step, and so we count it in evaluating point contention.)

Below we describe a simple but non-adaptive implementation of *SpecialSet* for N processes, with step complexity $O(\log N)$. Then, we give an informal overview of how the implementation can be made adaptive using existing ideas.

4.1 Non-adaptive Implementation

The data structure underlying the implementation of *SpecialSet* for N processes is a full binary tree of height $\lceil \log N \rceil$. Each node in the tree stores a process ID or $\bot$, initially $\bot$. We denote the value stored at node n by $NodeVal[n]$. In addition to the tree, we use an array $MyNode[1..N]$ of pointers to tree nodes or $\bot$, initially all $\bot$. For any process ID p, we will refer to $MyNode[p]$ as *p's node*. Informally, if $MyNode[p] = n_p$ for some tree node n_p then p is in the set and p uses tree nodes on the path between n_p and the root node to record information about itself. In the non-adaptive implementation, n_p will be a unique and statically determined leaf node, referred to as *p's leaf node*.

The InsertSelf() access procedure for process p first determines p's leaf node at line 25, and then passes control to the helper function InsertHelper(p), which is also used by RemoveSelf(). (Here the ID of p's leaf node is statically determined, but in the adaptive version of the algorithm it is not.) In function

```
shared variables:
  $NodeVal$: mapping from node ID to process ID or $\bot$, initially all $\bot$
  $MyNode$: array$[1..N]$ of pointer to tree node or $\bot$, initially all $\bot$
```

Fig. 2. Variables used in *SpecialSet* implementation

```
25 $MyNode[p]$ := ID of $p$'s leaf node
26 InsertHelper($p$)
```

Fig. 3. InsertSelf() for process $p \in \{1, \ldots, N\}$

```
$l := MyNode[z]$
foreach node $n$ on path from $l$ to root do
    $NodeVal[n] := z$
end
```

Fig. 4. InsertHelper(z)

```
Output: process ID or $\perp$
$l := MyNode[p]$
foreach node $n$ on the path from $l$ to root do
    if $n$ has a sibling node then
        $n' :=$ sibling of $n$
        $q := NodeVal[n']$
        if $q \neq \perp$ and $MyNode[q] \neq \perp$ then
            InsertHelper($q$)
            $MyNode[p] := \perp$
            return $q$
        end
    end
end
$MyNode[p] := \perp$
return $\perp$
```

Fig. 5. RemoveSelf() for process $p \in \{1, \ldots, N\}$

InsertHelper(p), the calling process traverses the binary tree from p's node to the root and writes p's ID at each node visited.

The RemoveSelf() access procedure works as follows. The caller, say process p, first determines its tree node, say l, by reading $MyNode[p]$. Next, p traverses the tree from l to the root. For each node visited, p reads the ID stored at the sibling node ($O(\log N)$ nodes in total). For each process ID encountered, say q, p checks whether q's node is not $\perp$. If the latter condition holds, then p stops its traversal immediately after inspecting q's node, and executes InsertHelper(q). (Here $q \neq p$ holds because p's leaf node is statically determined.) By calling InsertHelper(q) at this point, p ensures that if there are any nodes between the current node and the root that contain the ID p, they will be overwritten with an ID that is still in the set. If this were not done, then future calls to RemoveSelf() might behave as though there are no remaining items in the set, and erroneously return $\perp$. Finally, p's execution of RemoveSelf() overwrites $MyNode[p]$ with $\perp$ and returns q. Otherwise, if no such q is found, then upon reaching the root node, p's execution of RemoveSelf() overwrites $MyNode[p]$ with $\perp$ and returns $\perp$.

4.2 Adaptive Implementation

The non-adaptive implementation of *SpecialSet* described above can be altered so that its step complexity becomes adaptive to k – the point contention (as defined earlier for executions involving a *SpecialSet* object). The main idea is to choose p's node so that it has distance $O(\min(k, \log N))$ from the root, which is difficult since p's node must be unique among all processes that are in the set. One approach is to build the tree dynamically using splitter-like objects, which are based on Lamport's "fast path" mechanism. Anderson and Kim used such objects to construct an adaptive ME algorithm based on reads and writes only [13]. The RMR complexity of this algorithm is $O(\min(k, \log N))$, and key portions of it have step complexity $O(\min(k, \log N))$.

Rather than using pieces of the Anderson and Kim algorithm to create our adaptive implementation of *SpecialSet*, we execute the entire ME algorithm in our implementation and extract certain useful information from that execution. This allows us to re-use complex synchronization machinery directly rather than modifying it and re-proving its correctness properties. The wait-free portion of the trying protocol of the Anderson-Kim algorithm is executed inside InsertSelf(), and the remainder in RemoveSelf(). Since RemoveSelf() is executed in mutual exclusion by Condition 1, this means that the executing process will never busy-wait inside the Anderson-Kim algorithm. (In fact, we can replace the locks used therein with "no-ops".)

5 Ticket Dispenser

Our mutual exclusion algorithm internally uses numbered tickets, much like Lamport's bakery algorithm [2]. Tickets are obtained by calling function ObtainTicket(), which is used in conjunction with function DoneWithTicket() according to the following etiquette:

Condition 2. *The calls to* ObtainTicket() *and* DoneWithTicket() *made by any process occur in an alternating sequence, beginning with* ObtainTicket(), *and ending with* DoneWithTicket().

Informally, we can think of ObtainTicket() as returning a (not necessarily unique) element of some pool of free tickets, and DoneWithTicket() as cleaning up some internal state once a process is done using a particular ticket. (Using a pair of functions in this way makes the ticket dispenser somewhat more complex to specify, but easier to implement.)

We say that a process is *participating* in the ticket dispenser if it has begun its call to ObtainTicket() but not yet completed its subsequent call to DoneWithTicket(). If a participating process has completed its call to ObtainTicket(), then we say that it *holds* the ticket returned by that call. A ticket is *active* if it is held by some process, otherwise it is *inactive*. Tickets satisfy the following properties:

Specification 1

(a) The domain of tickets is the set of integers modulo mN for some integer $m \geq 3$.
(b) At any time, the set of tickets that are active is confined to some interval of fewer than $mN/2$ consecutive integers modulo mN.

We will use (a) and (b) as follows to define a total order on the set of tickets that are simultaneously active. Given two active tickets i and j, where $i < j$, we will say that i is *less than* j (denoted $i \lhd j$) if $j - i < mN/2$, otherwise we will say that i is *greater than* j (denoted $i \rhd j$). (We will also use $\unlhd$ and $\unrhd$ to denote weak inequalities.) Finally, if $i = j$ then we will say i is *equal to* j. For technical reasons, we also define a special *dummy* ticket, denoted -1, which can be compared against and is less than any active ticket. We say that two tickets are *comparable* if they are simultaneously active (or one or both is -1), and *incomparable* otherwise. Finally, note that our mutual exclusion algorithm compares tickets only implicitly, inside the priority queue.

Having defined an ordering among simultaneously active tickets, we are now ready to specify the correctness properties of ObtainTicket().

Specification 2. *Consider any execution at the end of which distinct processes p and q hold tickets t_p and t_q, respectively. Let C_p and C_q denote the calls to* ObtainTicket() *that generated these tickets, respectively.*

- *If C_p occurred before C_q, then $t_p \lhd t_q$.*
- *If C_p occurred after C_q, then $t_p \rhd t_q$.*
- *If C_p and C_q were concurrent, then the ordering among t_p and t_q is arbitrary*

To simplify the implementation of the operations ObtainTicket() and DoneWithTicket(), we restrict concurrent executions of these functions as follows:

Condition 3

(a) Function DoneWithTicket() *is executed in mutual exclusion.*
(b) Moreover, if processes p and q are participating simultaneously and hold tickets t_p and t_q, respectively, where $t_p \lhd t_q$, then p subsequently completes a call to DoneWithTicket() *before q does. (In other words, p stops participating before q does.)*

Condition 4. *For any execution fragment during which some process p is (contiguously) participating in the ticket dispenser, every other process participates at most three times during that execution fragment.*

Condition 5. *For any execution fragment during which some process p is (contiguously) executing inside* ObtainTicket()*, if another process q executes* ObtainTicket() *(partially or completely) during that fragment, then q does not subsequently call* DoneWithTicket() *before p finishes its call to* ObtainTicket() *under consideration.*

For our purposes, an implementation of the ticket dispenser must satisfy the following: given that Conditions 2–5 hold, Specifications 1–2 must hold, and the InsertSelf() and RemoveSelf() operations must have step complexity $O(\min(k, \log N))$, where k denotes point contention. (Note that by Condition 2, if a process has completed ObtainTicket() but not yet started its subsequent call to DoneWithTicket(), then it is enabled to execute another step, and so we count it in evaluating point contention.)

5.1 Adaptive Implementation

Next, we describe an implementation of the ticket dispenser that is adaptive in the number of participating processes.

```
shared variables:
  $Tickets$: array$[0..7N\text{-}1]$ of $\{$INUSE, FREE$\}$
  initially $Tickets[0..(3N\text{-}1)] =$ FREE and $Tickets[3N..(7N\text{-}1)] =$ INUSE

  $lastTicket$: $0..7N\text{-}1$ initially $7N\text{-}1$

private variables:
  $ticket$: $0..7N\text{-}1$ uninitialized
```

Fig. 6. Variables used in ticket dispenser implementation

```
$first := lastTicket$
$i := 1$
// Find upper bound on the smallest FREE ticket.
while $i < 3N \wedge Tickets[(first + i) \bmod 7N] =$ INUSE do
    $i := \min\{3N, i \times 2\}$
// Now do binary search to find the ticket.
$last := first + i$
while $first < last$ do
    $midpoint := \lfloor (first + last)/2 \rfloor$
    if $Tickets[midpoint \bmod 7N] =$ INUSE then
        $first := midpoint + 1$
    else
        $last := midpoint$
// At this point $first = last$ holds.
$ticket := first \bmod 7N$
$Tickets[ticket] :=$ INUSE
return $ticket$
```

Fig. 7. Implementation of ObtainTicket()

```
   // Reset a ticket that was previously active.
59 $Tickets[(ticket + 3N) \bmod 7N] :=$ FREE
60 $lastTicket := ticket$
```

Fig. 8. Implementation of DoneWithTicket()

The algorithm uses a shared circular array *Tickets* of length $7N$, whose entries represent the state of the correspondingly numbered tickets. Each entry is either `INUSE` or `FREE`, indicating, as we explain later, whether the corresponding ticket is active. The shared variable *lastTicket* stores the ticket that was held by the last process that stopped participating in the ticket dispenser, i.e., the last process that called DoneWithTicket(), and is used by ObtainTicket() to efficiently find a `FREE` ticket. ObtainTicket() uses a two-stage search mechanism to determine the next `FREE` ticket. First, the algorithm attempts to find an interval of consecutive tickets, starting at *lastTicket*, that contains a `FREE` ticket. This is done at lines 45–49 by searching rightwards from *lastTicket* in steps of exponentially increasing size, up to a distance of $3N$. Starting at *lastTicket* ensures that the search is adaptive to point contention, k, and taking steps of exponentially increasing size bounds the total number of steps taken to be $O(\log k)$. We only need to search up to a distance of $3N$ from *lastTicket*, since, by Condition 4, every other process participates at most three times while the search is being done. This means there will be at most $3(N-1)$ `INUSE` tickets after *lastTicket*.

Once a `FREE` ticket is found, the interval from *lastTicket* to the `FREE` ticket is guaranteed to contain at least one `FREE` ticket. However, there may be another `FREE` ticket earlier in the interval. The algorithm performs a binary search of the interval at lines 50–55 to pinpoint such a ticket if it exists. The ticket computed is stored in the private variable *ticket* at line 56, and marked `INUSE` at line 57.

Function DoneWithTicket() simply resets a previously-active ticket at line 59 (so that it can be reused later), and then updates *lastTicket* at line 60.

6 Correctness of FCFS ME Algorithm

In this section we provide a very high-level overview of why the FCFS ME Algorithm defined in Figure 1 is correct, and why it has RMR complexity $O(\min(k, \log N))$.

The correctness of the FCFS ME algorithm relies on the correctness of the *SpecialSet* and ticket dispenser implementations outlined in Sections 4 and 5. These implementations are correct only if they are used according to the etiquette outlined in Conditions 1–5. Our proof that these conditions hold in Figure 1 relies on the ME algorithm satisfying FCFS. Our proof for FCFS, however, relies on the correctness of the *SpecialSet* and ticket dispenser, which leads to a cycle of dependencies. We deal with this cycle through careful induction on the length of the *execution history*. (An execution history is an alternating sequence of *states* and process steps, where a state consists of the values assigned to all private and shared variables in the system, and a step is a shared memory operation by a process.)

The proof proceeds in two parts. The first part shows that FCFS holds in any execution history in which the *SpecialSet* and ticket dispenser are correct. The second part uses induction to show that Conditions 1–5 hold in any execution history, and hence that the *SpecialSet* and ticket dispenser are correct. We proceed in reverse, sketching the second part of the proof first, and then sketching the remaining details.

Lemma 1. *Conditions 1–5 hold in any execution history of the algorithm.*

Proof sketch. By inspection of the ME algorithm in Figure 1, Conditions 1–3(a) hold. To show that Conditions 3(b)–5 hold, we use induction on the length of the execution history H. In the initial state of H, no process has taken a step, and so the conditions hold trivially. We assume that the conditions hold up to some state s in the execution history, and show that the conditions also hold in the next state s' after s. Suppose, by way of contradiction, that the conditions do not hold in state s'. Since all conditions are satisfied up to s, it can be shown that exactly one condition is not satisfied in s'. Due to space limitations, we only argue for a contradiction when Condition 4 does not hold. In this case, there must be some process p contiguously participating in the ticket dispenser while another process q participates four times. Process q must have started participating for the fourth time when it took a step between s and s'. It turns out (by Condition 5) that p must have finished executing ObtainTicket() before q went through the CS when it participated in the ticket dispenser the second time. Thus, during q's third time participating, p will have finished the doorway before q starts it. FCFS holds prior to s', and so q cannot execute through the CS and participate a fourth time until p has executed through the CS. But this means that when q participates for the fourth time, p will no longer be participating contiguously, contradicting the assumption that it is.

Lemma 2. *The algorithm satisfies mutual exclusion.*

Proof. The lemma follows from the correct use of the auxiliary lock, which surrounds (among other things) the CS.

Lemma 3. *The algorithm satisfies bounded exit.*

Proof. The Kim and Anderson [13] algorithm, which we use for the auxiliary lock, satisfies bounded exit. Consequently, it follows from the structure of our algorithm that it too satisfies bounded exit.

Lemma 4. *The algorithm satisfies FCFS.*

Proof sketch. Assume that some process p finishes the doorway before some process q starts the doorway, and suppose, by way of contradiction, that q enters the CS before p in the corresponding passages. Immediately before q does so, p and q hold their tickets simultaneously. Since p finished the doorway before

q started it, p's call to ObtainTicket() finished executing before q's call to ObtainTicket() started. This and the ticket dispenser specification imply that p's ticket is smaller than q's. If p adds itself to Q at line 11 before q, then q has no hope of entering the CS before p since p will be in front of q in Q. So it must be the case that q adds itself to Q before p by executing the locked segment of code at lines 6–13 before p. In this case, however, q's call to RemoveSelf() at line 7 returns $tmp\_id \neq \perp$ (possibly $tmp\_id = p$), since Set contains p. This means that at line 9, q adds some process to Q with a dummy ticket. Process q cannot be signalled to enter the CS while there is a dummy ticket in Q, and it turns out the latter condition holds at least until p adds itself with its proper ticket to Q. When p does add itself to Q, it will be in front of q, since p has a smaller ticket than q. This implies that p will enter the CS before q, which contradicts the assumption that q enters before p.

Lemma 5. *The algorithm satisfies deadlock freedom.*

Proof sketch. Suppose, by way of contradiction, that deadlock freedom does not hold. That is, some process p loops forever in the trying protocol, and after some point in the execution, no process enters the CS. It turns out that the only place where p may be looping is at line 15, while waiting to be signalled to enter the CS. Furthermore, since there is a point after which no process enters the CS, there must be a last call to Q.FindMin() (line 12 or 20). The contradiction that we derive is to show that after the last call to Q.FindMin(), there must be another call to Q.FindMin().

When the last call to Q.FindMin() occurs, it cannot return $\perp$. If it did return $\perp$, this would mean Q is empty. But then p's final execution of the locked segment of code at lines 6–13 must occur after the last call to Q.FindMin(), otherwise p would already be in the queue and at (or about to execute) line 15 when the latter call occurs. This implies that p executes Q.FindMin() after the last call to Q.FindMin().

It also follows that when the last call to Q.FindMin() occurs, it returns the ID of a process q that is not associated with a dummy ticket. If q were associated with a dummy ticket, then q must be in the trying protocol before the locked segment of code. This means that q eventually executes Q.FindMin() after the last call to Q.FindMin().

Thus, one of the two following cases must hold: (i) the last call to Q.FindMin() is at line 12 and returns the ID of the caller, a process q; or (ii) the last call to Q.FindMin() is at line 20 and returns the ID of a process q that is at lines 14–16 at the time. In both cases, q will eventually be signalled to enter the CS, and so q will eventually call Q.FindMin() at line 20, after the last call to Q.FindMin().

Lemma 6. *The algorithm satisfies lockout freedom.*

Proof. Lockout freedom follows directly from FCFS (Lemma 4) and deadlock freedom (Lemma 5).

Lemma 7. *The algorithm has RMR complexity* $O(\min(k, \log N))$ *in both the DSM and CC models.*

Proof sketch. The ticket dispenser operations and *SpecialSet* operations have step complexity $O(\min(k, \log N))$. For the auxiliary lock at lines 5, 14, 16, 23, we use the adaptive mutual exclusion algorithm of Kim and Anderson [13], which has RMR complexity $O(\min(k, \log N))$. For the priority queue, we use a min-heap implementation, which has step complexity $O(\log k)$. The busy-wait loop at line 15 incurs $O(1)$ RMRs in the CC model, and no RMRs in the DSM model. Every other line of the algorithm causes at most $O(1)$ RMRs per passage.

The preceding lemmas culminate in the following theorem:

Theorem 1. *The algorithm defined by Figure 1 is a correct FCFS mutual exclusion algorithm, and it has RMR complexity* $O(\min(k, \log N))$ *in both the DSM and CC models, where* k *is the point contention and* N *is the number of processes in the system.*

Acknowledgements. We are grateful to Vassos Hadzilacos for his insightful feedback during the writing of this paper. We also thank the anonymous referees for their helpful comments.

References

1. Dijkstra, E.: Solution of a problem in concurrent programming control. Communications of the ACM 8(9), 569 (1965)
2. Lamport, L.: A new solution to Dijkstra's concurrent programming problem. Communications of the ACM 17(8), 453–455 (1974)
3. Mellor-Crummey, J., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems 9(1), 21–65 (1991)
4. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1(1), 6–16 (1990)
5. Kim, Y.-J., Anderson, J.: A time complexity bound for adaptive mutual exclusion. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 1–15. Springer, Heidelberg (2001)
6. Attiya, H., Hendler, D., Woelfel, P.: Tight RMR lower bounds for mutual exclusion and other problems. In: Proc. STOC 2008, pp. 217–226 (2008)
7. Golab, W., Hadzilacos, V., Hendler, D., Woelfel, P.: Constant-RMR implementations of cas and other synchronization primitives using read and write operations. In: Proc. PODC 2007, pp. 3–12. ACM, New York (2007)
8. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst 5(1), 1–11 (1987)
9. Styer, E.: Improving fast mutual exclusion. In: PODC 1992: Proceedings of the eleventh annual ACM symposium on Principles of distributed computing, pp. 159–168. ACM, New York (1992)
10. Choy, M., Singh, A.K.: Adaptive solutions to the mutual exclusion problem. Distrib. Comput. 8(1), 1–17 (1994)

11. Attiya, H., Bortnikov, V.: Adaptive and efficient mutual exclusion. Distrib. Comput. 15(3), 177–189 (2002)
12. Yang, J.H., Anderson, J.H.: A fast, scalable mutual exclusion algorithm. Distributed Computing 9(1), 51–60 (1995)
13. Kim, Y.J., Anderson, J.: Adaptive mutual exclusion with local spinning. Dist. Computing 19(3), 197–236 (2007)
14. Afek, Y., Stupp, G., Touitou, D.: Long lived adaptive splitter and applications. Distrib. Comput. 15(2), 67–86 (2002)
15. Cypher, R.: The communication requirements of mutual exclusion. In: SPAA 1995: Proc. of the 7th annual ACM symposium on Parallel algorithms and architectures, pp. 147–156. ACM Press, New York (1995)
16. Anderson, J., Kim, Y.J.: An improved lower bound for the time complexity of mutual exclusion. Distributed Computing 15(4), 221–253 (2002)
17. Fan, R., Lynch, N.: An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In: PODC 2006: Proc. of the 25th annual ACM symposium on Principles of distributed computing, pp. 275–284. ACM Press, New York (2006)
18. Jayanti, P.: f-arrays: Implementation and applications. In: PODC 2002: Proceedings of the twenty-first annual symposium on Principles of distributed computing, pp. 270–279. ACM, New York (2002)
19. Taubenfeld, G.: The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 56–70. Springer, Heidelberg (2004)

The Weakest Failure Detector for Message Passing Set-Agreement

Carole Delporte-Gallet[1], Hugues Fauconnier[1], Rachid Guerraoui[2], and Andreas Tielmann[1,*]

[1] Laboratoire d'Informatique Algorithmique, Fondements et Applications (LIAFA), University Paris VII, France
[2] School of Computer and Communication Sciences, EPFL, Switzerland

Abstract. In the set-agreement problem, n processes seek to agree on at most $n-1$ different values. This paper determines the weakest failure detector to solve this problem in a message-passing system where processes may fail by crashing. This failure detector, called the *Loneliness* detector and denoted $\mathcal{L}$, outputs one of two values, *"true"* or *"false"* such that: (1) there is at least one process where $\mathcal{L}$ outputs always *"false"*, and (2) if only one process is correct, $\mathcal{L}$ eventually outputs *"true"* at this process.

Keywords: set-agreement, failure detectors.

1 Introduction

The set-agreement problem [1] has no deterministic solution in asynchronous systems where any number of processes can fail by crashing [2,3,4] and the remaining processes have no information about such failures. With failure detection however, the impossibility can be circumvented [5]. For instance, with a perfect failure detection mechanism that accurately detects crashes, it is trivial for the processes to reach agreement. A natural question is what failure information is necessary and sufficient to reach agreement. In the parlance of [6], this question can be precisely formulated using the notion of "weakest failure detector": In short, the weakest failure detector to solve a problem is one that (a) indeed solves the problem and (b) can be emulated by any failure detector that solves the problem. Property (a) conveys the sufficiency of the failure detector whereas property (b) conveys its necessity.

Several papers have been devoted to determine the weakest failure detector to solve the set-agreement problem in a distributed system where any number of processes can fail by crashing [7,8,9,10]. In particular, Zieliński proved recently that anti-Ω – a failure detector that outputs id's of processes such that the id of at least one correct process is output only finitely many times – is the weakest failure detector for set-agreement in a shared memory system [10]. The proof of

[*] Work was supported by grants from Région Ile-de-France.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 109–120, 2008.

the result is particularly involved and builds on earlier proof techniques from [6] and [8].

In the context of message passing however, the weakest failure detector for set-agreement has not been determined yet and one might have hoped to derive it somehow from anti-Ω. Indeed, Zieliński conjectured in [11] that failure detector Σ [12] – the weakest failure detector to build a shared memory in a message passing system – is both sufficient and necessary to implement set-agreement. This would mean that some common denominator of anti-Ω and Σ would constitute the weakest failure detector for set-agreement in message passing. Nevertheless, Delporte et al. recently disproved Zieliński's conjecture by showing that Σ is not necessary, albeit sufficient [13]. The question of the weakest failure detector to solve set-agreement in a message passing system remained thus open. The contribution of this paper is precisely to close the question.

We introduce the *Loneliness* failure detector, denoted $\mathcal{L}$, and we show that it is the weakest failure detector for set-agreement in a message passing system. Failure detector $\mathcal{L}$ outputs, whenever queried by a process, one of two values: *"true"* or *"false"* such that the following two properties are satisfied: (1) there is at least one process where the output is always *"false"*, and (2) if only one process is correct (does not crash), then the output at this process is eventually *"true"* forever. We first give an algorithm that solves set-agreement using $\mathcal{L}$. The particularity of the algorithm is its non-symmetric nature as it heavily exploits the total order on the identity of the processes. We then assume that there is an algorithm A that solves set-agreement (with some failure detector), and we show how to "extract" from A the output of $\mathcal{L}$. Our approach here is, on the one hand, different from the approach of [6] where each process locally simulates several runs of A and, on the other hand, different from the approach of [10], as well as [8], where the extraction relies on the asynchronous impossibility of a problem. In our case, the processes execute one instance of A, without knowing the automaton of A performed at each process. The processes obtain the output of $\mathcal{L}$ by "simply" intercepting communication between these automata. This leads to a very simple, almost trivial, extraction algorithm.

Our proof that $\mathcal{L}$ is the weakest in message passing is thus remarkably simple and this might be surprising compared to the rather involved proof of Zieliński [10] in shared memory systems. Somehow, we show that – contrary to a wide belief – results in message passing systems are sometimes easier to prove than in shared memory.

We prove that – not surprisingly – failure detector $\mathcal{L}$ is strictly stronger than anti-Ω, the weakest in a shared memory system. (Indeed a message passing system can be emulated by a shared memory system but the converse requires additional assumptions, e.g., a majority of correct processes [14].) Furthermore, we show that no failure detector that may behave arbitrarily for any finite amount of time is stronger than $\mathcal{L}$ (but nevertheless such failure detectors can be incomparable with $\mathcal{L}$). We also show that for $n > 2$, Σ is strictly stronger than $\mathcal{L}$, confirming the result of [13] that emulating a shared memory requires more information about failures than reaching agreement (Figure 1).

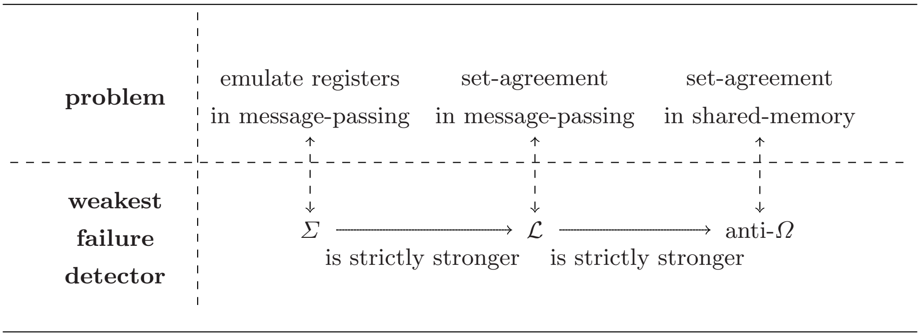

problem
emulate registers in message-passing
set-agreement in message-passing
set-agreement in shared-memory
weakest failure detector
Σ
ℒ
anti-Ω
is strictly stronger
is strictly stronger

Fig. 1. Relations between failure detector classes

The rest of the paper is organized as follows. We first define our model in Section 2. Then we show that $\mathcal{L}$ is sufficient for set-agreement in Section 3 and that $\mathcal{L}$ is also necessary in Section 4. In Section 5, we show that $\mathcal{L}$ is strictly stronger than anti-Ω. And finally, in Section 6 we show that for $n > 2$, Σ is strictly stronger than $\mathcal{L}$.

2 Model and Definitions

2.1 Processes and Failure Detectors

The system model we consider is that of Chandra et al. [6] which we briefly recall here. We consider a set $\Pi = \{p_1, \ldots, p_n\}$ of $n \geq 2$ processes which communicate by message passing over a fully connected network with reliable links. Any number of processes may fail by prematurely halting, i.e. they crash. However, no process can otherwise deviate from its protocol. We assume a global clock $\mathcal{T}$ that is used to depict steps in an execution; the clock is not accessible to the processes.

A failure pattern is a function from time $\mathcal{T}$ to 2^{Π} that specifies for every time t which processes have crashed by time t. A process p_i that does not crash in a failure pattern $\mathcal{F}$ is said to be correct in $\mathcal{F}$ ($p_i \in \mathit{correct}(\mathcal{F})$). A process is said to be alive until it crashes. Processes that are not correct are called faulty. An environment $\mathcal{E}$ is a set of possible failure patterns. In this paper, we consider every environment, i.e. any number of processes may crash and in particular any process may crash at any time.

A failure detector $\mathcal{D}$ is a distributed oracle that provides the processes with information about failures. A failure detector is defined by its histories. Given a failure pattern $\mathcal{F} \in \mathcal{E}$, a history H of a failure detector $\mathcal{D}$ is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}_{\mathcal{D}}$, the failure detector range of $\mathcal{D}$, i.e. the set of possible outputs of $\mathcal{D}$: $\mathcal{D}(\mathcal{F})$ denotes a set of failure detector histories that are allowed for $\mathcal{F}$.

An algorithm A is modeled as a set of n deterministic automata, one for every process in the system. A run of A proceeds in steps and at every time t at most one process executes a step. We assume only fair runs, i.e. every correct process executes infinitely many steps. A step consists of receiving a (possibly empty) message, reading a value of a failure detector, changing the state accordingly, and outputting a (possibly empty) message.

A failure detector is said to solve a problem in a given environment $\mathcal{E}$ if there is an algorithm that solves the problem using message passing and that failure detector (and no other information about failures) for every failure pattern in $\mathcal{E}$. A failure detector $\mathcal{D}$ is said to be stronger than another failure detector $\mathcal{D}'$ in an environment $\mathcal{E}$ if there is an algorithm that uses only $\mathcal{D}$ to emulate the output of $\mathcal{D}'$ for every failure pattern in $\mathcal{E}$. Similarly, detector $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$ if $\mathcal{D}'$ is stronger than $\mathcal{D}$ in $\mathcal{E}$. Failure detector $\mathcal{D}$ is said to be strictly stronger than failure detector $\mathcal{D}'$ in environment $\mathcal{E}$ if $\mathcal{D}$ is stronger than $\mathcal{D}'$ in $\mathcal{E}$ but not vice versa.

The weakest failure detector [6] $\mathcal{D}$ to solve a given problem in an environment $\mathcal{E}$ is a failure detector that is sufficient to solve the problem in $\mathcal{E}$ and that is also necessary to solve the problem, i.e. $\mathcal{D}$ is weaker than any failure detector that solves the problem in $\mathcal{E}$.

We define $\mathcal{D}$ to be (strictly) stronger (resp. weaker) than $\mathcal{D}'$ if $\mathcal{D}$ is (strictly) stronger (resp. weaker) in every environment. Similarly, a weakest failure detector for a problem is defined to be a weakest failure detector for this problem for every environment.

2.2 Set-Agreement

In the set-agreement problem, every process p_i starts with some proposal value v_i and tries to decide a value such that the following three properties are satisfied:

Agreement: At most $n-1$ different values are decided.
Validity: Every value that has been decided must have been a proposal value of some process.
Termination: Eventually, every correct process decides a value.

2.3 Failure Detector $\mathcal{L}$

We now define the *Loneliness* detector $\mathcal{L}$. This failure detector outputs one of two values *"true"* and *"false"*. The intuition behind the semantics of this failure detector is that if the output at some correct process is *"false"* forever, then there is another alive process in the system. By convention, we assume that if a process is crashed at time t, then its failure detector output at time t is *"false"*. The following properties are satisfied:

- at least one process never outputs *"true"*, and
- if only one process is correct, then it eventually outputs *"true"* forever.

More formally:

Definition 1. *The range of $\mathcal{L}$ is {"true", "false"}. For every environment $\mathcal{E}$, for every failure pattern $\mathcal{F} \in \mathcal{E}$, and every history $H \in \mathcal{L}(\mathcal{F})$:*

$$\exists p_i \in \Pi, \forall t, H(p_i, t) \neq \text{“true”} \tag{1}$$

$$\wedge\ \forall p_i \in \Pi, correct(\mathcal{F}) = \{p_i\} \Rightarrow \exists t, \forall t' \geq t, H(p_i, t') = \text{“true”} \tag{2}$$

3 The Sufficient Part

To show that failure detector $\mathcal{L}$ is sufficient to solve set-agreement in our model, we give an algorithm that implements set-agreement with $\mathcal{L}$. The algorithm is depicted in Figure 2.

To ensure that at most $n-1$ proposal values are decided, every process tries to agree with another process on one value. To achieve this, initially some processes send their values. To prevent a circular value exchange, i.e. a situation where the proposal values are simply permuted, the values are only sent to processes with a higher id. This means, that process p_1 sends its value to everybody (except itself), process p_i to all processes from p_{i+1} to p_n, and process p_n to nobody.

If some process receives[1] one of these values, it sends this value to all other processes and decides. As long as there is another correct process, every correct process decides either through one of the messages that were initially sent or, if it does not receive such a message (e.g., because it has a lower id than the other correct processes), it decides through a message of an already decided process. Note that it may be possible that a process receives its initial value back in such a message. In this case, the sender of this message does not decide its own proposal value.

To deal with crashes, we only execute these steps if the output of the failure detector is *"false"*. But in the case of only one correct process in the system, we do not want to wait for messages of other processes forever. Therefore, if the output of the failure detector changes to *"true"* – and by its property (2) in the case of only one correct process it will eventually do so – this process simply decides its own proposal value. We can do this without violating agreement, because by property (1) there will always be one process that does not decide

```
Algorithm for process $p_i$:
1 to propose($v$):
2    initially:
3       send $\langle v \rangle$ to all $p_j$ with $j > i$;
4    on receive $\langle v' \rangle$ do:
5       send $\langle v' \rangle$ to all;
6       decide $v'$; halt;                              (* decision D1 *)
7    on $\mathcal{L} =$ "true" do:
8       send $\langle v \rangle$ to all;
9       decide $v$; halt;                               (* decision D2 *)
```

Fig. 2. Implementing set-agreement with $\mathcal{L}$

[1] For simplicity of the presentation, we assume that the code Lines 5-6 and Lines 8-9 are executed atomically.

due to a *"true"* output, and as we have argued before, processes that decide due to a message exchange eliminate at least one value.

Proposition 1. *The algorithm in Figure 2 implements set-agreement in every environment $\mathcal{E}$.*

Proof. We have to prove the three properties of set-agreement, namely agreement, validity, and termination.

Agreement. We start with the agreement property of set-agreement. We assume a run where all processes decide and every process p_i has a distinct initial value v_i. Without this assumption, agreement is trivially met.

By Property (1) of $\mathcal{L}$, not all processes can have decided by decision D2. Therefore, in such a run at least one process decides by D1. This means that it is sufficient to show that if at least one process decides by D1, then at most $n-1$ values are decided.

Among the processes that decide by D1, consider p_i as the process with the highest id and let v' be the decided value. We distinguish between the two cases where p_i decides its initial value ($v' = v_i$), and where it does not.

Case 1: The only possibility that the decided value v' is equal to p_i's value v_i is that a process p_j with $j > i$ has received p_i's initial message and decided v_i. Therefore, p_i and p_j decide the same value and at most $n-1$ values are decided.

Case 2: If v' is not equal to v_i and $i = n$, then v_n will never be decided because process p_n does not send its value to anybody. If $i < n$, then the only possibility that v_i is decided is if a process p_k with $k > i$ has received v_i from p_i and decided by D1. But as p_i is the process with the highest id that decides by D1, such a k does not exist. And therefore, v_i is never decided.

Validity. The validity property of set-agreement is trivially satisfied, since only proposal values are sent.

Termination. If some correct process decides by D1 or D2, then it sends its decided value to all processes and all correct processes that have not yet decided eventually receive this value and also decide.

Therefore, it remains to show that in every run some correct process decides by D1 or D2. We distinguish two cases: the case when there exist at least two correct processes in a run with a failure pattern $\mathcal{F} \in \mathcal{E}$, and the case with only one correct process.

Case 1: If there are at least two correct processes and none decides by D2, then eventually, the one with the highest id receives the initial message of the other ones and decides by D1.

Case 2: If there is only one correct process and it does not decide by D1, then by property (2) of $\mathcal{L}$, this process eventually decides by decision D2. □

4 The Necessary Part

Following the approach of Chandra et al. [6], we show that failure detector $\mathcal{L}$ is necessary to solve set-agreement in our model by providing an algorithm that emulates the output of $\mathcal{L}$ given any algorithm A and failure detector $\mathcal{D}$, such that A using $\mathcal{D}$ solves set-agreement. Figure 3 presents such an emulation algorithm. The output of our emulation of $\mathcal{L}$ is provided through a special variable *output*.

The idea for the emulation of $\mathcal{L}$ is that if all messages that are sent by algorithm A get delayed for a very long time, the safety properties of set-agreement still have to hold, while for the case that only one process is correct, even the liveness property has to hold, i.e. the algorithm has to terminate. Therefore, every process executes A with $\mathcal{D}$, omits to send any messages that are generated by algorithm A to other processes, and outputs *"false"* until A terminates.

Property (1) of $\mathcal{L}$ is thus always fulfilled, because otherwise the executions at all processes would have terminated without ever receiving a message and therefore agreement could not have been guaranteed. But nevertheless, if there is only one correct process p_i, the algorithm A executed at p_i has to terminate and property (2) of $\mathcal{L}$ is also guaranteed.

Interestingly, this technique works for every non-trivial problem in which communication between processes is necessary, i.e. where not all processes may terminate without receiving messages from other processes. Therefore, $\mathcal{L}$ is necessary for all of these problems.

Proposition 2. *The algorithm in Figure 3 implements $\mathcal{L}$ in every environment $\mathcal{E}$.*

Proof. Assume there exists a run r, where the algorithm in Figure 3 does not fulfill property (1) of $\mathcal{L}$ with a failure pattern $\mathcal{F} \in \mathcal{E}$. This means, that in run r, for every process, there exists a time when *output* = *"true"*, i.e. the execution of algorithm A has terminated at all processes without receiving any message from other processes at all.

Let t be the time at which A has terminated at all processes in run r. Then, since the system is totally asynchronous, it is possible to construct a valid run r' of A with the same failure pattern $\mathcal{F}$, where all messages to other processes get delayed to a time after t, and all processes have terminated A at time t.

```
Algorithm for process $p_i$:
output := "false";
execute $A$ with value $i$ and detector $\mathcal{D}$, but omit sending messages to others;
if $A$ has terminated, then output := "true";
```

Fig. 3. Implementing $\mathcal{L}$ with an algorithm A and a failure detector $\mathcal{D}$ that solve set-agreement

Note that a failure detector is solely specified as a function over a failure pattern in an execution, i.e. it is not allowed to output any information about the state of other processes or to give hints about the proposal values.

Therefore, to fulfill the validity property of set-agreement, the decision value at every process p_i can only be its proposal value i. A contradiction with the agreement property of set-agreement. Therefore, property (1) of $\mathcal{L}$ is always satisfied.

If for some run r of our algorithm, for some process p_i, $\mathcal{F}$ is the failure pattern in run r and $\mathit{correct}(\mathcal{F}) = \{p_i\}$, then it is possible to construct a run r_A of A in which no faulty process is able to send a message (because the system is totally asynchronous) and p_i takes exactly the same steps as in r. By the termination property of set-agreement, eventually algorithm A has to terminate in run r_A at p_i. Since r and r_A are indistinguishable for p_i, it terminates the execution of A also in r and the output changes to *"true"*. Thus, property (2) is also satisfied. □

Theorem 1. *$\mathcal{L}$ is the weakest failure detector for set-agreement in a message passing system.*

Proof. We have shown in Proposition 1 that $\mathcal{L}$ is sufficient and in Proposition 2 that it is necessary for set-agreement in all environments. □

5 Comparing $\mathcal{L}$ and Anti-Ω

To keep our proofs as generic as possible, we first introduce the notion of eventual failure detectors. We say that a failure detector is an eventual failure detector if the detector can behave arbitrarily for any finite amount of time. A more formal definition can be found in [15] where such failure detectors are called strongly unreliable failure detectors.

Zieliński shows in [16] that every eventual failure detector (that satisfies some other assumptions that are irrelevant here) is stronger than anti-Ω, the weakest failure detector for set-agreement in a shared memory [10]. Each query to the anti-Ω detector returns a process id. The failure detector guarantees that there is a correct process whose id will be returned only finitely many times. Clearly, anti-Ω is an eventual failure detector and $\mathcal{L}$ is not. We show that $\mathcal{L}$ is strictly stronger than anti-Ω. This means, that to implement set-agreement in message passing there is a strictly stronger failure detector necessary than in shared memory.

Lemma 1. *$\mathcal{L}$ is stronger than anti-Ω.*

Proof. An implementation of anti-Ω using $\mathcal{L}$ is given in Figure 4. The basic idea is simple: Every process p_i outputs the id j of a process p_j such that j is the lowest id of all processes from which p_i has not yet heard that they have had a *"true"* as failure detector output. For this, the processes remember the ids of processes that have received a *"true"* from $\mathcal{L}$ in a set *lonely*. The output of anti-Ω is emulated in a special variable *output*.

Algorithm for process p_i:

```
initially:
    $lonely := \emptyset$;
    $output := \{1\}$;
on $\mathcal{L} =$ "true" do:
    $lonely := lonely \cup \{i\}$;
    send $\langle lonely \rangle$ to all;
    $output := \min(\{1, \ldots, n\} \setminus lonely)$;
on receive $\langle lonely' \rangle$ do:
    if $lonely \neq lonely'$ then send $\langle lonely \cup lonely' \rangle$ to all;
    $lonely := lonely \cup lonely'$;
    $output := \min(\{1, \ldots, n\} \setminus lonely)$;
```

Fig. 4. Implementation of anti-Ω using $\mathcal{L}$

We now show that this transformation indeed emulates anti-Ω. From property 1 of the definition of $\mathcal{L}$, the output of at least one process is never a *"true"*. Therefore, there is always at least one id that is output (i.e. it is never $lonely = \{1, \ldots, n\}$).

To prove that there is a correct process whose id is output only finitely often, note that eventually the set *lonely* is the same at all correct processes because it can only grow and will always be a subset of $\{1, \ldots, n\}$ (and every correct process relays it after every change). Therefore, eventually all correct processes have the same output. Now assume the id of every correct process is output infinitely often at the processes. This implies that there is only one correct process, because all processes always output the minimum of $\{1, \ldots, n\} \setminus lonely$ which can only shrink and therefore never oscillates between different process ids. But from property 2 of the definition of $\mathcal{L}$, a single correct process eventually receives a *"true"* and therefore belongs to its set *lonely*. A contradiction. □

Lemma 2. *No eventual failure detector is stronger than* $\mathcal{L}$.

Proof. Assume there exists an algorithm A that transforms an eventual failure detector $\mathcal{D}$ to $\mathcal{L}$. Then, assume for every $1 \leq i \leq n$, a run r_i of A with failure pattern $\mathcal{F}_i$ and $correct(\mathcal{F}_i) = \{p_i\}$ and where the faulty processes take no steps. If A is correct, then eventually the output at process p_i in run r_i has to be *"true"*, say at time t_i. Similarly, assume a run r of A with a failure pattern $\mathcal{F}$ with $correct(\mathcal{F}) = \Pi$, but no process p_i receives a message from any other process before or at time t_i and every p_i is scheduled as in r_i. Let the output of $\mathcal{D}$ at every process p_i before time t_i be exactly as in run r_i (this is possible, since $\mathcal{D}$ may behave arbitrarily for any finite amount of time). Then, for every process p_i, run r_i is indistinguishable from run r before time t_i and every process p_i outputs *"true"* at time t_i. But this contradicts property 1 of $\mathcal{L}$. □

Theorem 2. *$\mathcal{L}$ is strictly stronger than anti-Ω.*

Proof. Follows directly from Lemma 1 and Lemma 2. □

6 Comparing $\mathcal{L}$ and Σ

We now show that Σ, the weakest failure detector to emulate a shared memory in message-passing systems [12] is strictly stronger than $\mathcal{L}$. In a sense, this indicates that emulating a shared memory in message passing is strictly harder than solving set-agreement, confirming the result of [13]. By convention, we assume that if a process is crashed at time t, then its failure detector output is Π at time t. At each invocation, Σ outputs a list of trusted processes and it satisfies two properties:

Intersection: Given any two lists of trusted processes, possibly at different times and by different processes, at least one process belongs to both lists.
Completeness: Eventually no faulty process is ever trusted by any correct process.

Lemma 3. *Σ is stronger than $\mathcal{L}$.*

Proof. The reduction is simple: At the beginning, every process outputs *"false"*. For every process p_i, if the output of Σ is $\{p_i\}$, output *"true"*.

Assume that for every process there is some time when the output of $\mathcal{L}$ is *"true"*. Since this happens only if at every process p_i, $\{p_i\}$ is output, the intersection property of Σ is clearly violated. Therefore, this will never happen and property 1 of $\mathcal{L}$ is never violated.

From the completeness property follows that if a process p_i is the only correct process, the output will eventually be *"true"* (property 2 of $\mathcal{L}$). □

For the special case that the system consists only of two processes, the specifications of set-agreement and consensus are equivalent. Delporte-Gallet et al. show in [17] that for this case Σ is the weakest failure detector for consensus. Together with Theorem 1 this immediately implies that $\mathcal{L}$ and Σ are also equivalent for this case. However, in the following lemma we show that for $n > 2$ this is not the case.

Lemma 4. *$\mathcal{L}$ is not stronger than Σ, if $n > 2$.*

Proof. Assume there exists an algorithm A that transforms $\mathcal{L}$ into Σ. Let $P = P_1, P_2, P_3$ be any partitioning of Π. Then assume two runs r_1 and r_2 where the processes in P_i are correct in run r_i and all other processes are faulty from the beginning, and the output of $\mathcal{L}$ at the processes in partition P_i is *"true"*. Since A fulfills completeness, it eventually has to output in every run r_i a subset of P_i, say at time t_i.

Now imagine a run r in which the processes in P_1 and P_2 are correct and the output of $\mathcal{L}$ is *"true"*. Additionally, no message of a process from a different partition is received in partition P_1 and P_2 before time t_1 (respectively t_2) and the messages between the processes in P_1 and P_2 are exactly scheduled as in runs r_1 and r_2. The runs r_1 and r_2 are indistinguishable from run r before time t_1 (respectively t_2). Therefore, the output at time t_i will be a subset of P_i for partition $i = 1, 2$. But this contradicts to the intersection property of Σ. So there exists no such algorithm A. □

Theorem 3. *If $n > 2$, then Σ is strictly stronger than $\mathcal{L}$.*

Proof. Lemma 3 shows that Σ is stronger than $\mathcal{L}$ and Lemma 4 shows that it is strictly stronger. □

7 Summary

We have determined the weakest failure detector for set-agreement in a message-passing system where processes may fail by crashing. The failure detector is called $\mathcal{L}$ and it returns at every invocation *"true"* or *"false"*. It ensures that (1) there is at least one process where the output is always *"false"*, and (2) if there is only one correct process, then the output at this process is eventually *"true"* forever.

Acknowledgments. We are grateful to Sam Toueg for helpful suggestions on the sufficient part of our proof. Furthermore, we would like to thank the reviewers for their helpful comments.

References

1. Chaudhuri, S.: More choices allow more faults: Set consensus problems in totally asynchronous systems. Inf. Comput. 105(1), 132–158 (1993)
2. Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM J. Comput. 29(5), 1449–1483 (2000)
3. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for t-resilient asynchronous computations. In: STOC 1993: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp. 91–100. ACM, New York (1993)
4. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM 46(6), 858–923 (1999)
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
6. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
7. Raynal, M., Travers, C.: In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 3–19. Springer, Heidelberg (2006)
8. Guerraoui, R., Herlihy, M., Kouznetsov, P., Lynch, N., Newport, C.: On the weakest failure detector ever. In: PODC 2007: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, pp. 235–243. ACM, New York (2007)

9. Chen, W., Zhang, J., Chen, Y., Liu, X.: Weakening failure detectors for k -set agreement via the partition approach. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 123–138. Springer, Heidelberg (2007)
10. Zieliński, P.: Anti-Omega: the weakest failure detector for set agreement. In: PODC 2008: Proceedings of the twenty-seventh annual ACM symposium on Principles of distributed computing (2008)
11. Zieliński, P.: Anti-Omega: the weakest failure detector for set-agreement. Technical report, UCAM-CL-TR-694 (2007)
12. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Shared memory vs message passing. Technical report, LPD-REPORT-2003-001 (2003)
13. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Sharing is harder than agreeing. In: PODC 2008: Proceedings of the twenty-seventh annual ACM symposium on Principles of distributed computing (2008)
14. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM 42(1), 124–142 (1995)
15. Guerraoui, R.: Indulgent algorithms. In: PODC 2000: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (2000)
16. Zieliński, P.: Automatic classification of eventual failure detectors. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 465–479. Springer, Heidelberg (2007)
17. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R. (Almost) all objects are universal in message passing systems. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724. Springer, Heidelberg (2005)

Local Maps: New Insights into Mobile Agent Algorithms

Bilel Derbel[*]

Laboratoire d'Informatique Fondamentale de Lille (LIFL),
Université des Sciences et Technologies de Lille, France
bilel.derbel@lifl.fr

Abstract. We address the problem of computing with mobile agents having small local maps. Several trade-offs concerning the radius of the local maps, the number of agents, the time complexity and the number of agent moves are proven. Our results are based on a generic simulation scheme allowing to transform any message passing algorithm into a mobile agent one. For instance, we show that using a near linear (resp. sublinear) number of agents having local maps of polylogarithmic (resp. sublinear) radius allows us to obtain a polylogarithmic (resp. sublinear) ratio between the time complexity of a message passing algorithm and its mobile agent counterpart. As a fundamental application, we show that there exists a universal algorithm that computes, from scratch, any global labeling function of any graph using n mobile agents knowing their $o(n^{\epsilon})$-neighborhood (resp. without any neighborhood knowledge) in $\widetilde{O}(D)$ time (resp. $\widetilde{O}(\Delta + D)$ expected time)[1], where n, D, Δ are respectively the size, the diameter, the maximum degree of the graph and ϵ is an arbitrary small constant. For the leader election problem (resp. BFS tree construction), we obtain $\widetilde{O}(D)$ time algorithms under the additional restriction of using mobile agents having only $\log^{O(1)} n$ (resp. $\widetilde{O}(n)$) memory bits.

To the extent of our knowledge, the impact of local maps on mobile agent algorithms has not been studied in previous works. Our results prove that small local maps can have a strong global impact on the power of computing with mobile agents. Thus, we believe that the local map concept is likely to play an important role to a better understanding of the locality nature of mobile agent algorithms.

Keywords: Mobile agents, local maps, time complexity.

1 Introduction

Motivation and Goals. In a mobile agent algorithm, we are given a set of mobile entities equipped with a memory and able to move from a node to another in the network in order to perform some computations. To evaluate the efficiency

[*] Supported by the DOLPHIN INRIA project team.

[1] $\widetilde{O}(f) \overset{\text{def}}{=} f \cdot (\log n)^{O(1)}$.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 121–136, 2008.

of a mobile agent algorithm, the most common complexity measures are time, number of agents, number of agent moves, and memory size of the agents. The main motivation of this paper is to understand the relationship of these classical complexity measures, especially the *time complexity*, to the *initial local view* complexity measure.

For an intuitive definition of the *initial local view* of a mobile agent, and to show how it can interfere with the way of solving a distributed problem, let us consider the following example. Take a *non-anonymous* static graph that models a set of pairwise connected locations where some robots (i.e., mobile agents) are scattered. The robots can move from one location to a neighboring one. They can communicate by leaving a message in the location where they pass. Suppose that initially each robot is given a map of its surrounding locations and they have to agree on some location to meet, i.e., elect a location. It is clear that if initially the robots know the entire location map (that is the common graph), then they can choose the location having the smallest name to meet there. The time needed to gather the robots is then dominated by the time needed to reach that elected location. Suppose now that initially each robot is given only an incomplete or a restricted map of its environment, for example a map of only the closest locations. First, it is not straightforward how the information provided by the local maps can help the robots to meet. Second, assuming that the information given by the local maps does so, it is not obvious that the robots can meet as fast as if they know the whole environment.

Roughly speaking, the initial local view of a mobile agent measures how small is the local map given to it initially. More precisely, it measures the *radius* of the initial map given to the agent. The purpose of this paper is to show that small local maps have a strong global impact. In particular, our aim is to show how local maps can help designing efficient mobile agent solutions.

Methodology and results. We tackle this problem independently of both the distributed task and the underlying network. For that purpose, we adopt a "simulation" based approach that allows us to transform any message passing algorithm into a mobile agent one. In fact, the initial local view of a mobile agent can be interpreted from a message passing perspective as the amount of knowledge that a node may have about its neighborhood. Since many distributed problems have been shown to have efficient local solutions in the message passing setting, providing a generic and efficient simulation method will also lead to design efficient local solutions using mobile agents.

Following this intuitive approach, we develop a generic scheme that allows us to simulate message passing algorithms in the mobile agent model (see model details in Section 2). Our scheme is based on a partition of the graph into a set of clusters. Roughly speaking, each cluster is controlled by some agents that simulate the message passing algorithm in that cluster while collaborating with the other agents in neighboring clusters. The computations inside a cluster are almost for free, only computing at the border of a cluster costs high. One should remark that the idea of using clustered representations is not new in distributed computing and it has already been proved to be extremely useful to solve many distributed problems, e.g., [1, 2, 3, 13, 33]. In this paper, we carefully combine several related techniques concerning sparse partitions [5, 13], small dominating sets [27], efficient synchronizers [2, 29] and balanced tree structures [9] to derive an efficient simulation scheme that deals with mobile agent specific issues.

Using partitions optimizing either the number of clusters, the radius of clusters, or the inter-connexion between clusters, we obtain trade-offs concerning the complexity of our simulation scheme. For instance, we show that using $O(n^{1+1/\ell})$ mobile agents, a message passing algorithm with time complexity τ can be simulated in $\widetilde{\tau}$ time such that $\widetilde{\tau}/\tau = \ell^{O(1)}$ (n is the size of the graph, ℓ is an integer). If the number of agents is fixed to be $k < n$, then we obtain a time ratio of at most $O(n/\sqrt{k})$. The latter results hold when the initial local view of agents is $\ell^{O(1)}$ and $O(n/\sqrt{k})$ respectively. Trade-offs concerning the number of agent moves are also given. The most interesting corollaries are obtained for $\ell = \Theta(\log n)$ or $k = o(n)$, where our results show that using near linear (resp. sublinear) number of agents having local maps of polylogarithmic (resp. sublinear) radius allows to obtain a polylogarithmic (resp. sublinear) time ratio.

Since our simulation scheme is based on some partitions with particular properties, one major difficulty to apply our scheme is to construct these partitions distributively and to initialize the mobile agents from scratch. We describe efficient distributed initialization techniques to cope with these issues. In particular, we show that the cost of initializing the agents from scratch can be made negligible in comparison to the cost of the simulation itself. Our initialization technique is based on an adaptation of a distributed decomposition algorithm described in [13] and allowing us to gain a log factor in the size of the spanners constructed in [13] which could be of independent interest.

Based on the latter results, we derive a universal distributed algorithm that computes in $\widetilde{O}(D)$ time *any labeling function* on *any graph* using in every node *one* mobile agent knowing its $o(n^{\epsilon})$-neighborhood, $\epsilon > 0$ is any arbitrary small constant and D is the diameter of the graph. Using mobile agents with no initial information at all, the time complexity of our algorithm becomes $\widetilde{O}(\Delta + D)$ in expectation (Δ is the maximum degree of the graph). These results suggest that from a time complexity point of view mobile agents having small initial knowledge could be as powerful as message passing.

Finally, we investigate the time complexity of computing two fundamental distributed problems with *limited memory* agents: the leader election and the Breadth First Spanning (BFS) tree. Using n mobile agents knowing their n^{ϵ}-neighborhood, and having respectively $\log^{O(1)} n$ and $\widetilde{O}(n)$ memory bits, we obtain $\widetilde{O}(D)$ time algorithms for both problems.

Overview of related works. To the extent of our knowledge, the relation between the initial local view as defined in this paper and the complexity of mobile agent algorithms has not been studied in previous works. However, one can find many seemingly related concepts.

The concept of *limited visibility* (see e.g., [14, 15, 21, 22, 35]) is perhaps the most closely related to the issues addressed in this paper. Roughly speaking, a robot is said to have a limited visibility if it can sense its surrounding up to a fixed distance, i.e., it can look and see the positions of other robots in a ball of a fixed radius at any time of the execution of the algorithm. It is a fact that the computation models used to study limited visibility agents are different from ours. Moreover, the problems studied therein are in general different from the *labeling* problems addressed in this paper.

Another related concept, called *oracles* (or *advice*), araised recently for some specific problems, e.g., broadcast / wake-up [17], tree exploration [18], coloring [16], graph searching [31]. The concept of oracles is tightly related to the concept of *labeling schemes* (see e.g., [19, 23, 24, 25]). These concepts may apply for the mobile agent model as well as for the message passing model. Informally speaking, an oracle with respect to a given problem P is an algorithm that, given a graph, outputs a labeling of nodes in such a way solving P by a distributed algorithm can be done efficiently. The challenges in that context are *(i)* to design an oracle minimizing the number of bits used for the labeling of nodes, and *(ii)* to come out with a distributed algorithm that, using the information provided by the oracle (the labels), solves P efficiently. Although studying oracles helps understanding the locality of a given problem, the issues addressed by oracles are different from those studied in this paper in many ways. In fact, paraphrasing the discussion made in the introduction of [26], we can say that oracles "are centralized, in the sense that they are based on a sequential algorithm which given a description of the entire graph outputs the entire set of node labels[2]. Hence, while the resulting short labels reflect local knowledge and can be used locally, their generation process is centralized and global".

Many other local issues concerning specific mobile agent problems have been extensively studied over the last few years (e.g., graph exploration [10], decontamination [15], election [7], etc). Few of them have considered the impact of the initial local knowledge on the complexity of mobile agent algorithms.

In this paper, we are addressing the problem of solving *any labeling task*, on *any network*, in a *fully distributed way*, provided that some local maps are known. One should remark that the existing approaches were not concerned with the three previous issues. More generally, our interest in the local map concept stems mainly from the fact that it allows to bring a new approach to understand the locality nature of mobile agent algorithms and to think classical mobile agents problems in a different way.

From a message passing perspective, the impact of local knowledge on solving distributed tasks has been intensively studied over the last years. For instance, the results presented in [4] show that broadcasting in a network, where each node knows its ρ-neighborhood, can be done using $\Theta(\min\left\{m, n^{1+O(1)/\rho}\right\})$ messages, where m (resp. n) is the number of links (resp. nodes) in the network. Reviewing all the existing results on the locality of message passing algorithms is beyond the scope of this paper. The reader is referred to [28, 30, 34] and the pointers there for further details concerning the particularly rich state-of-the-art concerning the locality of distributed message passing algorithms. One should keep in mind that

[2] Exceptions exist for labeling schemes but they are problem specific or topology specific, e.g., [26].

the simulation technique presented in this paper is a tool to answer the question: how powerful small local maps could be when designing mobile agent algorithms? Although our results allow to give an answer to "how powerful mobile agent algorithms could be compared to message passing ones?", our primary goal is to study the impact of local maps from a pure mobile agent perspective.

Only few works have studied the relationship between mobile agent algorithms and message passing algorithms. In [6, 8], a simulation based approach provides equivalence results between tasks that can be computed with messages and those that can be computed with mobile agents. Roughly speaking, the research concern there is on determining what tasks can be computed by mobile agents and under what conditions, but not at what cost. More recently, it is shown in [11] how to simulate message passing algorithms with mobile agents under failures with a polynomial overhead in the number of moves per agent (for simulating one message sending). In [20, 32], the authors showed how mobile agents (from a system engineering point of view) can define a *navigational* programming style for distributed parallel computing which is competitive in many points compared to classical message passing and shared memory systems.

Outline. In Section 2, we define the notations and the distributed models. In Section 3, we describe two basic and independent simulation techniques. In Section 4, we describe and analyze our generic simulation scheme. In Section 5, we show how to couple our simulation scheme with some clustered representations. In Section 6, efficient initialization techniques are described. In section 7, we apply our results to compute any function of a given graph, then to compute a leader and a BFS tree using agents having limited memory.

2 Definitions and Models

We model a network by a connected undirected graph $G = (V, E)$ where V is the set of nodes and E is the set of edges. A labeled graph is a graph where nodes and edges are assigned labels. We denote by n, m, Δ, and D respectively the number of nodes ($n = |V|$), the number of edges ($m = |E|$), the maximum degree, and the diameter of G. Each node v of G is given a unique integer identifier of at most $O(\log n)$ bits. The ports, that is the incident links, of a node v have distinct identifiers taken from 1 to deg_v, where deg_v is the degree of v. The size n of the graph G is known. We define the radius of a subgraph H of G as follows: $\mathrm{Rad}(H) = \min_{u \in H} \{\max_{v \in H} \{d_G(u,v)\}\}$, where $d_G(u,v)$ is the distance between the nodes u and v in G. Given a set of nodes C of G, we denote $G[C]$ the subgraph of G induced by the nodes of C. Given a set $\mathcal{C}$ containing some sets of nodes, we denote $\mathrm{Rad}(\mathcal{C}) = \max_{C \in \mathcal{C}} \{\mathrm{Rad}(C)\}$ where $\mathrm{Rad}(C) = \mathrm{Rad}(G[C])$. We denote $G_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ the graph induced by $\mathcal{C}$, i.e., there is an edge between two nodes in $G_{\mathcal{C}}$ if their corresponding sets are neighbors in G. We denote by G^t the graph obtained by adding an edge between every two nodes at distance at most t in G. For the sake of simplicity, we will write n^{ϵ} instead of $n^{O(1/\sqrt{\log n})}$. In fact, $n^{O(1/\sqrt{\log n})} = o(n^{\epsilon})$ for any arbitrary small constant $\epsilon > 0$. Also, we will use the $\widetilde{O}$ notation ($\widetilde{O}(f) \stackrel{\mathrm{def}}{=} f \cdot (\log n)^{O(1)}$) and omit precising some polylogarithmic overheads when they can be easily deduced from the context. In the next paragraphs, we detail the two distributed models we will be concerned with.

The message passing model. In this model, each node of a graph G is an autonomous entity of computation that can communicate with its neighbors by sending and receiving messages. The output of a message passing algorithm is given by the final labeling of the graph. In the remainder, we write MsA to denote a message passing algorithm running on a graph G. We concentrate on message passing algorithms that *detect the local termination*. More precisely, this assumption requires that each node *can detect that it will not make no more computations*. Actually, our termination assumption is made only for the sake of simplicity and clarity, but fundamentally, it does not affect our results as we will point in Remark 1. We assume the classical *synchronous message passing model* [34]. In other words, we assume that there exists a global clock generating the same pulses for all nodes. At each pulse a node can process some messages, do some local computations and send messages to its neighbors. A message sent at a given pulse arrives before the next pulse. We denote by Time(MsA) the time complexity of algorithm MsA, that is the number of pulses from the beginning of the algorithm up to its termination, and Message(MsA) the message complexity of the algorithm, that is the total number of messages exchanged by nodes.

As we will precise later, our mobile agent model is however *asynchronous*. This allows us to consider the most general scenarios in term of synchrony.

The mobile agent model. In this model, a mobile agent is a computation entity which is able to move from a node to another to perform computations. We assume that a mobile agent is equipped with an internal memory of unlimited capacity. Each node has a whiteboard of unlimited capacity where agents can write and read information in a mutual exclusion manner. When an agent arrives at a node v, it is able to distinguish the edge from which it has arrived to v among all other edges of v. We assume that the output result computed by a mobile agent algorithm is encoded by mean of the whiteboards of the nodes. In other words, the output is a labeling of the graph. Let us remark that our assumptions concerning the memory are introduced in order to focus on the high level locality issues. However, we will take a special care to derive algorithms using small memory agents.

We consider the fully *asynchronous* mobile agent model. More precisely, the mobile agent algorithms that we will describe do *not* exploit any assumptions on time such as the existence of a global clock or an upper bound on the agent delay to do some actions, etc.

In the following, we denote by AgA a mobile agent algorithm computing some labeling of a graph G. We say that a node is in a final state if its whiteboard will not be changed by any mobile agent. We say that a mobile agent algorithm terminates, if the algorithm ends up with all nodes in a final state. The number of agents used by algorithm AgA is denoted by Size(AgA). The total number of agent moves from the beginning of the algorithm up to its termination is denoted Cost(AgA). The time complexity of algorithm AgA, denoted by Time(AgA), is defined as the total number of time units from the beginning of the algorithm up to its termination, assuming that an agent move induces a delay of one time unit and that the computations done by agents are time negligible. The time units are introduced only for the pure sake of analysis. We emphasis on the fact that time units in the latter definition are with regard to a pure theoretical external clock which is not available to agents in any way.

Now, let us define the initial local view complexity measure. Let $\mathcal{A}$ be the set of mobile agents used by algorithm AgA. For every mobile agent $A \in \mathcal{A}$, we call h_A the homebase of agent A if at the beginning of the algorithm, agent A is at node h_A. We say that an agent $A \in \mathcal{A}$ has an *initial local view* H_A, if at the beginning of the algorithm agent A knows a labeled connected subgraph H_A of G containing its homebase h_A. *The initial local view* of algorithm AgA, denoted $\mathcal{ILV}(\text{AgA})$, is then defined by $\mathcal{ILV}(\text{AgA}) = \max_{A \in \mathcal{A}} \{\text{Rad}(H_A)\}$. In the remainder, given a message passing (resp. mobile agent) algorithm MsA (resp. AgA) solving a problem P, we will term *time ratio*: Time(AgA)/Time(MsA).

3 Basic Techniques

First, suppose that we have only one mobile agent knowing the whole graph G, i.e., its initial local view is D. Then, any global labeling function of G can be computed in $O(n)$ time as follows: First, the agent computes a copy of the output labeling locally at his homebase. Second, the agent explores the graph in order to assign the final state of each node. According to our model assumptions, only the graph exploration is time consuming. Exploring a known graph can be done in $O(n)$ time, using a depth first traversal for example. Thus, any labeling function can be computed by one agent knowing the whole graph in $O(n)$ time.

Since the time complexity of the previous technique depends only on the time needed to traverse the graph and mark the final states of nodes, one may ask whether we can go faster by allowing more than one agent. The answer is positive. In fact, an efficient graph structure was given in [9] in order to explore a graph searching for black holes. That data structure represents the graph using a forest of k trees (spanning the whole graph), each tree has at most $O(n/k)$ nodes. Thus, using k mobile agents, the agents can mark the nodes of the graph in parallel using the *"balanced tree"* structure of [9]. Hence, we can prove:

Theorem 1. *Given an integer parameter $k \geqslant 1$, any labeling function of G can be computed by an asynchronous mobile agent algorithm* AgA *such that:*

$\mathcal{ILV}(\text{AgA})$	D
Size(AgA)	k
Time(AgA)	$O(n/k + D)$
Cost(AgA)	$O(n + k \cdot D)$

Now, suppose that each node is assigned a mobile agent having no initial local view. Suppose we want to simulate a given message passing algorithm MsA. Then, a simple idea is to make each agent A_u in node u simulate the instructions that algorithm MsA would have done at node u. Let us first consider the following simple technique. **(i)** Using the whiteboard of u, agent A_u creates deg_u *receive-buffers* corresponding to the ports of u. Each receive-buffer is identified by its corresponding port. **(ii)** To simulate a send instruction of a message from node u to a node v, agent A_u stores the message in its internal memory. Then, it crosses the edge connecting u to node v. Once at node v, it writes the message in the corresponding receive-buffer of the whiteboard of v. After that, the agent goes back to u. **(iii)** To simulate a local computation instruction, agent A_u makes the same local computation using the whiteboard of u.

Clearly, the previous technique is correct when algorithm MsA is asynchronous. However, it may fail when the algorithm is synchronous. Furthermore, it may fail even if we assume that the mobile agent model is synchronous. A solution that solves the problems due to synchrony is to use a synchronizer-α like algorithm [2, 29] in order to "synchronize" an agent A_u with the other agents in the neighborhood of u. More precisely, in order to simulate the send instructions of a given pulse p, each agent A_u proceeds in two stages. At the first stage, agent A_u simulates the send instructions as explained before. At the second stage, the agent moves to each node v in its neighborhood and it writes a special <*IamSafe in p*> message in the corresponding receive buffer of v saying that it has finished simulating the instructions of pulse p. When the agent A_u learns that all the agents in its neighborhood are also safe, then it can proceed with pulse $p+1$ and so on. Clearly, simulating one pulse is at most $O(\Delta)$ time consuming, and requires the agents to move $O(m)$ times. Using $deg_u + 1$ agents per node, one can see that the time ratio can be reduced to $O(1)$.

The previous techniques are resumed by the following theorem:

Theorem 2. *Any synchronous message passing algorithm* MsA *can be simulated by an asynchronous mobile agent algorithm* AgA *such that:*

$\mathcal{ILV}$(AgA)	0	
Size(AgA)	n	$O(m)$
Time(AgA)	$O(\Delta \cdot \text{Time(MsA)})$	$O(\text{Time(MsA)})$
Cost(AgA)	$O(m \cdot \text{Time(MsA)})$	

4 A Generic Simulation Scheme

The idea of our generic scheme is to combine the previous basic techniques with a clustered representation of the graph. In this section, we will consider the following assumption:

H($\mathcal{C}$)*: We are given a precomputed partition $\mathcal{C}$ of the graph G into disjoint regions. Each region C has a distinguished node r_C called the center and a precomputed spanning tree T_C rooted at the center. At node r_C, there is a mobile agent called the master, together with some other agents called workers. Each master knows a labeled copy of the neighborhood of its region. In particular, he knows the edges leading to different regions.*

Input: a graph G with $\mathbf{H}(\mathcal{C})$ satisfied; a message passing algorithm MsA.
Output: algorithm AgA in a region C.

1. For each pulse p in algorithm MsA, the master agent of a region C simulates the instructions of pulse p for all nodes in C (locally in r_C):
   (a) For every node in C, the master creates a collection of receive-buffers in the whiteboard of r_C.
   (b) To simulate a send instruction from node $u \in C$ to node $v \in C$, the master writes the message in the corresponding receive-buffer created in r_C.
   (c) If some nodes in C send some messages in algorithm MsA to some other nodes belonging to a region $C' \neq C$, then the master concatenates those messages into only one message. Then, one worker is chosen to deliver that message to $r_{C'}$. The worker returns to r_C once its job finished.
   (d) If a node $u \in C$ makes some computations in algorithm MsA, then the master of C makes the same computations using the witheboard of r_C.
   (e) Once the master finishes simulating the sending of messages of pulse p, it synchronizes with other masters in neighboring region: the workers move to neighboring regions and deliver a special <*IamSafe* (C, p)> message.
   (f) Once the master is safe and learns that all the neighboring masters are safe, then it proceeds with the next pulse $p + 1$.
2. Once a master and its workers have finished simulating the instructions of algorithm MsA in C:
   (a) They inform other masters in neighboring regions that the simulation is terminated for region C by delivering an <*IamDone* C > message.
   (b) They continue synchronizing with a neighboring master in region C' until an <*IamDone* C' > message is delivered by that master.
   (c) They mark every node in C with its final state computed in Step 1. (Once an <*IamDone*> message is delivered by each neighboring master).

Fig. 1. High level description of the simulation scheme

Distributed methods to cope with assumption $\mathbf{H}(\mathcal{C})$ will be presented later in sections 5 and 6. For now, we concentrate on describing and analyzing our generic simulation scheme under the hypothesis $\mathbf{H}(\mathcal{C})$.

Provided that $\mathbf{H}(\mathcal{C})$ is true, any synchronous message passing algorithm MsA can be simulated by the generic scheme described in Fig. 1. There are two key points in our scheme: **(i)** The messages sent by nodes in a region C to nodes in a region C' ($C \neq C'$) are concatenated into only one message $M_{C \to C'}$ (Step 1.c). The message $M_{C \to C'}$ is delivered to the master in center node $r_{C'}$ by one worker at once, thus avoiding to deliver the messages one by one. **(ii)** The pulses of algorithm MsA are simulated using a combination of a synchronizer-γ like algorithm and a synchronizer-α like algorithm [2, 29]. In fact, the master synchronizes the nodes in its region by itself (This is made possible since the master knows the entire topology of its region). Then the master synchronizes its region with neighboring ones.

Clearly, the simulation of a send instruction between two nodes *not* in the same region dominates the overall time complexity per pulse. The concatenation mechanism allows to deliver the messages exchanged by nodes in a region C and nodes in a different region C' at once. Therefore, assuming that each master agent has as many workers as neighboring regions, it takes at most $2\text{Rad}(C) + 2\text{Rad}(C') + 1$ time to simulate all the send instructions of a given pulse. Thus, by using at most $|\mathcal{C}|$ master agents plus $2\,|E_{\mathcal{C}}|$ workers, each pulse can be simulated in $O(\text{Rad}(\mathcal{C}))$ time. As for the cost complexity, one can see that simulating each pulse requires the workers to deliver the messages of the original algorithm and to synchronize with neighboring masters. Thus, the cost complexity is $O(\text{Rad}(\mathcal{C})\cdot |E_{\mathcal{C}}|)$ per pulse.

After finishing the simulation of the pulses of algorithm MsA, the nodes must be marked with their final states (Step 2.c). This step has also a time and a cost complexity. In the following, we denote by $\tau(\mathcal{C})$ the maximum time needed for each master agent (and its workers) to mark the whiteboards of nodes in its region with their final states. We also denote by $\eta(\mathcal{C})$ the cost complexity of marking the nodes with their final states, that is the total number of moves that all agents make in order to mark the whiteboards of all nodes in the graph. The analysis of $\tau(\mathcal{C})$ and $\eta(\mathcal{C})$ is delayed to section 5.

Lemma 1. *Assuming* $\mathbf{H}(\mathcal{C})$*, any synchronous message passing algorithm* MsA *can be simulated by an asynchronous mobile agent algorithm* AgA *such that:*

$\mathcal{ILV}$(AgA)	$\text{Rad}(\mathcal{C})$
Size(AgA)	$O(\|\mathcal{C}\| + \|E_{\mathcal{C}}\|)$
Time(AgA)	$O(\text{Rad}(\mathcal{C}) \cdot \text{Time(MsA)}) + \tau(\mathcal{C})$
Cost(AgA)	$O\left(\text{Rad}(\mathcal{C}) \cdot \|E_{\mathcal{C}}\| \cdot \text{Time(MsA)}\right) + \eta(\mathcal{C})$

Remark 1. Since we have assumed the local termination detection property in our message passing model, a master can detect when to execute Step 2. Nevertheless, suppose that we have a message passing algorithm that does not detect the local termination, e.g., at least one node can not say whether it has finished the computations. Then, we can modify our simulation scheme so that, *at each round* a master and its workers mark the nodes in their region with the labels computed by the message passing algorithm, instead of waiting until the end of the simulation. Thus, at each round, the labeling of nodes will be the same as in the message passing algorithm. Of course, this is achieved at the price of increasing the time and the cost complexities. Roughly speaking, the overhead is order of $\tau(\mathcal{C})$ (time) and $\eta(\mathcal{C})$ (cost) at each round. This can be shown to be negligible compared to the complexity of the simulation itself.

5 Generic Trade-Offs

Sparse decompositions (see, e.g., [5]) allow to represent a graph by a set of clusters with a good compromise between the radius of the clusters and the number of inter-cluster edges. Many distributed constructions of sparse decompositions exist in the literature. In this paper, we use an adaptation of a distributed algorithm that appeared in [13] and obtain the following key Lemma. We remark that the sparseness of the partition stated in our lemma is better than the one of [13] by a $\log k$ factor. Due to lack of space, our distributed construction and its analysis will appear in the full version of this paper (see also [12]). We also remark that partitions with slightly better properties exists (see [5]) but constructing them *distributively* is time consuming.

Lemma 2. *In the message passing model, there exists a deterministic (resp. randomized) distributed algorithm that given an n-node graph G and an integer parameter k, constructs a partition structure $\mathcal{C}$ such that* $\mathrm{Rad}(\mathcal{C}) = O(k^c)$ *and* $|E_{\mathcal{C}}| = O(n^{1+1/k})$ *in* $k^c \cdot n^\epsilon$ *time (resp.* $O(k^c \cdot \log n)$ *expected time) where c is a constant ($c = \log_2 5$).*

By applying Theorem 2 in the context of the sparse partition algorithm given by Lemma 2, one can derive a preprocessing mobile agent algorithm constructing the required partition. In the following, we will denote by Pre_Part such an algorithm. Bounding $\tau(\mathcal{C})$ and $\eta(\mathcal{C})$ in Lemma 1 is easy when using $O(n)$ agents. Thus, one can prove the following:

Theorem 3. *Given an integer parameter k, any graph G can be preprocessed by an asynchronous mobile agent algorithm* Pre_Part *such that after the preprocessing phase, any synchronous message passing algorithm* MsA *can be simulated by an asynchronous mobile agent algorithm* AgA *satisfying:*

$\mathcal{ILV}$(Pre_Part)	0
Size(Pre_Part)	n
Time(Pre_Part)	$k^c\Delta\ n^\epsilon$
$\mathbb{E}$ (Time(Pre_Part))	$k^c\Delta\ \log n$

$\mathcal{ILV}$(AgA)	$O(k^c)$
Size(AgA)	$O(n^{1+1/k})$
Time(AgA)	$O(k^c \cdot \mathrm{Time}(\text{MsA}))$
Cost(AgA)	$O\left(k^c\ n^{1+1/k} \cdot \mathrm{Time}(\text{MsA})\right)$

Now, we want to fix the number of agents used by our simulation scheme to be a parameter $k < n$. For that purpose, we use a graph structure based on small dominating sets. A ρ-dominating set S of G is a set of nodes satisfying: $\forall v \in V$, $\exists s \in S$ such that $d_G(v,s) \leqslant \rho$. Given a ρ-dominating set of G, a partition of G can be obtained by clustering the nodes of the graph around the nodes of the dominating set. Based on [27], we have:

Lemma 3 ([27]). *In the message passing model, there exists a deterministic distributed algorithm that given an n-node graph G and a parameter ρ $(< n)$ constructs a partition structure $\mathcal{C}$ such that* $\mathrm{Rad}(\mathcal{C}) = O(\rho)$ *and* $|\mathcal{C}| = O(n/\rho)$ *in* $O(\rho \cdot \log^* n)$ *time.*

Applying Theorem 2 to the partition given by Lemma 3 allows us to derive a preprocessing algorithm called Pre_Dom to construct the required partition. However, coupling algorithm Pre_Dom with Lemma 1 is less straightforward. First, we choose $\rho = n/\sqrt{k}$ to obtain a number of at most k agents. Then, we apply Theorem 1 inside each region of the partition to efficiently bound $\tau(\mathcal{C})$ and $\eta(\mathcal{C})$ in Lemma 1 because using a trivial traversal to mark the final states of nodes will dominate the complexity of the simulation itself. More precisely, we use $\sqrt{k}$ masters with $\sqrt{k}$ workers each, and a "balanced tree" structure inside each $O(n/\sqrt{k})$-radius region. Thus, we can prove:

Theorem 4. *Given an integer parameter $k < n$, any graph G can be preprocessed by an asynchronous mobile agent algorithm* Pre_Dom *such that after the preprocessing phase, any synchronous message passing algorithm* MsA *can be simulated by an asynchronous mobile agent algorithm* AgA *satisfying:*

$\mathcal{ILV}$(Pre_Dom)	0
Size(Pre_Dom)	n
Time(Pre_Dom)	$O\left(\Delta \cdot n/\sqrt{k} \cdot \log^* n\right)$

$\mathcal{ILV}$(AgA)	$O\left(n/\sqrt{k}\right)$
Size(AgA)	k
Time(AgA)	$O\left(n/\sqrt{k} \cdot \text{Time(MsA)}\right)$
Cost(AgA)	$O\left(n\ \sqrt{k} \cdot \text{Time(MsA)}\right)$

6 Efficient Distributed Initialization of Agents

In the previous section, we did not care about how the agents are initialized distributively after the preprocessing phase, i.e., how the masters and the workers are initialized distributively. In the following, we will argue that the complexity of initializing the agents is negligible compared to the preprocessing itself.

Suppose that initially we have one agent per node. After the preprocessing of Theorem 3 or 4, each mobile agent can say whether it is a center of a region or not. Thus, it is easy to initialize the master agents: Each mobile agent who identifies its homebase as the center of a region becomes master in that region. A trivial solution to initialize the workers would be to allow a master to create as many workers as needed. In the following, we will *not* make such an assumption.

A first idea to initialize the workers is to let an agent whose homebase is not a center of a region be a worker in its region. This idea will work in the case of Theorem 4. In fact, the ρ-dominating set algorithm of [27] also allows to partition the graph into regions having at least n/ρ nodes (This non trivial property is proved in [27]). Hence, if each agent in a non-center node becomes worker and joins his master (at distance $\rho = n/\sqrt{k}$), then each master will have the required number of workers. Hence, the complexity of initializing the agents is negligible compared to the preprocessing of Theorem 4.

The same idea will not work for Theorem 3 since the number of required workers could be $\Theta(n^{1+1/k})$. One could think that if initially there are $\Theta(n^{1/k})$ mobile agents per node, then the previous initialization technique would hold. This is not true since the maximum degree of the graph $G_{\mathcal{C}}$, where $\mathcal{C}$ is the partition constructed in Lemma 2, is not bounded by $O(n^{1/k})$. Hence, the question is: assuming that we have $\Theta(n^{1/k})$ agent per node, how can we initialize the workers needed for Theorem 3 efficiently?

We use the algorithm of Lemma 2 to answer the previous question. The general idea is to construct the partition of Lemma 2 and at the same time to choose some *preferred edges* connecting the clusters, which will help to initialize the workers. Assuming that we have $\Theta(n^{1/k})$ agents per node and using the set of these preferred edges, we are able to show that the workers are initialized in $k^{O(1)}$ time (and cost) after the preprocessing of Theorem 3, which is negligible (see [12] for a complete proof).

7 Fundamental Applications

7.1 On Computing Any Labeling of a Graph

We remark that any labeling function of G can be computed by a message passing algorithm in $O(D)$ time in the message passing model (by collecting the topology of G at one node, computing the labeling locally and broadcasting the result). Moreover, if a task can be computed by a deterministic message passing algorithm in t time, then all the information used by each node is in its t-neighborhood. Thus, if we have a mobile agent per node, and if the mobile agents know the $O(t)$-neighborhood of their homebases, then each mobile agent can construct in $O(1)$ time a labeled copy of its $O(t)$-neighborhood where the labels assigned to nodes and edges of that neighborhood copy correspond to the final labeling computed by the message passing algorithm.

For $k = \log n$, we have $n^{1/k} = O(1)$. Hence, we can derive an efficient algorithm for computing any labeling of a graph using $O(1)$ agents per node. Actually, the following theorem provides a stronger result that holds when *initially there is exactly one agent per node*:

Theorem 5. *Given any graph G with one mobile agent assigned to each node, any labeling function of G can be computed by a deterministic (resp. randomized) asynchronous mobile agent algorithm* Det_AgA *(resp.* Rand_AgA*) satisfying:*

$\mathcal{ILV}$(Det_AgA)	0	n^{ϵ}
Time(Det_AgA)	$n^{\epsilon} \cdot \Delta + \widetilde{O}(D)$	$\widetilde{O}(D)$

$\mathcal{ILV}$(Rand_AgA)	0
$\mathbb{E}$(Time(Rand_AgA))	$\widetilde{O}(\Delta + D)$

Remark 2. Note that for any graph G, there exists a labeling function of G that can not be computed by n mobile agents having no initial local views in time better than $\Omega(D)$ or $\Omega(m/n)$.

7.2 On Computing with Small Memory Agents

In previous sections, we have assumed that the agents are equipped with unlimited memory in our simulation scheme. Actually, only the workers need to have enough internal memory. In fact, a worker has to **(i)** store the concatenated messages (and the synchronization messages), **(ii)** store the route used to go from a center node to another one, and **(iii)** store the information needed to mark nodes with their final labeling. Thus, by denoting b the maximum size (in bits) of a message of the simulated algorithm and ℓ the maximum size (in bits) of the output labels, we can show that for $k = \log n$ in Theorem 3 the required memory size per agent is at most $\widetilde{O}(\max\{mb, \ell\})$ bits. Thus, assuming that ℓ is small compared to mb, the memory size needed by a worker is dominated by the size of the concatenated messages. Nevertheless, we remark that if a node in the original message passing algorithm sends the *same* message to all its neighbors then a worker can store only one message for that node in its internal memory, i.e., it does not need to store the same message many times. More generally, the memory size can be drastically improved for algorithms having particular behaviors. For instance, consider a message passing algorithm such that, at each round, a node **(i)** sends the same message M_1 to a bounded number of neighbors and **(ii)** sends the same message M_2 to the other neighbors. Then, it is not difficult to see that we can modify our concatenation mechanism so that the memory size needed by a worker is at most $\widetilde{O}(n \cdot b)$ bits. In particular, we can prove:

Theorem 6. *For any n-node graph G, a BFS tree with respect to a given node can be computed in $\widetilde{O}(D)$ time by using n mobile agents with n^{ϵ} initial local view and $\widetilde{O}(n)$ memory bits.*

Theorem 7. *For any n-node graph G, a leader can be computed in $\widetilde{O}(D)$ time by using n mobile agents with n^{ϵ} initial local view and $\log^{O(1)} n$ memory bits.*

References

1. Afek, Y., Ricklin, M.: Sparser: a paradigm for running distributed algorithms. Journal of Algorithms 14, 316–328 (1993)
2. Awerbuch, B.: Complexity of network synchronization. Journal of the ACM 32, 804–823 (1985)
3. Awerbuch, B., Goldberg, A.V., Luby, M., Poltkin, S.A.: Network decomposition and locality in distributed computation. In: 30th Symposium on Foundations of Computer Science (FOCS), pp. 364–369 (1989)
4. Awerbuch, B., Goldreich, O., Vainish, R., Peleg, D.: A trade-off between information and communication in broadcast protocols. Journal of the ACM 37, 238–256 (1990)
5. Awerbuch, B., Peleg, D.: Sparse partitions. In: 31st Symposium on Foundations of Computer Science (FOCS), pp. 503–513 (1990)
6. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Can we elect if we cannot compare? In: 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 324–332 (2003)
7. Barriere, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Rendezvous and Election of Mobile Agents: Impact of Sense of Direction. Theory of Computing Systems (ToCS) 40, 143–162 (2007)
8. Chalopin, J., Godard, E., Métivier, Y., Ossamy, R.: Mobile agent algorithms versus message passing algorithms. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 187–201. Springer, Heidelberg (2006)
9. Cooper, C., Klasing, R., Radzik, T.: Searching for black-hole faults in a network using multiple agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 320–332. Springer, Heidelberg (2006)
10. Das, S., Flocchini, P., Nayak, A., Santoro, N.: Distributed exploration of an unknown graph. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 99–114. Springer, Heidelberg (2005)
11. Das, S., Flocchini, P., Santoro, N., Yamashita, M.: Fault-tolerant simulation of message-passing algorithms by mobile agents. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 289–303. Springer, Heidelberg (2007)

12. Derbel, B.: Local maps: New insights into mobile agent algorithms, Tech. Report RR-6511, INRIA - LIFL - USTL (April 2008), http://hal.inria.fr/
13. Derbel, B., Gavoille, C.: Fast deterministic distributed algorithms for sparse spanners. In: Flocchini, P., sieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 100–114. Springer, Heidelberg (2006)
14. Flocchini, P., Nayak, A., Schulz, A.: Decontamination of arbitrary networks using a team of mobile agents with limited visibility. In: 6th IEEE/ACIS International Conference on Computer and Information Science, pp. 469–474 (2007)
15. Flocchini, P., Santoro, N.: Distributed security algorithms by mobile agents. In: 8th Conference on Distributed Computing and Networking, pp. 1–14 (2006)
16. Fraigniaud, P., Gavoille, C., Ilcinkas, D., Pelc, A.: Distributed computing with advice: Information sensitivity of graph coloring. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 231–242. Springer, Heidelberg (2007)
17. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Oracle size: a new measure of difficulty for communication tasks. In: 25th Symposium on Principles of Distributed Computing (PODC), pp. 179–187 (2006)
18. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Tree exploration with an oracle. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 24–37. Springer, Heidelberg (2006)
19. Fraigniaud, P., Korman, A., Lebhar, E.: Local mst computation with short advice. In: 19th Symp. on Parallel Algo. and Arch (SPAA), pp. 154–160 (2007)
20. Fukuda, M., Bic, L.F., Dillencourt, M.B., Cahill, J.M.: Messages versus messengers in distributed programming. Journal of Parallel and Distributed Computing 57, 188–211 (1999)
21. Isler, V., Kannan, S., Khanna, S.: Randomized pursuit-evasion with limited visibility. In: 15th Symp. on Discrete algorithms (SODA), pp. 1053–1063 (2004)
22. Kazazakis, G.D., Argyros, A.A.: Fast positioning of limited-visibility guards for the inspection of 2d workspaces. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2843–2848 (2002)
23. Korman, A.: General compact labeling schemes for dynamic trees. Distributed Computing 20, 179–193 (2007)
24. Korman, A., Kutten, S.: Labeling schemes with queries. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 109–123. Springer, Heidelberg (2007)
25. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. In: 24th Symp. on Principles of distributed computing (PODC), pp. 9–18 (2005)
26. Korman, A., Peleg, D., Rodeh, Y.: Labeling schemes for dynamic tree networks. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 76–87. Springer, Heidelberg (2002)
27. Kutten, S., Peleg, D.: Fast distributed construction of small k-dominating sets and applications. Journal of Algorithms 28, 40–66 (1998)
28. Linial, N.: Locality in distributed graphs algorithms. SIAM Journal on Computing 21, 193–201 (1992)
29. Moran, S., Snir, S.: Simple and efficient network decomposition and synchronization. Theoretical Computer Science 243, 217–241 (2000)
30. Naor, M., Stockmeyer, L.: What can be computed locally? SIAM Journal on Computing 24, 1259–1277 (1995)
31. Nisse, N., Soguet, D.: Graph searching with advice. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 51–65. Springer, Heidelberg (2007)

32. Pan, L., Bic, L.F., Dillencourt, M.B., Huseynov, J.J., Lai, M.K.: Distributed parallel computing using navigational programming. Journal of Parallel Programming 32, 1–37 (2004)
33. Panconesi, A., Srinivasan, A.: On the complexity of distributed network decomposition. J. Algo. 20, 356–374 (1996)
34. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM Monographs on Discrete Mathematics and Applications (2000)
35. Souissi, S., Défago, X., Yamashita, M.: Using eventually consistent compasses to gather oblivious mobile robots with limited visibility. In: 8^{th} Symposium on Stabilization, Safety, and Security of Distributed Systems, pp. 484–500 (2006)

r^3: Resilient Random Regular Graphs

S. Dimitrov[1,⋆], P. Krishnan[2], C. Mallows[2], J. Meloche[2], and S. Yajnik[2]

[1] Dept. of Ind. and Op. Eng., Univ. of Michigan, Ann Arbor, MI 48109
sdimitro@umich.edu
[2] Avaya Labs, 233 Mt. Airy Rd., Basking Ridge, NJ 07920
{pk,colinm,jmeloche,shalini}@avaya.com

Abstract. Efficiently building and maintaining resilient regular graphs is important for many applications. Such graphs must be easy to build and maintain in the presence of node additions and deletions. They must also have high resilience (connectivity). Typically, algorithms use offline techniques to build regular graphs with strict bounds on resilience and such techniques are not designed to maintain these properties in the presence of online additions, deletions and failures. On the other hand, random regular graphs are easy to construct and maintain, and provide good properties with high probability, but without strict guarantees. In this paper, we introduce a new class of graphs that we call r^3 (*resilient random regular*) graphs and present a technique to create and maintain r^3 graphs. The r^3 graphs meld the desirable properties of random regular graphs and regular graphs with strict structural properties: they are efficient to create and maintain, and additionally, are highly connected (i.e., $1 + d/2$-node and d-edge connected in the worst case). We present the graph building and maintenance techniques, present proofs for graph connectedness, and various properties of r^3 graphs. We believe that r^3 graphs will be useful in many communication applications.

1 Introduction

Regular graphs [1], i.e., graphs with fixed degree at each node, have been studied as candidates in several computing and networking scenarios. Examples include overlay, multicast and peer-to-peer network design [2,3] and optical networks [4]. In most problems in these domains, the graph captures the topology of the network and this topology changes as nodes and edges arrive and leave the network. Building and maintaining such *evolving graphs* [5,6] efficiently is currently of great research interest. Additionally, in most system contexts, it is highly desirable that the graphs be resilient to node and edge failures. In this paper, we study the problem of building regular graphs that are *provably highly resilient*. In particular, we desire graph connectivity even with several node and edge failures.

A number of algorithms presented in the literature use offline techniques to construct d regular graphs (i.e., regular graphs of degree d at each node) with guaranteed bounds on resilience [7,8,9,10,11]. All these techniques are not

[⋆] Portions of this work were done when the author was visiting Avaya Labs Research.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 137–151, 2008.

designed to maintain the required properties in the presence of joins and leaves of the graph nodes and edges, and are not suitable for evolving graphs. In particular, using these techniques for evolving graphs incurs large number of edge cuts and rejoins for each node addition or deletion to maintain connectivity properties. Additionally, some of these techniques [9,10] require a large number of computations to explore the solution space and provide strict bounds on the resilience.

Randomized algorithms can be effectively used to solve problems very efficiently while providing good guarantees either in the average case, or with provably high probability [12,13]. Random graphs are typically good candidates for distributed design. In particular, Pandurangan et al [3] present a randomized graph building scheme for low diameter peer-to-peer networks with a bounded degree. However, their scheme focuses on building low-diameter connected graphs and not on guarantees on the resilience of the resulting network to node and edge failures, which is the main focus of our paper.

Random regular graphs [14] are fixed degree graphs built using a randomized approach. Such random d regular graphs have interesting properties like d-connectivity with very high probability (specifically, asymptotically almost surely), but not in the worst case. As an example, the Araneola multicast overlays [2] are built using random regular graphs and rely on the connectedness of random regular graphs to ensure that the multicast overlay is resilient with a high probability. There are, however, no strict guarantees on the resilience of such graphs.

In this paper, we concentrate on the problem of constructing resilient regular graphs. The main contribution of our paper is a simple and efficient construction technique for regular graphs. We call the resulting graphs r^3 (*resilient random regular*) graphs. The r^3 graphs are easy to build and maintain in the presence of node arrivals and departures. The graph building is done using *constrained* random choices as each node is added to the graph, providing efficiency. We show that the resulting graph is $1 + d/2$-*node and* d *edge-connected* resulting in very high guaranteed resilience. Intuitively, the algorithm melds the best properties of random graph building with structured maintenance to achieve efficiency as well as guarantees on resilience. We also describe other interesting properties of r^3 graphs that relate to their efficient construction and maintenance.

The rest of the paper is organized as follows. Section 2 explains the algorithm for building r^3 graphs. Section 3 proves the guarantees on the connectivity properties of these graphs in the presence of edge and node removals. The theoretical proofs are also supported by simulation results given in the latter half of the section. Section 4 gives the algorithm for deletion and introduces the concept of "natal nodes" in the graph. Section 5 describes some properties of natal nodes in the r^3 graphs and supports them with simulations and numerical calculations. Section 6 concludes with a discussion of our results and some open problems.

2 Construction

We start with some preliminaries. The degree of a node v in a graph is the number of edges incident on node v. A d regular graph is a graph in which every node has the same degree d. If a d regular graph g has $|g| = n$ nodes, the number of edges must be $nd/2$, so that at least one of n, d must be even. The theory is simpler when d is even, so let us assume that d is even, and $d \geq 4$ (the case $d = 2$ is trivial). A graph is *connected* if there is a path between every pair of nodes in the graph. A graph is said to be k-node (k-edge) connected if there does not exist a set of $k - 1$ nodes (edges) whose removal disconnects the graph. The graphs we consider in this paper are labeled. However, we do not distinguish among different labellings if the graphs are isomorphic.

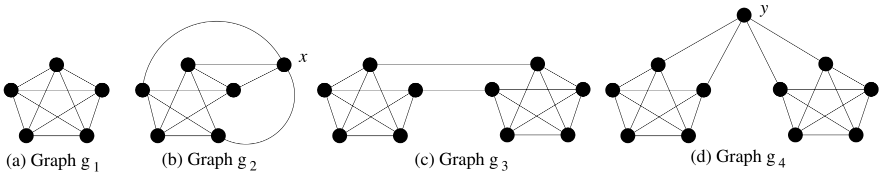
For the purposes of this paper, *resilience* of a graph is a measure of how large k is, given that the graph is k-node connected. The typical cost in constructing and maintaining graphs includes communication costs to discover appropriate nodes/edges, and the cost for breaking and creating edges. Since communication cost is very specific to the system implementation, in this paper, we compute *efficiency* based on the graph operations (i.e., node and edge creation/deletion). However, we do provide information on structural properties of the graphs as they relate to communication costs. In Appendix B we provide more information about key algorithms presented in the paper, including discussion of some implementation issues.

2.1 r^2 and r^3 Graphs

Our starting point is the following algorithm for growing regular graphs:

Algorithm A**.** (adding a node). Given a d-regular graph g with n nodes, choose a set of d vertices (v_1, v_2, ..., v_d) such that each of the edges (v_{2i-1}, v_{2i}) is present in g. Delete these edges, and insert new edges connecting each of (v_1, v_2, ..., v_d) to a new node. The result is a regular graph with $n + 1$ nodes. Figure 1 illustrates algorithm A: Graph g_2 is obtained by adding node x to graph g_1, and graph g_4 is obtained by adding node y to graph g_3. Note that algorithm A incurs $O(d)$ edge deletions and additions for each node addition; specifically, a node addition involves $d/2$ edge deletions and d edge additions.

Starting with an arbitrary d-regular graph g_0, repeated application of algorithm A will generate an infinite set of d-regular graphs that we name $G(g_0)$. Some graphs may be in more than one such set. An *atomic* graph g is a graph



(a) Graph g_1 (b) Graph g_2 (c) Graph g_3 (d) Graph g_4

Fig. 1. Some 4-regular graphs

that is in only one set, namely $G(g)$. The complete graph K_{d+1} on $d+1$ nodes is atomic. In Figure 1, graphs g_1 and g_3 are atomic, $g_2 \in G(g_1)$ and $g_4 \in G(g_3)$. We look more closely at atomic graphs in Section 2.2.

Our main interest is in the set $G(K_{d+1})$, which we call r^2 (*resilient regular*) graphs and denote by G_d. For the graphs in Figure 1, graphs $g_1, g_2 \in G_4$, and $g_3, g_4 \notin G_4$. Note that algorithm A *does not* specify how the edges to delete are chosen when a new node joins the graph. We denote by r^3 (*resilient random regular*) or *random* r^2 those graphs that are generated by algorithm A when the edges to delete are chosen randomly.

There are several methods for random edge selection for algorithm A. For example, one method is an iterative procedure that keeps a list of all *eligible* edges (i.e., edges that are node-disjoint with currently chosen edges), chooses an edge randomly from this list, and removes all edges that become ineligible because of this choice. We use this random selection procedure for our simulation and probabilistic results presented in this paper. Clearly, the randomization can be done in several other ways. Two other possibilities include (a) iteratively selecting an eligible node at random followed by an eligible edge incident at this node, or (b) choosing a random $d/2$ matching from all possible matchings. It is easy to verify that the edge selection probabilities under the three methods outlined above are not always the same. While adjustments can be made [15] to ensure that the edge selection probabilities are the same, the joint edge selection probabilities may vary. Furthermore, some of these random procedures can be more easily approximated with a distributed approach than others. Computation for small n, d suggests that none of these randomization rules make the n-node elements of G_d equally likely. Also, if nodes are repeatedly added using algorithm A and deleted (using the algorithm for node deletion presented in Section 4), the probability distribution over the n-node elements of G_d does not tend to the uniform distribution.

Our worst case connection properties shown in Section 3 are for *all* r^2 graphs (and hence, for r^3 graphs also). Our simulation results in this paper are for r^3 graphs.

2.2 Characterizing Atomic Graphs

While atomic graphs are not central to the main discussion in this paper (namely, efficient construction and resiliency of r^2 and r^3 graphs), understanding atomic graphs is useful in understanding the construction of such graphs. We discuss atomic graphs here.

Lemma 1. *For each d, there is an infinite number of atomic graphs.*

Proof. For $n = k(d+1)$ ($k \geq 2$) we can construct an atomic graph with n vertices by taking a k-cycle and replacing each node by a copy of K_{d+1} with one edge removed.

For any graph g, let g' denote the complementary graph. Graph g' has edges exactly where g does not. If g is regular with degree d, g' is regular with degree

$d' = n - 1 - d$. In the following discussions, d is even and $d \geq 4$. We show the following result; the proof appears in Appendix A.

Lemma 2. *For any regular graph g with $n \geq d + 4$ nodes, if g' is bipartite (so n is necessarily even), then g is an atomic graph.*

3 Connectivity

A major emphasis in this paper is efficiently building regular graphs that are highly connected. Clearly, d regular graphs can at best be d-node connected. However, some d regular graphs are not even 2-node connected (e.g., graph g_4 in Figure 1(d)).

An important property for our application is that the r^2 graphs that are generated by algorithm A (and in particular, the r^3 graphs) have good connectivity properties in the worst-case. Let $e(g)$ denote the edge-connectivity and $n(g)$ denote the node connectivity of a graph g. Our two main results of this section are:

Theorem. (Edge Connectivity: Theorem 1) For $g \in G(g_0)$ we have $e(g) \geq e(g_0)$. Hence, for $g \in G_d$, $e(g) \geq e(K_{d+1}) = d$.

Theorem. (Node Connectivity: Theorem 2) For $g \in G_d$, $n(g) \geq 1 + d/2$.

We now prove these results in the rest of this section.

If S and T are disjoint sets of nodes in a graph g, we use the notation $c_g(S, T)$ to denote the number of edges in g between S and T, which we refer to as *cross edges*. We define a *lineage* of g as a sequence of graphs g_0, …, g_n starting from the atomic graph g_0 and ending with $g_n = g$ that results from a construction of g. We first show that the number of cross edges between two sets of nodes that partition a graph never decreases as nodes are added to the graph.

Lemma 3. *If g is formed by adding one node v to $\hat{g}$ and if S and T form a partition of g then $c_g(S, T) \geq c_{\hat{g}}(S \cap \hat{g}, T \cap \hat{g})$.*

Proof. An edge $e \in \hat{g}$ going from $s \in S \cap \hat{g}$ to $t \in T \cap \hat{g}$ is either left unchanged by the construction of g or is replaced by two edges, one going from s to v and the other from v to t, exactly one of which is a cross edge between S and T. Thus, the construction of g cannot decrease the number of cross edges between S and T.

Lemma 4. *If $g \in G_d$ can be partitioned as $S + T$, then $c_g(S, T) \geq x(1 + d - x)$, where $x = \min\{d/2, |S|, |T|\}$.*

Proof. Let $g_0 = K_{d+1}$, …, $g_n = g$ be a lineage of g. Starting with g_n, we go back through the ancestors until one is found such that $\min(|S \cap g_i|, |T \cap g_i|) = x$. There will be such an i because $|g_0| = d + 1$. At this point, at least one of the two partition elements $S \cap g_i$ or $T \cap g_i$ has exactly x elements. Because that component can have at most $x(x-1)$ internal connections it must have $xd - x(x-1) = x(1+d-x)$ external ones. Using Lemma 3 we can write

$$\begin{aligned} c_g(S,T) &= c_{g_n}(S \cap g_n, T \cap g_n) \\ &\geq c_{g_{n-1}}(S \cap g_{n-1}, T \cap g_{n-1}) \\ &\quad\vdots \\ &\geq c_{g_i}(S \cap g_i, T \cap g_i) = x(1+d-x). \end{aligned}$$

Theorem 1. *If $g \in G(g_0)$ then $e(g) \geq e(g_0)$. Hence, for $g \in G_d$, $e(g) \geq e(K_{d+1}) = d$.*

Proof. The first part of the theorem follows from Lemma 3. For any partition $g = S + T$, $g \in G_d$, with $|g| > d$, $|S| > 0$ and $|T| > 0$, $x = \min\{d/2, |S|, |T|\} \geq 1$. Because $x(1+d-x)$ is minimized at $x = 1$ over the range $\{1, \ldots, d/2\}$, Lemma 4 implies that $c_g(S,T) \geq 1(1+d-1) = d$.

We now prove one of the main results of our paper: that graphs in G_d are highly node-connected; i.e., removing even $d/2$ nodes in such a graph maintains connectivity.

Theorem 2. *If $g \in G_d$ then g is at least $1 + d/2$ node connected.*

Proof. The result is trivial if $|g| = d+1$. Assume $|g| > d+1$. Let M be a cut set of size n that could disconnect the remaining nodes: There is a a partition $g = S + M + T$ such that $c_g(S,T) = 0$, $|S| > 0$, $|T| > 0$ and $|M| = n$. Let the nodes in the cut set be a_1, …, a_n. Given a partition $M = M_S + M_T$, we can define

$$u_i = \begin{cases} \text{connections between } a_i \text{ and } T + M_T \text{ if } a_i \in M_S \\ \text{connections between } a_i \text{ and } S + M_S \text{ if } a_i \in M_T \end{cases}$$

and $u = \max_{1 \leq i \leq n} u_i$. We now construct a partition $M = M_S + M_T$ for which $u \leq d/2$. We start with $M_S = M$ and $M_T = \phi$ and iteratively move the nodes a_i (illustrated in Figure 2) from M_S to M_T whenever $u_i > d/2$. Moving node a_i from M_S to M_T will reduce u_i to something no larger than $d/2$. In addition, u_j

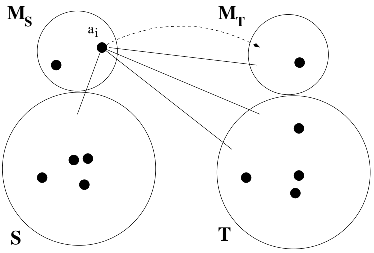


Fig. 2. If $u_i > d/2 = 2$, a_i moves from M_S to M_T

for $a_j \in M_T$ will not increase. The iteration stops when $u_i \leq d/2$ for all $a_i \in M_S$ and at that point, $u = \max_{1 \leq i \leq n} u_i \leq d/2$.

Consider now the edges out of any of the nodes a_i for which $u = u_i$ and assume without loss of generality that $a_i \in M_S$. Because $u = u_i$, $T + M_T$ must have at least u nodes. On the other hand, the remaining $d - u \geq u$ edges out of a_i must connect to nodes in $S + M_S$ and we conclude that both $S + M_S$ and $T + M_T$ must have at least u nodes.

Clearly, the number of cross edges, $\sum_{1 \leq i \leq n} u_i \leq nu$. Using Lemma 4 along with the observation that $f(u) = u(1 + d - u)$ is an increasing function for $u \leq d/2$, we get

$$c_g(S + M_S, T + M_T) \geq u(1 + d - u).$$

Hence,

$$nu \geq \sum_{1 \leq i \leq n} u_i \geq c_g(S + M_S, T + M_T) \geq u(1 + d - u),$$

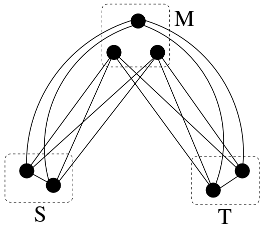
which implies $n \geq 1 + d - u > d/2$.

3.1 Tightness of Theorems 1 and 2

We now show that the bounds of Theorems 1 and 2 are tight. As far as edge connectivity is concerned, it is clear that $g \in G_d$ is not $d + 1$ edge connected because the set of d edges originating at a node constitutes a cut set. As for node connectivity, consider three sets S, T and M, where S and T each have $d/2$ nodes and M has $1 + d/2$ nodes. The graph g has the node set $S \cup T \cup M$, and the adjacency matrix as in Figure 3a, where $\mathbf{1}$ is the matrix with all 1's, $\mathbf{I}$ is the identity matrix, and $\mathbf{0}$ is the matrix with all 0's (the dimensions of the matrices are obvious from the context). It is easy to verify that graph g can be generated from K_{d+1} using algorithm A. The rows and the columns of this adjacency matrix are easily seen to add up to d and the set of nodes M is critical to the connectivity between S and T. The graph corresponding to this construction with $d = 4$ is displayed in Figure 3b.

$$\begin{array}{c|ccc} & S & M & T \\ \hline S & \mathbf{1\text{-}I} & \mathbf{1} & \mathbf{0} \\ M & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ T & \mathbf{0} & \mathbf{1} & \mathbf{1\text{-}I} \end{array}$$

(a) Adjacency Matrix



(b) Graph for $d = 4$

Fig. 3. Adjacency matrix for a $(1 + d/2)$ node connected graph in G_d, and a graph corresponding to $d = 4$

3.2 Simulation Results

As discussed earlier, our construction is particularly attractive because it guarantees $1 + d/2$ node-connectivity and in addition, the connectivity property is independent of the random selections. Note that a regular graph picked uniformly at random from the set of all regular graphs is known to have desirable connectivity properties with high probability a.a.s. [14], but not in the worst case. In particular, a regular graph picked at random is likely to be d-node connected provided that the selection was made with uniform probabilities. Whether some constrained random selection scheme (like algorithm A) enjoys the same asymptotic properties is not obvious.

To investigate how r^3 graphs perform in practice, we conducted a simulation study. As indicated in Section 2.1, for our simulations, node addition was performed by deleting edges selected uniformly at random. The simulation consisted of generating independent r^3 graphs, with degrees ranging from $d \leq 50$ and sizes ranging from $n = d + 2$ to $4d$. For each graph, we computed the node connectivity using network flow techniques with MAXFLOW [16]. In most of the cases, the resulting graph is d-connected and in a few cases, it is slightly deficient with a connectivity of $d - 1$. In our simulations, we never observed a connectivity of $d - 2$ or lower, although it is clearly possible as we have shown in Section 3.1. Most of the deficiencies we observed were for graphs of size $d + 2$. Less than 1% of graphs of size greater than $d + 5$ were observed to be deficient.

4 Deletion

The connectivity properties of the graph make the graph resilient to transient node and edge failures, i.e., even with a set of nodes and edges failing, the graph still remains connected. However, if a node or an edge leaves the graph permanently, the graph needs to be repaired, such that it retains its connectivity properties. The following subsections discuss algorithms for graceful node and edge deletions. These algorithms ensure that the graph resiliency properties are retained in the presence of interleaved additions and deletions.

4.1 Node Deletion and Natal Nodes

One can imagine the following simple deletion algorithm that is an inverse of algorithm A.

Algorithm D': A node v_0 of a graph g in G_d can be deleted if the neighbors of v_0 can be written in order as $(v_1, v_2, \ldots, v_d)$ with each edge (v_{2i-1}, v_{2i}) absent in g. The node v_0 and the edges (v_i, v_0) are deleted, and edges (v_{2i-1}, v_{2i}) are inserted.

For atomic graphs, no node can be deleted. Using algorithm D', for some graphs, some nodes can be deleted in more than one way. For example, in the graph M9.4.10 (i.e., the tenth in Meringer's list of $n = 9$, $d = 4$ graphs [17] shown in Figure 4), which is in G_4, the 7-th node can be deleted in two ways. One way of deletion gives M8.4.4 which is in G_4, while the other gives M8.4.2 which is not

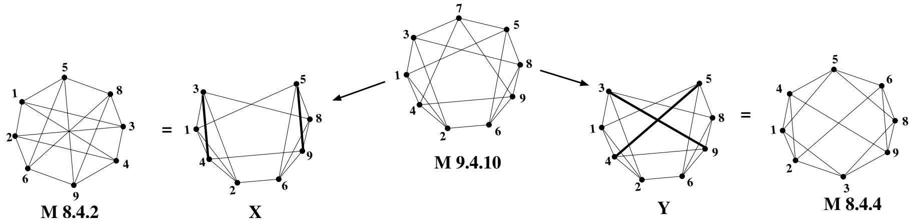


Fig. 4. Two ways of deleting node 7 from graph M9.4.10 by Algorithm D'

in G_4 (in fact, it is atomic). The bold lines in Figure 4 (i.e., edges (3,4) and (5,9) in X and (3,9) and (4,5) in Y) indicate the newly added edges.

Recall that our node connectivity results are valid only for graphs in G_d. To preserve the desired connectivity properties, we need to allow only deletions that exactly reverse the effect of previous additions. To achieve this, we describe our node deletion algorithm D below.

Algorithm D**.** We require that whenever a node is added (by algorithm A), a record is kept of the edges that were deleted to insert this node, so that these edges can be reinstated when this node is deleted. A node may be deleted only when it is still connected to the nodes it was first connected to (when it joined the graph). We term such nodes *natal*. Note that two adjacent nodes cannot both be natal. If a *non-natal* node needs to be deleted, it can first swap neighbors in the graph with a natal node and then remove itself from the graph.

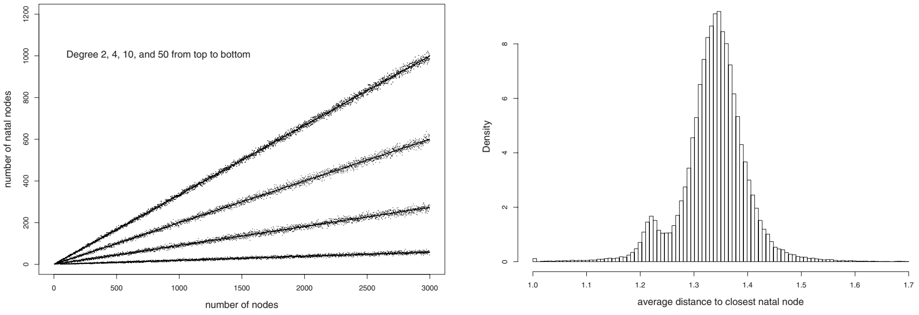
Clearly, deleting a node requires $O(d)$ edge removals and additions; specifically, deleting a natal node requires d edge removals and $d/2$ edge additions, and deleting a non-natal node requires $2d$ edge removals and $3d/2$ edge additions. From a graph point of view, the interesting metrics related to finding a natal node are the number of and distance to natal nodes. Natal nodes and their properties are studied in more detail in Section 5.

4.2 Edge Removal

In some network situations (e.g., overlays), a node A may change its preference on keeping another node B as its neighbor. This effectively translates to "removing" edge AB. This can be achieved conceptually by deleting a natal node C and reinserting it while ensuring that node C chooses edge AB during insertion. (In most cases, this can be efficiently implemented without going through the full deletion and reinsertion process.) A similar procedure can be used for bringing a node X "closer" to node Y by connecting them both to a natal node. The more general problem of associating costs with edges and, in particular, disallowing certain edges (associating infinite costs with edges) is a subject of future study.

5 Properties of Natal Nodes

The distribution of natal nodes in the graph is a key factor that determines the ease of implementing the deletion algorithm in a distributed way. The number



(a) Number of natals

(b) Distance to natal

Fig. 5. Number of natals and distance to the closest natal

of natal nodes in the graph and the distance of each node to a closest natal node determine the complexity of the implementation. In this section, we aim to show that in an r^3 graph, there are a large number of natal nodes in the graph and that most non-natal nodes have a natal neighbor nearby.

5.1 Simulation Results

We evaluated through simulations, the number of natal nodes and the average distance to the closest natal node. The simulation consisted of generating independent r^3 graphs, with degree d upto 50 and sizes ranging from $n = 5$ to $n = 3000$. For each graph, we counted the number of natal nodes and the average distance to the closest natal node. The averaging is performed over nodes that are not natal and accordingly, the smallest distance to the closest natal node is 1. The results of the simulations are shown in Figure 5.

The plot in Figure 5a shows the number of natal nodes as a function of the number of nodes n in the graph. The plot shows only the values corresponding to d=2, 4, 10 and 50 for clarity. The graph demonstrates the linear relationship between the number of natal nodes and n for the four values of d. The points in the plot cluster along lines that have slopes very close to $1/(d+1)$. We show below in Section 5.2 for $d = 2, 4$ that the expected number of natal nodes is exactly $n/(d+1)$. The graph in Figure 5b is a histogram of the average distance to the closest natal node for all values of n and d. When $n = d + 1$, there is exactly one natal node and all the other nodes are at distance one from it, which is the reason for the small bump over 1 in the histogram. The hump at 1.22 is caused by the graphs of degree $d = 2$. The distribution has an average of 1.34 with a standard deviation of 0.06.

5.2 Expected Number of Natal Nodes

Simulation results given in the Section 5.1 show that there are a large number of natal nodes in the graph and with a high probability a node will have a natal node in its neighbor set. The simulations can be supported by numerical results on the expected value of natal nodes for small values of d. The results for $d = 2$ and $d = 4$ are given below.

Case $d = 2$. In the case $d = 2$ there is only one atomic graph, namely K_3, and so only one family of graphs, which contains only cycles. We choose to label one of the nodes in K_3 as natal. Suppose a cycle C_n contains W_n natal nodes, and a new node is added in a random position. If the deleted edge has a natal end-point (it cannot have two) then the number of natal nodes is unchanged; otherwise it increases by 1. So we have the Markov recurrence

$$P(W_{n+1} = W_n) = \frac{2W_n}{n},$$
$$P(W_{n+1} = W_n + 1) = 1 - \frac{2W_n}{n}.$$

Computing the conditional expectation of W_{n+1} given W_n, we get the recurrence $E(W_{n+1}) = ((n-2)/n)E(W_n) + 1$. Since $W_3 = 1$, it follows that for $n \geq 3$, $E(W_n) = n/3$. A similar approach shows that for $n \geq 4$, $Var(W_n) = 2n/45$ [18].

Case $d = 4$. In this case there are some simple exact results. We consider only the family $G(K_5)$. In our discussion in this section, g_n is an n node graph with $d = 4$. The number of edges in a regular graph with n nodes is $2n$. The number of disjoint pairs of edges is $n(2n-7)$.

Lemma 5. *If a node is natal in a graph g_n, the probability that it remains natal in g_{n+1} is $(n-4)/n$.*

Proof. Suppose node i is natal. The number of disjoint pairs of edges, one of which has a node i as an end-point, is $4(2n-7)$. So the probability that node i is not affected when the next node is added is $(n-4)(2n-7)/n(2n-7) = (n-4)/n$.

We number the nodes in K_5 as 1, 2, 3, 4, 5 and we choose to label node 5 as being natal.

Theorem 3. *For all $n \geq 5$, the expected number of natal nodes in g_n is $n/5$.*

Proof. For $j = 5, 6, ..., n$, the probability that the j^{th} node is natal in g_n is

$$P_j(n) = \frac{(j-4)}{j}.\frac{(j-3)}{(j+1)}.\frac{(j-2)}{(j+2)}...\frac{(n-5)}{(n-1)} = \frac{(j-1)^{(4)}}{(n-1)^{(4)}}$$

where $a^{(h)} = a!/(a-h)!$. So the expected number of natal nodes is

$$\text{E(natal nodes)} = \sum_{j=5}^{n} P_j(n) = \frac{n^{(5)}}{5(n-1)^{(4)}} = \frac{n}{5}.$$

6 Conclusion

In this paper, we have introduced a new class of random regular graphs (r^3 graphs) that are efficient to construct and maintain in an evolutionary way and are highly resilient (connected). We have described algorithms to add and delete nodes from such graphs, while retaining resilience properties. We have shown properties of the graphs, including number and distance to specific types of nodes (natal nodes) that are important for efficiency of the node deletion algorithms. Our simulation results show that, in practice, when constructing the graphs randomly, the connectivity of r^3 graphs exceeds our proven tight bound of $d/2+1$ for node connectivity.

Some analytical open problems remain, including tight analytical bounds for the expected number and variance of natal nodes for $d > 4$. The use of r^3 graphs in a specific system context (namely, overlay networks), and its interaction with routing and multicast protocols is currently under investigation.

References

1. Harary, F.: Graph Theory. Addison-Wesley Publishing Company, Inc, Reading (1969)
2. Melamed, R., Keidar, I.: Araneola: A scalable reliable multicast system for dynamic environment. In: 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA), pp. 5–14 (September 2004)
3. Pandurangan, G., Raghavan, P., Upfal, E.: Building low-diameter peer-to-peer networks. IEEE Journal on Selected Areas in Communications 21(6), 995–1002 (2003)
4. Haque, A., Aneja, Y.P., Bandyopadhyay, S., Jaekel, A., Sengupta, A.: Some studies on the logical topology design of large multi-hop optical networks. In: Proc. of OptiComm 2001: Optical Networking and Comm, pp. 227–241 (August 2001)
5. Bui-Xuan, B., Ferreira, A., Jarry, A.: Evolving graphs and least cost journeys in dynamic networks. In: Proc. of WiOpt 2003 – Modeling and Optimization in Mobile, Ad-Hoc and Wireless Networks, Sophia Antipolis, pp. 141–150 (March 2003)
6. Gaertler, M., Wagner, D.: A hybrid model for drawing dynamic and evolving graphs. In: Algorithmic Aspects of Large and Complex Networks (2006)
7. Harary, F.: The maximum connectivity of a graph. Proc Natl Acad Sci U S A. 48(7), 1142–1146 (1962)
8. Doty, L.L.: A large class of maximally tough graphs. OR Spectrum 13(3), 147–151 (1991)
9. Hou, X., Wang, T.: An algorithm to construct k-regular k connected graphs with maximum k-diameter. Graphs and Combinatorics 19, 111–119 (2003)
10. Hou, X., Wang, T.: On generalized k-diameter of k-regular k-connected graphs. Taiwanese Journal of Mathematics 8(4), 739–745 (2004)
11. Angskun, T., Bosilca, G., Dongarra, J.: Binomial graph: A scalable and fault-tolerant logical network topology. In: The Fifth International Symposium on Parallel and Distributed Processing and Applications, pp. 471–482 (2007)
12. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
13. Mahlmann, P., Schindelhauer, C.: Peer-to-peer networks based on random transformations of connected regular undirected graphs. In: SPAA 2005: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures, pp. 155–164. ACM Press, New York (2005)

14. Wormald, N.: Models of random regular graphs. Surveys in Combinatorics, 239–298 (1999)
15. Cochran, W.: Sampling Techniques. Wiley, Chichester (1977)
16. Boykov, Y., Kolmogorov, V.: Maxflow - software for computing mincut/maxflow in a graph, http://www.adastral.ucl.ac.uk/~vladkolm/software.html
17. Meringer, M.: Regular graphs (website), http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html
18. Mallows, C., Shepp, L.: The necklace process. Journal of Applied Probabilty 45 (to appear, 2008)

A Characterizing Atomic Graphs

In this section, we present the proof of Lemma 2. Recall that g' is the complementary graph of g and d is even.

Proof. (of Lemma 2) Graph g is atomic iff for each node A, then for every way of labeling the nodes adjacent to A as B_1, B_2, ..., B_d, at least one of the edges (B_1, B_2), (B_3, B_4),... is present in g, so there is no way to remove node A. In terms of g', for each node A in g', the nodes $B_1, ..., B_d$ are those that are non-adjacent to A; if g is atomic, then for every way of labeling these nodes, at least one of the pairs $(B_1, B_2), (B_3, B_4)$... must fail to represent an edge in g'.

If g' is bipartite, each node can be colored white or black such that in g', only edges that connect nodes of different colors are present. Since g' is regular, the number of white nodes must be $n/2$. Pick any node A; w.l.o.g. assume A is white. There are d' nodes adjacent to A in g', and all are black. The d nodes $B_1, B_2, ...B_d$ include $(n/2 - 1)$ white nodes and $(n/2 - d')$ blacks. Note that $(n/2 - 1) = (n/2 - d') + (d' - 1)$ and $(d' - 1) \geq 3$ (because $n \geq d + 4$).

So for every way of labeling the nodes $B_1, ..., B_d$ there must be at least one pair (B_{2i-1}, B_{2i}) with both members of the pair white, i.e. not joined in g', i.e. joined in g. So A cannot be removed.

It is necessary that $d+4 \leq n \leq 2d+2$. The case $n = 2d+2$ is not very interesting because g is then two disjoint K_{d+1} graphs.

B Discussion of Algorithms *A* and *D*

In Section 2.1 we provided an abstract description of how algorithm A adds a node to a graph g, and in Section 4.1 we presented algorithm D, the inverse of algorithm A, to delete a node from a graph g. Below, we elaborate on a particular implementation of algorithms A and D.

B.1 Discussion of Algorithm *A*

Assume that a node j is being added to graph g which has more than $d + 1$ nodes (otherwise node j connects to all nodes in graph g). Let $\mathcal{I}$ be some subset of nodes in the graph g; we refer to $\mathcal{I}$ as the *introduction set*. Let $\mathcal{S}$ be the current list of neighbors of j; at the start, $\mathcal{S} = \phi$, and at the end of the procedure, $|\mathcal{S}| = d$. Initially, node j makes contact with one node i in the introduction set $\mathcal{I}$. Node i chooses a random time-to-live (TTL) greater than zero. It sends a message to a randomly selected neighbor with request to add node j, and includes the chosen TTL in the message. Whenever a node in graph g receives a message to add node j and the TTL is greater than 1, the node forwards the message to one of its neighbors selected uniformly at random and decrements the TTL by 1. Once the TTL counter hits 1, the message is forwarded to a neighbor $v_1 \notin \mathcal{S}$. Such a neighbor exists because $|\mathcal{S}| < d$ and every node has d neighbors. Node v_1 now selects a neighbor $v_2 \notin \mathcal{S}$, and nodes v_1 and v_2 are added to (ordered) list $\mathcal{S}$. The procedure is repeated until $\mathcal{S}$ has d elements. When $|\mathcal{S}| = d$, break every edge (v_{2k-1}, v_{2k}) and add edges (v_{2k-1}, j) and (v_{2k}, j) to g. A possible pseudo-code for the above implementation of algorithm A when TTL is 1 appears in Figure 6 and uses additional notation from Definition 1.

The set of edges, $\mathcal{N}(j)$, that were broken when j was added to g must be stored in the graph since the edges in $\mathcal{N}(j)$ will have to be restored when deleting node j using algorithm D. Standard reliable data storage techniques need to be used. For example, when storing $\mathcal{N}(j)$ we may not want to store the entire information in node j since a failure of node j leads to our inability to rebuild the graph. One way to get around this issue is to build in redundancy into the storage of $\mathcal{N}(j)$. Furthermore, any appropriate locking granularity may be used when implementing the algorithm.

B.2 Discussion of Algorithm *D*

Assume that a node j is being removed from graph g. If j is natal, we know from Section 4.1 that the nodes that are neighbors of node j are the same as the nodes in $\mathcal{N}(j)$. Therefore, we can simply restore all of the broken edges as defined in $\mathcal{N}(j)$ and delete node j from the graph. However, if j is *not* a natal node, j finds a natal node, η, in the graph and swaps positions. During a swap, the two nodes exchange neighbor sets and their sets $\mathcal{N}$ of initially broken edges, and the neighbors of these nodes also update their neighbor lists to reflect the swap. After a swap, node j is a natal node and can be deleted as one.

Finding a natal node reduces to the problem of node j broadcasting in graph g. As we have seen in Section 5.1, on average a node is no more than two nodes away from a natal node. Therefore, almost every node in the graph has a natal node either as its neighbor or as its neighbors' neighbor. Using this observation, finding a natal node will require node j asking first its neighbors if any one of them is a natal node, then its neighbors' neighbor, etc. until a natal node if found. The broadcasts can be limited using TTL values. In particular, broadcasts with increasing TTL values can expand the search for a natal node if one has not yet been found. As with the node addition process, any appropriate locking granularity can be used. A possible pseudo-code for the above implementation of algorithm D and some of its helper procedures appears in Figure 6 and uses notation from Definition 1. The procedure *SWAP_NODES* is simply the swap process described earlier.

Definition 1. *MSG.send(M,i,j,[T]) denotes a node i sending message M to node j with an optional time-to-live T. MSG.receive(M,i,j) denotes a node j receiving message M from node i. In both cases, a message is never forwarded to a node that the message has already traversed. NEIGHBORS(i) denotes nodes directly connected to node i. CONNECT(i, j)/BREAK(i, j) denote procedures to form/break the link between nodes i and j.*

```
Algorithm A
MSG.send (Add, $j$, $i \in \mathcal{I}$)
$s$ =a random node selected by node $i$
Ordered list $\mathcal{S}$ = NULL
while ($|\mathcal{S}| < d$) do
  $\mathcal{S} = \mathcal{S} \bigcup s$
  $s = s' : s' \in \text{NEIGHBORS}(s), s' \notin \mathcal{S}$
end
for ($k = 0$; $k < d/2$; $k++$) do
  BREAK($\mathcal{S}[2 \cdot k]$, $\mathcal{S}[2 \cdot k + 1]$)
  CONNECT($j$, $\mathcal{S}[2 \cdot k]$)
  CONNECT($j$, $\mathcal{S}[2 \cdot k + 1]$)
end
```

```
Algorithm D
if ($j$ is natal) then
  REMOVE_NATAL($j$)
end
else
  $\eta$ = FIND_NATAL($j$)
  SWAP_NODES($j, \eta$)
  REMOVE_NATAL($j$)
end
```

```
Procedure FIND_NATAL
Natal_found = FALSE
asynchronous event $\mathcal{E}$ =
  MSG.receive(Looking_for_Natal, $\nu, j, T$)
$\mathcal{E} \implies$ Natal_found = TRUE
$T = 0$ (Time to Live)
while (NOT Natal_found) do
  $T++$
  for ($\eta \in$ NEIGHBORS($j$)) do
    MSG.send(Looking_for_natal, $j, \eta, T$)
  end
  Wait some time in anticipation of event $\mathcal{E}$
end
return $\nu$
```

```
Procedure REMOVE_NATAL
for ($\eta \in$ NEIGHBORS($j$)) do
  BREAK($j, \eta$)
end
for ($link \in \mathcal{N}(j)$) do
  CONNECT ($link$[0], $link$[1])
end
DELETE($j$)
```

Fig. 6. Algorithms A, D and helper procedures. The algorithms are presented from a system (graph)-wide perspective and can be interpreted for each node in the graph.

Online, Dynamic, and Distributed Embeddings of Approximate Ultrametrics

Michael Dinitz*

Computer Science Department, Carnegie Mellon University, Pittsburgh,
PA 15213, USA
mdinitz@cs.cmu.edu

Abstract. The theoretical computer science community has traditionally used embeddings of finite metrics as a tool in designing approximation algorithms. Recently, however, there has been considerable interest in using metric embeddings in the context of networks to allow network nodes to have more knowledge of the pairwise distances between other nodes in the network. There has also been evidence that natural network metrics like latency and bandwidth have some nice structure, and in particular come close to satisfying an ϵ-three point condition or an ϵ-four point condition. This empirical observation has motivated the study of these special metrics, including strong results about embeddings into trees and ultrametrics. Unfortunately all of the current embeddings require complete knowledge about the network up front, and so are less useful in real networks which change frequently. We give the first metric embeddings which have both low distortion and require only small changes in the structure of the embedding when the network changes. In particular, we give an embedding of semimetrics satisfying an ϵ-three point condition into ultrametrics with distortion $(1+\epsilon)^{\log n+4}$ and the property that any new node requires only $O(n^{1/3})$ amortized edge swaps, where we use the number of edge swaps as a measure of "structural change". This notion of structural change naturally leads to small update messages in a distributed implementation in which every node has a copy of the embedding. The natural offline embedding has only $(1+\epsilon)^{\log n}$ distortion but can require $\Omega(n)$ amortized edge swaps per node addition. This online embedding also leads to a natural dynamic algorithm that can handle node removals as well as insertions.

1 Introduction

Many network applications, including content distribution networks and peer-to-peer overlays, can achieve better performance if they have some knowledge of network latencies, bandwidths, hop distances, or some other notion of distance between nodes in the network. Obviously the application could estimate these

* This work was supported in part by an NSF Graduate Research Fellowship and an ARCS Scholarship, and was partially done while the author was an intern at Microsoft Research-Silicon Valley.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 152–166, 2008.

distances by an on-demand measurement, but they get even more benefit from an instantaneous estimate that does not require recurring measurements. Many solutions have been proposed for this problem, most of which involve measuring either a random or a carefully chosen subset of pairwise distances and then embedding into a low-dimension coordinate space, e.g. a Euclidean space [6, 7, 9, 10, 12]. Because of this methodology, the problem of giving good approximations of network latencies is sometimes referred to as the *network coordinate problem*.

We take our inspiration from the work of Abraham et al. [1] who tried to use embeddings into *tree metrics* instead of into Euclidean space. Through experimental measurements they discovered that many networks have latencies that "locally" look like trees, and in particular satisfy an ϵ-four point condition that they invented. They gave an embedding into a single tree which has small distortion for small ϵ and allows instantaneous distance measurements like in the coordinate case, but also due to its tree structure has a natural notion of hierarchy and clustering that corresponds to an intuitive model of the Internet. However, their embedding was mainly of theoretical interest, since in practice it required having global knowledge of the complete network and complete recomputation whenever the network changed. Since real networks change regularly and do not normally allow a global viewpoint, the applicability of their tree embedding was unclear.

In this paper we take a first step towards a practical tree embedding for network distances. We define an ϵ-three point condition (as opposed to the four point condition of [1]) and show that there are good embeddings into ultrametrics when ϵ is small. Since ultrametrics are a special case of tree metrics this is also a tree embedding, but they have the extra property that they can be represented by *spanning* trees, so no Steiner nodes are required. They also satisfy a"bottleneck" condition which allows ultrametrics to represent bandwidths (actually 1 over the bandwidth) particularly well. We do not want to assume a static network like in previous work, so we give an *online* embedding which works no matter what order nodes in the network arrive (but assumes that nodes never leave) and a *dynamic* embedding which extends the online embedding to handle the case of nodes leaving.

In this paper we allow our online embedding to change the already-embedded points when a new one arrives, unlike the traditional model of online algorithms which does not allow change. However, an important part of this work is *minimizing* that change, and in fact trading it off with the distortion. We try to minimize the "structural change" of the embedding, where we measure the structural change by the number of edges in the disjoint union of the tree before a node insertion and the tree after a node insertion. We show that there is a tradeoff between distortion and structural change: the offline algorithm that minimizes distortion has large structural change, while by slightly relaxing the distortion requirement we can decrease the structural change by a polynomial amount. This immediately results in a bound on the communication complexity of a distributed algorithm that sends update messages across an underlying routing fabric.

Aside from the original practical motivation, we believe that our embedding is also of theoretical interest. Designing online embeddings is a very natural problem, but has proven surprisingly difficult to solve. In fact, there is basically no previous work on embeddings in any model of computation other than the standard offline full information model, even while online, dynamic, and streaming algorithms are becoming more important and prevalent. While in many cases it is easy to see that it is basically impossible to give an online embedding that doesn't change the embedding of previous points at all, we hope that this paper will begin the study of online embeddings that minimize structural change, or at least tradeoff structural change for extra distortion.

1.1 Definitions and Results

A *semimetric* is a pair (V, d) where $d : V \times V \rightarrow \mathbb{R}_{\geq 0}$ is a distance function with three properties that hold for all $x, y \in V$. First, $d(x, y) \geq 0$. Second, $d(x, y) = 0$ if and only if $x = y$. Lastly, $d(x, y) = d(y, x)$. Note that we are not assuming a metric space, so we do not assume the triangle inequality. Instead, throughout this paper we assume that the underlying semimetric is approximately an ultrametric, in the following sense.

Definition 1 (ϵ-three point condition). *A semimetric (V, d) satisfies the ϵ-three point condition if $d(x, y) \leq (1 + \epsilon) \max\{d(x, z), d(y, z)\}$ for all $x, y, z \in V$. We let $\mathsf{3PC}(\epsilon)$ denote the class of semimetrics satisfying the ϵ-three point condition.*

Ultrametrics are exactly the metrics that satisfy the 0-three point condition, i.e. $\mathsf{ULT} = \mathsf{3PC}(0)$ where ULT is the set of all ultrametrics. So for any given semimetric, the smallest ϵ for which it satisfies the ϵ-three point condition is a measure of how "close" it is to being an ultrametric. Clearly every metric satisfies the 1-three point condition, since by the triangle inequality $d(x, y) \leq d(x, z) + d(y, z) \leq 2 \max\{d(x, z), d(y, z)\}$. Thus when the semimetric actually satisfies the triangle inequality, we know that it is in $\mathsf{3PC}(1)$.

This definition is in contrast to the ϵ-four point condition defined in [1], which is a condition on the lengths of matchings on four points rather than the lengths of edges on three points. In particular, a metric (V, d) satisfies the ϵ-four point condition if $d(w, z) + d(x, y) \leq d(w, y) + d(x, z) + 2\epsilon \min\{d(w, x), d(y, z)\}$ for all $x, y, z, w \in V$ such that $d(w, x) + d(y, z) \leq d(w, y) + d(x, z) \leq d(w, z) + d(x, y)$. This definition also has the property that all metrics satisfy the 1-four point condition, but instead of being related to ultrametrics it is related to trees, in that tree metrics are exactly the metrics satisfying the 0-four point condition.

While ultrametrics are just semimetrics in $\mathsf{3PC}(0)$, we will represent them as trees over the point set. Given a tree $T = (V, E)$ with weights $w : E \rightarrow \mathbb{R}_{\geq 0}$, for $x, y \in V$ let $P(x, y)$ denote the set of edges on the unique path in T between x and y. Define $d_T : V \times V \rightarrow \mathbb{R}_{\geq 0}$ by $d_T(x, y) = \max_{e \in P(x,y)} w(e)$. It is easy to see that (V, d_T) is an ultrametric, and that every ultrametric can be represented in this way. We say that (V, d_T) is the ultrametric *induced* by T.

Since these are the only embeddings we will use, we define the contraction, expansion, and distortion of an embedding in the obvious ways. Given a semimetric (V, d) and a tree T on V, the *expansion* of the embedding is $\max_{u,v \in \binom{V}{2}} \frac{d_T(u,v)}{d(u,v)}$, the *contraction* of the embedding is $\max_{u,v \in \binom{V}{2}} \frac{d(u,v)}{d_T(u,v)}$, and the *distortion* is the product of the expansion and the contraction.

In this paper we want to solve embedding problems *online*, when vertices arrive one at a time and we want a good embedding at every step. An online ultrametric embedding is simply an algorithm that maintains a tree (and thus an induced ultrametric) on the vertices that have arrived, and outputs a new spanning tree when a new node arrives. There are two measures of the quality of an online embedding: its distortion and its update cost. An online embedding has distortion at most c if every tree that it outputs has distortion at most c relative to the distances between the nodes that have already arrived.

The *update cost* of the algorithm is the average number of edges in the tree that have to be changed when a node arrives. Let $\mathcal{A}(\epsilon)$ be an online ultrametric embedding, and suppose that it is run on a semimetric $(V, d) \in \mathsf{3PC}(\epsilon)$ where the order in which the nodes arrive is given by π. Let $T_i = (V_i, E_i)$ be the tree given by the $\mathcal{A}$ after the ith node arrives. Then $cost(\mathcal{A}(\epsilon), V, d, \pi) = \sum_{i=1}^{|V|} |E_i \setminus E_{i-1}|/|V|$. We define the overall update cost of $\mathcal{A}(\epsilon)$ to be its worst case cost over semimetrics satisfying the ϵ-three point condition and ordering of the nodes of the semimetric, i.e. $\sup_{(V,d) \in \mathsf{3PC}(\epsilon), \pi:[n] \rightarrow V} cost(\mathcal{A}, v, d, \pi)$.

Now we can state our main result in the online framework. All logs are base 2 unless otherwise noted.

Theorem 1. *For any $\epsilon \geq 0$ there is an online ultrametric embedding algorithm for $\mathsf{3PC}(\epsilon)$ which has distortion at most $(1+\epsilon)^{\lceil \log n \rceil + 4}$ and update cost at most $n^{1/3}$, where n is the number of points in the semimetric.*

The dynamic setting is similar to the online setting, except that nodes can also leave after they arrive. The cost is defined similarly, as the average number of edges changed by an operation, where now an operation can be either an arrival or a departure. We modify our online algorithm to handle departures, giving a similar theorem.

Theorem 2. *For any $\epsilon \geq 0$ there is a dynamic ultrametric embedding for $\mathsf{3PC}(\epsilon)$ which has distortion at most $(1+\epsilon)^{\lceil \log n \rceil + 5}$ (where n is the number of nodes currently in the system) and update cost at most $O(n_{max}^{1/3})$ (where n_{max} is the maximum number of nodes that are ever in the system at one time).*

Note that this notion of update cost is essentially the communication complexity of the natural distributed implementation. In particular, suppose that every node in a network has a copy of the tree (and thus the embedding). When a new node arrives, it obtains a copy of the tree from some neighbor, and then runs an algorithm to determine what the new embedding should be. The update cost is just the expected size of the update message which must be sent across the network to inform all nodes of the new embedding, so the communication complexity of the algorithm is at most $O(tn_{max})$ times the update cost where t is the number of operations (since for each operation we have to send an update to every node in the system). We describe this application in more detail in the full version, but the main result is the following upper bound.

Theorem 3. *There is a distributed dynamic ultrametric embedding algorithm for* $\mathsf{3PC}(\epsilon)$ *that allows every node to estimate the distance between any two nodes up to a distortion of* $(1+\epsilon)^{\lceil \log n \rceil + 5}$ *while only using space* $O(n)$ *at each node and communication complexity* $O(tn_{max}^{4/3})$ *in the dynamic arrival model.*

1.2 Related Work

Our work is heavily influenced by the work of Abraham et al. [1], who gave an embedding of metrics satisfying the related ϵ-four point condition into tree metrics. While our techniques are not the same (since we focus on ultrametrics), the motivation (network latency prediction) is, and part of the original motivation for this work was an attempt to make a distributed version of the algorithm of [1].

This work also fits into the "local vs. global" framework developed by [3] and continued in [5]. These papers consider embeddings from one space into a class of spaces where it is assumed that all subsets of the original metric of size at most some k embed well into the target class. It is easy to see that a semimetric satisfies the ϵ-three point condition if and only if every subset of size 3 embeds into an ultrametric with distortion at most $(1+\epsilon)$. They essentially solve the offline problem for ultrametrics by showing that if every subset of size k embeds into an ultrametric with distortion at most $1+\epsilon$ then the entire space embeds into an ultrametric with distortion at most $(1+\epsilon)^{O(\log_k n)}$. The case of $k=3$ is special, though, since it is the smallest value for which it is true that if every subset of size k embeds isometrically into an ultrametric then the entire space can be embedded isometrically into an ultrametric. Thus it is the most important case, and we tailor our algorithms to it.

We also note that the definitions of the ϵ-three point condition given by Abraham et al. [1] and our definition of the ϵ-four point condition are closely related to the notion of δ-hyperbolicity defined by Gromov in his seminal paper on hyperbolic groups [8]. In particular, his notion of hyperbolicity is an additive version of our notions. While we do not use any of the results or techniques of [8] directly, the methods and results are somewhat similar.

Finally, while there has been extensive work in the theoretical computer science community on metric embeddings, none of it has been concerned with models of computation other than the standard offline case. This seems to be a glaring omission, and this work is the first to provide positive results. We hope that it will be the first in a line of work examining metric embeddings in other models, especially online, dynamic, and streaming computation.

2 Online Embedding

We begin with the following lemma, which is an easy corollary of [3, Theorem 4.2]. We will use it throughout to bound the contraction of our embeddings.

Lemma 1. *Let* (V, d) *be a semimetric in* $\mathsf{3PC}(\epsilon)$*. Then for any spanning tree* T *on* V *and nodes* $u, v \in V$*,* $d(u, v) \leq (1 + \epsilon)^{\lceil \log n \rceil} d_T(u, v)$

This immediately implies that the minimum spanning tree is a good embedding:

Theorem 4. *Let* T *be a minimum spanning tree of the semimetric* $(V, d) \in \mathsf{3PC}(\epsilon)$*. Then* $d_T(u, v) \leq d(u, v) \leq (1 + \epsilon)^{\lceil \log n \rceil} d_T(u, v)$ *for all* $u, v \in V$

Proof. We know from Lemma 1 that $d(u, v) \leq (1 + \epsilon)^{\lceil \log n \rceil} d_T(u, v)$, so it just remains to prove that $d_T(u, v) \leq d(u, v)$. Let e be the maximum length edge on the path between u and v in T, and let ℓ be its length. If $\ell > d(u, v)$, then the spanning tree obtained by adding the edge $\{u, v\}$ and removing e is smaller than T, which is a contradiction since T is a minimum spanning tree. Thus $\ell \leq d(u, v)$, and so by definition $d_T(u, v) \leq d(u, v)$. □

It is actually easy to see that this is basically the same embedding used by [3], and furthermore that by replacing the metric distance with the Gromov product (as defined in [8]) and min/max with max/min we recover the embedding of hyperbolic spaces given in [8].

While the MST is a good embedding from the standpoint of distortion, computing and maintaining it requires possibly changing many edges on every update. For example, consider the metric on $\{x_1, \ldots, x_n\}$ that has $d(x_i, x_j) = 1 - i\delta$ for $i > j$, where δ is extremely small. If the nodes show up in the network in the order $x_1, x_2, \ldots, x_n$, then after x_i enters the system the MST is just the star rooted at x_i. So to maintain the MST we have to get rid of every edge from the old star and add every edge in the new one, thus requiring $\Omega(n)$ average edge changes per update.

One thing to note about this example is that it is not necessary to maintain the MST. Instead of relocating the center of the star to the new node every time, if that new node just adds one edge to the first node then the induced ultrametric only expands distances by a small amount. This suggests that maybe it is enough to just add the smallest edge incident on v when v arrives. Unfortunately this is not sufficient, as it is easy to construct semimetrics in $\mathsf{3PC}(\epsilon)$ for which this algorithm has distortion $(1+\epsilon)^{\Omega(n)}$. So maintaining the MST has good distortion but large update cost, while just adding the smallest edge has small update cost but large distortion. Can these be balanced out, giving small distortion and small update cost? In this section we show that, to a certain extent, they can.

We define the *k-threshold algorithm* as follows. We maintain a spanning tree T of the vertices currently in the system. When a node u enters the system, examine the edges from u to every other node in increasing order of distance. When examining $\{u, v\}$ we test whether $d_T(u, v) > (1 + \epsilon)^k d(u, v)$, and if so we add $\{u, v\}$ to T and remove the largest edge on the cycle that it induces. If not, then do not add $\{u, v\}$ to T. A formal definition of this algorithm is given in Algorithm 1.

This algorithm has the useful property that the embedded distance between two nodes never increases. More formally, consider a single step of the algorithm, in which it considers the edge $\{u, v\}$ and either adds it or does not. Let T be the

```
Input: Weighted tree $T = (V, E, w : E \to \mathbb{R}_{\geq 0})$, new vertex $u$, distances
       $d : u \times V \to \mathbb{R}_{\geq 0}$
Output: Weighted tree $T' = (V \cup \{u\}, E', w')$
Sort $V$ so that $d(u, v_1) \leq d(u, v_2) \leq \ldots \leq d(u, v_n)$;
Let $V' = V \cup \{u\}$, let $E' = E \cup \{\{u, v_1\}\}$, and set $w(\{u, v_1\}) = d(u, v_1)$;
for $i = 2$ to $n$ do
    Let $e$ be the edge of maximum weight on the path between $u$ and $v_i$ in
    $T' = (V', E')$;
    if $w(e) > (1 + \epsilon)^k d(u, v_i)$ then
        $E' \leftarrow E' \setminus \{e\} \cup \{\{u, v_i\}\}$;
        $w(\{u, v_i\}) = d(u, v_i)$;
    end
end
return $T' = (V', E', w)$
```

Algorithm 1. k-threshold algorithm

tree before and T' the tree after this decision (note that both T and T' share the same node set V).

Lemma 2. *$d_{T'}(x, y) \leq d_T(x, y)$ for all nodes $x, y \in V$*

Proof. If the algorithm does not add $\{u, v\}$ then $T = T'$, so $d_T(x, y) = d_{T'}(x, y)$. If it adds $\{u, v\}$ then it removes the largest edge on the path in T between u and v, say $\{w, z\}$ (without loss of generality we assume that the path from u to v in T goes through w before z). Clearly $d(w, z) > (1 + \epsilon)^k d(u, v)$ or else the algorithm would not have added $\{u, v\}$. If the path in T between x and y does not go through $\{w, z\}$ then it is the same as the path between x and y in T', so $d_T(x, y) = d_{T'}(x, y)$. If the path does go through $\{w, z\}$ then the path from u to w is unchanged, every edge on the path in T' from w to z has length at most $d(w, z)$, and the path from z to v is unchanged. Thus $d_{T'}(x, y) \leq d_T(x, y)$. □

Given this lemma, the following theorem is immediate.

Theorem 5. *After adding n nodes, the k-threshold algorithm results in an embedding into an ultrametric with distortion at most $(1 + \epsilon)^{k + \lceil \log n \rceil}$*

Proof. We know from Lemma 1 that the contraction is at most $(1 + \epsilon)^{\lceil \log n \rceil}$, so we just need to show that the expansion is at most $(1 + \epsilon)^k$. Let u and v be any two vertices, and without loss of generality let u enter the system after v. Then when u enters the system it chooses to either add the edge to v or not, but in either case immediately after that decision the expansion of $\{u, v\}$ is at most $(1 + \epsilon)^k$. Lemma 2 implies that this distance never increases, and thus the expansion is always at most $(1 + \epsilon)^k$. □

The hope is that by paying the extra $(1 + \epsilon)^k$ in distortion we get to decrease the number of edges added. Our main theorems are that this works for small k, specifically for k equal to 2 and 4.

Theorem 6. *The 2-threshold algorithm has update cost at most $n^{1/2}$*

Theorem 7. *The 4-threshold algorithm has update cost at most $n^{1/3}$*

We believe that this pattern continues for larger k, so that the k-threshold algorithm would have update cost $n^{2/(k+2)}$. Unfortunately our techniques do not seem strong enough to show this, since they depend on carefully considering the local structure of the embedding around any newly inserted node, where the definition of "local" increases with k. This type of analysis becomes infeasible for larger k, so a more general proof technique is required. However, the cases of small k are actually the most interesting, in that they provide the most "bang-for-the-buck": if our conjecture is true, then when k gets larger the same increase in distortion provides a much smaller decrease in update cost.

The following lemma about the operation of the k threshold algorithm will be very useful, as it gives us a necessary condition for an edge to be added that depends not just on the current tree but on all of the edges that have been inserted by the algorithm.

Lemma 3. *Suppose that u arrives at time t, and let v be any other node that is already in the system. Suppose there exists a sequence $u = u_0, u_1, \ldots, u_k = v$ of nodes with the following three properties:*

1. *The edge $\{u, u_1\}$ is considered by the k-threshold algorithm before $\{u, v\}$*
2. $d(u_i, u_{i+1}) \leq (1 + \epsilon)^k d(u, v)$
3. *If $\{u_i, u_{i+1}\}$ has never been inserted by the algorithm, then $d(u_i, u_{i+1}) \leq d(u, v)$*

Then the k-threshold algorithm will not add the edge $\{u, v\}$

Proof. For any two nodes x and y, let $x \sim y$ denote the path between them in the algorithm's tree just before it considers $\{u, v\}$. We know from the first property that $d(u, u_1) \leq d(u, v)$, and so when $\{u, v\}$ is considered the maximum edge on $u \sim u_1$ is at most $(1 + \epsilon)^k d(u, u_1) \leq (1 + \epsilon)^k d(u, v)$. Fix i, and suppose that $\{u_i, u_{i+1}\}$ was already inserted by the algorithm. Then when it was inserted the maximum edge on the path between u_i and u_{i+1} was just that edge, which had length $d(u_i, u_{i+1})$. So by Lemma 2, the maximum edge of $u_i \sim u_{i+1}$ is at most $d(u_i, u_{i+1})$, which by the second property is at most $(1 + \epsilon)^k d(u, v)$. On the other hand, suppose that $\{u_i, u_{i+1}\}$ was not inserted by the algorithm. Then when it was considered the path between u_i and u_{i+1} had maximum edge at most $(1+\epsilon)^k d(u_i, u_{i+1})$, and so by Lemma 2 and the third property we know that the maximum edge of $u_i \sim u_{i+1}$ is at most $(1 + \epsilon)^k d(u_i, u_{i+1}) \leq (1 + \epsilon)^k d(u, v)$.

So for all $i \in \{0, \ldots, k - 1\}$, the path $u_i \sim u_{i+1}$ has maximum edge at most $(1 + \epsilon)^k d(u, v)$. Since $u \sim v$ is a subset of $u \sim u_1 \sim \ldots \sim u_k$, this implies that $u \sim v$ has maximum edge at most $(1 + \epsilon)^k d(u, v)$, so the k-threshold algorithm will not add the edge $\{u, v\}$. □

2.1 2-Threshold

We now prove Theorem 6. Let G_t be the "all-graph" consisting of every edge ever inserted into the tree by the 2-threshold algorithm before the tth node arrives. In order to prove Theorem 6 we will use the following lemma, which states that nodes which are close to each other in G_t have smaller expansion than one would expect.

Lemma 4. *Let u and v be any two nodes. If there is some t_0 such that there is a path of length 2 between u and v in G_{t_0}, then for all $t \geq t_0$ the expansion of $\{u, v\}$ in the induced ultrametric is at most $(1 + \epsilon)$*

Proof. Let $u - w - v$ be a path of length two in G_t. We want to show that $\max\{d(u, w), d(w, v)\} \leq (1+\epsilon)d(u, v)$, since this along with Lemma 2 will imply that the the largest edge on the path between u and v in G_t has length at most $(1+\epsilon)d(u, v)$. If $d(u, v) \geq d(u, w)$, then $d(w, v) \leq (1 + \epsilon) \max\{d(u, w), d(u, v)\} = (1 + \epsilon)d(u, v)$, which would prove the lemma. Similarly, if $d(u, v) \geq d(w, v)$ then $d(u, w) \leq (1+\epsilon) \max\{d(w, v), d(u, v)\} = (1+\epsilon)d(u, v)$, again proving the lemma. So without loss of generality we assume that $d(u, w) > d(u, v)$ and $d(w, v) > d(u, v)$. Together with the ϵ-three point condition this implies that $d(u, w) \leq (1 + \epsilon)d(w, v)$ and $d(w, v) \leq (1 + \epsilon)d(u, w)$. We will now derive a contradiction from these assumptions, which proves the lemma since it implies that either $d(u, v) \geq d(u, w)$ or $d(u, v) \geq d(w, v)$.

Suppose that u is the last node of the three to enter the system. Then the sequence u, v, w satisfies the properties of Lemma 3, so the algorithm would not add $\{u, w\}$, giving a contradiction since we assumed that $\{u, w\}$ was in G_t. If v is the last of the three to arrive, then v, u, w satisfies the properties of Lemma 3, so $\{v, w\}$ would not have been added, giving a contradiction. If w is the last node of the three to be added, then we can assume by symmetry that $\{u, w\}$ was considered by the algorithm before $\{w, v\}$, in which case w, u, v satisfies the properties of Lemma 3, so $\{w, v\}$ would not have been added. Thus no matter which of the three arrives last we get a contradiction, proving the lemma. □

We will also need to relate the girth of a graph to its sparsity, for which we will use the following simple result that can be easily derived from [4, Theorem 3.7] and was stated in [2] and [11]. Recall that the girth of a graph is the length of the smallest cycle.

Lemma 5. *If a graph G has girth at least $2k + 1$, then the number of edges in G is at most $n^{1+\frac{1}{k}}$*

Now we can use a simple girth argument to prove Theorem 6.

Proof of Theorem 6. Suppose that G_{n+1} (which is the final graph of all edges ever inserted by the algorithm) has a cycle of length $h \leq 4$ consisting of the nodes $x_1, x_2, \ldots, x_4$, where x_1 is the last of the h nodes on the cycle to have entered the system. Without loss of generality, assume that $\{x_1, x_2\}$ is considered by the algorithm before $\{x_1, x_h\}$. Let e be the largest edge on the cycle other than $\{x_1, x_h\}$ and $\{x_1, x_2\}$. In the case of $h = 4$, Lemma 4 implies that $e \leq (1+\epsilon)^{h-3}d(x_2, x_h)$, and if $h = 3$ then this is true by definition since then $e = \{x_2, x_h\}$. But now we know from the ϵ-three point condition that this is at most $(1+\epsilon)^{h-2}\max\{d(x_1, x_2), d(x_1, x_h)\} = (1+\epsilon)^{h-2}d(x_1, x_h) \leq (1+\epsilon)^2 d(x_1, x_h)$. Thus every edge on the cycle is at most $(1+\epsilon)^2 d(x_1, x_h)$, so applying Lemma 3 to $x_1, x_2, \ldots, x_h$ implies that $\{x_1, x_h\}$ would not be added by the algorithm, and therefore would not be in G_{n+1}. Thus G_{n+1} has girth at least 5, so by Lemma 5 we have that the number of edges added by the algorithm is at most $n^{3/2}$, and thus the 2-threshold algorithm has update cost at most $n^{1/2}$. □

2.2 4-Threshold

In order to prove Theorem 7 we will prove a generalization of Lemma 4 for the 4-threshold algorithm. The proof follows the same outline as the proof of Lemma 4: first we show a series of sufficient conditions, and the assuming them all to be false we derive a contradiction with the operation of the algorithm. In one case this technique is insufficient, and instead we have to assume that not only are the sufficient conditions false, but so is the lemma, and then this is sufficient to derive a contradiction.

Lemma 6. *Let u and v be any two nodes, and let $h \leq 4$. Then if there is some t_0 such that there is a path of length h between u and v in G_{t_0}, then for all $t \geq t_0$ the expansion of $\{u, v\}$ in the induced ultrametric is at most $(1+\epsilon)^{h-1}$*

Proof. The case of $h = 1$ is obvious, since a path of length 1 is just an edge and Lemma 2 implies that the embedded distance from then on does not increase. The case of $h = 2$ is basically the same as Lemma 4, just with $k = 4$ instead of $k = 2$. Since the proof is basically the same, we defer it to the full version.

For the case of $h = 3$, consider a path $u - x - y - v$ in G_t. As before, we want to show that $\max\{d(u, x), d(x, y), d(y, v)\} \leq (1+\epsilon)^2 d(u, v)$, since along with Lemma 2 this will prove the lemma. Due to symmetry, we can assume without loss of generality that $d(u, x) \geq d(v, y)$. So there are two cases, corresponding to whether the maximum edge on the path is $\{u, x\}$ or whether it is $\{x, y\}$. In either case the theorem is trivially true if the maximum edge is at most $d(u, v)$, so without loss of generality we will assume that it is greater that $d(u, v)$.

Suppose that $\{u, x\}$ is the maximum edge on the path. Then we know from the $h = 2$ case that $d(u, x) \leq (1+\epsilon)d(u, y) \leq (1+\epsilon)^2 \max\{d(u, v), d(v, y)\}$. If $d(u, v) \geq d(v, y)$ then this proves the lemma, so without loss of generality we can assume that $d(u, v) < d(v, y)$, which implies that $d(u, x) \leq (1+\epsilon)^2 d(v, y)$. Now by using Lemma 3 we can derive a contradiction by finding an edge in G_t that the algorithm would not have added. If u is the last node of u, x, y, v to enter the system, then the sequence u, v, y, x satisfies the properties of Lemma 3, so the edge $\{u, x\}$ would not have been added. If x is the last of the four to enter then we can apply Lemma 3 to the sequence x, y, v, u, so again the edge $\{u, x\}$ would not have been added. If v is the last of the four to enter then we can apply Lemma 3 to the sequence v, u, x, y, so the edge $\{v, y\}$ would not have been added. Finally, if y is the last of the four to arrive then if $\{x, y\}$ is considered before $\{y, v\}$ then applying Lemma 3 to the sequence y, x, u, v contradicts the addition of the edge $\{y, v\}$, and if $\{y, v\}$ is considered before $\{x, y\}$ then applying Lemma 3 to the sequence y, v, u, x contradicts the addition of the edge $\{x, y\}$. Thus no matter which of the four nodes enters last we can find an edge in G_t that the algorithm would not have added.

The second case is that $\{x, y\}$ is the largest of the three edges. In this case we want to prove that $d(x, y) \leq (1 + \epsilon)^2 d(u, v)$. We know from the $h = 2$ case that $d(x, y) \leq (1+\epsilon)d(u, y) \leq (1+\epsilon)^2 \max\{d(u, v), d(y, v)\}$, which implies that we are finished if $d(u, v) \geq d(y, v)$, and thus without loss of generality $d(y, v) > d(u, v)$ and $d(x, y) \leq (1 + \epsilon)^2 d(y, v)$. Similarly, we know that $d(x, y) \leq (1 + \epsilon)d(v, x) \leq (1 + \epsilon)^2 \max\{d(u, v), d(u, x)\}$, so without loss of generality $d(u, x) > d(u, v)$ and $d(x, y) \leq (1 + \epsilon)^2 d(u, x)$. So all three of the edges are larger than $d(u, v)$, and they are all within a $(1 + \epsilon)^2$ factor of each other. Thus no matter which of the four nodes enters last there is a sequence to which we can apply Lemma 3 in order to find an edge in G_t which the algorithm wouldn't add. If the last node is u then the sequence is u, v, y, x and the edge is $\{u, x\}$, if the last node is x then the sequence is x, u, v, y and the edge is $\{x, y\}$, if the last node is y then the sequence is y, v, u, x and the edge is $\{x, y\}$, and if the last node is v then the sequence is v, u, x, y and the edge is $\{v, y\}$. Thus we have a contradiction in every case, which proves the lemma for $h = 3$.

For the $h = 4$ case, consider a path $u - x - z - y - v$ of length 4 in G_t. We want to show that $\max\{d(u, x), d(x, z), d(z, y), d(y, w)\} \leq (1+\epsilon)^3 d(u, v)$. By symmetry there are only two cases: either $d(u, x)$ is the maximum or $d(x, z)$ is the maximum. Let e be the length of this maximum edge. The lemma is clearly true if $e \leq d(u, v)$, so in both cases we can assume that the maximum edge is at least $d(u, v)$. We also know from the $h = 3$ case that $e \leq (1 + \epsilon)^2 d(u, y) \leq (1 + \epsilon)^3 \max\{d(u, v), d(y, v)\}$, so if $d(u, v) \geq d(y, v)$ then the lemma is true. So without loss of generality we can assume that $d(y, v) > d(u, v)$ and $e \leq (1 + \epsilon)^3 d(y, v)$.

We start with the case when the maximum edge is $\{x, z\}$, i.e. $e = d(x, z)$. Then from the $h = 3$ case we know that $d(x, z) \leq (1 + \epsilon)^2 d(x, v) \leq (1 + \epsilon)^3 \max\{d(u, x), d(u, v)\}$. If $d(u, v) \geq d(u, x)$ we are finished, so without loss of generality we can assume that $d(u, x) > d(u, v)$ and $d(x, z) \leq (1+\epsilon)^3 d(u, x)$. Now no matter which of the five vertices enters last, we can derive a contradiction using Lemma 3. This is because all of $d(u, x), d(x, z), d(v, y)$ are greater than $d(u, v)$ (which is the only edge in the cycle not in G_t) and are all within a $(1+\epsilon)^3$ factor of each other. $d(z, y)$ is not necessarily larger than $d(u, v)$ or close to the other edges, but it's enough that it is smaller than $d(x, z)$ since that means that it is not too large and if it is very small then we will not have to use it to derive a contradiction. So if u enters last then the $\{u, x\}$ edge would not be added, if x or z enters last then the $\{x, z\}$ edge wouldn't be added, if v enters last then the $\{y, v\}$ edge would not be added, and if y enters last whichever of $\{z, y\}$ and $\{y, v\}$ is considered second would not be added. These are all contradictions, which finishes the case of $e = d(x, z)$.

The final case, when the maximum edge is $\{u, x\}$, is slightly more complicated because now it is not obvious that z has to be incident to a large edge, which we need in order to derive the type of contradictions that we have been using. If $d(x, y) \leq d(v, y)$, though, then this isn't a problem and we can easily derive the same kind of contradictions using Lemma 3: if u or x enters the system last among u, x, y, v (note the exclusion of z) then the $\{u, x\}$ edge will not be added, and if v or y enters the system last of the four then then $\{v, y\}$ edge will not be added. This is because both $d(x, y)$ and $d(u, v)$ are at most $d(v, y)$, which is at most $d(u, x)$, and $e = d(u, x) \leq (1 + \epsilon)^3 d(v, y)$.

So without loss of generality we assume that $d(x, y) > d(v, y)$. We still get an immediate contradiction if the last of the five nodes to enter is not z since we still have that $d(u, v) < d(v, y) \leq d(u, x)$ and $d(y, v) \leq (1 + \epsilon)^3 d(u, x)$ and $d(x, z), d(y, z) \leq d(u, x)$. This implies that if it is u or x which arrives last then we can apply Lemma 3 to the sequence u, v, y, z, x (or its reverse) to get that $\{u, x\}$ will not be added. If it is v then the sequence v, u, x, z, y implies that $\{v, y\}$ will not be added, and if it y then the same around-the-cycle sequence construction implies that whichever of $\{y, z\}$ and $\{y, v\}$ is larger will not be added.

Thus the only difficult case is when z is the last of the five to arrive. Note that we are trying to prove that $d(u, x) \leq (1 + \epsilon)^3 d(u, v)$, so we will assume that $d(u, x) > (1 + \epsilon)^3 d(u, v)$ and derive a contradiction. If $d(v, y) < d(u, x)/(1 + \epsilon)^2$ then the ϵ-three point condition and the fact that $d(v, y) > d(u, v)$ imply that $d(u, y) < d(u, x)/(1 + \epsilon)$. Now the ϵ-three point condition implies that $d(x, y) \geq d(u, x)/(1+\epsilon)$, which in turn implies that $\max\{d(x, z), d(z, y)\} \geq d(u, x)/(1+\epsilon)^2$. But now we have our contradiction, since this means that whichever of the two edges incident on z is considered second is at most $(1 + \epsilon)^2$ times smaller than $d(u, x)$ and thus is also larger than $d(u, v)$, and so will not be added.

So finally we are left with the case that $d(v, y) \geq d(u, x)/(1 + \epsilon)^2$. Recall that $d(x, y) > d(v, y)$, so $d(x, y) > d(u, x)/(1 + \epsilon)^2$ and thus $\max\{d(x, z), d(z, y)\} \geq d(u, x)/(1 + \epsilon)^3 > d(u, v)$. So we again have the contradiction that whichever of the two edges incident on z is considered second will not be added by the algorithm. □

With this lemma in hand we can now prove Theorem 7:

Proof of Theorem 7. Suppose that G_{n+1} has a cycle of length $h \leq 6$ consisting of the nodes $x_1, x_2, \ldots, x_h$, where x_1 is the last of the h nodes on the cycle to have entered the system. Without loss of generality, assume that $\{x_1, x_2\}$ is considered by the algorithm before $\{x_1, x_h\}$. Let e be the largest edge on the cycle other than $\{x_1, x_h\}$ and $\{x_1, x_2\}$. Lemma 6 implies that $e \leq (1 + \epsilon)^{h-3} d(x_2, x_h) \leq (1 + \epsilon)^{h-2} \max\{d(x_1, x_2), d(x_1, x_h)\} = (1 + \epsilon)^{h-2} d(x_1, x_h) \leq (1 + \epsilon)^4 d(x_1, x_h)$. Thus every edge on the cycle is at most $(1 + \epsilon)^4 d(x_1, x_h)$, so applying Lemma 3 to $x_1, x_2, \ldots, x_h$ implies that $\{x_1, x_h\}$ would not be added by the algorithm, and therefore would not be in G_{n+1}. Thus G_{n+1} has girth at least 7, so by Lemma 5 we have that the number of edges added by the algorithm is at most $n^{4/3}$, and thus the 4-threshold algorithm has cost at most $n^{1/3}$. □

A natural question is whether our analysis of the girth is tight. We show in the full version that it is: for any k (and thus in particular for $k = 2$ and $k = 4$) and any $\epsilon \geq 0$, there is a semimetric in $\mathsf{3PC}(\epsilon)$ and an ordering of the points such that the graph of all edges inserted by the k-threshold algorithm results in a cycle of length $k + 3$. Obviously we would like to also show that not only is the girth analysis tight, but so is the update cost analysis that we use it for. Unfortunately we are not able to do this, but for good reason. We conjecture that the graph of edges inserted by the k-threshold algorithm has girth at least $k + 3$ for all even integer k, and if this is true then proving the matching lower bound would solve the celebrated Erdös girth conjecture.

3 Dynamic Embedding

The algorithms in the previous section work in the online setting, when nodes enter the system one at a time and then never leave. While slightly more realistic than the standard offline setting, this still does not model real networks particularly well since it does not handle the removal of nodes, which happens all the time in reality. What we would really like is a *dynamic* embedding that can handle both insertions and removals with the same kind of guarantees that we obtained in Theorems 6 and 7. Fortunately it turns out that insertions are the difficult case, and removals are easy to handle by just allowing Steiner nodes in the embedding.

Our removal algorithm is as follows. We say that a node in the embedding is *active* if it is currently in the system and *inactive* if it is not. Nodes can change from being active to being inactive, but not from being inactive to active. Suppose that u leaves the system. Let a be the number of active nodes after u leaves, and let b be the number of inactive nodes. If $a \leq \alpha(a + b)$ (for some parameter $0 < \alpha < 1$ to be specified later) then remove all inactive nodes and let T be a minimum spanning tree on the active nodes. We call this a *cleanup* step. Otherwise do nothing, in which case u stays in the current tree as an inactive (i.e. Steiner) node. When a new node u enters the system we use the k-threshold algorithm to insert it, i.e. we consider the distances from u to the other active nodes in non-decreasing order and perform an edge swap if the current expansion is greater than $(1 + \epsilon)^k$. We call this the k-dynamic algorithm.

It is easy to see that Lemma 4 basically hold in this setting; it just needs to be changed to only hold for paths for which both of the endpoints are active. The proof of the lemma still goes through since for a path $u - x - v$ in which u and v are active, when the path was created x had to be active as well (since when an edge is formed its endpoints are active, so no matter which of the three nodes arrives last x must be active). Thus Theorem 6 still holds since when a cycle is formed in the G_t by the addition of a node u, clearly the two nodes adjacent to u in the cycle must be active so the modified version of Lemma 4 applies.

It is slightly more complicated to see that Lemma 6 still holds with the same change, that its guarantee only applies to paths where the endpoints are both active. The arguments have to be made much more carefully because we can only use smaller cases (e.g. depending on the $h = 3$ case when proving the $h = 4$ case) when we are guaranteed that both endpoints of the subpath that we are applying it to are active. We also cannot assume that any distance involving an inactive node is expanded by only $(1+\epsilon)^k$, so property 3 of Lemma 3 has to be changed so that if $\{u_i, u_{i+1}\}$ has never been inserted by the algorithm then $d(u_i, u_{i+1}) \leq d(u, v)$ and u_i and u_{i+1} are both active. Due to the similarity with the proof of Lemma 6, we defer the proof to the full version.

We now show that this algorithm has small distortion in terms of the number of nodes actually in the system. Setting $\alpha = \frac{1}{2}$ gives a reasonable tradeoff, so that is what we will do. Due to space constraints we defer the proofs of theorems in this section to the full version, but we note that combining Theorems 8 and 9 for $k = 4$ gives us Theorem 2.

Theorem 8. *Suppose that at time t there are n nodes in the system and T is the tree that the k-dynamic algorithm is currently maintaining. Then for all x, y in the system, $\frac{d(x,y)}{(1+\epsilon)^{\lceil \log n \rceil + 1}} \leq d_T(x,y) \leq (1+\epsilon)^k d_T(x,y)$*

Define the cost of an operation to be the number of edge insertions that it requires, so the cost of an insertion is the same as in the online case and the cost of a removal is 0 if it does not result in a cleanup and the $a-1$ if it does (recall that a is the number of active nodes).

Theorem 9. *The amortized cost of an operation in the k-dynamic algorithm (for $k = 2$ or 4) is at most $O(n_{max}^{2/(k+2)})$, where n_{max} is the largest number of nodes that are ever in the system at one time.*

4 Conclusion and Open Questions

We have shown that when a semimetric satisfies an ϵ-three point condition (or equivalently has the property that every subset of size three embeds into an ultrametric with distortion at most $1+\epsilon$) it is possible to embed it online into a single ultrametric in a way that has both small distortion and small "structural change". We measure structural change by the number of edges that have to be added to the tree representing the ultrametric, which in a distributed setting is a natural goal since it results in small update messages when the network changes. Furthermore, a trivial example shows that the best offline embedding might require very large structural change, while we get a polynomial reduction in the structural change by losing only a constant factor in the distortion.

The obvious open question is whether the k-threshold algorithm is good for all values of k. In particular, if the analog of Lemma 6 holds for all k then the $\Theta(\log n)$ threshold algorithm would have update cost at most $O(1)$ and distortion only $(1+\epsilon)^{O(\log n)}$. We believe that this is true, but our techniques do not extend past $k = 4$. In particular, the proof of Lemma 6 proceeds via case analysis, and when k gets large there are just too many cases. Nevertheless, we conjecture that for even k the graph of all edges ever inserted by the k-threshold algorithm has girth at least $k+3$, and thus has update cost at most $n^{\frac{2}{k+2}}$.

Acknowledgements. We would like to thank Dahlia Malkhi for pointing out this line of research and for many helpful discussions.

References

1. Abraham, I., Balakrishnan, M., Kuhn, F., Malkhi, D., Ramasubramanian, V., Talwar, K.: Reconstructing approximate tree metrics. In: PODC 2007: Proceedings of the twenty-sixth annual ACM Symposium on Principles of Distributed Computing, pp. 43–52. ACM, New York (2007)
2. Althöfer, I., Das, G., Dobkin, D., Joseph, D., Soares, J.: On sparse spanners of weighted graphs. Discrete Comput. Geom. 9(1), 81–100 (1993)
3. Arora, S., Lovász, L., Newman, I., Rabani, Y., Rabinovich, Y., Vempala, S.: Local versus global properties of metric spaces. In: SODA 2006: Proceedings of the seventeenth annual ACM-SIAM Symposium on Discrete Algorithm, pp. 41–50. ACM, New York (2006)
4. Bollobás, B.: *Extremal graph theory.* London Mathematical Society Monographs, vol. 11. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London (1978)
5. Charikar, M., Makarychev, K., Makarychev, Y.: Local global tradeoffs in metric embeddings. In: FOCS 2007: Proceedings of the forty-eighth annual IEEE Symposium on Foundations of Computer Science, pp. 713–723 (2007)
6. Costa, M., Castro, M., Rowstron, A., Key, P.: Pic: Practical internet coordinates for distance estimation. In: ICDCS 2004: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004), Washington, DC, USA, pp. 178–187. IEEE Computer Society, Los Alamitos (2004)
7. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. SIGCOMM Comput. Commun. Rev. 34(4), 15–26 (2004)
8. Gromov, M.: Hyperbolic groups. In: Essays in group theory. Math. Sci. Res. Inst. Publ, vol. 8, pp. 75–263. Springer, New York (1987)
9. Lim, H., Hou, J.C., Choi, C.-H.: Constructing internet coordinate system based on delay measurement. In: IMC 2003: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, pp. 129–142. ACM, New York (2003)
10. Tang, L., Crovella, M.: Virtual landmarks for the internet. In: IMC 2003: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, pp. 143–152. ACM, New York (2003)
11. Thorup, M., Zwick, U.: Approximate distance oracles. J. ACM 52(1), 1–24 (2005)
12. wei Lehman, L., Lerman, S.: Pcoord: Network position estimation using peer-to-peer measurements. In: NCA 2004: Proceedings of the Network Computing and Applications, Third IEEE International Symposium on (NCA 2004), Washington, DC, USA, pp. 15–24. IEEE Computer Society, Los Alamitos (2004)

Constant-Space Localized Byzantine Consensus

Danny Dolev* and Ezra N. Hoch

School of Engineering and Computer Science,
The Hebrew University of Jerusalem, Israel
{dolev,ezraho}@cs.huji.ac.il

Abstract. Adding *Byzantine* tolerance to large scale distributed systems is considered non-practical. The time, message and space requirements are very high. Recently, researches have investigated the *broadcast problem* in the presence of a f_ℓ-local *Byzantine* adversary. The local adversary cannot control more than f_ℓ neighbors of any given node. This paper proves sufficient conditions as to when the synchronous *Byzantine consensus problem* can be solved in the presence of a f_ℓ-local adversary.

Moreover, we show that for a family of graphs, the *Byzantine* consensus problem can be solved using a relatively small number of messages, and with time complexity proportional to the diameter of the network. Specifically, for a family of bounded-degree graphs with logarithmic diameter, $O(\log n)$ time and $O(n \log n)$ messages. Furthermore, our proposed solution requires constant memory space at each node.

1 Introduction

Fault tolerance of a distributed system is highly desirable, and has been the subject of intensive research. *Byzantine* faults have been used to model the most general and severe failures. Classic *Byzantine*-tolerant research has concentrated on an "all mighty" adversary, which can choose up to f "pawns" from the n available nodes (usually, $f < \frac{n}{3}$). These *Byzantine* nodes have unlimited computational power and can behave arbitrarily, even colluding to "bring the system down". Much has been published relating to this model (for example, [11], [12], [2]), and many lower bounds have been proven.

One of the major drawbacks to the classic *Byzantine* adversarial model is its heavy performance cost. The running time required to reach agreement is linear in f (which usually means it is also linear in n), and the message complexity is typically at least $O(n^2)$ (when $f = O(n)$, see [7]). This drawback stems from the "global" nature of the classic *Byzantine* consensus problem - i.e., the *Byzantine* adversary has no restrictions on the spread of faulty nodes. Thus, the communication graph must have high connectivity (see [6] for exact bounds) - which leads to a high message load. Moreover, the global nature of the adversary requires every node to agree with every other node, leading to linear termination time and to a high out degree for each node (at least $2f + 1$ outgoing connections,

* Supported in part by ISF.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 167–181, 2008.

see [6]). Such algorithms cannot scale; thus - as computer networks grow - it becomes infeasible to address global *Byzantine* adversaries.

To overcome this limitation, some research has assumed computationally-bounded adversaries, for example [5]. Alternatively, recent work has considered a f_ℓ-local *Byzantine* adversary. The f_ℓ-local adversary is restricted in the nodes it can control. For every node p, the adversary can control at most f_ℓ neighboring nodes of p. We call this stronger model "local" and the classic model "global". [10] has considered the *Byzantine* broadcast problem, which consists of all non-*Byzantine* nodes accepting a message sent by a given node. [13] classifies the graphs in which the broadcast problem can be solved, in the presence of a f_ℓ-local adversary. In the current work we consider the *Byzantine* consensus problem, which consists of all non-*Byzantine* nodes agreeing on the same output value, which must be in the range of the input values of the non-*Byzantine* nodes. Sufficient conditions are given to classify graphs on which the problem can be solved, in the presence of a f_ℓ-local adversary.

Solving the *Byzantine* consensus problem when facing a global adversary requires $O(n)$ memory space. When considering a local adversary, the memory space requirements can be reduced. This raises the question of possible tradeoff between fault tolerance and memory space requirement, which has been previously investigated in the area of self-stabilizing (see [3], [8], [9]). The current work provides a new tradeoff: for a family of graphs, consensus can be solved using constant memory space, provided that the *Byzantine* adversary is f_ℓ-local.

Contribution: This work solves the *Byzantine* consensus problem in the local adversary model. For a large range of networks (a family of graphs with constant degree and logarithmic diameter), *Byzantine* consensus is solved within $O(\log n)$ rounds and with message complexity of $O(n \cdot \log n)$, while requiring $O(1)$ space at each node[1]. These results improve exponentially upon the classic setting which requires linear time, $O(n^2)$ messages, and (at least) linear space for reaching a consensus. We also present two additional results which are of special interest while using constant memory: first, we show a means of retaining identities in a local area of the network (see Section 4.1); second, we present a technique for waiting $\log n$ rounds using $O(1)$ memory space at each node (see Section 4.2).

2 Model and Problem Definition

Consider a synchronous distributed network of n nodes, $\{p_0, \ldots, p_{n-1}\} = \mathcal{P}$, represented by an undirected graph $G = (E, V)$, where $V = \mathcal{P}$ and $(p, p') \in E$ if p, p' are connected. Let $\Gamma(p)$ be the set of neighbors of p, including p. The communication network is assumed to be synchronous, and communication is done via message passing; each communication link is bi-directional.

Byzantine nodes have unlimited computational power and can communicate among themselves. The *Byzantine* adversary may control some portion of the nodes; however, the adversary is limited in that each node may not have more

[1] Formally, we show that each node uses space polynomial in its degree; for constant degree it is $O(1)$.

than f_ℓ *Byzantine* neighbors, where f_ℓ is a system-wide constant. Formally, a subset $S \subset \mathcal{P}$ is f_ℓ-local if for any node $p \in \mathcal{P}$ it holds that $|S \bigcap \Gamma(p)| \leq f_\ell$. A *Byzantine* adversary is said to be f_ℓ-local if (in any run) the set of *Byzantine* nodes is f_ℓ-local.

2.1 Problem Definition

The following is a formal definition of the *Byzantine* consensus problem, followed by a memory-bounded variant of it.

Definition 1. *The* ***Byzantine*** **consensus problem** *consists of the following: Each node p has an input value v_p from an ordered set $\mathcal{V}$; all nodes agree on the same output $V \in \mathcal{V}$ within a finite number of rounds (agreement), such that $v_p \leq V \leq v_q$ for some correct nodes p, q (validity).*

The goal of the current work is to show that for f_ℓ-local adversaries, efficient deterministic solutions to the *Byzantine* consensus problem exist; efficient with respect to running time, message complexity and space complexity. A node p's memory space depends solely on the local network topology - i.e. p's degree or the degree of p's neighbors.

Denote by ℓ-MaxDeg(p) the maximal degree of all nodes that are up to ℓ hops away from p. Notice that 0-MaxDeg$(p) = |\Gamma(p)|$.

Definition 2. *The* **ℓ-hop local *Byzantine*** *consensus problem is the* Byzantine *consensus problem with the additional constraint that each correct node p can use memory space that is polynomially bounded by ℓ-*MaxDeg(p).

Informally, the above definition states that a node cannot "know" about all the nodes in the system, even though it can access all its local neighbors. That is, consensus must be reached with "local knowledge" only.

Remark 1. Definition 2 can be replaced by a requirement that the out degree of each node as well as the memory space are constant. We choose the above definition instead, as it generalizes the constant-space requirement.

3 *Byzantine* Consensus

In this section sufficient conditions for solving the *Byzantine* consensus problem for a f_ℓ-local adversary are presented. First we discuss solving the *Byzantine* consensus problem in a network that is fully connected. We later give some definitions that allow us to specify sufficient conditions to ensure that *Byzantine* consensus can be solved in networks that are not fully connected.

3.1 Fully Connected *Byzantine* Consensus

We now define ByzCon, which solves the *Byzantine* consensus problem in a fully connected network. Take any *Byzantine* agreement[2] algorithm (for example, see

[2] *Byzantine* agreement is sometimes called *Byzantine* broadcast. This problem consists of a single leader broadcasting some value v using point-to-point channels.

[14]), and denote it by BA. ByzCon executes n instance of BA, one for each node p's input value, thus agreeing on the input value v_p. As a result, all correct nodes have an agreed vector of n input values (notice that this vector may contain "$\perp$" values when BA happens to return such a value for a *Byzantine* node). The output value of ByzCon is the median of the values of that vector, where "$\perp$" is considered as the lowest value.

Claim. ByzCon solves the *Byzantine* consensus problem in a fully-connected network.

Definition 3. *Given an algorithm $\mathcal{A}$ and a set of nodes $S \subset \mathcal{P}$,* Valid*($\mathcal{A}$, S) = true if the set S upholds the connectivity requirements of $\mathcal{A}$.* Faulty*($\mathcal{A}$, S) denotes the maximal number of faulty nodes that $\mathcal{A}$ can "sustain" when executed on S (for example,* Faulty$(BA, S) = \lfloor \frac{|S|-1}{3} \rfloor$*).*

In a similar manner, Time$(\mathcal{A}, S)$ *is the maximal number of rounds it takes $\mathcal{A}$ to terminate when executed on S, and* Msg$(\mathcal{A}, S)$ *is the maximal number of messages sent during the execution of $\mathcal{A}$ on S.*

Notice that for all S: Valid(ByzCon, S)= Valid(BA, S), Faulty(ByzCon, S)= Faulty(BA, S), Time(ByzCon, S) = Time(BA, S) Msg(ByzCon, S) = $|S| \cdot$ Msg(BA, S). That is, ByzCon's connectivity requirements, fault tolerance ratio and running time, are the same as in the *Byzantine* agreement algorithm of [14]; *i.e.,* ByzCon requires a fully connected graph among the nodes participating in ByzCon and it supports up to a third of them being *Byzantine*.

3.2 Sparsely Connected *Byzantine* Consensus

Consider a given algorithm that solves the *Byzantine* consensus problem when executed on a set of nodes S. ByzCon, defined in the previous section, requires S to be fully-connected. The following discussion assumes ByzCon's existence and correctness.

The following definitions hold with respect to any f_ℓ-local adversary.

Definition 4. *Given a subset $S \subset \mathcal{P}$, denote by f_ℓ-Byz(S) the maximal number of nodes from S that may be* Byzantine *for a f_ℓ-local adversary.*

Definition 5. *A non-empty subset S, $\emptyset \neq S \subset \mathcal{P}$, is a f_ℓ-***decision group** *if* Valid*(*ByzCon*, S) = true and* Faulty(ByzCon, S) $\geq$ *f_ℓ-Byz(S).*

When considering ByzCon as constructed in Section 3.1, the above definition states that S must be fully connected, and $|S| > 3 \cdot f_\ell$. Since 0-local adversaries are of no interest, the minimal size of S satisfies, $|S| \geq 4$.

Definition 6. *A non-empty subset $S' \subseteq S$ of a f_ℓ-decision group is a f_ℓ-***common** *source if f_ℓ-Byz$(S') + |S - S'| \leq$* Faulty(ByzCon, S)*.*

Claim. If $S' \subseteq S$ is a common source and all correct nodes in S' have the same initial value, ν, then the output value of ByzCon when executed on S, is ν.

Proof. Denote by W the set of correct nodes in S' and by Y the set of all nodes in S that are not in W; *i.e.,* $Y := S - W$. Since S' is a common source, $|Y| \leq \textsc{Faulty}(\textsc{ByzCon}, S)$. Assume by way of contradiction that all nodes in W have the same initial value ν, and the output of ByzCon (when executed on S) is not ν; denote this execution by $\mathcal{R}$. If all the nodes in Y are *Byzantine* and they simulate their part of $\mathcal{R}$, then it holds that all correct nodes in S have the same initial value ν, the number of *Byzantine* nodes is less than $\textsc{Faulty}(\textsc{ByzCon}, S)$, and yet the output is not ν. In other words, the "validity" of ByzCon does not hold. Therefore, if all nodes in $W = S - Y$ are correct and have the same initial value ν, that must be the output value of ByzCon when executed on S. □

Definition 7. *Two subsets $S_1, S_2 \subset \mathcal{P}$ are f_ℓ-***connected** *if S_1, S_2 are f_ℓ-decision groups, and if $S_1 \cap S_2$ is a f_ℓ-common source for both S_1 and S_2.*

Definition 8. *A list $C = S_1, S_2, \ldots, S_l$ is a f_ℓ-***value-chain** *between S_1 and S_l if the subsets S_i, S_{i+1} are f_ℓ-connected, for all $1 \leq i \leq l-1$. The* **length** *of the value-chain C is $l-1$.*

Definition 9. *Let $\mathcal{G}$ be a set of f_ℓ-decision groups,* i.e., *$\mathcal{G} \subseteq 2^{\mathcal{P}}$. $\mathcal{G}$ is an f_ℓ-***entwined structure** *if for any two subsets $g, g' \in \mathcal{G}$ there is a f_ℓ-value-chain $C = g_1, g_2, \ldots, g_l \in \mathcal{G}$ between g, g'. The* **distance** *between g, g' is the minimal length among all f_ℓ-value-chains between g, g'.*

Definition 10. *The* **diameter** *$\mathcal{D}$ of an f_ℓ-entwined structure $\mathcal{G}$ is the maximal distance between any two f_ℓ-decision groups $g, g' \in \mathcal{G}$.*

Definition 11. *An f_ℓ-entwined structure $\mathcal{G}$ is said to* **cover** *graph G if for any node p in G there is a subset $g \in \mathcal{G}$ such that $p \in g$. Formally: $\bigcup_{g \in \mathcal{G}} \{g\} = \mathcal{P}$.*

Remark 2. All entwined structures in the rest of this paper are assumed to cover their respective graphs. We will therefore sometimes say "an entwined structure $\mathcal{G}$" instead of "an entwined structure $\mathcal{G}$ that covers graph G".

Some of the above definitions were parameterized by f_ℓ. When f_ℓ is clear from the context, we will remove the prefix f_ℓ. (*i.e.,* "decision group" instead of "f_ℓ-decision group").

Definition 12. *Let G be a graph. Denote by $\Phi(G)$ the maximal value f_ℓ s.t. there is a f_ℓ-entwined structure that covers G.*

The following theorem states the first result of the paper.

Theorem 1. *The* Byzantine *consensus problem can be solved on graph G for any $\Phi(G)$-local adversary.*

Section 3.3 contains the algorithm LocalByzCon that given a f_ℓ-entwined structure $\mathcal{G}$, solves the *Byzantine* consensus problem for any f_ℓ-local adversary. Section 3.4 proves the correctness of LocalByzCon, thus completing the proof of Theorem 1.

```
Algorithm LocalByzCon                                          /* executed at node $p$ */
                                              /* $\mathcal{BC}_i$ is an instance of ByzCon */
Initialization:
  1. set $v_p := p$'s initial value;
  2. set $Output_p := \emptyset$;                               /* $\mathcal{G}_p := \{g \in \mathcal{G} | p \in g\}$ */
  3. for all $g_i \in \mathcal{G}_p$ start executing $\mathcal{BC}_i$ with initial value $v_p$;

For $\Delta_{max} \cdot (2\mathcal{D} + 1)$ rounds:          /* $\Delta_{max} := max_i\{\text{TIME}(\text{ByzCon}, g_i)\}$ */
  1. execute a single round of each $\mathcal{BC}_i$ that $p$ participates in;
  2. for each $\mathcal{BC}_i$ that terminated in the current round with value $V$:
       set $Output_p := Output_p \bigcup \{V\}$;
  3. for each $\mathcal{BC}_i$ that terminated in the current round:
       start executing $\mathcal{BC}_i$ with initial value $\min\{Output_p\}$;

Return value:
   return output as $\min\{Output_p\}$;
```

Fig. 1. Solving the *Byzantine* consensus problem for a f_ℓ-local adversary

3.3 Algorithm LocalByzCon

Figure 1 introduces the algorithm LocalByzCon that solves the *Byzantine* consensus problem for f_ℓ-local adversaries, given an f_ℓ-entwined structure $\mathcal{G}$. The main idea behind LocalByzCon is to execute ByzCon locally in each decision group. Each node takes the minimal agreement value among the decision groups it participated in. The fact that any two decision groups in $\mathcal{G}$ have a value-chain between them ensures that the minimal agreement value among the different invocations of ByzCon will propagate throughout $\mathcal{G}$. Since $\mathcal{G}$ covers G, all nodes will eventually receive the same minimal value.

Consider $\mathcal{G}$ to be a f_ℓ-entwined structure, and $g_1, g_2, \ldots, g_m \in \mathcal{G}$ to be all the decision groups (of $\mathcal{P}$) in $\mathcal{G}$. For a node p, $\mathcal{G}_p$ is the set of all decision groups that p is a member of; that is, $\mathcal{G}_p := \{g \in \mathcal{G} | p \in g\}$. Each node p participates in repeated concurrent executions of ByzCon instances, where for each $g_i \in \mathcal{G}_p$, node p will execute a ByzCon instance, and once that instance terminates p will execute another instance, etc. For each $g_i \in \mathcal{G}_p$ denote by $\mathcal{BC}_i^1$ the first execution of ByzCon by all nodes in g_i; $\mathcal{BC}_i^2$ denotes the second instance executed by all nodes in g_i, and so on.

According to Definition 1 all nodes participating in ByzCon terminate within some finite time Δ. Furthermore, each node can wait until Δ rounds elapse and terminate, thus achieving simultaneous termination. Therefore, in LocalByzCon, all nodes that participate in $\mathcal{BC}_i^j$ terminate it at the same round and start executing $\mathcal{BC}_i^{j+1}$ together at the following round.

3.4 Correctness Proof

For every $g_i \in \mathcal{G}$ denote by $r_i(1)$ the round at which the first consecutive instance of ByzCon executed on g_i has terminated. Denote by $r(1) := \max\{r_i(1)\}$. Let $Output_p^r$ denote the value of $Output_p$ at the end of round r. Notice that $Output_p^r \subseteq Output_p^{r+1}$ for all correct p and all r. Denote $Output^r := \bigcup\{Output_p^r\}$, the union of all $Output_p$ (for correct p) at the end of some round r. Using this notation, $Output^{r(1)}$ represents all the values in any $Output_p$ (for correct p) after at least one instance of ByzCon has terminated for each $g \in \mathcal{G}$. Consider some instance of ByzCon that terminates after round $r(1)$: it must be (at least) a second execution of that instance, thus all correct nodes that participated in it had their input values chosen from $Output^{r(1)}$. Thus, due to *validity*, the output value is in the range of $[\min\{Output^{r(1)}\}, \max\{Output^{r(1)}\}]$. Hence, we conclude that $\min\{Output^r\} \geq \min\{Output^{r(1)}\}$ for any $r \geq r(1)$. Denote by $v_{min} := \min\{Output^{r(1)}\}$; clearly no correct node p will hold a lower value (in $Output_p$) for any $r \geq r(1)$.

Lemma 1. *If a correct node p has $\min\{Output^r_p\} = v_{min}$ then it will never have a lower value in $Output_p$ for any round $r' \geq r$.*

Lemma 2. *At round $r(1)$, there exists $g_i \in \mathcal{G}$ such that for every correct node $p \in g_i$ it holds that $\min\{Output_p\} = v_{min}$.*

Proof. By definition, $v_{min} \in Output^{r(1)}_p$. Thus, v_{min} was the output of some $\mathcal{BC}_i$ instance (on decision group g_i) at some round $r \leq r(1)$. Consider the nodes in g_i, they have all added v_{min} to their $Output_p$. Thus, $v_{min} \in Output^{r(1)}_p$ and by definition it is the lowest value in $Output_p$ at round $r(1)$. Thus, at round $r(1)$, all correct nodes in g_i have $\min\{Output_p\} = v_{min}$. □

Lemma 3. *If* LocalByzCon *has been executed for at least $\Delta_{max} \cdot (2\mathcal{D} + 1)$ rounds, then all correct nodes have $\min\{Output_p\} = v_{min}$.*

Proof. Divide the execution of LocalByzCon into "stages" of Δ_{max} rounds each. Clearly there are at least $2\mathcal{D} + 1$ stages, and in each stage each $\mathcal{BC}_i$ is started at least once. From the above lemma, for some decision group g_i, all correct nodes $p \in g_i$ have $\min\{Output_p\} = v_{min}$ at the end of the first stage.

Let g' be some decision group, and let $g_i = g_1, g_2, \ldots, g_l = g'$ be a value-chain between g_i and g'; there exists such a value-chain because $\mathcal{G}$ is an entwined structure, and its length is $\leq \mathcal{D}$.

Consider the second stage. Since g_1, g_2 are connected, and since all nodes in g_1 have the same initial value (v_{min}) during the entire stage 2, then in $g_1 \cap g_2$ all nodes have v_{min} as their initial value during stage 2. Since $g_1 \cap g_2$ is a common source of g_2, it holds that instance $\mathcal{BC}_2$ that is started in stage 2 is executed with all nodes in $g_1 \cap g_2$ having initial value v_{min}. Thus, when $\mathcal{BC}_2$ terminates (no later than the end of stage 3), it terminates with the value v_{min}, thus all nodes in g_2 also have $v_{min} \in Output_p$. By Lemma 1, all nodes in g_2 choose v_{min} as their initial value for any instance of ByzCon started during stage 4 and above. Repeating this line of proof leads to the conclusion that after an additional $2\mathcal{D}$ stages all correct nodes in g' have $v_{min} \in Output_p$.

Since any decision group g' has a value-chain of no more than $\mathcal{D}$ length to g_i, we have that after $2\mathcal{D} + 1$ stages, all correct nodes in all decision groups have $v_{min} \in Output_p$. Since $\mathcal{G}$ covers G, each correct node is a member of some decision group, thus all correct nodes in G have $v_{min} \in Output_p$. Since v_{min} is the lowest possible value, $\min\{Output_p\} = v_{min}$ for all $p \in \mathcal{P}$. □

Remark 3. Consider the value-chain $g_1, g_2, \ldots, g_l$ in the proof above. The proof bounds the time (in rounds) it takes v_{min} to "propagate" from g_1 to g_l. The given bound $(2 \cdot l \cdot \Delta_{max})$ is not tight. In fact, instead of assuming $2 \cdot \Delta_{max}$ rounds for each "hop along the chain", one can accumulate $2 \cdot \textsc{Time}(\textsc{ByzCon}, g_i)$ when moving from g_{i-1} to g_i. Thus, define $\textsc{Time}_{dist}(g, g')$ to be the shortest such sum on any value-chain between g, g', and $\mathcal{D}_{\text{TIME}}$ as the maximal $\textsc{Time}_{dist}(g, g')$ on any $g, g' \in \mathcal{G}$; and we can conclude that it is enough to run LocalByzCon for $\mathcal{D}_{\text{TIME}}$ rounds. Notice that this analysis it tight up to a factor of 2.

Lemma 4. *Given an f_ℓ-entwined structure $\mathcal{G}$ that covers G,* LocalByzCon *solves the* Byzantine *consensus problem, for a f_ℓ-local adversary.*

Proof. From the above lemmas, after $\Delta_{max} \cdot (2\mathcal{D} + 1)$ rounds, all correct nodes terminate with the same value, v_{min}. Notice that v_{min} is the output of some ByzCon instance. Thus, there are two correct nodes p, q such that $v_p \leq v_{min} \leq v_q$. Therefore, both "agreement" and "validity" hold. □

3.5 Complexity Analysis

The time complexity of LocalByzCon is $(2\mathcal{D} + 1) \cdot \Delta_{max}$. In other words, let g_{max} be the largest decision group in $\mathcal{G}$, that is $g_{max} := argmax_{g_i \in \mathcal{G}}\{|g_i|\}$; using this terminology we have that, $\textsc{Time}(\textsc{LocalByzCon}, \mathcal{P}) = (2\mathcal{D} + 1) \cdot O(g_{max})$.

Similarly, the message complexity of LocalByzCon per round is the sum of all messages of all ByzCon instances each round, which is bounded by $\sum_{g_i} |g_i|^3 \leq |\mathcal{G}| \cdot |g_{max}|^3$. Thus, $\textsc{Msg}(\textsc{LocalByzCon}, \mathcal{P}) \leq (2\mathcal{D} + 1) \cdot |\mathcal{G}| \cdot O(|g_{max}|^4)$ (messages per round times rounds).

4 Constant-Space *Byzantine* Consensus

Section 3 proves a sufficient condition for solving the *Byzantine* consensus problem on a given graph G for a f_ℓ-local adversary. The current section gives a sufficient condition for solving the ℓ-hop local *Byzantine* consensus problem.

Definition 13. *An f_ℓ-entwined structure $\mathcal{G}$ is called* **ℓ-hop local** *if for every $g \in \mathcal{G}$, ℓ bounds the distance between any two nodes in g.*

For ByzCon constructed in Section 3.1, any entwined structure $\mathcal{G}$ is 1-hop local, since for every decision group $g \in \mathcal{G}$, it holds that $\textsc{Valid}(\textsc{ByzCon}, g) = true$, thus g is fully connected. However, the following discussion holds for any algorithm that solves the *Byzantine* consensus problem, even if it does not require decision groups to be fully connected.

Definition 14. *An f_ℓ-entwined structure $\mathcal{G}$ is called* **ℓ-lightweight** *if $\mathcal{G}$ is ℓ-hop local and for every $p \in \mathcal{P}$, $|\mathcal{G}_p|$ and $|g|$ (for all $g \in \mathcal{G}_p$) are polynomial in $|\Gamma(p)|$.*

Definition 15. *Let G be a graph. Denote by ℓ-$\Psi(G)$ the maximal value f_ℓ s.t. there is a f_ℓ-entwined structure that is ℓ-lightweight and covers G.*

The following theorem states the second contribution of this paper.

Theorem 2. *The ℓ-hop local* Byzantine *consensus problem can be solved on graph G for any $[\ell\text{-}\Psi(G)]$-local adversary.*

By Theorem 1, any f_ℓ-entwined structure $\mathcal{G}$ that covers graph G can be used to solve the *Byzantine* consensus problem on G, for a f_ℓ-local adversary. To prove Theorem 2 we show that when LocalByzCon is executed using an ℓ-lightweight f_ℓ-entwined structure, each node p's space requirements are polynomial in ℓ-MaxDeg(p). There are 3 points to consider: first, we show that the memory footprint of the different ByzCon instances that p participates in is "small". Second, ByzCon assumes unique identifiers, which usually require $\log n$ space. We need to create identifiers that are locally unique (thus requiring space that is independent of n), such that ByzCon can be executed properly on each decision group. Third, the main loop of LocalByzCon requires to count at least up to $\mathcal{D}$, which requires $\log \mathcal{D}$ bits, possibly requiring space that is dependent on n.

The second point is discussed in Section 4.1 and the third in Section 4.2. To solve the first point notice that $\mathcal{G}$ is ℓ-lightweight, thus each node participates in no more than $poly(|\Gamma(p)|)$ ByzCon instances concurrently, and each such instance contains $poly(|\Gamma(p)|)$ nodes. Assuming that the identifiers used by ByzCon require at most $polylog(\ell\text{-}\textsc{MaxDeg}(p))$ bits (see Section 4.1), p requires at most $poly(\ell\text{-}\textsc{MaxDeg}(p))$ space to execute the ByzCon instances of LocalByzCon.

4.1 Locally Unique Identifiers

Consider the algorithm LocalByzCon in Figure 1 and an ℓ-lightweight entwined structure $\mathcal{G}$. Different nodes communicate only within decision groups, *i.e.,* node p sends or receives messages from node q only if $p, q \in g$ for some $g \in \mathcal{G}$. Thus, the identifiers used can be locally unique.

To achieve this goal, node p numbers each node q in each decision group $g \in \mathcal{G}_p$, sequentially. Notice that the same node q might "receive" different numbers in different decision groups that p participates in. Thus, we can define Num(p, g, q) to be the number p assigns to q for the decision group $g \in \mathcal{G}_p$. Notice that for an ℓ-lightweight entwined structure, Num$(p, *, *)$ requires polynomial space in $|\Gamma(p)|$. Each node z holds Num$(p, g, *)$ for all $g \in \mathcal{G}_z \bigcap \mathcal{G}_p$, along with a mapping between Num(p, g, q) and Num(z, g, q). The memory footprint of this mapping is again polynomial in $|\Gamma(z)|$ (for ℓ-lightweight entwined structures).

In addition, each node p numbers the decision groups it is a member of: let Indx(p, g) be the "number" of $g \in \mathcal{G}_p$ according to p's numbering. Any node z (such that $\mathcal{G}_p \bigcap \mathcal{G}_z \neq \phi$) holds a mapping between Indx(p, g) and Indx(z, g), for all $g \in \mathcal{G}_p \bigcap \mathcal{G}_z$. Notice that Indx$(p, *)$ is polynomial in $|\Gamma(p)|$, and the mapping requires memory space of size polynomial in $max\{|\Gamma(p)|, |\Gamma(z)|\}$. For ℓ-lightweight entwined structures, the distance between p and z is $\leq \ell$, thus $max\{|\Gamma(p)|, |\Gamma(z)|\} \leq \ell$-MaxDeg$(p)$, resulting in a memory space footprint (of all the above structures) that is polynomial in ℓ-MaxDeg(p) for any node p.

When node p wants to send node q's identifier to z regarding decision group g (notice that $p, q, z \in g$ and $g \in \mathcal{G}_p, \mathcal{G}_q, \mathcal{G}_z$), it sends "$(\textsc{Num}(p, g, q), \textsc{Indx}(p, g))$" and node z uses its mapping to calculate $\textsc{Indx}(z, g)$, from which node z can discern what g is, and use its Num mapping to calculate $\textsc{Num}(z, g, q)$. Therefore, nodes can identify any node in their decision groups and can communicate these identities among themselves. Thus, "identities" can be uniquely used locally, with a low memory footprint. *i.e.,* the required memory space is polynomial in ℓ-$\textsc{MaxDeg}(p)$. Notice that the above structures are constructed before executing LocalByzCon, once the system designer knows the structure of $\mathcal{G}$.

4.2 Memory-Efficient Termination Detection

LocalByzCon as given in Figure 1 loops for $\Delta_{max} \cdot (2\mathcal{D}+1)$ rounds. Therefore, for $\mathcal{D}$ that depends on n, the counter of the loop will require too much memory. From the analysis of the execution of LocalByzCon it is clear that any node terminating after it ran for more than $\Delta_{max} \cdot (2\mathcal{D}+1)$ rounds, terminates with the same value. Thus, it is only important that all nodes eventually terminate, and that they do so after at least $\Delta_{max} \cdot (2\mathcal{D}+1)$ rounds; how can this be done without counting rounds? The following is an example of a solution requiring constant memory, using a simpler model.

Consider a synchronous network without any *Byzantine* nodes, where each node p has $poly(|\Gamma(p)|)$ memory. Given that $\mathcal{D}$ (the diameter of the network) is not constant, how can one count until $\mathcal{D}$ in such a network? Mark two nodes u, v as "special" nodes, such that the distance between u and v is $\mathcal{D}$ (clearly there exist u, v that satisfy this condition). When the algorithm starts, u floods the network with a *"start"* message. When v receives this message, it floods the network with an *"end"* message. When any node receives the *"end"* message, it knows that at least $\mathcal{D}$ rounds have passed, and it can therefore terminate.

Using the above example, we come back to our setting of entwined structures and f_ℓ-local *Byzantine* adversaries: consider two "special" decision groups g_1, g_2 from $\mathcal{G}$, such that the $\textsc{Time}_{dist}$ between g_1 and g_2 is $\mathcal{D}_{\textsc{time}}$ (see Remark 3). Each node p, in addition to its initial value v_p, has two more initial values v_p^1, v_p^2 which are both set to "1". All nodes in g_1 set $v_p^1 :=$"0". Instead of executing a single LocalByzCon, each node executes 3 copies of LocalByzCon: one on v_p, one on v_p^1 (denoted $\textsc{LocalByzCon}_1$) and one on v_p^2 (denoted $\textsc{LocalByzCon}_2$). Only nodes in g_2 perform the following rule: once g_2's output in $\textsc{LocalByzCon}_1$ is "0", set $v_p^2 :=$ "0". Lastly, all nodes terminate one repetition after $Output_p$ of $\textsc{LocalByzCon}_2$ contains "0".

The analysis of this addition is simple: the value "0" in $\textsc{LocalByzCon}_1$ propagates throughout the network until it reaches g_2. Once it reaches g_2, the value "0" of $\textsc{LocalByzCon}_2$ propagates throughout the network, causing all nodes to terminate. Notice that before g_2 updates its input value of $\textsc{LocalByzCon}_2$, no correct node will have a value of "0" for $\textsc{LocalByzCon}_2$. Lastly, notice that at least $\frac{1}{2}\mathcal{D}_{\textsc{time}}$ rounds must pass before g_2 changes the input value to the third LocalByzCon (see Remark 3). Thus, if the second and third LocalByzCon are executed “at half speed” (every round, the nodes wait one round), then all correct nodes terminate not before $\mathcal{D}_{\text{TIME}}$ rounds pass, and no later than $2\mathcal{D}_{\text{TIME}}$ rounds.

The above schema requires the same memory space as the “original” LocalByzCon (i.e. independent of n), up to a constant factor, while providing termination detection, as required.

The decision groups g_1, g_2 must be selected prior to LocalByzCon's execution. An alternative option is to select g_1 using some leader election algorithm, and then use an MST algorithm to find g_2. However, in addition to *Byzantine* tolerance, these algorithms' memory requirements must not depend on n, which means that global identifiers cannot be used. There is a plethora of research regarding leader election / spanning trees and their relation to memory space (see [1], [3], [4], [15]). However, as far as we know, there are no lower or upper bounds regarding the exact model this work operates in (e.g. local identities, but no global identities). Thus, it is an open question whether it is required to choose g_1, g_2 a priori, or if they can be chosen at runtime.

4.3 Complexity Analysis

Consider graph G with maximal degree d_{max}, and an ℓ-lightweight entwined structure $\mathcal{G}$ (with diameter $\mathcal{D}$) that covers G. Denote $g_{max} := argmax_{g_i \in \mathcal{G}}\{|g_i|\}$, since $\mathcal{G}$ is ℓ-lightweight, g_{max} is polynomial in d_{max}, and $|\mathcal{G}| = n \cdot poly(d_{max})$. Therefore, by Section 3.5, LocalByzCon's time complexity is $O(\mathcal{D}) \cdot poly(d_{max})$, and its message complexity is $O(\mathcal{D}) \cdot n \cdot poly(d_{max})$.

From the above, if G's maximal degree is independent of n, and if $\mathcal{D} = O(\log n)$, then the running time of LocalByzCon is $O(\log n)$ and its message complexity is $O(n \log n)$, while using $O(1)$ space. (Section 5 shows entwined structures that reach these values). This achieves an exponential improvement on the time and message complexity of *Byzantine* consensus in the “global” model, which are $O(n)$ and $O(n^2)$ respectively. [3]

5 Constructing 1-Lightweight Entwined Structures

In this section we present a family of graphs for which 1-lightweight entwined structures exist. The family of graphs is given in a constructive way: for each graph $G' = (V', E')$ (where $|V'| = n'$) and for any value of f_ℓ ($f_\ell << n'$) we construct a graph $G = (V, E)$ ($|V| = n$) with a 1-lightweight f_ℓ-entwined structure $\mathcal{G}$. Our construction achieves $|\mathcal{G}| = O(n)$ and $|g|$ is small for all $g \in \mathcal{G}$, thus ensuring that $\mathcal{G}$ is indeed 1-lightweight. Furthermore, G's diameter and maximal degree are a function of those of G's, thus ensuring that for G' with bounded degree and logarithmic diameter, G has similar properties.

First we show how to construct an entwined structure, given a graph with “ring topology”. Then we show how “to combine” two graphs with ring topologies. Lastly, for every graph G', we create a graph G that “blows up” each node p

[3] In fact, known algorithms that use the transformation in [6] to operate in not-fully-connected graphs might even require $O(n^3)$ messages.

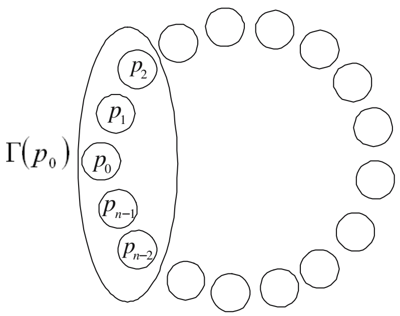


Fig. 2. p_0's neighbors in an extended ring topology of order 2

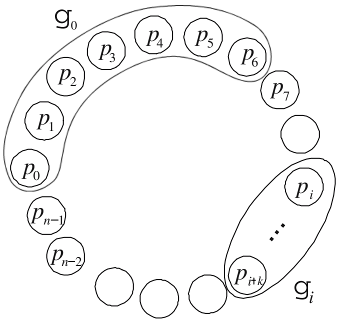


Fig. 3. g_0 for $f_\ell = 1$ and $k = 3(f_\ell + 1) = 6$

in G' to be a ring (containing $O(|\Gamma(p)|)$ nodes), and then combines the different rings of G'.

5.1 A Simple "Ring" Entwined Structure

This section presents a simple construction of an entwined structure $\mathcal{G}$ for a ring topology network. The constructed $\mathcal{G}$ has $|g_{max}|$ constant (independent of n), but a diameter of $O(n)$.

Definition 16. *A graph $G = (V, E)$ is said to have an* **extended ring topology** *of order k, if $E = \bigcup_{p_i \in V} \bigcup_{1 \leq j \leq k} \{(p_i, p_{i+j})\}$ (addition is done modulo n).*

Informally, an extended ring topology of order k means that each node is connected to the k neighbors "ahead" of it in the ring; since G is bi-directional, each node is also connected to the k neighbors "behind" it (see Figure 2 for an example). Notice that a "regular" ring topology is actually an extended ring topology of order 1.

Consider a graph $G = (V, E)$ with extended ring topology of order $k = 3 \cdot (f_\ell + 1)$. Define $\mathcal{G}$ as follows: $g_i := \{p_i, p_{i+1}, p_{i+2}, \ldots, p_{i+k}\}$ (see Figure 3).

Claim. Any $g \in \mathcal{G}$ is af_ℓ-decision group.

Claim. For any i, g_i and g_{i+1} are f_ℓ-connected.

Lemma 5. *$\mathcal{G}$ is an f_ℓ-entwined structure with diameter at most n, and $|g| = 3 \cdot f_\ell + 4$ for any $g \in \mathcal{G}$.*

Proof. By definition of $g_i \in \mathcal{G}$, $|g_i| = k + 1 = 3 \cdot f_\ell + 4$. Let $g_i, g_j \in \mathcal{G}$ be some decision groups. $0 \leq i, j \leq n - 1$ and either $i \geq j$ or $j \geq i$. Assume w.l.o.g. that $j \geq i$. Consider the list $C := g_i, g_{i+1}, \ldots, g_j$. By the previous claim, g_i and g_{i+1} are f_ℓ-connected, and therefore C is a value chain. Thus, there exists a f_ℓ-value chain between any two decision groups in $\mathcal{G}$ of length at most n. Thus, $\mathcal{G}$ is an f_ℓ-entwined structure with diameter at most n. □

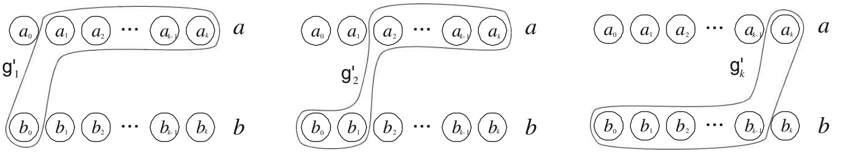

a_0 a_1 a_2 ··· a_{k-1} a_k a
g'_1
g'_2
g'_k
b_0 b_1 b_2 ··· b_{k-1} b_k b

Fig. 4. The original decision groups a, b and the constructed decision groups g'_1, g'_2 and g'_k

5.2 Connecting Rings Together

Consider two extended ring topology graphs G_1, G_2 and their respective f_ℓ-entwined structures $\mathcal{G}_1, \mathcal{G}_2$ (as built in the previous subsection). Let $a \in \mathcal{G}_1$ and $b \in \mathcal{G}_2$ be 2 decision groups that are fully connected and are of the same size, e.g. $|a| = |b| = k + 1$.

Mark by a_i the nodes in a and by b_i the nodes in b (for $0 \leq i \leq k$). Define $g'_j := \{a_j, a_{j+1}, \ldots, a_k, b_0, \ldots, b_{j-1}\}$ for $1 \leq j \leq k$; see Figure 4 for an example. Add edges such that each g'_j is fully connected. Notice that $|g'_j| = k+1$ and that for $1 \leq j < k$, it holds that g'_j and g'_{j+1} are f_ℓ-connected. In addition a and g'_0 are f_ℓ-connected, as well as b and g'_k.

Take $\mathcal{G}' = \mathcal{G}_1 \bigcup \mathcal{G}_2 \bigcup \{g'_j\}$. $\mathcal{G}'$ is a f_ℓ-entwined structure over the graph that is the union of G_1, G_2 with edges that are induced by the different g'_js. Notice that the diameter of $\mathcal{G}'$ is the diameter of $\mathcal{G}_1$ plus the diameter of $\mathcal{G}_2$ plus $k + 1$ (the length of the distance between a and b). Furthermore, $|\mathcal{G}'| = |\mathcal{G}_1| + |\mathcal{G}_2| + k$. In addition every node $p \in a \bigcup b$ participates in no more than k additional decision groups.

5.3 General Entwined Structure Construction

Let G' be any graph on n' nodes, and let d_v be the degree of node v; denote $d_{max} := max_{v \in G'} d_v$. Consider a graph G_v per node v with $d_v \cdot k$ nodes, such that G_v has an extended ring topology of order k. Consider the "ring" entwined structure constructed in the previous sections $\mathcal{G}_v$ with decision groups denoted as $g_1(v), g_2(v), \cdots$. Notice that $g_i(v) \cap g_{i+k}(v) = \phi$ (see Figure 5). For each node $v \in G'$, consider the list of its neighbors $\Gamma(v) = v_1, v_2, \ldots, v_{d_v}$, and define a function $N(u, v)$ that returns the index of v as a neighbor of u; *i.e.,* $N(v, v_i) = i$.

Using the operation defined in the previous subsection: for any edge (u, v) in G', "connect" the rings in the following way: $g_{N(u,v)*k}(u)$ with $g_{N(v,u)*k}(v)$. That is, let v be the ith neighbor of u ($N(u, v) = i$ or $u_i = v$), and u be the jth neighbor of v ($N(v, u) = j$, $v_j = u$) (see Figure 6); the following two decision groups are to be combined: $g_{i*k}(u)$ with $g_{j*k}(v)$. Denote the new decision groups created due to each such "connection" as $\mathcal{G}_{u,v}$.

Consider G to be the union of all G_v (both nodes and edges), and let $\mathcal{G}$ be the union of the entwined structures. That is, $\mathcal{G} = (\bigcup_{v \in G'} \mathcal{G}_v) \cup (\bigcup_{u,v \in G'} \mathcal{G}_{u,v})$.

Claim. Let $g \in \mathcal{G}_v$ and $g' \in \mathcal{G}_u$ such that (v, u) is an edge in G', then there is a f_ℓ-value-chain between g and g' of length $\leq (d_{max} + 1) \cdot k$.

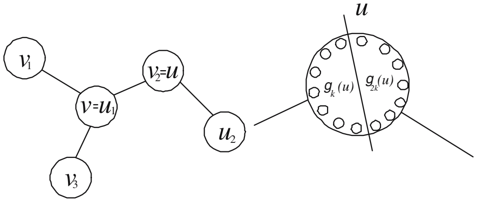


Fig. 5. A graph with 5 nodes, and node u's "ring" $\mathcal{G}_u$ and 2 decision groups $g_k(u), g_{2k}(u)$

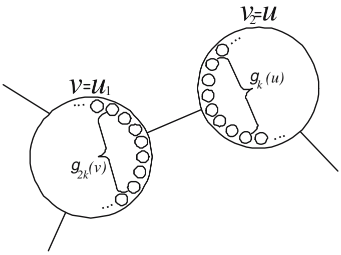


Fig. 6. v, u connecting, combining $g_{2k}(v), g_k(u)$

Claim. $\mathcal{G}$ is a f_ℓ-entwined structure with diameter $\mathcal{D} \leq D_{G'} \cdot (d_{max} + 1) \cdot k + 2k$, where $D_{G'}$ is the diameter of G'.

Theorem 3. *$\mathcal{G}$ is a 1-lightweight f_ℓ-entwined structure.*

5.4 Analysis

The above construction shows that for any graph G' there is a graph G and an f_ℓ-entwined structure $\mathcal{G}$ (covering G) s.t. the diameter of $\mathcal{G}$ is linear in the diameter of G', the maximal degree of G', and in k. Thus, by taking G' to be any graph with constant degree and logarithmic diameter, we have that the diameter of $\mathcal{G}$ is $O(D_{G'} \cdot k)$. In other words, $\mathcal{D} = O(k \cdot logn)$; by taking k to be constant as well (this means that the graph G has a constant degree), we have that $\mathcal{D} = O(logn)$. Therefore, the above construction produces graphs and their respective entwined structures, such that the diameter is logarithmic and the degree is constant, fulfilling the promise of Section 4.3.

6 Conclusion and Future Work

A sufficient condition for solving *Byzantine* consensus in the presence of a localized *Byzantine* adversary was given. Furthermore, for a family of graphs it was shown how to solve the *Byzantine* consensus problem using memory space that is constant (independent of the size of the network).

We consider few points for future research: first, find tighter bounds for when *Byzantine* consensus can be solved on a graph G. Second, allow for dynamic changes in the system, such as nodes joining or leaving or even constructing the entwined structure on-the-fly. Third, adapt the entwined structure-construction such that if some portion of the network does not uphold the required *Byzantine*-to-correct threshold, then the rest of the network may still reach consensus. Lastly, it would be interesting to find other constructions of entwined structures and see what additional types of graphs can solve *Byzantine* consensus.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments.

References

1. Afek, Y., Stupp, G.: Optimal time-space tradeoff for shared memory leader election. J. Algorithms 25(1), 95–117 (1997)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons, Chichester (2004)
3. Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. In: PODC 1999: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing, pp. 199–207. ACM, New York (1999)
4. Beauquier, J., Gradinariu, M., Johnen, C.: Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Distributed Computing 20(1), 75–93 (2007)
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (2002)
6. Dolev, D.: The byzantine generals strike again. Journal of Algorithms 3, 14–30 (1982)
7. Dolev, D., Reischuk, R.: Bounds on information exchange for byzantine agreement. J. ACM 32(1), 191–204 (1985)
8. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
9. Dolev, S., Gouda, M.G., Schneider, M.: Memory requirements for silent stabilization. In: PODC 1996: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, pp. 27–34. ACM, New York (1996)
10. Koo, C.-Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: PODC 2004: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, pp. 275–282. ACM, New York (2004)
11. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 301–382 (1982)
12. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
13. Pelc, A., Peleg, D.: Broadcasting with locally bounded byzantine faults. Inf. Process. Lett. 93(3), 109–115 (2005)
14. Toueg, S., Perry, K.J., Srikanth, T.K.: Fast distributed agreement. SIAM Journal on Computing 16(3), 445–457 (1987)
15. Yamashita, M., Kameda, T.: Computing on anonymous networks. i. characterizing the solvable cases. Parallel and Distributed Systems, IEEE Transactions 7(1), 69–89 (1996)

Optimistic Erasure-Coded Distributed Storage*

Partha Dutta[1], Rachid Guerraoui[2], and Ron R. Levy[2]

[1] IBM India Research Lab, Bangalore, India
[2] EPFL IC, Lausanne, Switzerland

Abstract. We study erasure-coded atomic register implementations in an asynchronous crash-recovery model. Erasure coding provides a cheap and space-efficient way to tolerate failures in a distributed system. This paper presents ORCAS, Optimistic eRasure-Coded Atomic Storage, which consists of two separate implementations, ORCAS-A and ORCAS-B. In terms of storage space used, ORCAS-A is more efficient in systems where we expect large number of concurrent writes, whereas, ORCAS-B is more suitable if not many writes are invoked concurrently. Compared to replication based implementations, both ORCAS implementations significantly save on the storage space. The implementations are optimistic in the sense that the used storage is lower in synchronous periods, which are considered common in practice, as compared to asynchronous periods. Indirectly, we show that tolerating asynchronous periods does not increase storage overhead during synchronous periods.

1 Introduction

1.1 Motivation

Preventing data loss in storage devices is one of the most critical requirements in any storage system. Enterprise storage systems in particular have multiple levels of redundancy built in for fault tolerance. The cost of a specialized centralized storage server is very high and yet it does not offer protection against unforseen consequences such as fires and floods. Distributed storage systems based on commodity hardware, as alternatives to their centralized counterparts, have gained in popularity since they are cheaper, can be more reliable and offer better scalability. However, implementing such systems is more complicated due to their very distributed nature.

Most existing distributed storage systems rely on data replication to provide fault tolerance [15]. Recently however, it has been argued that erasure coding is a better alternative to data replication since it reduces the cost of ensuring fault tolerance [7, 8]. In erasure-coded storage systems, instead of keeping an identical version of a data V on each server, V is encoded into n fragments such that V can be reconstructed from any set of at least k fragments (called k-of-n encoding), where the size of each fragment is roughly $|V|/k$. A different encoded

[*] Part of this work was done when Partha Dutta and Ron R. Levy were at Bell Labs Research, India.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 182–196, 2008.

fragment is stored on each of the n servers, and ideally such a system can tolerate the failure of $f = n - k$ servers.

The main advantage of erasure-coded storage over replicated storage is its *storage usage*, i.e., less storage space is used to provide fault tolerance. For instance, it is well-known that a replicated storage system with 4 servers can tolerate at most 1 failure in an asynchronous environment. If each server has a storage capacity of 1 TB, the total capacity of the replicated storage system is still 1 TB. In this case the storage overhead (total capacity/useable capacity) is 4, i.e. only 1/4 of the total capacity is available. Erasure coding allows the reduction of this overhead to 2 in an asynchronous system, i.e., makes 2 TB useable. In a synchronous system (with 4 servers and at most 1 failure), it is even possible to further reduce this overhead and make 3 TB available to the user. Clearly, the synchronous erasure-coded storage is more desirable in terms of storage usage. Unfortunately, synchrony assumptions are often not realistic in practice. Even if we expect the system to be synchronous most of the time, it is good practice to tolerate asynchronous periods. The idea underlying our contribution is the common practice of designing distributed systems that can cope with worst case conditions (e.g., asynchrony and failures) but are optimized for best case situations (e.g., synchrony and no failures) that are considered common in practice.

1.2 Contributions

In this paper we investigate one of the fundamental building blocks of a fault-tolerant distributed storage – multi-writer multi-reader atomic register implementations [3, 13, 15]. An atomic register is a distributed data-structure that can be concurrently accessed by multiple processes and yet provide an “illusion” of a sequential register. (A sequential register is a data-structure that is accessed by a single process with read and write operations, where a read always returns the last value written.) We consider implementations over a set of n server processes in an asynchronous crash-recovery message-passing system where (1) each process may crash and recover but has access to a stable storage, (2) in a run, at most f out of n servers are faulty (i.e., eventually crash and never recover), and (3) channels are fair-lossy.

We present two wait-free atomic register implementations ORCAS-A and ORCAS-B (Optimistic eRasure-Coded Atomic Storage). Our implementations are the first wait-free atomic register implementations in a crash-recovery model that have an “optimistic” (stable) storage usage. Suppose that all possible write values are of a fixed size Δ.[1] Then in both of our implementations, during synchronous periods with q alive (non-crashed) servers and when there is no write operation in progress, the stable storage used at every alive server is $\frac{\Delta}{q-f}$, whereas during asynchronous periods when there is no write operation in progress, the storage used is $\frac{\Delta}{n-2f}$ at all but f servers (in ORCAS-A, at most f servers may use

[1] Through out the paper we assume that, other than the write value and its encoded fragments, all other values (e.g., timestamp) at a server are of negligible size.

Δ). However, the two implementations differ in their storage usage when there is a write in progress. In ORCAS-A, when one or more writes are in progress, the storage used at a server can be Δ, but even in the worst-case the storage used is never higher than Δ. In contrast, if there are w concurrent writes in progress in ORCAS-B then, in the worst-case, the storage used at a server can be $\frac{w\Delta}{n-2f}$. Thus in terms of storage space used, ORCAS-A is more efficient in systems where we expect large number of concurrent writes, whereas, ORCAS-B is more suitable if not many writes are invoked concurrently. We also show how the number of messages exchanged in ORCAS-A can be significantly reduced by weakening the termination condition of the read from wait-free to Finite Write (FW) termination [1].[2]

Both ORCAS implementations are based on a simple but effective idea. The write first "gauges" the number of alive servers in the system by sending a message to all servers and counting the number of replies received during a short waiting period. Depending on the number of replies, the write decides how to erasure code its value. Additionally, to limit the communication overhead, the ORCAS implementations ensure that the write value or the encoded fragments are sent to the servers in only one of the phases of a write; later, the servers can locally compute the final encoded fragments on receiving a small message that specifies how the value needs to be encoded (but the message does not contain the final encoded fragments).

In particular, in ORCAS-A, the write sends the unencoded write value to all servers and waits for replies. If it receives replies from q servers, the write sends a message to the servers that requests them to locally encode the received value with $(q-f)$-of-n encoding. (Note that $q \geq n-f$ because at most f servers can be faulty.) Roughly speaking, a subsequent read can contact at least $q-f$ of the servers that reply to the write, and (1) either the read receives an unencoded value from one of those servers, or (2) it receives $q-f$ encoded fragments. In both cases, as the write does a $(q-f)$-of-n encoding, the read can reconstruct the written value. Note that, as $q-f \geq n-2f$, in the worst-case ORCAS-A does a $(n-2f)$-of-n encoding, and in synchronous periods with q alive servers, it does a $(q-f)$-of-n encoding.

On the other hand, ORCAS-B, like previous erasure-coded atomic register implementations, never sends an unencoded write value to the servers. Ideally, to obtain the same storage usage as ORCAS-A, in a write of ORCAS-B we would like to send an $(n-2f)$-of-n encoded fragment to each server, and on receiving replies from q servers, request the servers to keep a $(q-f)$-of-n encoded fragment. However, in general, it is not possible to extract a particular fragment of a $(q-f)$-of-n encoding from a *single* fragment of a $(n-2f)$-of-n encoding. Thus with this naive approach, either a write would need to send another set of fragments to the servers or the servers would need to exchange their fragments, resulting in significant increase in communication overhead. We solve this problem in ORCAS-B by a novel approach of storing multiple, much smaller fragments at

[2] A similar idea was earlier used in [10] where the read satisfied obstruction-free termination.

each server (instead of a single large one). Suppose the write estimates that there are q alive servers. Then it encodes the value with x-of-nz encoding, where x is any common multiple of $n-2f$ and $q-f$, and $z=\frac{x}{n-2f}$, and sends z fragments to each server. If the write receives replies from q servers, then it requests each server to *trim* its stored fragments in the next phase, i.e., retain any $y=\frac{x}{q-f}$ out of its z fragments and delete the rest. Roughly speaking, since a read can miss at most f servers that replied to the write, if a subsequent read sees a trimmed server then it will eventually receive y fragments from at least $q-f$ servers, and if the read does not see a trimmed server, then it will receive z fragments from at least $n-2f$ servers. In both cases, it receives $y(q-f)=z(n-2f)=x$ fragments, and therefore, can reconstruct the written value. The advantage of our approach over the naive approach is that, our approach has the same storage usage as latter, but has lower communication overhead.

The detailed presentation of our algorithms can be found in [6]. Due to space limitations, this paper focusses on ORCAS-A. Also in [6], we show lower bounds on storage space usage in atomic register implementation in synchronous and asynchronous systems, for a specific class of implementations (that include both ORCAS-A and ORCAS-B, and the implementation in [7]). Roughly speaking, we show that implementations – (1) which at the end of a write store equal number of encoded fragments in the stable storage of the servers, and (2) do not use different encoding schemes in the same operation – cannot have a stable storage usage better than ORCAS-A (and ORCAS-B) in either synchronous or asynchronous periods.

1.3 Related Work

Recently there has been lot of work on erasure-coded distributed storage [2, 5, 7, 8, 10, 11]. We discuss below three representative papers that are close to our work.

Frolund et al. [7] describe an erasure-coded distributed storage (called FAB) in the same system model as this paper, i.e., an asynchronous crash-recovery model. The primary algorithm in FAB implements an atomic register. Servers have stable-storage and keep a log containing old versions of the data, which is periodically garbage collected. The main difference with our approach is that in FAB the stable storage is not used optimistically. In particular, ORCAS-B has the same storage overhead as FAB during asynchronous periods (even when writes are in progress) but performs better during synchronous periods. Another difference is that FAB provides strict linearizability, which ensures that partial write operations appear to take effect before the crash or not at all. The price that is paid by FAB is to give up wait-freedom: concurrent operations may abort. ORCAS-B ensures that write operations are at worst completed upon recovery of the writer and guarantees wait-freedom: all operations invoked by correct clients eventually terminate despite the concurrent invocations of other clients.

Aguilera et al. [2] present an erasure-coded storage (that we call AJX) for *synchronous systems* that is optimized for $f << n$. AJX provides the same low storage overhead as ORCAS during failure-free synchronous periods, and performs better than ORCAS when there are failures. However, AJX provides consistency guarantees of only a regular register and puts a limit on the maximum number of client failures. Also, wait-freedom is not ensured since concurrent writes may abort.

Cachin et al. [5] propose a wait-free atomic register implementation for the byzantine model. It uses a reliable broadcast like primitive to disseminate the data fragments to all servers, thus guaranteeing that if one server receives a fragment, then all do. The storage required at the servers when there is no write in progress is $\frac{\Delta}{n-f}$. At first glance, one might be tempted to compare our implementations with a crash-failure restriction of the algorithm in [5], and conclude that our implementations have worse storage requirements in asynchronous periods ($\frac{\Delta}{n-2f}$). However, one of the implications of our lower bounds in [6] is that there is no obvious translation of the algorithm in [5] to a crash-recovery model while maintaining the same storage usage. (We discuss this comparison further in [6].)

2 Model and Definitions

Processes. We consider an asynchronous message passing model, without any assumptions on communication delay or relative process speeds. For presentation simplicity, we assume the existence of a global clock. This clock however is inaccessible to the servers and clients.

The set of servers is denoted by S and $|S| = n$. The j^{th} server is denoted by s_j, $1 \leq j \leq n$. The set of clients is denoted by C and it is bounded in size. Clients know all servers in S, but the set of clients is unknown to the servers. A client or a server is also called a *process*.

Every process executes a deterministic algorithm assigned to it, unless it *crashes*. (The process does not behave maliciously.) If it crashes, the process simply stops its execution, unless it possibly *recovers*, in which case the process executes a *recovery procedure* which is part of the algorithm assigned to it. (Note that in this case we assume that the process is aware that it had crashed and recovered.) A process is *faulty* if there is a time after which the process crashes and never recovers. A non-faulty process is also called a *correct* process. The set of faulty processes in a run is not known in advance. In particular, any number of clients can fail in a run. However, there is an known upper bound $f \geq 1$ on the number of faulty servers in a run. We also assume $f < n/2$ which is necessary to implement a register in asynchronous model.

Every process has a volatile storage and a stable storage (e.g., hard disk). If a process crashes and recovers, the content of its volatile storage is lost but the content of its stable storage is unaffected. Whenever a process updates one of its variables, it does so in its volatile storage by default. If the process decides to store information in its stable storage, it uses a specific operation **store**: we also say that the process *logs* the information. The process retrieves the logged information using the operation **retrieve**.

Fair-lossy channels. We assume that any pair of processes, say p_i and p_j, communicate using fair-lossy channels [4, 14], which satisfies the following three properties: (1) If p_j receives a message m from p_i at time t then p_i sent m to p_j at time t, (2) if p_i sends a message m to p_j a finite number of times, then p_j receives the message a finite number of times, and (3) if p_i sends a message m to p_j an infinite number of times and p_j is correct, then p_j receives m from p_i an infinite number of times.

On top of the fair-lossy channels we can implement more useful stubborn communication procedures (*s-send* and *s-receive*) which are used to send and receive messages reliably [4]. In addition to the first two properties of fair-lossy channels, stubborn procedures satisfy the following third property: If p_i s-sends a message m to a correct process p_j at some time t, and p_i does not crash after time t, then p_j eventually s-receives m. We would like to note that stubborn primitives can be implemented without using stable storage [4].

Registers. A sequential register is a data structure accessed by a single process that provides two operations: write(v), which stores v in the register and returns OK, and read(), which returns the last value stored in the register. (We assume that the initial value of the register is $\perp$, which is not a valid input value for a write operation.) An atomic register is a distributed data-structure that can be concurrently accessed by multiple processes and yet provide an "illusion" of a sequential register to the accessing processes [13, 14]. An algorithm *implements* an atomic register if all runs of the algorithm satisfy the *atomicity* and *termination* properties. We follow the definition of atomicity for a crash-recovery model given in [9], which in turn extends the definition given in [12]. We recall the definition in [6].

We use the following two termination conditions in this paper. (1) An implementation satisfies wait-free termination (for clients) if for every run where at most f of the servers are faulty (and any number of clients are faulty), every operation invoked by a correct client completes. (2) An implementation satisfies Finite-Write (FW) termination [1] if for every run where at most f of the servers are faulty (and any number of clients are faulty), every write invocation by a correct client is complete, and moreover, either every read invocation by a correct client is complete, or infinitely many writes are invoked in the run. (Note that wait-free termination implies FW-termination.)

Erasure coding. A k-of-n erasure coding [17] is defined by the following two primitives:

- **encode**(V, k, n) which returns a vector $[V[1], \ldots, V[n]]$, where $V[i]$ denotes i^{th} encoded fragment. (For presentation simplicity, we will assume that encode returns a *set* of n encoded fragments of V, where each fragment is tagged by its fragment number.)

- **decode**(X, k, n) which given a set X of at least k fragments of V (that were generated by encode(V, k, n)), returns V.

For our algorithms, we make no assumption on the specific implementation of the primitives except the following one: each fragment in a k-of-n encoding of V is roughly of size $|V|/k$.

In the next two sections, we present two algorithms that implement erasure-coded, multi-writer multi-reader, atomic registers in a crash-recovery model, ORCAS-A and ORCAS-B. Both implementations have low storage overhead when no write operation is in progress. The implementations differ in the storage overhead during a write, and in their message sizes.

3 ORCAS-A

We now present our first implementation which we call ORCAS-A. (The pseudocode is given in Figures 1 and 2.) The implementation is inspired by the well-known atomic register implementations in [3, 15]. Also, the registration process of a read at the servers is inspired by the listeners communication pattern in [16]. The first two phases of the write function are similar to that in [3, 15]$-$ they store the unencoded values at $n-f$ (a majority) of servers with an appropriate timestamp. Additionally in ORCAS-A, depending on the number of servers from which the write receives a reply, it selects an encoding r-of-n. Then, the write performs another round trip where it requests the servers to encode the value using r-of-n encoding and retain the fragment corresponding to its server id. The crucial parts of the implementation are choosing an encoding r-of-n and the condition for waiting for fragments at a read, such that, any read can recover the written value without blocking permanently. We now describe the implementation in more detail.

3.1 Description

Local variables. The clients maintain the following local variables: (1) ts: part of the timestamp of the current write operation, and (2) wid, rid: the identifiers of write and read operations, respectively, which are used to distinguish between messages from different operations of the same client, and (3) a timer T_c whose timeout duration is set to the round-trip time for contacting the servers in synchronous periods. The pair $[ts, wid]$ form the timestamp for the current write. The local variables at a server s_j are as follows: (1) A_j: its share of the value stored in the register, which can either be the unencoded value or the j^{th} encoded fragment, (2) τ, δ: the ts and the wid, respectively, associated with the value in A_j, and (3) ρ: the encoding associated with the value in A_j, namely, A_j is the j^{th} fragment of a ρ-of-n encoding of some value. (In particular, $\rho = 1$ implies that A_j contains an unencoded value.)

Write operation. The write operation consists of three phases, where each phase is a round-trip of communication from the client to the servers. The first phase is used to compute the timestamp for the servers, the second phase to write the unencoded value at the servers, and the final phase is used to encode the value at the servers. On invoking a write(V), the client first increments and logs its wid. This helps in distinguishing messages from different operations of the same server even across a crash-recovery. It also logs $ts = 0$ so as to detect an incomplete write across a crash-recovery. Next, the client sends get_ts

```
function initialization:
    $ts, wid, rid \leftarrow 0; r \leftarrow 1; T_c \leftarrow timer()$   $\{at\ every\ client\}$
    $A_j \leftarrow \perp; \tau, \delta \leftarrow 0; \rho \leftarrow 1$   $\{at\ every\ server\ s_j\}$

function write $(V)$ at client $c_i$
    $wid \leftarrow wid + 1$; $ts \leftarrow 0$
    store$(wid, ts)$
    repeat
        send$(\langle get\_ts, wid\rangle, S)$
    until s-receive $\langle ts\_ack, *, wid\rangle$ from $n - f$ servers
    $ts \leftarrow 1+$ max$\{ts_j$ : s-received $\langle ts\_ack, ts_j, wid\rangle\}$
    store$(ts, V)$
    trigger$(T_c)$
    repeat
        send$(\langle write, ts, wid, 0, V\rangle, S)$
    until s-receive $\langle w\_ack, ts, wid, 0\rangle$ from $n - f$ servers and expired$(T_c)$
    $r \leftarrow$ (number of servers from which s-received $\langle w\_ack, ts, wid, 0\rangle$ messages) $-f$
    if $r > 1$ then
        repeat
            $S' \leftarrow$ set of servers from which s-received $\langle w\_ack, ts, wid, 0\rangle$ until now
            send $(\langle encode, ts, wid, r\rangle, S')$
        until s-receive $\langle enc\_ack, ts, wid, r\rangle$ from $n - f$ servers
    return(OK)

upon receive $\langle get\_ts, wid\rangle$ from client $c_i$ at server $s_j$ do
    s-send$(\langle ts\_ack, \tau, wid\rangle, \{c_i\})$

upon receive $\langle write, ts', wid', rid', V'\rangle$ from client $c_i$ at server $s_j$ do
    if $rid' > 0$ then
        $\mathcal{R} \leftarrow \mathcal{R} \setminus \{[rid', *, *, i]\}$
    if $V' \neq \perp$ then
        if $[ts', wid'] >_{lex} [\tau, \delta]$ then
            $\tau \leftarrow ts'; \delta \leftarrow wid'; \rho \leftarrow 1; A_j \leftarrow V'$
            store$(\tau, \delta, \rho, A_j)$
        for all $[rid, ts, id, l] \in \mathcal{R}$ do
            s-send$(\langle r\_ack, rid, ts', wid', 1, V'\rangle, \{c_l\})$
    s-send$(\langle w\_ack, ts', wid', rid'\rangle, \{c_i\})$

upon receive $\langle encode, ts', wid', r'\rangle$ from client $c_i$ at server $s_j$ do
    if $[ts', id'] = [\tau, \delta]$ then
        $A_j \leftarrow j^{th}$ fragment of encode$(A_j, r', n)$
        $\rho \leftarrow r'$
        store$(\rho, A_j)$
    s-send$(\langle enc\_ack, ts', wid', r'\rangle, \{c_i\})$

upon recovery() at server $s_j$ do
    $[\tau, \delta, \rho, A_j] \leftarrow$ retrieve()

upon recovery() at client $c_i$ do
    $[rid, ts, wid, r, V] \leftarrow$ retrieve()
    if $ts \neq 0$ then
        repeat
            send$(\langle write, ts, wid, 0, V\rangle, S)$
        until s-receive $\langle w\_ack, ts, wid, 0\rangle$ from $n - f$ servers
```

Fig. 1. ORCAS-A: initialization, write and recovery procedures

messages to all servers and waits until it receives ts from at least $n - f$ servers. (The notation $send(m, X)$ is a shorthand for the following: for every processes $p \in X$, send the message m to p. It is not an atomic operation.) To overcome the effect of the fair-lossy channels, a client encloses the sending of its messages to the servers in a repeat-until loop, and the servers reply back using the s-send primitive. On receiving the ts from at least $n - f$ servers, the client increments

```
1: function read() at client $c_i$
2:     $rid \leftarrow rid + 1$; $\Gamma \leftarrow 0$; $M \leftarrow \emptyset$; $once \leftarrow false$
3:     store($rid$)
4:     repeat
5:         send($\langle read, rid \rangle, S$)
6:         $\mathcal{M} \leftarrow \{msg = \langle r\_ack, rid, *, *, *, * \rangle$ : s-received $msg\}$
7:         $TS \leftarrow \max_{lex}\{[ts, id]$ : $\langle r\_ack, rid, ts, id, *, * \rangle \in \mathcal{M}\}$
8:         if ($\mathcal{M}$ contains messages from at least $n - f$ servers) and ($once = false$) then
9:             $\Gamma \leftarrow TS$; $once \leftarrow true$
10:            if $TS = [0, 0]$ then return($\perp$)
11:    until ($once = true$) and ($\exists\ r', ts', id'$ such that ($[ts', id'] \geq_{lex} \Gamma$) and
       ($|\{A_j$ : $\langle r\_ack, rid, ts', id', r', A_j \rangle \in \mathcal{M}\}| \geq r'$))
12:    $\mathcal{A} \leftarrow$ set of $A_j$ satisfying the condition in line 11
13:    if $r' = 1$ then
14:        $V \leftarrow$ any $A_j$ in $\mathcal{A}$; $V' \leftarrow V$
15:    else
16:        $V \leftarrow$ decode($\mathcal{A}, r', n$); $V' \leftarrow \perp$
17:    repeat
18:        send($\langle write, ts', id', rid, V' \rangle, S$)
19:    until s-receive $\langle w\_ack, ts', id', rid \rangle$ from $n - f$ servers
20:    return($V$)

21: upon receive $\langle read, rid \rangle$ from client $c_i$ at server $s_j$ do
22:     if $\mathcal{R}$ does not contain any $[rid, *, *, i]$ then
23:         $\mathcal{R} \leftarrow \mathcal{R} \cup [rid, \tau, \delta, i]$
24:         s-send($\langle r\_ack, rid, \tau, \delta, \rho, A_j \rangle, \{c_i\}$)
```

Fig. 2. ORCAS-A: read procedure

by one the maximum *ts* received, to obtain the *ts* for this write. It then logs *ts* and V so that in case of a crash during the write, the client can complete the write upon recovery. Next, it starts its timer, and sends a *write* message with the timestamp $[ts, wid]$ and the value V, to the servers. (To distinguish this message from the *write* message sent by a read operation, the message also contains a *rid* field which is set to 0.) A server on receiving a *write* message with a higher timestamp than its current timestamp $[\tau, wid]$, updates A_j, τ and δ to V, *ts* and *wid* of the message, respectively. It also updates the encoding ρ to 1 (to denote that the contents of A_j is unencoded), and logs the updated variables. (The server also sends some message to the readers which we will discuss later.) The client waits until it receives *w_ack* messages from at least $n - f$ servers, and the timer expires. (Waiting for the timer to expire ensures that the client receives a reply from all non-crashed processes in synchronous periods.)

Next, the client select the encoding for the write to be $r = q - f$, where q is the number of *w_ack* messages received by the client. Note $r \geq 1$ because $q \geq n - f$ and $f < n/2$. Then the client sends an *encode* message to all servers which have replied to the *write* message. A server s_j on receiving this message encodes it value A_j using r-of-n encoding, and retains only the j^{th} fragment in A_j. It also updates its encoding ρ to r, logs A_j and ρ, and replies to the client. The client returns from the write on receiving $n - f$ replies. (Note that the encode phase is skipped if $r = 1$, because 1-of-n encoding is same as not encoding the value at all.)

Read operation. The read operation consists of two phases. The first phase gathers enough fragments to reconstruct a written value, and the second phase writes back the value at the servers to ensure that any subsequent reader does not read an older value.

On invoking a read, the client increments and logs its *rid*. It then sends a *read* message to the servers. On receiving a *read* message, a server *registers* the read[3] by appending it to a local list $\mathcal{R}$ with the following parameters: the *rid* of the *read* message, and the timestamp $[\tau, \delta]$ at the server when the *read* message was received. (The client *de-registers* in the second phase of the read: line 27, Figure 1.) The server then replies with its current value of A_j and its associated timestamp and encoding. In addition, whenever the server receives a new *write* message with a higher timestamp, it forwards it to its registered readers. The client on the other hand, first chooses a timestamp Γ which is greater than or equal to the timestamp seen at $n - f$ processes,[4] and then waits for enough fragments to reconstruct a written value that has an associated timestamp greater than or equal to Γ: the condition in line 11 of Figure 2 simply requires that (1) the client receives r_ack from at least $n-f$ servers, and (2) there is an encoding r' and timestamp $[ts', id']$ such that the client has received at least r' fragments of the associated value, and $[ts', id']$ is greater than or equal to Γ. In [6], we show that this condition is eventually satisfied for every read whose invoking client does not crash.

The second phase of a read is very similar to the second phase of a write except for the following case. If the read selects a value in the first phase that was encoded by the corresponding write ($r' > 1$), then the read does not need to write back the value to the servers because the write has already completed its second phase. In this case, the second phase of the read is only used to deregister the read at the servers.

Recovery Procedures. The recovery procedure at a server is straightforward: it retrieves all the logged values. The client, in addition to retrieving the logged values, also completes any incomplete write. (Note than, even if the last write invocation, before the crash at a client, is complete, *ts* can be greater than 0. In this case, the recovery procedure tries to rewrite the same value with the same timestamp. It is easy to see that this attempt to rewrite the value is harmless.)

3.2 Correctness

The proof of the atomicity of ORCAS-A is similar to the implementations in [3, 15]. The only non-trivial argument in the proof of wait-free termination is proving that the waiting condition in line 11 in Figure 2 eventually becomes true in every run where the client does not crash after invoking the read. In this section, we give an intuition for this proof by considering a simple case where a (possibly incomplete) write is followed by a read, and there are no other operations.

Suppose there is a write(V) that is followed by a read(). We claim that the read() can always reconstruct V or the initial value of the register, and it can

[3] When there is no ambiguity, we also say that the server registers the client.

[4] The Γ selected in this way is higher than or equal to the timestamp of all preceding writes because two server sets of size $n - f$ always has a non-empty intersection.

always reconstruct V if the write is complete. The write() operation has two phases that modify the state of the servers: the write phase and the encode phase. Suppose that during the write phase, the writer receives replies from q servers (denoted by set Q) such that $q \geq n - f > f$. If the writer fails without completing this phase, the read() can return the initial value of the register, which does not violate atomicity. In the encode phase, an r-of-n encode message is sent to all servers, where $r = q - f \geq n - 2f > 0$. If the writer crashes, this message reaches an arbitrary subset of servers. Subsequently, the read() contacts a set R containing at least $n - f$ servers. We denote the intersection of the read and write sets, by U, i.e. $U = Q \cap R$, and it follows that $|U| \geq q - f = r > 0$. There are two cases:

Case 1: There is at least one server in U which still has the unencoded value V. The read can thus directly obtain V from this server.

Case 2: All the servers in U have received the encode message and encoded V. Since $|U| \geq r$ and an r-of-n erasure code was used, there are enough fragments for the read to reconstruct V.

However, we must also consider the case where the read() is concurrent with multiple writes. If there is a series of consecutive writes, the write procedure ensures that all values are eventually encoded. If the read is slow, it could receive an encoded fragment of a different write from each server, making it impossible for the read to reconstruct any value. But the reader registration ensures that the servers will send all new fragments to the reader until the reader is able to reconstruct some written value. A detailed proof of wait-freedom is given in [6].

3.3 Algorithm Complexity

In this section we discuss the theoretical performance of ORCAS-A.

Timing guarantees. For timing guarantees we consider periods of a run where links are timely, local computation time is negligible, at least $n - f$ servers are alive, and no process crashes or recovers. It is easy to show that a write operation completes in three round-trips (i.e., six communication steps), as compared to two round-trips in the implementation of [15]. (We discuss this comparison further in Section 5.) Also it is straightforward to show that a read can complete in two round-trips if there is no write in progress. In [6], we show that even in the presence of concurrent writes, the read registration ensures that a read operation terminates within five communication steps.

Messages. Except the *r_ack* messages, the number of messages used by an operation is linear in the number of servers. In [6], we show how to circumvent the reader registration by slightly weakening the termination condition of the read. Message sizes in ORCAS-A are as large as those in the replication based register implementations of [3, 15]: the first phase of the write in ORCAS-A sends the unencoded value to all servers.

Worst-case bound on storage. Suppose that all possible write values are of a fixed size Δ, and the size of variables, other than those containing a value of a write operation or an encoded fragment of such a value, is negligible. Consider a partial run *pr* that has no incomplete write invocation. (An invocation that has no matching return event in the partial run is called incomplete.) The r computed in line 16 of Figure 1 of every write is at least $n - 2f$. Thus, every encoded fragment is at most of size $\Delta/(n - 2f)$. Since, there are no incomplete write invocation in *pr*, and every write encodes the value at $n-f$ processes before it returns, the size of (stable) storage at $n - f$ servers is at most $\Delta/(n - 2f)$ at the end of *pr*. In addition, note that the size of the storage at *all* servers is *always* bounded by Δ. This is in contrast to the implementation in [7] and ORCAS-B implementation that we describe later, where the worst-case storage size is dependent on the maximum number of concurrent writes.

Bound on storage in synchronous periods. Consider a partial run *pr* which has no incomplete write invocation. Let *wr* be the write with the highest timestamp in *pr*. Let t be the time when *wr* was invoked. Now, assume that (1) the links were timely in *pr* from time t onwards, and (2) at least $n - f$ servers are alive at time t, and no process crashes or recovers from time t onwards. Let $q \geq n - f$ be the set of servers that are alive at time t. Then, it is easy to see that the r computed in line 16 of Figure 1 is $q - f$ in *wr*, and hence, the size of storage at *all* alive servers is at most $\Delta/(q - f)$ at the end of *pr*. It also follows that, if *pr* is a synchronous failure-free partial run, then the size of storage at all servers is at most $\Delta/(n - f)$ at the end of *pr*.

FW-termination. Consider the case in the above implementation when a client invokes a read, registers at all the servers, and then crashes. If a server does not crash, its s-send module will send the *r_ack* message to the client forever. Since, these messages are of large sizes, it may significantly increase the load on the system. Following [1], we show in [6] that if we slightly weaken the waif-free termination condition of the read to Finite-Write (FW) termination, then such messages are not required.

4 ORCAS-B

Although the ORCAS-A implementation saves storage space in synchronous periods, it has two important drawbacks because it sends the unencoded values to the servers in the first phase of the write. First, it uses larger messages compared to implementations which never send any unencoded values to the servers. Second, if a client crashes before sending an *encode* message during a write, servers are left with an unencoded value in the stable storage.[5] In this section, we present our second implementation, ORCAS-B, which like most erasure-coded register implementations, never sends an unencoded value to the servers.

Due to lack of space, we discuss only those parts of ORCAS-B that significantly differ from ORCAS-A. (The pseudocode is presented in [6].) The crucial

[5] In practice, the second case might not cause a significant overhead because any subsequent complete write will erase such unencoded values.

difference between ORCAS-A and ORCAS-B is how the write value is encoded during a write and how it is reconstructed during a read. In ORCAS-B, the write consists of three phases. The first phase finds a suitable timestamp for the write, and tries to guess the number of alive servers, say r'. The write then encodes the value such that the following three conditions holds. (1) If the second phase of the write succeeds in contacting only $n - f$ servers (a worst-case scenario), a subsequent read can reconstruct the value. (2) If the second phase succeeds in contacting r' servers (the optimistic case), then in the third phase, the write can "trim" (i.e., reduce the size of) the stored encoded value at the servers, and still a subsequent read can reconstruct the value. (3) The size of the stored encoded value at a server should be equal to the size of a fragment in $(n-2f)$-of-n encoding in the first case, and $(r' - f)$-of-n encoding in the second case. The motivation behind these three conditions is to have the same optimistic storage requirements as in ORCAS-A.

It is not difficult to see that if the write uses a $(n - 2f)$-of-n encoding, then a server cannot *locally* extract its trimmed fragment in the third phase from the encoded fragment it receives in the second phase, without making extra assumptions about n or the erasure coding algorithm. Thus with $(n - 2f)$-of-n encoding in the second phase, in the third phase of the write, either the write needs to send the trimmed fragment to each server, or the servers need to exchange their (second-phase) fragments. In ORCAS-B we avoid this issue by simply storing multiple fragments at a server, while still satisfying the three conditions above.

We define the following variables: (1) $r = r' - f$, (2) x be the *lcm* (least common multiple) or r and $n-2f$, (3) $z = x/(n-2f)$, and (4) $y = x/r$. Now the second phase of the write encodes the value using x-of-(nz) encoding. It then tries to store z fragments at each server. If the write succeed in storing the fragments at r' servers, then in the next phase, it sends a *trim* message that requests the servers to retain y out of its z fragments (and delete the remaining fragments). Now it is easy to verify the above three conditions. If the second phase of the write stores the fragments at $n - f$ servers, a subsequent read can access at least $n - 2f$ of those servers, and thus receive at least $(n - 2f)z = x$ fragments. On the other hand, if the stored fragments at some server are trimmed, then at least r' servers have at least y fragments, and therefore a subsequent read receives y fragments from at least $r' - f = r$ servers; i.e., $ry = x$ fragments in total. In both cases, since the write has used x-of-(nz) encoding, the read can reconstruct the value. To see that the third condition is satisfied, notice that the total size of z stored fragments at a server after the second phase of the write is $z(\Delta/x) = \Delta/(n-2f)$. After trimming, the size of the stored fragments become $y(\Delta/x) = \Delta/(r' - f)$.

Another significant difference between ORCAS-A and ORCAS-B is the condition for deleting an old value at a server. In ORCAS-A, whenever a server receives an unencoded value with a higher timestamp, the old fragment or the old unencoded value is overwritten. However in ORCAS-B, if the server receives fragments with a timestamp *ts* that is higher than its current timestamp, the server adds the fragments to a set L of received fragments. Subsequently, if it receives a trim message (i.e., a message from the third phase of a write) with timestamp ts, it deletes all fragments in L with a lower timestamp. Also, the server sends the whole set L in its *r_ack* reply messages to a read. (Thus the trim message also acts as a garbage collection message.) This modification is necessary in ORCAS-B because, until a sufficient number of encoded fragments are stored at the servers, the newly written value is not recoverable from the stored data obtained from any set of servers. The trim message acts as a confirmation that enough fragments of the new value have been stored. A similar garbage collection mechanism is also present in the implementation in [7]. On the other hand, since a server in ORCAS-A receives an unencoded value first, it can directly overwrite values with lower timestamps. An important consequence of this modification is that the worst-case storage size of ORCAS-B (and the implementation in [7]) is proportional to the number of concurrent writes, whereas, the storage requirement in ORCAS-A is never worse than that in replication (i.e., storing the unencoded value at all servers). We show the wait-free termination property of ORCAS-B in [6].

5 Discussion and Future Work

There are two related disadvantages of ORCAS-A when compared to most replication based implementations. The write needs three phases to complete as compared to two phases in the latter. Also, the write needs four stable storage accesses (in its critical path) as compared to two such accesses in replication based implementations. Both disadvantages primarily result from the last phase that is used for encoding the value at the servers, and which can be removed if we slightly relax the requirement on the storage space. ORCAS-A ensures that the stable storage is encoded whenever there is no write in progress. Instead, if we require that the stable storage is *eventually* encoded whenever there is no write in progress, then (with some minor modifications in ORCAS-A) the write operation can return without waiting for the last phase. The last phase can then be executed "lazily" by the client. (The two disadvantages and the above discussion hold for ORCAS-B as well.) On a similar note, in ORCAS-A, if a read selects a value in the first phase that is already encoded at some server, then it can return after the first phase, and lazily complete the second phase (which in this case is used only for deregistering at the servers, and not for writing back the value). It follows that, a read that has no concurrent write in ORCAS-A can return after the first phase.

An important open problem is to study storage lower bounds on register implementations in a crash-recovery model. In particular, it would be interesting to study if our lower bounds (that are presented in [6]) hold when some of the underlying assumptions are removed. Another interesting direction for investigation can be implementations that tolerate both process crash-recovery with fair-lossy channels and malicious processes.

References

1. Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with byzantine shared memory. Distributed Computing 18(5), 387–408 (2006)
2. Aguilera, M.K., Janakiraman, R., Xu, L.: Using erasure codes efficiently for storage in a distributed system. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 336–345 (2005)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in a message passing system. Journal of the ACM 42(1), 124–142 (1995)
4. Boichat, R., Guerraoui, R.: Reliable and total order broadcast in the crash-recovery model. Journal of Parallel and Distributed Computing 65(4), 397–413 (2005)
5. Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded byzantine distributed storage. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 115–124 (2006)
6. Dutta, P., Guerraoui, R., Levy, R.R.: Optimistic erasure-coded distributed storage. Technical report, EPFL-IC-LPD, Lausanne, Switzerland (2008)
7. Frolund, S., Merchant, A., Saito, Y., Spence, S., Veitch, A.: A decentralized algorithm for erasure-coded virtual disks. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), pp. 125–134 (2004)
8. Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient byzantine-tolerant erasure-coded storage. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN) (2004)
9. Guerraoui, R., Levy, R.R., Pochon, B., Pugh, J.: The collective memory of amnesic processes. ACM Transactions on Algorithms 4(1) (2008)
10. Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead byzantine fault-tolerant storage. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP), pp. 73–86 (2007)
11. Hendricks, J., Ganger, G.R., Reiter, M.K.: Verifying distributed erasure-coded data. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 139–146 (2007)
12. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
13. Lamport, L.: On interprocess communication - part i: Basic formalism, part ii: Algorithms. DEC SRC Report, 8 (1985); Also in Distributed Computing, 1, pp. 77-101 (1986)
14. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo (1996)
15. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proceedings of the International Symposium on Fault-Tolerant Computing Systems (FTCS) (1997)
16. Martin, J.-P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: Proceedings of the International Symposium on Distributed Computing (DISC), pp. 311–325 (2002)
17. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. SIAM Journal of Applied Mathematics 8, 300–304 (1960)

On the Emulation of Finite-Buffered Output Queued Switches Using Combined Input-Output Queuing

Mahmoud Elhaddad and Rami Melhem

University of Pittsburgh
Department of Computer Science
Pittsburgh, Pennsylvania, 15260
{elhaddad,melhem}@cs.pitt.edu

Abstract. We study the emulation of Output Queuing (OQ) using Combined Input-Output Queuing (CIOQ) switches in the setting where the emulated OQ switch and the CIOQ switch have buffer capacity $B \geq 1$ packets at every output. We analyze the resource requirements of CIOQ policies in terms of the required fabric speedup and the additional buffer capacity needed at the CIOQ inputs.

For the family of work-conserving scheduling algorithms, we find that whereas every greedy CIOQ policy is valid for OQ emulation at speedup B, no CIOQ policy is valid at speedup $s < \sqrt[3]{B-2}$ when preemption is allowed. We also find that CCF in particular is not valid at any speedup $s < B$. We then introduce a CIOQ policy, CEH, that is valid at speedup $s \geq \sqrt{2(B-1)}$. Under CEH, the buffer occupancy at any input never exceeds $1 + \left\lfloor \frac{B-1}{s-1} \right\rfloor$.

For non-preemptive scheduling algorithms, we characterize a trade-off between the CIOQ speedup and the input buffer occupancy. Specifically, we show that for any *greedy* policy that is valid at speedup $s > 2$, the input buffer occupancy cannot exceed $1 + \left\lceil \frac{B-1}{s-2} \right\rceil$. We also show that at speedup 2, a greedy variant of the CCF policy requires input buffer capacity of only B packets for the emulation of non-preemptive OQ algorithms with PIFO service disciplines.

1 Introduction

In most Internet switches (routers), each switch output is equipped with a packet buffer, and employs an output scheduling algorithm to resolve contention among packets attempting to access the attached link. A switch output can transmit one packet at a time from the buffer, this packet then departs the switch. In addition to a service discipline that determines the packet transmission order, the output scheduling algorithm defines a drop policy (also known as the buffer management policy) to deal with buffer overflow events. The most commonly used algorithm is FIFO/Drop Tail where an incoming packet is dropped only if there is no space to store it in the appropriate output buffer, and packets in

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 197–211, 2008.

the buffer are served in FIFO order. A switch's inputs may also be equipped with buffers to hold the incoming packets until they can be delivered to the proper outputs, across the switch fabric. In this paper, we consider the setting where packets arrive online, and all switch links have equal speed (capacity). Each output can transmit one packet per time step, and there is at most one new arrival at each switch input per step.

Performance analysis of output scheduling algorithms in the above setting, for example [1,2], often assume that switches are of the Output Queuing (OQ) type. In an OQ switch, at each time step all newly arriving packets are switched to their respective outputs, where they are stored awaiting transmission. This switch architecture allows modeling packet networks as networks of queues where each switch output is accurately represented by a single-server queue controlled by an instance of the output scheduling algorithm, independently of other switch ports. However, a well-known limitation of output queuing is that in a switch with N ports, the switch must have an internal fabric speed that is N times the speed (capacity) of a link [3]: N packets destined to some output may arrive at the same time step at different inputs. The switch fabric must then be able to simultaneously transfer the N packets to that output port (i.e., at N time the speed of the switch links). This limits the applicability of output queuing in current switches where scalability, in terms of link speed and the number of ports, is a primary design objective.

To avoid the fabric speed as a scalability bottleneck, most packet switches today use Combined Input-Output Queuing (CIOQ): At each time step, up to s ($s \ll N$) packets can be switched from any input port to their respective outputs, and up to s packets can be switched to any output port, so that the switch's fabric may operate at a speedup of only s, relative to the speed of the links. CIOQ switches require packet buffers at the input ports, and a policy (the CIOQ policy) to arbitrate access to the switch fabric among packets stored at the inputs. Contention for access to the switch fabric among packets destined to different outputs complicates the analysis of scheduling algorithms in CIOQ switches.

A question that naturally arises is whether packet loss, throughput, and delay guarantees (possibly, per-session guarantees) provided by any output scheduling algorithm in a network of OQ switches carry over to networks of CIOQ switches. This is indeed the case if replacing each OQ switch with a CIOQ switch does not change the sequence of packet departures from any of the switch's outputs, which is the motivation for studying OQ switch emulation using CIOQ.

1.1 The OQ Emulation Problem

OQ emulation is defined informally as follows: A CIOQ switch with N input/output ports emulates an OQ switch of the same size if for any output scheduling algorithm employed by the OQ switch (henceforth, OQ algorithm) and any sequence of packet arrivals, the sequence of packet departures from each CIOQ switch output is identical to the sequence of departures from the corresponding OQ output. The CIOQ switch can emulate the OQ switch if, given its fabric speedup, the CIOQ policy transfers incoming packets to their respective outputs through the fabric in time to meet their departure times from the emulated switch. If this is the case for every arrival sequence, irrespective of the switch size and the capacity of the output buffers, we say that the CIOQ policy is *valid* for the emulation of the OQ algorithm. A CIOQ policy may be valid for the emulation of a given OQ algorithm under explicitly stated restrictions. In particular, it may be valid only in the *infinite-buffers setting*, in which the output buffers in the OQ switch (and the CIOQ switch) are considered to be of unlimited capacity. A CIOQ policy is valid for the emulation of a family of OQ algorithms if it is valid for the emulation of every algorithm in that family. A formal definition of validity is introduced in Section 2.2.

The *OQ emulation problem* was proposed by Chuang et al. [3], where the objective is to identify CIOQ policies that are valid, in the infinite-buffers setting, for the emulation of a family of OQ algorithms of practical interest, while imposing minimal requirements on the fabric speedup. In the OQ emulation problem, neither the CIOQ policy nor the emulated OQ algorithm has knowledge of future arrivals, and no statistical assumptions about the sequence of packet arrivals are made.

In the infinite-buffers setting, the drop policy is never exercised and, as such, the OQ algorithm can be defined by its service discipline. In that setting, Chuang et al. [3] introduced Critical Cells First (CCF),[1] a CIOQ policy that is valid at speedup 2 for the emulation of the family of Push-In-First-Out (PIFO) service disciplines, which includes many well-known disciplines such as FIFO (FCFS), Strict Priority, and Weighted Fair Queuing.[2] They also showed, using FIFO as an example, that no CIOQ policy is valid for the emulation of all PIFO service disciplines at speedup $< 2 - 1/N$. Similar results were obtained simultaneously and independently by Stoica and Zhang [4].

In this work, we investigate CIOQ policies for the emulation of OQ switches with fixed buffer capacity $B > 0$ at every output. Our interest in this setting is motivated by the emergence of technological constraints on buffer capacity in high-speed electronic and optical switches, which may limit B to a few dozen packets [5,6].

Before summarizing our results we describe the framework within which OQ emulation is set [3,7,8]: To emulate a given OQ algorithm, the CIOQ switch maintains, at all time, complete information about the internal state of the OQ algorithm and the configuration (content) of the emulated switch buffers. This information is leveraged so that:

(i) The CIOQ policy can move the packets presently at the inputs to the output side in time for departure.

(ii) The output ports dequeue and transmit each packet that reaches its departure time.

[1] The terms "packets" and "cells" are used interchangeably throughout the paper.

[2] In a PIFO service discipline, a packet arriving to an output queue can be inserted at any queue location. In each time step, the packet at the head of the queue, if any, departs from the switch.

At any time, a packet that is dropped by the emulated OQ algorithm is immediately discarded from the CIOQ buffer where it resides. To implement this framework, the CIOQ switch maintains a model of the OQ switch's output buffers, which is controlled by the OQ algorithm. In every time step, the CIOQ switch updates the model with any new arrivals and observes the algorithm's decisions. Note that this emulation framework applies to randomized as well as deterministic algorithms: Given an arrival sequence, the CIOQ switch emulates the sample path taken by the randomized algorithm.

1.2 Our Results

We evaluate CIOQ policies in terms of the CIOQ speedup required for the emulation of work-conserving OQ algorithms, and the additional buffer capacity needed to prevent buffer overflow events at the CIOQ inputs.The CIOQ switch is assumed to have buffer capacity B at every output (the same output buffer capacity as the OQ switch). To find the buffer capacity needed at each input, we adopt a CIOQ switch model where the buffer capacity at the inputs is infinite, and bound the maximum buffer occupancy, over all arrival sequences, for the CIOQ policy under consideration. The bounds depend only on the switch parameters such as the speedup and the output buffer capacity.

A CIOQ policy is said to be (s, b)-valid for the emulation of a given OQ algorithm if it is valid for the emulation of the algorithm at speedup s and, at that speedup, the buffer occupancy at any CIOQ input does not exceed b. For the family of work-conserving OQ algorithms, we find that whereas every greedy CIOQ policy is valid for the emulation of any algorithm at speedup B, no CIOQ policy is valid for the emulation of all algorithms at speedup $s < \sqrt[3]{B-2}$, when preemption is allowed. We also show, using FIFO/Drop Front [9,10] as example, that CCF is not valid for the emulation of preemptive PIFO algorithms at any speedup $s < B$. We then introduce a greedy CIOQ policy, CEH, that is valid for the emulation of all work-conserving OQ algorithms at speedup $s \geq \left\lfloor \sqrt{2(B-1)} \right\rfloor$. Under CEH, the buffer occupancy at any input never exceeds $1 + \left\lfloor \frac{B-1}{s-1} \right\rfloor$. Beside ensuring that packets meet their departure time from the emulated OQ switch, CEH transfers packets destined to the same output in their order of arrival, whenever possible. This prevents the buildup of excessively large queues at the inputs.

For the family of non-preemptive OQ algorithms, which may drop packets only by rejecting them upon arrival to an OQ switch's buffer, we characterize a tradeoff between the CIOQ speedup and the input buffer occupancy. Specifically, we show that for any *greedy* policy[3] that is valid at speedup $s > 2$, the input buffer occupancy cannot exceed $1 + \left\lceil \frac{B-1}{s-2} \right\rceil$. We also show that a greedy variant of the CCF policy is $(2, B)$-valid for the emulation of non-preemptive OQ algorithms with PIFO service disciplines.

[3] A greedy policy is one that transfers a maximal set of packets from the inputs to the outputs in every time step.

Although FIFO/Drop Tail is the most well-known algorithm, many algorithms of practical and theoretical interest use preemptive drop policies. In addition to Drop Front, preemptive policies include Nearest-To-Go [1], which resolves contention in favor of the packets with nearest destination, Strict Priority, and Random Drop, which chooses the packets to drop at random among those in the buffer.

The reason that there is no CIOQ policy capable of OQ emulation at constant CIOQ speedup is that when preemption is allowed all packets buffered at some CIOQ input port may immediately become needed at the corresponding outputs for departure. Thus, the CIOQ speedup must be at least equal to the maximum input buffer occupancy (over all possible arrival sequences). Although we obtain the lower bound using FIFO/Drop Front, similar examples can be constructed for OQ algorithms using the above-mentioned preemptive drop policies.

1.3 Related Work

Whereas OQ emulation in the infinite-buffers setting has been studied extensively, only few studies investigated the emulation of OQ switch with finite buffers. A simulation-based study in [6], suggests that under light traffic conditions, a CIOQ switch with speedup 2 and an input buffer capacity of 2 packets exhibits a loss behavior similar to that of an OQ switch with small output buffers employing the FIFO/Drop Tail scheduling algorithm. This motivated our investigation of whether a similar result can be obtained for any OQ algorithm and under all traffic patterns.

Kesselman and Rosén [7] showed that CCF is $(2, 2B)$-valid for the emulation of the FIFO/Drop Tail algorithm. It is straightforward to show that this result applies to all OQ algorithms combining a PIFO service discipline and a non-preemptive drop policy. The greedy variant of CCF, we describe here improves the maximum input buffer occupancy to B at the same computational complexity. Such savings could be of practical significance in all-optical switches.

Attiya, Hay, and Keslassy [8] proposed CIOQ policies for a relaxed version of the emulation problem: For any arrival sequence, each packet that successfully departs the OQ switch must depart the CIOQ switch within a bounded delay. They introduce a frame-based CIOQ policy that observes the packets departing from the OQ switch in each time frame, and transfers them from the input to the output in the following frame. The proposed CIOQ policy guarantees a relative packet delay and maximum buffer occupancy at most twice the output buffer capacity ($2B$), at speedup 2. Remarkably, the result holds for any OQ algorithms, even for those with preemptive drop policies. The reason is that even if all packets buffered at some CIOQ input depart simultaneously from the emulated OQ switch, the CIOQ policy can spread their transfer to the output side over a time frame duration (B time steps) without violating the relative delay guarantee. Although the throughput of a CIOQ switch using the frame-based policy is identical to the throughput of the emulated OQ switch and the relative packet delay is small, exact guarantees (e.g., throughput) obtained for a multihop network of OQ switches do not carry over to networks of CIOQ switches because of permitted delay. Composing approximate bounds over multiple hops leads to loose bounds where the error increases with the number of hops [11]. As a result, in this work we choose to investigate the cost of exact OQ emulation.

Finally, we should note that Minkenberg [12] studied the emulation of OQ switches with finite buffers, and reported a result that appears to contradict the results in this paper and in [7]. The result states that no CIOQ policy that does not starve some input queue can be work-conserving at any speedup $< N$ (the size of the switch). Thus, no policy can emulate an OQ switch employing a work-conserving scheduling algorithm. The result is obtained by constructing an example where the number of packets present in the CIOQ switch and destined to the same output can exceed the output buffer capacity. This is in contrast to the framework considered here and in [7,8], where the CIOQ switch immediately discards any packet that is dropped by the OQ algorithm.

2 Preliminaries

Consider an OQ switch with N input/output ports equipped with buffer capacity $B \geq 1$ packets at every output, and a CIOQ switch of the same size and output buffer capacity. Our goal is to identify CIOQ policies that enable the CIOQ switch to *emulate* the OQ switch. In this section, we give a precise characterization of such policies and introduce notation and definitions used throughout the remainder of the paper.

A switch's input and output ports are labeled $I_1, \ldots, I_N$ and $O_1, \ldots, O_N$, respectively. Given the foreseen technological limitations on buffer capacity and the demand for switch scalability (hundreds of ports), we assume $N \gg B$. Time proceeds in discrete steps indexed by the natural numbers. A time step is divided into three phases: the arrival, switching, and departure phases, in that order. During the arrival phase, arriving packets are received at the input ports (at most one per port), whereas in the switching phase, the switch may transfer packets from the input side to the output side across its fabric. Finally, in the departure phase each output port can transmit one packet along the attached link.

A sequence of packet arrivals σ is a non-empty finite set of triplets $\langle I, \tau, p \rangle$, each representing the arrival of a packet p at input I and time step τ.

2.1 OQ Algorithms

In an OQ switch, the fabric provides a dedicated point-to-point channel between each input and output. This enables the switch to simultaneously transfer up to N packets to each output port. Given that at most N packets arrive during a time step, all packets are transferred to their respective outputs in the switching phase immediately following their arrival.

At the output ports, each packet received from the input side is stored in the output buffer awaiting departure, or is dropped if no buffer space is available to store it. The output scheduling algorithm decides the departure order of packets in the buffer, and which packets are dropped in the case of overflow. For brevity, an output scheduling algorithm employed in an OQ switch is henceforth called an *OQ algorithm.*

Each output port in the OQ switch independently executes a copy of the OQ algorithm. Let σ be the arrival sequence. At any time, the *configuration of an output buffer* is the set of packets stored in the output's buffer. At the start of the departure phase of each time step t, the algorithm takes the current output configuration, and the history of packet arrivals and packet drops up to t as input, and decides which packets to drop, if any, and which packet to transmit during the departure phase. These decisions, along with any additional information (e.g., packets' queue positions in the case of FIFO-based algorithm), is called the *state of the OQ algorithm* at time t. Note that the OQ algorithm does not necessarily arrange the packets in the buffer into a queue. It may, for example, randomly choose a packet to transmit in each step.

The sequence of packet departures given arrival sequence σ is represented by a set D_{σ}^{B}. Each element in the set is a triplet $\langle O, \tau, p \rangle$ denoting the departure of packet p from port O at time τ.

Within the OQ emulation framework described in Section 1.1, the CIOQ switch "simulates" a complete step (all three phases) of the OQ switch at the start of each CIOQ switching phase. This allows the CIOQ to keep track of the OQ algorithm's decisions. The CIOQ switch emulates the OQ switch if for every arrival sequence σ, the sequence of departures from the CIOQ switch ports is the same as the departure sequence from the emulated OQ switch, that is D_{σ}^{B}. This is the case if and only if, given the CIOQ speedup, the CIOQ policy transfers each packet from the input to its output in time for departure.

2.2 CIOQ Policies

Suppose σ is the arrival sequence at the CIOQ switch. At the start of the switching phase of every time step t, a CIOQ policy, π, maps the current *input configuration* (the set of packets stored at the inputs) and the current state of the OQ algorithm at each of the emulated OQ outputs to a subset of the packets available at the input ports. Packets in this subset are moved to the outputs across the CIOQ fabric during the switching phase. The choice of the packets to move to the output in a given step is deterministic and is subject to the speedup constraint: Given a fabric speedup $s \geq 1$, the policy must choose the packets to transfer so that at most s packets are moved from each input, and at most s packets are moved to each output in a given step.

A CIOQ policy that enables the CIOQ to emulate a given OQ algorithm is called a *valid* policy for the emulation of the algorithm.

Definition 1 (Valid CIOQ Policy). *A CIOQ policy is valid for (the emulation of) a given OQ algorithm if, for any switch size N, output buffer capacity $B \geq 1$, and for every arrival sequence, it transfers the packets through the CIOQ fabric so that for every time step t, any packet that would depart from the emulated OQ switch during t is transferred to the corresponding CIOQ output before t's departure phase. A CIOQ policy is valid for a family of OQ algorithms if it is valid for every algorithm in that family.*

A CIOQ policy may be valid for the emulation of an algorithm only under some restrictions. For example, only in the infinite-buffers setting where the output buffer capacity is considered unlimited.

For a given OQ algorithm, switch parameters, and arrival sequence, a valid policy is said to *meet the OQ departure time* of every packet. Valid policies for the emulation of a particular OQ algorithm (or a family thereof) may differ in the buffer capacity requirements at the CIOQ inputs and the required CIOQ speedup. A CIOQ policy that is valid at speedup s, and for which the input buffer occupancy does not exceed b under any arrival sequence, is called an (s, b)-valid CIOQ policy.[4]

We focus our attention on CIOQ policies that are *greedy*. A greedy policy transfers a maximal set of packets to the output in every time step. As a result, for every non-greedy CIOQ policy π and CIOQ speedup s, one can define a greedy policy π', that, at every time step transfers a super-set of the packets transferred by π. Obviously, if π is valid (for the emulation of some OQ algorithm) at speedup s, then π' is also valid at the same speedup.

The following definitions lead to a formal characterization of greedy policies, and are used in subsequent sections:

Definition 2 (Input Blocking). *A packet p at a CIOQ input port I is input blocked during a time step t if, during t's switching phase, the CIOQ policy transfers s packets from I to the output side, and these packets do not include p.*

Definition 3 (Output Blocking). *A packet buffered at some input port and destined to output port O is output blocked during time step t if, during t's switching phase the CIOQ policy transfers s packets to output O, and these packets do not include p.*

Definition 4 (Greedy CIOQ Policy). *A CIOQ policy is greedy if at every time step, every packet buffered at an input port is either transferred to the output, is input blocked, or is output blocked.*

2.3 Families of OQ Algorithms

The objective of the OQ emulation problem is to identify CIOQ policies that are valid for the emulation of all OQ algorithms, at minimum CIOQ speedup and input buffer capacity requirements. Toward this end, we seek upper and lower bounds on the resource requirements of greedy CIOQ policies for the emulation of families of work-conserving algorithms.

Because in the OQ switch an output buffer can accept at most B packets in a time step, a speedup of B is sufficient for the emulation of all work-conserving algorithms.

Proposition 1. *Every greedy CIOQ policy is $(B, 1)$-valid for the emulation of all work-conserving OQ algorithms.*

[4] It is easy to see that an (s, b)-valid policy is also (s', b')-valid for all (s', b') where $s' \geq s$ and $b' \geq b$, if at speedup s' it transfers at each time step a super-set of the packets it would transfer at speedup s.

Such speedup requirement is feasible only when B is very small (e.g., up to 5), but would be prohibitive even in high-speed packet switches with limited buffering capacity.

To obtain lower bounds on the resource requirements of greedy CIOQ policies, we consider subsets of work-conserving algorithms that include well-known and widely-used ones. Namely, the family of algorithms with *non-preemptive* drop policies (non-preemptive algorithms) and the family of algorithms with *PIFO* service disciplines (PIFO algorithms).

The drop policy of an OQ algorithm is non-preemptive if an incoming packet may be dropped upon arrival to the OQ switch, but may not be dropped once admitted to the output buffer. Otherwise, the drop policy is preemptive. Non-preemptive drop policies are collectively referred to as "Drop Tail." These policies differ in how the tie is broken when the number of arrivals destined to an output port in a given time step exceeds the space available in that output's buffer. Possible tie-breaking rules include randomly choosing the "victim" packets among those arrivals, and tie-breaking based on input port numbers, or based on information in the packets' headers.

A PIFO service discipline arranges the packets in the output buffer into a queue, where:

(P1) At each time step, the packet at the head of the output queue departs the OQ switch.
(P2) An arriving packet is inserted at some arbitrary position (defined by the service discipline) in the output queue.
(P3) For each pair of packets p, q in the output queue, if p precedes q relative to the head of the queue at some time t, then this order is preserved at every subsequent step where both packets remain in the buffer.

In the absence of further packet arrivals to the output port, the position of any packet in the queue determines the time it departs from the OQ switch. We refer to this as the *projected departure time* of the packet at time t.

In the next section we investigate the speedup and input buffer capacity required by greedy CIOQ policies for the emulation of non-preemptive OQ scheduling algorithms. Emulation of preemptive OQ algorithms is considered in the following section.

3 OQ Emulation of Non-preemptive Scheduling Algorithms

In this section, we study the emulation of non-preemptive OQ scheduling algorithms. First, we characterize a trade-off between speedup and the maximum input buffer occupancy. The trade-off applies to all greedy CIOQ policies that are valid at speedup $s > 2$. Then, we describe a greedy variant of the CCF policy introduced in [3] and show that this variant is $(2, B)$-valid for the emulation of non-preemptive PIFO OQ algorithms.

3.1 The Speedup — Buffer Capacity Trade-Off

Theorem 1. *Let π be a greedy CIOQ policy that is valid for the emulation of a non-preemptive OQ algorithm A at speedup $s > 2$ in the finite-buffers setting. Then, at speedup s the buffer occupancy at each of the CIOQ switch's inputs does not exceed $1 + \left\lceil \frac{B-1}{s-2} \right\rceil$.*

Proof. To reach contradiction, suppose that there is a CIOQ input $I_i, i \in \{1, \ldots, N\}$, with buffer occupancy exceeding $1 + \left\lceil \frac{B-1}{s-2} \right\rceil$ at some time step. Let t be the earliest such step and consider the following claim (proof omitted)

Claim. Let p be the earliest arriving packet among those in I_i's buffer just after the arrival phase of time step t, and let $t-x$ be p's arrival time. Then the greedy CIOQ policy transfers at least $x + B + 1$ packets to p's output port during the interval $[t - x, t)$.

Let $O_j, j \in \{1, \ldots, N\}$, be p's output port. Neither p nor the first $x + B$ packets transferred to O_j during $[t - x, t)$ are dropped by the non-preemptive OQ algorithm. Otherwise, these packets would have been dropped by the CIOQ upon arrival. That is, without being buffered for a complete time step at the input (as in p's case) or being transferred to the output. Since the emulated OQ output serves at most x packets during $[t - x, t)$ and the arrival sequence is the same for both the CIOQ and the emulated OQ switch, the emulated OQ output corresponding to O_j would hold more than B packets at the beginning of time step t, which contradicts the fact that the output buffer capacity of the emulated OQ switch is B packets. □

3.2 The Critical Cells First CIOQ Policy

In this section, we review the CCF CIOQ policy of [3] and introduce its greedy variant, G-CCF. We show that G-CCF is $(2, B)$-valid for the emulation of non-preemptive PIFO algorithms. In contrast to this result, we show in the next section that G-CCF is not valid for OQ emulation at any speedup less than B when preemption is allowed.

CCF and G-CCF consist of two components: Management of input buffers, and the selection of packets to transfer to the output in every step. We begin by describing the buffer management component, which is common to both policies, then specify packet selection, starting with G-CCF.

Input Buffer Management: Under both CCF and G-CCF, the input buffer is organized as a queue that permits insertion of packets at arbitrary locations and the removal of packets at arbitrary locations. Consider an arbitrary packet p and let t be its arrival time. Further, let l be the **output cushion** of p — the number of packets at p's output that have earlier projected departure time than p (as calculated after t's arrival phase). Packet p is inserted into the input queue at position $l + 1$ (from the head of the queue). If the queue has less than l packets, the arriving packet is inserted at the end of the queue.

Packet Selection in G-CCF: To choose the set of packets to transfer to the output, in each time step G-CCF computes a *many-to-many pairwise-stable matching* (details below) of input ports to output ports. For this, G-CCF uses the Gale-Shapley Deferred Acceptance algorithm [13] (a.k.a. the stable marriage algorithm), as adapted by Roth to the many-to-many setting [14].

Given a CIOQ speedup $s \geq 1$, each port participates with a quota of s packets in the many-to-many matching. That is, up to s packets at each input port are transferred to the output side and up to s packets are transferred to an output port. Matching is based on the preferences of the inputs and outputs. The output preference is represented by a list of packets and the respective inputs arranged in increasing order of the projected OQ departure time. The input preference is a list of the packets queued at the input (and their respective outputs) arranged in the same order as the input queue. A port prefers to be matched with ports that appear earlier in its preference list. In the following pseudo-code, an outstanding request for a packet is a request that the corresponding input has not already rejected.

```
Deferred-Acceptance-Algorithm
  while there are outputs with unfilled quota and outstanding requests
      do
        Each such output requests its preferred packets from the inputs to
          fulfill its quota
        Each input grants the requests it prefers without exceeding its quota
```

Note that in the second step of the while loop, an input may cancel previous grants to accept more preferred requests.

Per the definition pairwise stability [15], a matching is pairwise-stable given the G-CCF preference lists if at every time step t, for every packet p buffered at some input at the beginning of the switching phase, either:

- p is transferred to the corresponding output during t,
- s packets with earlier projected OQ departure times are transferred to p's output during t, or
- s packets ahead of p in its input queue are transferred to their corresponding outputs during t.

It follows that G-CCF is a greedy CIOQ policy (cf. Definition 4).[5]

Packet Selection in CCF: CCF computes s one-to-one stable matchings in every time step by repeatedly invoking the (one-to-one) Deferred Acceptance algorithm [13]. The one-to-one algorithm uses the same input and output preference lists as G-CCF. Each output can request at most 1 packet, and each input can grant at most 1 packet in an iteration of the **while** loop.

[5] A pairwise stable matching is guaranteed to exist at every time step since the input and output ports have *substitutable* preferences. See [14].

Though the resulting matchings are individually stable in the one-to-one sense, one can construct an example where the iterative matching procedure violates the definition of a greedy CIOQ policy by failing to transfer a maximal set of packets in a given time step. The reason is that in each invocation of the one-to-one Deferred Acceptance algorithm where the algorithm fails to match a given output to an input, the output's quota is effectively decreased by 1.

OQ Emulation Using G-CCF

Kesselman and Rosén proved that CCF is $(2, 2B)$-valid for the emulation of the FIFO/Drop Tail algorithm. The result also holds for any non-preemptive PIFO algorithms. Here, we give a similar result for G-CCF that lowers the input buffer capacity capacity required to B packets.

Theorem 2. *For any output buffer capacity $B > 0$, G-CCF is a $(2, B)$-valid CIOQ policy for the emulation of any non-preemptive PIFO OQ algorithms.*

4 OQ Emulation with Preemption Allowed

In this section we show that no greedy CIOQ policy is valid for the emulation of all OQ algorithms at speedup $s \leq \sqrt[3]{B-2}$ when preemption is allowed, and that G-CCF is not valid at any speedup $s < B$ under the same conditions.

Theorem 3. *No greedy CIOQ policy is valid for the emulation of all PIFO scheduling algorithms at any speedup $s \leq \sqrt[3]{B-2}$ when preemption is allowed in the emulated OQ algorithm, and the output buffer capacity is B.*

The proof of Theorem 3 proceeds by constructing an example where $N \geq 2B^2$. It uses FIFO/Drop Front, which is a PIFO OQ scheduling algorithm. The Drop Front policy has been proposed for the objective of minimizing the queuing delays incurred by successfully delivered packets [9], but has also been shown to improve TCP throughput compared to Drop Tail [10].

Next we show that when preemption is allowed, G-CCF (hence CCF) is in not valid for the emulation of all PIFO OQ algorithms at any $s < B$. To reach this result, we demonstrate using an example that G-CCF fails to emulate a variant of the FIFO/Drop Front OQ scheduling algorithm that recognizes two different classes of packets: a low-delay class, denoted as class L, and a bulk data transfer class denoted as class T. We refer to this variant as *2-class FIFO/Drop Front*. The proof exploits the fact that G-CCF favors packets with earlier projected OQ departure times in every time step, and the fact that "investing" in such packets may be futile if preemption is allowed.

In 2-class FIFO/Drop Front, each traffic class has a fixed allocation (a partition) of the emulated OQ buffer capacity. We specify the buffer allocations by a pair (B_L, B_T) where $B_L + B_T = B$. At any time, the number of class-L packets present in the buffer does not exceed B_L, and similarly for class-T. An incoming packet is inserted into the proper buffer partition based on its class. Each of the two partitions is a FIFO buffer, where Drop Front is used to resolve overflow events. In each time step, a class-T packet is served if and only if no class-L packets are present in the L-partition.

Theorem 4. *If preemption is allowed, G-CCF is not valid for the emulation of PIFO OQ scheduling algorithms at any $s < B$.*

In constructing the example, we exploit the fact that whenever possible, CCF transfers packets to each output in the order of their projected OQ departure times. As a consequence of preemption, some packets (the T-packets in our example) remain output blocked for an extended period of time, Thus allowing the occupancy of corresponding input buffers to build up. In the next section, we consider mitigating this buffer buildup at the inputs by ordering the output preference lists based on the time of packet arrival.

5 The CCF-EAF Hybrid CIOQ Policy

Early Arrivals First (EAF) is a CIOQ policy where every newly arrived packet is inserted at the head of the corresponding input queue. To choose the packets to transfer to the output in a given time step, EAF computes a many-to-many stable matching of input to output ports in the same way as G-CCF. However, unlike G-CCF, each output's preference list is a list of the packets buffered at the inputs and destined to that output, arranged in order of non-increasing arrival time, with ties broken based on port numbers (hence the name of the policy).

It is easy to see that also unlike CCF and G-CCF, EAF is not prone to input buffer buildup when preemption is allowed. However, it is obvious that EAF would fail to emulate a scheduling algorithm where a later arrival to the switch may have an earlier departure time; for example, the OQ scheduling algorithms based on the Last-In-First-Out service discipline and the Drop Front packet drop policy.

5.1 The CEH CIOQ Policy

Now we propose and investigate the performance of a greedy policy, CEH, which is a hybrid of CCF and EAF. Under CEH, new arrivals to the CIOQ switch are inserted at the head of the corresponding input buffer. Given CIOQ speedup $s \geq 2$, CEH chooses the packets to transfer from the input to the output by sequentially computing two pairwise stable matchings using the Deferred Acceptance algorithm (Section 3.2).

The first is a matching computed using the input and output CCF preference lists. In this matching, every output has a quota of 1 and every input has a quota of s. That is, whereas the number of packets participating in the stable matching destined to any given output does not exceed 1, an input may participate in the matching with up to s packets.

The quotas for the second matching are calculated as follows: Suppose some port P (an input or output port) participates in the first matching with $U(P)$ packets. Then, in the second matching its quota is $s - U(P)$ packets. The second matching is computed using the EAF preference lists described above.

The following lemma asserts that CEH is indeed a greedy policy.

Lemma 1. *At every time step t, for every packet p buffered at one of the inputs at the beginning of the arrival phase, either:*

(i) *p is transferred to the corresponding output, say O,*
(ii) *There exists a packet with earlier OQ departure time and $s-1$ packets with earlier arrival times that are transferred to output O, or*
(iii) *Exactly s packets ahead of p in its input queue are transferred to their corresponding outputs.*

5.2 Performance of CEH

Theorem 5. *At any speedup $s \geq \max\{2, \left\lceil\sqrt{2(B-1)}\right\rceil\}$, CEH is $(s, 1+\frac{B-1}{s-1})$-valid for the emulation of all work-conserving OQ algorithms.*

Proof. Suppose $s \geq 2$ and consider a packet p that is not dropped by the OQ scheduling algorithm. Suppose the packet arrives at time t and departs the OQ switch at time $t' > t$.

Upon arrival, at most $B-1$ packets with earlier arrival times than p and destined to the same output are buffered at the CIOQ's inputs. This is because at most B packets with a common destination can simultaneously exist in the CIOQ switch. Obviously, the number of packets buffered at the input and have earlier arrival times than p does not increase in subsequent time steps.

At time t, p is at the head of the input queue. At any step $\tau \in [t, t')$, p is not transferred to the output only if at least $s-1$ packets at the input with earlier arrival times than p participate in the second stable matching. These packets are then transferred to the output during τ. In every step p remains buffered at the input, the number of packets ahead of it in the input queue increases by at most 1. Thus, the number of packets ahead of p in its input buffer increases by one packet at most $\left\lfloor\frac{B-1}{s-1}\right\rfloor$ times.

During t', p cannot be output blocked since it has the earliest departure time among packets destined to its output. It follows that p is moved to the output if during t' if it is not input blocked. This is the case if $s > \left\lfloor\frac{B-1}{s-1}\right\rfloor$. That is, if $s \geq \left\lceil\sqrt{2(B-1)}\right\rceil$. Thus CEH is valid at any speedup $s \geq \left\lceil\sqrt{2(B-1)}\right\rceil$.

Observe that the number of packets ahead of any packet in an input queue is incremented at most $\left\lfloor\frac{B-1}{s-1}\right\rfloor$ times, irrespective of whether it is eventually dropped or transferred to the output. Thus, the buffer occupancy at any input port never exceeds $1 + \left\lfloor\frac{B-1}{s-1}\right\rfloor$. □

Notice that the proof allows for newly arriving packets to preempt packets already in the switch buffers. It also doesn't make any restrictions on changes to the service order induced by new arrivals or by packet drops.

6 Concluding Remarks

In this paper we investigated CIOQ policies for the emulation of finite-buffered OQ switches employing a work-conserving (OQ) scheduling algorithm. We showed that when preemption is allowed no CIOQ policy can emulate all PIFO OQ algorithms at constant speedup. We proposed a CIOQ policy, CEH, that can emulate any work conserving OQ algorithm at speedup $O(\sqrt{B})$ (the output buffer capacity). Such speedup may be feasible in high-speed switches, which are expected to have a small number of buffers. One possible avenue for future research is closing the gap between the $O(\sqrt[3]{B})$ speedup lower bound and the $O(\sqrt{B})$ upper bound.

References

1. Aiello, W., Ostrovesky, R., Kushilevitz, E., Rosén, A.: Dynamic routing on networks with fixed-size buffers. In: Symposium On Discrete Algorithms (SODA) (2003)
2. Harchol-Balter, M., Wolfe, D.: Bounding delays in packet-routing networks. In: The 27th Annual ACM Symposium on Theory of Computing (STOC) (May 1995)
3. Chuang, S.T., Goel, A., McKeown, N., Prabhakar, B.: Matching output queuing with a combined input output queued switch. IEEE Journal on Selected Areas in Communications 17(6), 1030–1039 (1999)
4. Stoica, I., Zhang, H.: Exact emulation of an output queueing switch by a combined input output. In: International Workshop on Quality of Service (1998)
5. Enachescu, M., Ganjali, Y., Goel, A., McKewon, N., Roughgarden, T.: Routers with very small buffers. In: IEEE Infocom. (2006)
6. Beheshti, N., Ganjali, Y., Rajaduray, R., Blumenthal, D., McKeown, N.: Buffer sizing in all-optical packet switches. In: Optical Fiber Communication (2006)
7. Kesselman, A., Rosen, A.: Scheduling policies for CIOQ switches. Journal of Algorithms 60(1), 60–83 (2006)
8. Attiya, H., Hay, D., Keslassy, I.: Packet-mode emulation of output-queued switches. In: ACM symposium on on parallel algorithms and architectures (January 2006)
9. Yin, N., Hluchyj, M., Mansfield, M.: Implication of dropping packets from the front of a queue. IEEE Trans. Communications (January 1993)
10. Lakshman, T., Neidhardt, A., Ott, T.: The drop from front strategy in TCP and in TCP over ATM. In: INFOCOM. (January 1996)
11. Le Boudec, J., Thiran, P.: Network Calculus: A theory of deterministic queues for the Internet, vol. 2050. Springer, Heidelberg (2002)
12. Minkenberg, C.: Work-conservingness of CIOQ packet switches with limited output buffers. Communications Letters, IEEE 6(10), 452–454 (2002)
13. Gale, D., Shapley, L.: College admissions and the stability of marriage. The American Mathematical Monthly (January 1962)
14. Roth, A.: Stability and polarization of interests in job matching. Econometrica (January 1984)
15. Sotomayor, M.: Three remarks on the many-to-many stable matching problem. Mathematical Social Sciences (January 1999)

On Radio Broadcasting in Random Geometric Graphs*

Robert Elsässer[1], Leszek Gąsieniec[2], and Thomas Sauerwald[3]

[1] Institute for Computer Science, University of Paderborn,
33102 Paderborn, Germany
elsa@upb.de

[2] Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, UK
leszek@csc.liv.ac.uk

[3] Paderborn Institute for Scientific Computation, University of Paderborn,
33102 Paderborn, Germany
sauerwal@upb.de

Abstract. One of the most frequently studied problems in the context of information dissemination in communication networks is the broadcasting problem. In this paper we consider radio broadcasting in random geometric graphs, in which n nodes are placed uniformly at random in $[0, \sqrt{n}]^2$, and there is a (directed) edge from a node u to a node v in the corresponding graph iff the distance between u and v is smaller than the transmission radius assigned to u. Throughout this paper we consider the distributed case, i.e., each node is only aware (apart from n) of its own coordinates and its own transmission radius, and we assume that the transmission radii of the nodes vary according to a power law distribution. First, we consider the model in which any node is assigned a transmission radius $r > r_{\min}$ according to a probability density function $\rho(r) \sim r^{-\alpha}$ (more precisely, $\rho(r) = (\alpha - 1)r_{\min}^{\alpha-1}r^{-\alpha}$), where $\alpha \in (1, 3)$ and $r_{\min} > \delta\sqrt{\log n}$ with δ being a large constant. For this case, we develop a simple radio broadcasting algorithm which has the running time $O(\log\log n)$, with high probability, and show that this result is asymptotically optimal. Then, we consider the model in which any node is assigned a transmission radius $r > c$ according to the probability density function $\rho(r) = (\alpha - 1)c^{\alpha-1}r^{-\alpha}$, where α is drawn from the same range as before and c is a constant. Since this graph is usually not strongly connected, we assume that the message which has to be spread to all nodes of the graph is placed initially in one of the nodes of the giant component. We show that there exists a fully distributed randomized algorithm which disseminates the message in $O(D(\log\log n)^2)$ steps, with high probability, where D denotes the diameter of the giant component of the graph.

Our results imply that by setting the transmission radii of the nodes according to a power law distribution, one can design energy efficient radio networks with low average transmission radius, in which broadcasting can be performed *exponentially* faster than in the (extensively studied) case where all nodes have the same transmission power.

* Partly supported by the Royal Society IJP 2007/R1 “Geometric Sensor Networks with Random Topology”.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 212–226, 2008.

1 Introduction

In view of recent technological developments in wireless/mobile communication the abstract model of packet radio networks became very popular and received a lot of attention in the algorithms community [2,5,9,12]. Most of the work on time efficient radio broadcasting done so far is devoted to radio networks with an arbitrary (in fact the worst case) topology. Our main intention is to derive efficient distributed algorithms for radio broadcasting in *random geometric graphs*, which are often used to model wireless communication networks.

1.1 Models and Motivation

A radio network is modeled by a directed graph $G = (V, E)$, where V represents the set of nodes of the network, and E contains ordered pairs of distinct nodes such that $(v, w) \in V \times V$ iff node v can directly send a message to node w. The total number of neighbors connected to a node by (in-)coming edges forms its (in-)degree. The *size of the network* is the number of nodes $n = |V|$. The set of nodes directly reachable from a node $v \in V$ is the *range* of v.

One of the radio network properties is that a message transmitted by a node is always sent to all nodes within its range. The communication in the network is synchronous and it consists of a sequence of (communication) steps. During one step, each node v either transmits or listens. If v transmits, then the transmitted message reaches each of its neighbors by the end of this step. However, a node w in the range of v successfully receives this message iff in this step w is listening and v is the only transmitting node which has w in its range. If node w is in the range of a transmitting node but is not listening, or is in the range of more than one transmitting node, then a *collision* (conflict) occurs and w does not retrieve any message in this step. In fact coping with collisions is one of the main challenges in efficient radio communication. A commonly used tool to handle this problem in radio networks with unknown topology is the concept of selected families of transmission sets [5,7,9,19].

The running time of an algorithm is the number of communication steps required to complete the considered communication task. Thus, any internal computation within individual nodes is neglected. In this paper we are mainly interested in the running time of distributed broadcasting algorithms using radio communication protocol. In the broadcasting problem it is assumed that a message is placed in one of the nodes of a radio network, and the goal is to spread this message to all nodes of the network using radio communication. In this paper we assume that each node knows its own position ((x, y) coordinates), its transmission radius, and the number of nodes in the network. However, the location of the other nodes or their transmission radii are not known.

It is of our particular interest to analyze radio communication in ad hoc sensor networks. Ad hoc sensor networks are often modeled by the so called $G(n, r)$ random geometric graph model (e.g. [18,26,28]), i.e., n vertices with radius r are placed within $[0, \sqrt{n}]^2$ uniformly at random[1], and two nodes are connected by an edge in the resulting graph iff their Euclidean distance is smaller than r. This simple model of radio network is applicable to wireless networks where similar stations are randomly distributed in a flat region without large obstacles. In such a terrain, the signal of a transmitter reaches receivers at the same distance in all directions.

In this paper we consider radio broadcasting in two different types of random geometric networks. Due to simplicity reasons, we assume that n points are distributed uniformly at random within $[0, \sqrt{n}]^2$, however, the radii of the nodes may vary according to a power law distribution, i.e., a node is assigned a transmission radius larger than some value r with probability proportional to $r^{1-\alpha}$, where $\alpha \in (1, 3)$ is a fixed constant. Similar graph models are known to have improved fault tolerance [22] and (as we show in this paper) these networks allow very fast broadcasting, in fact exponentially faster than $G(n, r)$ graphs with polylogarithmic transmission radii, while maintaining almost the same average energy consumption parameters as the corresponding $G(n, r)$ model. We should note that the graphs considered in this paper are not necessarily undirected, since a node u with large radius may contain some node v with smaller radius in its range, and thus u might fall outside the range of v. A precise definition of the graph models considered in this paper can be found in Section 1.3.

1.2 Related Work

The broadcasting problem has attracted a great deal of attention in the context of radio networks with an arbitrary topology. For networks with linearly bounded labels, in which the nodes do not possess any global knowledge about the topology of the network, the trivial $O(n^2)$ upper bound on deterministic broadcasting was first improved by Chlebus et al. [6] to $O(n^{11/6})$. The subsequent improvements included an $\tilde{O}(n^{5/3})$ time algorithm proposed by De Marco and Pelc [12], an $O(n^{3/2})$ time algorithm proposed by Chlebus et al. [5], and an $O(n \log^2 n)$ time algorithm developed by Chrobak et al. [7]. Clementi et al. [9] presented a deterministic broadcasting algorithm for *ad-hoc* radio networks which works in time $\tilde{O}(D\Delta)$, where D is the diameter of the network (the number of edges on the longest shortest path) and Δ is the maximum in-degree of a node. The $O(n \log^2 n)$ and $\tilde{O}(D\Delta)$ algorithms, presented in [7] and [9], respectively, can easily be adapted for polynomially bounded node labels. Brusci and Del Pinto [2] showed that for any deterministic broadcasting algorithm $\mathcal{A}$ in ad-hoc radio networks, there are networks on which $\mathcal{A}$ requires $\Omega(n \log n)$ time. Later, Czumaj and Rytter proposed a randomized algorithm which achieves with high probability linear broadcasting time on arbitrary networks [10]. Under the assumption that the network diameter is known, they presented a broadcasting algorithm which has a running time of $O(D \log(n/D) + \log^2 n)$. Independently, Kowalski and Pelc introduced a similar algorithm with the same running time [24].

[1] In the general model the vertices are placed in $[0, 1]^d$ for some $d > 0$, however, in this paper we only consider placement of n points on the plane $[0, \sqrt{n}]^2$.

In the model where the network topology is known to all nodes in advance Gaber and Mansour [17] proposed a centralized broadcasting procedure completing the task in time $O(D + \log^5 n)$. Elkin and Kortsarz improved this bound to $D+O(\log^4 n)$ in general graphs and to $D+O(\log^3 n)$ in planar graphs [14]. Gąsieniec et al. proposed an alternative solution with times $D + O(\log^3 n)$ and $O(D)$ respectively [20]. Very recently, the constructive upper bounds w.r.t. broadcasting in general graphs have been improved to $D + O(\log^3 n/ \log \log n)$ and $O(D + \log^2 n)$ in [8] and [25], respectively. Note that computing an optimal (radio) broadcast schedule for an arbitrary network is NP-hard [4,31].

In [15] the authors considered radio broadcasting in the traditional Erdős-Rényi random graph model. In this model, given a set of n nodes a graph $G_{n,p}$ is constructed by letting any two pair of vertices be connected with probability p, independently. They presented centralized as well as fully distributed procedures for the broadcasting problem in such graphs, and showed that these algorithms are asymptotically optimal. In [1] Berenbrink et al. considered efficient radio broadcasting algorithms w.r.t. running time and energy consumption in these types of random graphs.

In [13] Dessmark and Pelc analyzed radio broadcasting in geometric networks. They showed that if each node knows its neighbors, then broadcasting can be performed in $O(D)$ steps. If each node knows only its own position, then broadcasting can be performed in $O(n)$ steps, and, if the nodes are not able to detect collisions, this result cannot be improved.

In [16] Emek et al. considered the broadcasting problem in geometric graphs in which each node has the same transmission radius (UDG model). They determined the broadcasting time depending on the diameter D and the granularity g, which is the inverse of the minimum distance between any two nodes. First, it was shown that if the nodes other than the source are initially idle and cannot transmit until they hear a message for the first time, then broadcasting can be accomplished in time $O(Dg)$. For the case, in which all nodes may transmit messages from the beginning, an optimal broadcasting algorithm with running time $O(\min\{D + g^2, D \log g\})$ was presented.

Radio communication in the $G(n, r)$ model has been analyzed by Lotker and Navarra in [27]. In order to cope with radio broadcasting or gossiping on the $G(n, r)$ graph, these problems have first been solved on the grid. Then, Lotker and Navarra emulated the corresponding grid protocol on the $G(n, r)$ model, and obtained asymptotically optimal algorithms for the broadcasting and gossiping problem. That is, if $r = \Omega(\sqrt{\log n})$, then the time needed to spread a message is $O(D)$, with high probability, where $D = \Theta(\sqrt{n}/r)$ is the diameter of the graph, with probability $1 - o(1)$.

Recently, Czumaj and Wang considered radio gossiping under different locality assumptions in the $G(n, r)$ graph and generalized the results mentioned before [11]. However, these algorithms cannot be extended to random geometric graphs in which the distribution of the transmission radii varies according to some (e.g. power law) distribution.

1.3 Our Results

In this paper, we consider distributed radio broadcasting algorithms in random geometric graphs in which the transmitting radii of the nodes vary according to a power law distribution. More precisely, we consider the following graph models:

1. Let n vertices be placed uniformly at random within $[0, \sqrt{n}]^2$. In this case, a node is assigned transmission radius $r > r_{\min}$ according to the probability density function $\rho(r) = (\alpha - 1)r_{\min}^{\alpha-1} r^{-\alpha}$, independently, where $\alpha \in (1, 3)$ is a constant and $r_{\min} > \delta\sqrt{\log n}$ with δ being a (large) constant. In the resulting graph $G_{\geq r_{\min}}$ a node v is in the range of a node u if the Euclidean distance between u and v is smaller than the radius of u. The choice of δ implies that the graph is strongly connected with very high probability[2] (e.g. [29]).
2. Let n vertices be placed uniformly at random within $[0, \sqrt{n}]^2$. Here, a node is assigned radius $r > c$ according to the probability density function $\rho(r) = (\alpha - 1)c^{\alpha-1} r^{-\alpha}$, independently, where c is some (large) constant. The ranges of the nodes in the resulting graph $G_{\geq c}$ are defined by the same rules as in the previous model.

Throughout this paper we assume full synchronization, i.e., all nodes share a global clock. In the first model, the graph is (strongly) connected w.v.h.p. [29]. In the second model, the graph has a strongly connected giant component containing $\Theta(n)$ vertices, w.v.h.p. [30]. We develop for the graph model $G_{\geq r_{\min}}$ an efficient randomized broadcasting algorithm[3] which is able to distribute a message, placed initially in one of the nodes of the graph, to all nodes within $O(\log \log n)$ steps. Concerning the $G_{\geq c}$ model, we show that any message placed initially in one of the nodes of the giant component of the graph can be distributed to all nodes within $O(D(G_{\geq c})(\log \log n)^2)$ steps, w.v.h.p., where $D(G_{\geq c})$ denotes the diameter of the giant component of the graph. Notice that the nodes of the giant component can reach *any* node in the graph within $O(D)$ steps, w.v.h.p. (cf. Section 3).

A main implication of our results is that by setting the transmission radii in a set of nodes placed uniformly at random in the plane according to a power law distribution, we obtain a radio network which supports very fast broadcasting by keeping the energy consumption almost as low as in a $G(n, r)$ graph with the same average transmission radius. More precisely, in a graph $G(n, r)$ with $r = \log^{c'} n$, where $c' > 1/2$, a message is broadcasted to all nodes of the graph within $\tilde{\Theta}(\sqrt{n})$ steps, w.h.p., where $\tilde{\Theta}$ is the Θ-function omitting polylogarithmic terms [13]. The total energy consumption needed for transmission during the broadcasting process is $\tilde{\Theta}(n)$. In the $G_{\geq r_{\min}}$ graph with $r_{\min} = \log^{c'} n$, where $c' > 1/2$, a message can be broadcasted within $\Theta(\log \log n)$ steps, w.h.p., while the total energy consumption and the average transmission radius remain almost the same as in the corresponding $G(n, r)$ graph.

[2] When we write "with very high probability" or "w.v.h.p.", we mean with probability $1 - o(n^{-1})$.

[3] The running time of this algorithm is guaranteed with high probability. "With high probability" or "w.h.p." means with probability $1 - o(1)$.

2 Broadcasting in $G_{\geq r_{\min}}$

In this section, we consider the geometric random graph model $G_{\geq r_{\min}} = (V, E)$ defined in the previous section. In this graph, a vertex u has an outgoing edge to a vertex v in $G_{\geq r_{\min}}$ iff the corresponding Euclidean distance is smaller than the radius assigned to u. We assume that $r_{\min} \geq \delta\sqrt{\log n}$, where δ is a large constant. Then, $G_{\geq r_{\min}}$ is connected with very high probability [30]. In the rest of the paper $S((x, y), (x', y'))$ denotes the rectangle delimited by the points $(x, y), (x, y'), (x', y)$, and (x', y'), where $0 \leq x \leq x' \leq \sqrt{n}$ and $0 \leq y \leq y' \leq \sqrt{n}$. The distance between two nodes (x, y) and (x', y') means the Euclidean distance between them and is denoted by $dist((x, y), (x', y'))$. The number of hops from a node u to a node v represents the length of a shortest path from u to v in the resulting graph. The set of points in $[0, \sqrt{n}]^2$ lying within the transmission radius of at least one of the nodes of some subset $S \subseteq V$ is called the area covered by S. In the sequel (x_0, y_0) represents the node in which the message which has to be spread to all nodes is placed at time 0.

In order to show that a message can efficiently be spread to all nodes of such a graph, we first state the following proposition.

Proposition 1. *In a graph $G_{\geq r_{\min}}$ (or $G_{\geq c}$) there are $\Omega(n/r^{\alpha-1})$ nodes with radius at least r, with probability $1 - o(n^{-2})$, for any $r \geq r_{\min}$ (or $r \geq c$).*

Proof. We know that in this graph a node has been assigned radius r according to the probability density function $\rho(r) = (\alpha - 1)r_{\min}r^{-\alpha}$, independently of all other nodes. This implies that a node has radius larger than some r with probability $\int_r^\infty (\alpha - 1)r_{\min}^{\alpha-1}x^{-\alpha}\mathrm{d}x = r_{\min}^{\alpha-1}r^{-(\alpha-1)}$. Hence, using the Chernoff bounds [3,21] we conclude that there are less than $\epsilon n/r^{\alpha-1}$ nodes, which have radius at least r, with probability at most

$$\begin{aligned}
&\sum_{i=n-\epsilon n/r^{\alpha-1}}^{n} \binom{n}{i} \left(1 - \left(\frac{r_{\min}}{r}\right)^{\alpha-1}\right)^{i} \left(\frac{r_{\min}}{r}\right)^{(\alpha-1)(n-i)} \\
&\leq \left(\frac{1 - (r_{\min}/r)^{\alpha-1}}{1 - \epsilon/r^{\alpha-1}}\right)^{n(1-\epsilon/r^{\alpha-1})} \left(\frac{(r_{\min}/r)^{\alpha-1}}{\epsilon/r^{\alpha-1}}\right)^{n\epsilon/r^{\alpha-1}} \qquad (1)\\
&= \left(1 - \frac{r_{\min}^{\alpha-1} - \epsilon}{r^{\alpha-1} - \epsilon}\right)^{n(1-\epsilon/r^{\alpha-1})} \left(\frac{r_{\min}^{\alpha-1}}{\epsilon}\right)^{n\epsilon/r^{\alpha-1}}
\end{aligned}$$

which equals $o(n^{-2})$ whenever $r_{\min} = \Omega(1)$ and ϵ is small enough. □

Proposition 1 implies that in a graph $G_{\geq r_{\min}}$ there are $\Omega(1)$ nodes with radius at least $2\sqrt{n}$, with probability $1 - o(n^{-2})$. We should note that in the case of $G_{\geq c}$ we may replace Ω by Θ in the statement of Proposition 1.

Now we consider broadcasting in the $G_{\geq r_{\min}}$ graph. Here, we only consider the case $r_{\min} < 2^{\log^\epsilon n}$, where ϵ may be any constant smaller than 1, and show that for these graphs broadcasting can be performed in time $O(D(G_{\geq r_{\min}}))$, w.h.p., where $D(G_{\geq r_{\min}})$ denotes the diameter of the graph. The same results can also be shown for any $G_{\geq r_{\min}}$ with $r_{\min} = n^{o(1)}$, however, the case $r_{\min} > 2^{\log^{\epsilon} n}$ for *any* $\epsilon < 1$ would require an elaborate case analysis which is omitted in this extended abstract.

Now we concentrate on a lower bound on the diameter of $G_{\geq r_{\min}}$.

Theorem 1. *If* $r_{\min} < 2^{\log^{\epsilon} n}$ *for some constant* $\epsilon < 1$*, then the diameter of* $G_{\geq r_{\min}}$ *is* $\Omega(\log\log n)$*, w.v.h.p.*

The proof of this theorem is omitted due to space limitations. Intuitively, with some constant probability a node v with radius r_v can only reach nodes with radius at most $r_v^{\Theta(1)}$, and hence, there is a node with radius $r_{\min}$ which needs at least $\Omega(\log\log n)$ hops to reach a node with radius $\Theta(\sqrt{n})$, w.v.h.p.

Now we show that there exists an optimal distributed broadcasting algorithm in $G_{\geq r_{\min}}$. The idea behind the algorithm is that, with sufficient probability, each node u has an edge to a node v with a somewhat larger radius. Among the several such nodes v, one can be selected by having all such nodes v reply with a probability inversely proportional to their expected number, after which the chosen node can replace u and repeat the procedure. Then, after O(log log n) steps, the broadcast message reaches a node with an edge to every other node. A precise description of the algorithm is given in the next two paragraphs.

Let (x_0, y_0) denote the vertex which has the broadcast message at the beginning and assume that its radius r_0 is smaller than $\log^3 n$. In the first round this node transmits the message, and its transmission range r_0, together with a control bit set to 1. The succeeding rounds consist of several steps. In the second round the informed nodes which have their radii in the range $[3r_0, 6r_0]$ transmit in each odd step with probability $1/(r_{\min}^{\alpha-1} r_0^{3-\alpha})$ a control bit set to 0. If in some odd step (x_0, y_0) receives the control bit, i.e., exactly one of the informed nodes with the properties described above was transmitting, then (x_0, y_0) sends in the next (even) step a control bit set to 1. In the next *even* step the node that sent the control bit, received by (x_0, y_0) three steps before, transmits the message and its transmission range r_1, together with the control bit set to 1.

Generally, in some round $i > 1$ we consider two cases. If the radius r_{i-2} of the node (x_{i-2}, y_{i-2}) is smaller than $\log^{4/\epsilon} n$, where $\epsilon < 6 - 2\alpha$ is some constant, then in each odd step of this round, the nodes which received the message in the last step of round $i-1$ from the node (x_{i-2}, y_{i-2}) **and** have their radius in the range $[3r_{i-2}, 6r_{i-2}]$ transmit with probability $1/(r_{\min}^{\alpha-1} r_{i-2}^{3-\alpha})$ a control bit set to 0. If $r_{i-2} > \log^{4/\epsilon} n$, then the nodes which received the message in the last step of round $i-1$ from the node (x_{i-2}, y_{i-2}) and have their radius in the range $[r_{i-2}^{(4-\epsilon)/(2(\alpha-1))}, 2r_{i-2}^{(4-\epsilon)/(2(\alpha-1))}]$ transmit with probability $1/(r_{i-2}^{\epsilon/2} r_{\min}^{\alpha-1})$ the control bit set to 0. In both cases if in some odd step the node (x_{i-2}, y_{i-2}) receives the control bit, i.e., only one of the nodes in its range with the properties described above has sent a message in the most recent step, then (x_{i-2}, y_{i-2}) transmits in the next (even) step the control bit set to 1. In the following *even* step, the single node which transmitted the control bit three steps before transmists the message and its transmission range r_{i-1}, together with the control bit set to 1. This transmitting node is denoted after this step by (x_{i-1}, y_{i-1}), and round $i+1$ begins.

Theorem 2. *Let $G_{\geq r_{\min}}$ be the graph defined at the beginning of this section, where $r_{\min} \geq \delta\sqrt{\log n}$. Furthermore, let a message be placed in one of the nodes of $G_{\geq r_{\min}}$. Then, the randomized distributed radio broadcasting algorithm described above spreads the message to all nodes of $G_{\geq r_{\min}}$ in $O(\log\log n)$ steps, w.h.p.*

Proof. In order to show that the algorithm described above informs a node with radius $2\sqrt{n}$ within $O(\log\log n)$ rounds, w.h.p., we first prove that any node with some radius $r \in [\delta\sqrt{\log n}, \log^{4/\epsilon} n]$ reaches $\Theta(r_{\min}^{\alpha-1} r^{3-\alpha})$ nodes which have their radii in $[3r, 6r]$, w.v.h.p. As in the proof of Proposition 1, we can show that a node has its radius in the range $[3r, 6r]$ with probability $\int_{3r}^{6r} (\alpha-1) r_{\min} x^{-\alpha} \mathrm{d}x = (6^{\alpha-1} - 3^{\alpha-1})/18^{\alpha-1} \cdot r_{\min} r^{-(\alpha-1)}$, independently. Applying now the Chernoff bounds [3,21] we obtain that with probability $1-o(n^{-2})$ there are $\Theta(n r_{\min}^{\alpha-1} r^{-(\alpha-1)})$ nodes which have their radii in the range $[3r, 6r]$. These nodes fall into the range of a fixed node with radius r with probability $\pi r^2/n$, independently. Hence, the Chernoff bounds imply that there are $\Theta(r_{\min}^{\alpha-1} r^{3-\alpha})$ nodes in the range of a fixed node with radius r, w.v.h.p., whenever δ is large enough.

Next we show that any node with radius $r \geq \log^{4/\epsilon} n$ reaches $\Theta(r^{\epsilon/2} r_{\min}^{\alpha-1})$ nodes which have their radii in $\left[r^{(4-\epsilon)/(2(\alpha-1))}, 2r^{(4-\epsilon)/(2(\alpha-1))}\right]$, w.v.h.p. As before, we conclude that there are $\Theta(n r_{\min}^{\alpha-1} r^{-(4-\epsilon)/2})$ nodes which have their radii in the range $\left[r^{(4-\epsilon)/(2(\alpha-1))}, 2r^{(4-\epsilon)/(2(\alpha-1))}\right]$, w.v.h.p. Since any node falls into the range of a fixed node with radius r with probability $\pi r^2/n$ (we ignore border effects), independently, applying the Chernoff bounds we obtain that there are $\Theta(r_{\min}^{\alpha-1} r^{\epsilon/2})$ nodes in the range of a fixed node with radius r, w.v.h.p. Combining the results of the previous two paragraphs, we conclude that the diameter of $G_{\geq r_{\min}}$ is $O(\log\log n)$.

In order to conclude the proof, let $X_{i,j}$ be a random variable which is 1 if in the jth odd step of the ith round only one node transmits the control bit set to 0, and 0 otherwise. Furthermore, let $A_{i,j}$ denote the event that $E[X_{i,j}] = \Theta(1)$. Then, due to the choice of the nodes, $\Pr[X_{i,j}|A_{i,j}] = \Theta(1)$ for any i, j. We denote by Y_l a random variable which is 1 if exactly one node transmits the control bit set to 0 in the lth odd step (the odd steps are now counted over the whole time period), and A_l is the event that $E[X_l] = \Theta(1)$. We are looking now for some T such that $\Pr[\sum_{l=1}^{T} Y_l \geq \phi \cdot D(G_{\geq r_{\min}}) | \cup_{i=1}^{T} A_l] = 1 - o(1/D(G_{\geq r_{\min}}))$, where $D(G_{\geq r_{\min}})$ is the diameter of $G_{\geq r_{\min}}$ and ϕ is some (large) constant. Since $\Pr[Y_l = 1|A_l] = \Omega(1)$, independently, we can use the Chernoff bounds [3,21], and obtain that $T = \Theta(\log\log n)$. Since A_l occurs with very high probability, applying the Union bound over $O(\log\log n)$ steps we obtain that a node with radius at least $2\sqrt{n}$ gets the message within $O(\log\log n)$ steps, w.h.p. Such a node node transmits the message alone in a time step with constant probability. This implies that within additional $O(\log\log n)$ steps all nodes receive the message, w.h.p., and the theorem follows. □

Applying similar arguments as in the previous proof, one can show that if the algorithm presented above is run for $O(\log n)$ steps, one can disseminate a message to all nodes of $G_{\geq r_{\min}}$ with *very high probability* (instead of probability $1-o(1)$). Using the so called echo procedure from [23], we can derandomize the algorithm described in the proof of Theorem 2 (as well as the algorithm described in Theorem 5), and obtain the same results as before. The result of Theorem 2 can also be extended to random geometric graphs obtained from a homogeneous Poisson point process with some intensity which exceeds the connectivity threshold value. Please refer to [30] for details.

3 Broadcasting in $G_{\geq c}$

In this section we consider the $G_{\geq c}$ model defined in the introduction. Due to the choice of c, this graph is not necessarily strongly connected, however, it contains a strongly connected giant component of size $\Theta(n)$, w.v.h.p. [30]. Then, we can state the following theorem.

Theorem 3. *If $q_2 = 1/(3-\alpha)$, then the diameter of the giant component in $G_{\geq c}$ is $O(\log^{2q_2} n)$, w.v.h.p.*

Proof. In this proof we only show (due to simplicity reasons) that for any (slow-growing) function $f(n) \in [\omega(1), O(\log\log n)]$ the diameter of the giant component of $G_{\geq c}$ is $O(f(n)\log^{2q_2} n)$. To simplify the analysis, let the graph $G_{\geq c}$ be constructed in two steps. First, construct a graph G'_c by placing the nodes with radius $r \leq f^{2/5}(n)\log^{q_2} n$ in $[0,\sqrt{n}]^2$, uniformly at random. In a second step, place the remaining nodes and obtain the graph $G_{\geq c}$.

Let u be a node of G'_c, and let v be another node which is $f(n)\log^{2q_2} n$ hops away from u in G'_c (whenever such a node exists). Furthermore, let $P(u,v) = (u, u_1, u_2, \ldots, u_{f(n)\log^{2q_2} n-1}, v)$ denote a shortest path between u and v in G'_c. We show in the following that the nodes $u_1, \ldots, u_{f(n)\log^{2q_2} n-1}$ cover an area of $\Omega(f(n)\log^{2q_2} n)$.

Assume for simplicity that $\sqrt{n}$ is an integer, c is even, and $c/2$ divides $\sqrt{n}$. Let $C(i,j)$ denote the square $S((ic/2, jc/2), ((i+1)c/2, (j+1)c/2))$. Now we show that any such square contains at most two nodes which lie on $P(u,v)$. Let us assume that there is some square $C(i,j)$ which contains three nodes u_{s_1}, u_{s_2}, and u_{s_3} lying on $P(u,v)$. Since the diameter of $C(i,j)$ is $\sqrt{2}c/2 < c$ every node in this square reaches any other node within $C(i,j)$. Then, u_{s_1} has u_{s_2} and u_{s_3} in its range, and $P'(u,v) = (u, \ldots u_{s_1}, u_{s_3}, u_{s_3+1}, \ldots, v)$ is a valid path from u to v. Since $|P'(u,v)| < |P(u,v)|$, $P(u,v)$ cannot be a shortest path from u to v, which contradicts our assumption. Summarizing, the nodes of $P(u,v)$ cover an area of at least $f(n)\log^{2q_2} nc^2/8$.

According to Proposition 1 there are $\Omega(n/(f^{2/5}(n)\log^{q_2} n)^{\alpha-1})$ with radius larger than $f^{2/5}(n)\log^{q_2} n$, with probability $1-o(n^{-2})$. Given that there are $\Omega(n/(f^{2/5}(n)\log^{q_2} n)^{\alpha-1})$ nodes with radius larger than $f^{2/5}(n)\log^{q_2} n$ in $G_{\geq c}$, the area covered by $P(u,v)$ contains no node having radius $r > f^{2/5}(n)\log^{q_2} n$ with probability

$$\left(1 - \frac{\Omega(f(n)\log^{2q_2} n)}{n}\right)^{\Omega\left(\frac{n}{(f^{2/5}(n)\log^{q_2} n)^{\alpha-1}}\right)} \leq o(e^{-\Omega(\sqrt[5]{f(n)}\log n)}) \leq o(n^{-3}).$$

Therefore, there is some node with radius $r > f^{2/5}(n) \log^{q_2} n$ placed in the area covered by $P(u, v)$ with probability $1 - o(n^{-3})$. This implies that u reaches a node which has radius larger than $f^{2/5}(n) \log^{q_2} n$ in $O(f(n) \log^{2q_2} n)$ steps, with probability $1 - o(n^{-3})$. Applying now the Union bound over all nodes of G'_c, we conclude that all nodes, which are connected to some other node via $f(n) \log^{2q_2} n$ hops, can reach a node with radius larger than $f^{2/5}(n) \log^{q_2} n$ in $O(f(n) \log^{2q_2} n)$ steps, with probability $1 - o(n^{-2})$. If for some node w isn't any node w' at $f(n) \log^{2q_2} n$ hops from w in G'_c, but w is in the giant component of $G_{\geq c}$, then w must reach a node with radius $r > f^{2/5}(n) \log^{q_2} n$ in $O(f(n) \log^{2q_2} n)$ hops. This holds since w reaches every node in its strong component in G'_c within less than $O(f(n) \log^{2q_2} n)$ hops, and this component joins the giant component of $G_{\geq c}$ via a node of $G_{\geq c} \setminus G'_c$. According to the definition of G'_c, a node of $G_{\geq c} \setminus G'_c$ has radius larger than $f^{2/5}(n) \log^{q_2} n$.

Now we show that in the range of any node which has radius $r > f^{2/5}(n) \log^{q_2} n$ there is at least one node with radius larger than

$$f^{1/5}(n) \log^{q_2} n \cdot (r/(f^{1/5}(n) \log^{q_2} n))^{1+(3-\alpha)/(\alpha-1)},$$

with probability $1 - o(n^{-2})$. Given that there are $\Omega(n/r^{\alpha-1})$ nodes which have radii larger than r, there is no node with a radius larger than $f^{1/5}(n) \log^{q_2} n \cdot (r/(f^{1/5}(n) \log^{q_2} n))^{1+(3-\alpha)/(\alpha-1)}$ in the range of a node having radius r with probability

$$\left(1 - \frac{\pi r^2}{n}\right)^{\Omega\left(\frac{n}{\left(f^{1/5}(n) \log^{q_2} n \left(\frac{r}{f^{1/5}(n) \log^{q_2} n}\right)^{1+\frac{3-\alpha}{\alpha-1}}\right)^{\alpha-1}}\right)} \leq e^{-\omega(\log n)} = o(n^{-3}).$$

We conclude by applying the Union bound over all nodes with radius larger than $f^{2/5}(n) \log^{q_2} n$. Iterating this procedure $O(\log n)$ times we obtain that one can reach a node with radius $2\sqrt{n}$ within $O(\log n)$ additional hops.

Summarizing, any node of the giant component reaches within $O(f(n) \log^{2q_2} n)$ hops a node with radius $2\sqrt{n}$, w.v.h.p., and the theorem follows. □

We might ask whether the upper bound given in Theorem 3 is asymptotically tight. A related open question was formulated in [30] about the second largest component in $G_{\geq c}$, namely whether the second largest component of the traditional $G(n, r)$ model with $r = c$ is of size $\Theta(\log^2 n)$. Concerning the diameter of the giant component in $G_{\geq c}$ we can only prove a lower bound of $\Omega(\log n)$.

Theorem 4. *The diameter of the giant component of $G_{\geq c}$ is $\Omega(\log n)$, w.v.h.p.*

The proof of this theorem uses similar techniques as Theorem 3. Due to lack of space, we do not prove Theorem 4 here.

The results of Theorems 3 and 4 can be extended to further random geometric graph models. Consider for example the graph $G_{\geq c}$, in which we enlarge the radius of a node in any strongly connected component so that the graph becomes

```
DISTRIBUTED ALGORITHM FOR BROADCASTING IN $G_{\geq c}$
for $t = 1$ to $O(D(G_{\geq c}))$ do
    for $s = 1$ to $1024(c \log\log n)^2$ do
        for every vertex $v = (x', y')$ in parallel do
            $s' \leftarrow (s-1) - ((s-1) \mod 256c^2)$
            $i' - 1 \leftarrow \frac{s'}{256c^2} \mod 2\log\log n$
            $i - 1 \leftarrow \frac{1}{2\log\log n} \cdot (\frac{s'}{256c^2} - (i'-1))$
            $j \leftarrow \lfloor (x' \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor$
            $j' \leftarrow \lfloor (y' \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor$
            if $r'(v) \in [c^{i'}, c^{i'+1}]$ and $j = \lfloor \frac{s-1}{16c} \rfloor \mod 16c$ and $j' = (s-1) \mod 16c$ then
                $v$ transmitts with probability $\frac{1}{c^i}$
            end if
        end for
    end for
end for
```

Fig. 1. Algorithm used in the proof of Theorem 5. Here $r'(v)$ denotes the transmission radius of node v.

strongly connected. Another model is the extension of the point Poisson process on $[0, \sqrt{n}]^2$ with intensity c, whereas the radii are distributed as in the $G_{\geq c}$ model. In all these models it is possible to broadcast any message, placed initially in one of the nodes of the giant component, to all nodes of the graph within $O(\log^{2/(3-\alpha)} n)$ steps, w.v.h.p.

Before we start with the analysis of radio broadcasting in $G_{\geq c}$ we first give a high level description of our broadcasting algorithm. The algorithm consists of two main phases. In the first phase (cf. Figure 1) the goal is to let the message generated at a source node reach a node with radius larger than $c^{2\log\log n}$, w.v.h.p. In the second phase the message reaches a node with radius $2\sqrt{n}$, w.v.h.p. The second phase performs similarly to the algorithm presented for $G_{\geq r_{\min}}$, and thus, we omit the analysis of this phase in the paper. For the first phase, we show that the message traverses a shortest path from the source of the message to a node with radius larger than $c^{2\log\log n}$, w.v.h.p. In order to ensure that each node on this path transmits the message to the next node on the path, the algorithm consists of $O(D(G_{\geq c}))$ phases, and each phase is executed over $O((\log\log n)^2$ time steps. During these time steps, each informed node of radius r, where $r \in [c, c^{2\log\log n}]$, transmits at least once with some probability in the range $[r^{3-\alpha}/c, cr^{3-\alpha}]$. By ensuring that interferences can only occur if several nodes lying in the same square $I^{i',j,j'}$ (see Figure 2) transmit at the same time, one can show that the message will traverse the shortest path mentioned above within $O(D(G_{\geq c}))$ phases, w.v.h.p.

Formaly, the distributed algorithm that guarantees the running time given in the theorem below consists of $O(D(G_{\geq c}))$ initial rounds. In each round we have $1024(c\log\log n)^2$ steps. In step $256c^2(2(i-1)\log\log n + (i'-1)) + 16cj + j' + 1$ with $1 \leq i, i' \leq 2\log\log n$ and $0 \leq j, j' \leq 16c - 1$ any informed node

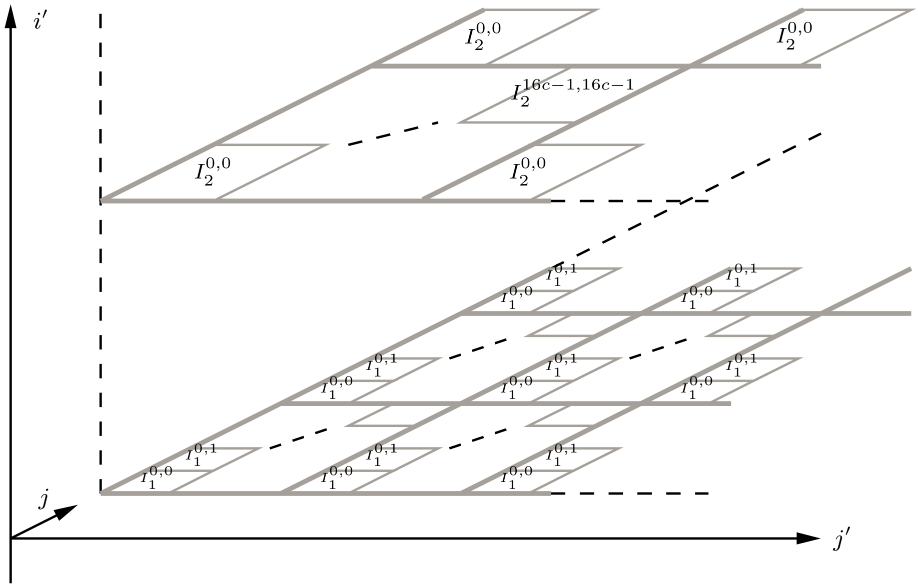


Fig. 2. The nodes with radius in the range $[c^{i'}, c^{i'+1}]$ placed in the squares denoted by $I_{i'}^{j,j'}$ transmit in step $256c^2(2(i-1)\log\log n + (i'-1)) + 16cj + j' + 1$ with probability $1/c^i$. The two planes consisting of the squares $I_1^{*,*}$ and $I_2^{*,*}$, respectively, are both embedded into $[0, \sqrt{n}]^2$ and contain the same set of points. Here, we have drawn two parallel planes for a better visualization.

(x', y') with radius $r' \in [c^{i'}, c^{i'+1}]$ and $j = \lfloor (x' \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor$, $j' = \lfloor (y' \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor$ transmits with probability $1/c^i$ (cf. Figure 1). For a pseudo code of these $O(D(G_{\geq c}))$ initial rounds see Figure 1.

After these $O(D(G_{\geq c}))$ initial rounds we reach a node with a radius in the range $[c^{2\log\log n}, 2c^{2\log\log n}]$, and then we apply a similar procedure as in Theorem 2. We only consider the first phase, which requires $O(D(G_{\geq c})(\log\log n)^2)$ steps. The second phase requires only $O(\log n)$ steps.

Now we state the main theorem of this section.

Theorem 5. *Let $G_{\geq c}$ be the graph defined at the beginning of this section, where c is a large constant. Furthermore, let a message be placed in one of the nodes of the giant component of $G_{\geq c}$. Then, the randomized distributed radio broadcasting algorithm described above spreads the message to all nodes of $G_{\geq c}$ (even to nodes outside of the giant component) in $O(D(G_{\geq c})(\log\log n)^2)$ steps, w.v.h.p., where $D(G_{\geq c})$ denotes the diameter of the giant component of $G_{\geq c}$.*

Proof. We show that within $O(D(G_{\geq c})(\log\log n)^2)$ steps any (x, y) receives the message, w.v.h.p. Obviously, two nodes (x_1, y_1) and (x_2, y_2) with radii $r_1, r_2 \in [c^{i'}, c^{i'+1}]$, where $i' \leq 2\log\log n$, cannot produce an interference at any node whenever $\lfloor (x_1 \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor \neq \lfloor (x_2 \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor$ or $\lfloor (y_1 \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor \neq \lfloor (y_2 \mod (4c^{i'+1}))/(c^{i'}/4) \rfloor$. Let now $P = (v_0, v_1, \ldots, v_k)$

be a shortest path from $(x_0, y_0) = v_0$ to $(x, y) = v_k$. We know that an informed node transmits at most $2 \log\log n$ times in a round, each time with a different probability. Let $t_{q,i,l}$ denote the time step in the lth round, in which v_q transmits with probability $1/c^i$. Furthermore, denote by $X_{q,i,l}$ a random variable which is 1 if the message reaches v_{q+1} in step $t_{q,i,l}$ and 0 otherwise. Now, v_q can produce an interference with at most $O(c^{(3-\alpha)i'} + \log n)$ other nodes, with probability $1 - o(n^{-2})$, where the radius r_q of v_q is in the range $[c^{i'}, c^{i'+1}]$. Thus, there is at least one i such that $\Pr[X_{q,i,l} = 1 \mid v_q \text{ is informed before round } l] = \Omega(1)$. Let $Y_l = X_{q,i,l}$, with $q = \max_{q'}\{v_{q'} \text{ is informed before round } l\}$, and let i be chosen such that $\Pr[X_{q,i,l} = 1] = \Omega(1)$. Then, $\Pr[Y_l = 1] = \Omega(1)$, independently. As in the proof of Theorem 2 we can show that there is some $T = O(|P| + \log n)$ such that $\Pr[\sum_{l=1}^{T} Y_l \geq |P|] = 1 - o(n^{-2})$. Since each round consists of $O((\log\log n)^2)$ steps, (x, y) becomes informed within $O((|P| + \log n)(\log\log n)^2)$ steps, with probability $1 - o(n^{-2})$. Applying now the Union bound over all nodes of the graph, we obtain that within $O(D(G_{\geq c})(\log\log n)^2)$ steps a node with radius in the range $[c^{2\log\log n}, 2c^{2\log\log n}]$ receives the message. If now c is large enough, using the same arguments as in the proof of Theorem 2 we conclude that within additional $O(\log n)$ steps the message reaches any node of the graph, w.v.h.p.

□

As in the case of Theorem 2, the result of Theorem 5 can also be extended to random geometric graphs obtained from a homogeneous Poisson process with a corresponding intensity.

We know that a message, placed on one of the nodes of a $G(n, r)$ graph, can be spread to all other nodes within $\tilde{\Theta}(\sqrt{n})$ steps [13,30], v.w.h.p., where $r = \log^{c'} n$ with $c' \geq 1/2$ and $\tilde{\Theta}$ is the Θ-fuction omitting polylogarithmic terms. The total energy consumption needed for transmission during the broadcasting process in the network is bounded by $\tilde{\Theta}(n)$. However, if we consider our results for α being a constant in the range $(2, 3)$, then we may perform broadcasting in time $\tilde{\Theta}(\log n)$, and the total energy consumption needed for transmissions is still bounded by $\tilde{\Theta}(n)$. Moreover, the average transmission radius is asymptotically the same as in the corresponding $G(n, r)$ graph. Thus, our results imply that if we are given n radio transmitters, and we are allowed to set the transmission radius of each of these devices before they are placed uniformly at random in $[0, \sqrt{n}]^2$, then we are able to design a radio network, which supports broadcasting in (poly)logarithmic time and keeps the energy consumption in the network very low. Furthermore, our results can also be extended to the case when the transmission radii of the nodes vary in time, independently, according to a power law distribution with some exponent $\alpha \in (1, 3)$.

4 Conclusion

As described in the introduction, our main intention was to derive efficient algorithms for radio broadcasting in wireless networks which are modeled by random geometric graphs containing nodes with different transmission radii. The results presented here can only be viewed as a first step in this direction, and there are still several interesting open problems in this field which are worth to be analyzed. In the case of the $G_{\geq c}$ model for example there is still a gap of $\log^{\Theta(1)} n$ between the upper and lower bound w.r.t. the diameter of the giant component of the graph, and it would be of great interest to close this gap. Another open problem is whether it is possible to broadcast a piece of information in G_c within $O(D(G_{\geq c}))$ steps.

References

1. Berenbrink, P., Cooper, C., Hu, Z.: Energy efficient randomised communication in unknown adhoc networks. In: Proc. of 19th SPAA 2007, pp. 250–259 (2007)
2. Brusci, D., Pinto, M.D.: Lower bounds for the broadcast problem in mobile radio networks. Distributed Computing 10(3), 129–135 (1997)
3. Chernoff, H.: A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. Ann. Math. Stat. 23, 493–507 (1952)
4. Chlamtac, I., Kutten, S.: On broadcasting in radio networks - problem analysis and protocol design. IEEE Transactions on Communications 33(12), 1240–1246 (1985)
5. Chlebus, B., Gąsieniec, L., Östlin, A., Robson, J.: Deterministic radio broadcasting. In: Welzl, E., Montanari, U., Rolim, J. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 717–728. Springer, Heidelberg (2000)
6. Chlebus, B., Gąsieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in ad hoc radio networks. Distributed Computing 15(1), 27–38 (2002)
7. Chrobak, M., Gąsieniec, L., Rytter, W.: Fast broadcasting and gossiping in radio networks. Journal of Algorithms 43(2), 177–189 (2002)
8. Cicalese, F., Manne, F., Xin, Q.: Faster centralized communication in radio networks. In: Asano, T. (ed.) ISAAC 2006. LNCS, vol. 4288, pp. 339–348. Springer, Heidelberg (2006)
9. Clementi, A., Monti, A., Silvestri, R.: Distributed broadcast in radio networks with unknown topology. Theoretical Computer Science 302, 337–364 (2003)
10. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. Journal of Algorithms 60(2), 115–143 (2006)
11. Czumaj, A., Wang, X.: Fast messgae dissemination in random geometric ad-hoc radio networks. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 220–231. Springer, Heidelberg (2007)
12. De Marco, G., Pelc, A.: Faster broadcasting in unknown radio networks. Information Processing Letters 79(2), 53–56 (2001)
13. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. Journal of Discrete Algorithms 5, 187–201 (2007)
14. Elkin, M., Kortsarz, G.: An improved algorithm for radio broadcast. ACM Transactions on Algorithms 3(1) (2007)
15. Elsässer, R., Gąsieniec, L.: Radio communication in random graphs. Journal of Computer and Systems Sciences 72, 490–506 (2006)
16. Emek, Y., Gąsieniec, L., Kantor, E., Pelc, A., Peleg, D., Su, C.: Broadcasting in udg radio networks with unknown topology. In: Proc. of PODC 2007, pp. 195–204 (2007)
17. Gaber, I., Mansour, Y.: Centralized broadcast in multihop radio networks. Journal of Algorithms 46(1), 1–20 (2003)

18. Ganesan, D., Govindan, R., Shenker, S., Estrin, D.: Highly resilient, energy-efficient multipath routing in wireless sensor networks. ACM SIGMOBILE Mobile Computing and Communication Review 5(4), 11–25 (2001)
19. Gąsieniec, L., Pagourtzis, A., Potapov, I., Radzik, T.: Deterministic communication in radio networks with large labels. Algorithmica 47(1), 97–117 (2007)
20. Gąsieniec, L., Peleg, D., Xin, Q.: Faster communication in known topology radio networks. Distributed Computing 19(4), 289–300 (2007)
21. Hagerup, T., Rüb, C.: A guided tour of Chernoff bounds. Information Processing Letters 36(6), 305–308 (1990)
22. Ishizuka, M., Aida, M.: Achieving power-law placement in wireless sensor networks. In: Proc. of ISADS 2005, pp. 661–666 (2005)
23. Kowalski, D., Pelc, A.: Time of deterministic broadcasting in radio networks with local knowledge. SIAM Journal on Computing 33, 870–891 (2004)
24. Kowalski, D., Pelc, A.: Broadcasting in undirected ad hoc radio networks. Distributed Computing 18(1), 43–57 (2005)
25. Kowalski, D., Pelc, A.: Optimal deterministic broadcasting in known topology radio networks. Distributed Computing 19(3), 183–195 (2007)
26. Krishnamachari, B., Wicker, S., Bejar, R., Pearlman, M.: Critical density thresholds in distributed wireless networks. In: Communications, Information and Network Security. Kluwer Academic Publishers, Dordrecht (2002)
27. Lotker, Z., Navarra, A.: Managing random sensor networks by means of grid emulation. In: Boavida, F., Plagemann, T., Stiller, B., Westphal, C., Monteiro, E. (eds.) NETWORKING 2006. LNCS, vol. 3976, pp. 856–867. Springer, Heidelberg (2006)
28. Meguerdichian, S., Koushanfar, F., Potkonjak, M., Srivastava, M.: Coverage problems in wireless ad-hoc sensor networks. In: Proc. of INFOCOM 2001, vol. 3, pp. 1380–1387 (2001)
29. Muthukrishnan, S., Pandurangan, G.: The bin-covering technique for thresholding geometric graph properties. In: Proc. of 16th SODA 2005, pp. 989–998 (2005)
30. Penrose, M.: Random Geometric Graphs. Oxford Studies in Probability (2003)
31. Sen, A., Huson, M.L.: A new model for scheduling packet radio networks. In: Proc. of INFOCOM 1996, pp. 1116–1124 (1996)

Ping Pong in Dangerous Graphs: Optimal Black Hole Search with Pure Tokens

Paola Flocchini[1], David Ilcinkas[2], and Nicola Santoro[3]

[1] SITE, University of Ottawa, Canada
flocchin@site.uottawa.ca
[2] CNRS, Université de Bordeaux, France
david.ilcinkas@labri.fr
[3] School of Computer Science, Carleton University, Canada
santoro@scs.carleton.ca

Abstract. We prove that, for the black hole search problem, the pure token model is computationally as powerful as the whiteboard model; furthermore the complexity is exactly the same. More precisely, we prove that a team of *two* asynchronous agents, each endowed with a single identical pebble (that can be placed only on nodes, and with no more than one pebble per node) can locate the black hole in an arbitrary network of known topology; this can be done with $\Theta(n \log n)$ moves, where n is the number of nodes, even when the links are not FIFO.

Keywords: distributed computing, graph exploration, mobile agents, autonomous robots, dangerous graphs.

1 Introduction

1.1 The Framework

Black Hole Search (Bhs) is the distributed problem in a networked system (modeled as a simple edge-labelled graph G) of determining the location of a *black hole* (Bh): a site where any incoming agent is destroyed without leaving any detectable trace. The problem has to be solved by a team of identical system agents injected into G from a safe site (the homebase). The team operates in presence of an adversary that chooses e.g., the edge labels, the location of the black hole, the delays, etc. The problem is solved if at least one agent survives and all surviving agents know the location of the black hole (e.g., see [15]).

The practical interest of Bhs derives from the fact that a black hole can model several types of faults, both hardware and software, and security threats arising in networked systems supporting code mobility. For example, the crash failure of a site in an asynchronous network turns such a site into a black hole; similarly, the presence at a site of a malicious process (e.g., a virus) that thrashes any incoming message (e.g., by classifying it as spam) also renders that site a black hole. Clearly, in presence of such a harmful host, the first step must be to to determine and report its location.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 227–241, 2008.

From a theoretical point of view, the natural interest in the computational and complexity aspects of this distributed problem is amplified by the fact that it opens a new dimension in the classical *graph exploration* problem. In fact, the black hole can be located only after all the nodes of the network but one have been visited and are found to be safe; in this exploration process some agents may disappear in the black hole. In other words, while the existing wide body of literature on *graph exploration* (e.g., see [1, 2, 8, 9, 16, 17]) assumes that the graph is *safe*, Bhs opens the research problems of the exploration of *dangerous graphs*.

Indeed Bhs has been studied in several settings, under a variety of assumptions on the power of the adversary and on the capabilities of the agents; e.g., on the level of synchronization of the agents; on whether or not the links are FIFO; on the type of mechanisms available for inter agent communication and coordination; on whether or not the agents have a map of the graph. In these investigations, the research concern has been to determine under what conditions and at what cost mobile agents can successfully accomplish this task. The main complexity measures are the size of the team (i.e., the number of agents employed) and the number of moves performed by the agents; sometimes also time complexity is considered.

In this paper we are interested in the weakest settings that still make the problem solvable. Thus we will make no assumptions on timing or delays, and focus on the *asynchronous* setting. Indeed, while the research has also focused on the *synchronous* case [5, 6, 7, 18, 19] where all agents are synchronized and delays are unitary, the main body of the investigations has concentrated on the asynchronous one (e.g., [4, 10, 11, 12, 13]).

1.2 The Quest and Its Difficulties

In the asynchronous setting, the majority of the investigations operate in the *whiteboard* model: every node provides a shared space for the arriving agents to read and write (in fair mutual exclusion). The whiteboard model is very powerful: it endows the agents not only with direct and explicit communication capabilities, but also with the means to overcome severe network limitations; in particular, it allows the software designer to assume FIFO links (even when not supported by the system). Additionally, whiteboards allow to break symmetry among identical agents. Indeed, whiteboards (and even stronger inter-agent coordination mechanisms) are supported by most existing mobile agent platforms [3]. The theoretical quest, on the contrary, has been for the weakest interaction mechanism allowing the problem to be solved.

A weaker and less demanding interaction mechanism is the one assumed by the *token* model, used in the early investigations on (safe) graph exploration; it is provided by identical *pebbles* (that can be placed on nodes, picked up and carried by the agents) without any other form of marking or communication (e.g., [2]).

The research quest is to determine if pebbles are computationally as powerful as whiteboards with regards to Bhs. The importance of this quest goes beyond the specific problem, as it would shed some light on the relative computational power of these two interaction mechanisms.

Two results have been established so far in this quest. In [10] it has been shown that $\Delta + 1$ agents[1] without a map (the minimum team size under these conditions), each endowed with an identical pebble, can locate the black hole with a (very high but) polynomial number of moves. In [13] it has been shown that two agents with a map (the minimum team size under these conditions), each endowed with a constant number of pebbles, can locate the black hole in a ring network with $\Theta(n \log n)$ moves, where n denotes the number of nodes in the network.

Although they indicate that Bhs can be solved using pebbles instead of whiteboards, these results do not prove yet the computational equivalence for Bhs of these two inter-agent coordination mechanisms. There are two main reasons for this. The first main reason is that both results assume *FIFO links*; note that the whiteboard model allows to work assuming FIFO links, but does not require them. Hence, the class of networks for which the results of [10, 13] apply is smaller than that covered with whiteboards; also such an assumption is a powerful computational help to any solution protocol. The second and equally important reason is that these results are *not* established within the “pure” token model used in the traditional exploration problem. In fact, in [10, 13] the agents are allowed to place pebbles not only on nodes but also on links (e.g., to indicate on which link it is departing); this gives immediately to a single token the computational power of $O(\log \Delta)$ bits of information. In [13], where the network considered is only a ring, each agent has available several tokens, and multiple tokens can be placed at the exact same place (node or link) to store more than one bit of information.

1.3 Our Results

In this paper, we provide the first proof that indeed the pure token model is computationally as powerful as the whiteboard model for Bhs.

The context we examine is the one of agents with a map in an arbitrary graph. For this context we prove that: A team of *two* asynchronous agents, each endowed with a single identical pebble (that can be placed only on nodes, and at no more than one pebble per node) and a map of the graph can locate the black hole with $\Theta(n \log n)$ moves, even if the links are not FIFO.

In other words, for networks of known topology, using pure tokens it is possible to obtain exactly the same optimal bounds for team size and number of moves as using whiteboards.

Note that our result implies as a corollary an optimal solution for the whiteboard model using only a single bit of shared memory per node; the existing solution [11] requires a whiteboard of $O(\log n)$ bits at each node.

Our results are obtained using a new and (surprisingly) simple technique called *ping pong*. In its bare form, this technique solves the problem but with

[1] Δ denotes the maximun node degree in G.

$O(n^2)$ moves. To obtain the optimal bound, the technique is enhanced by integrating it with additional mechanisms, exploiting two ideas developed in previous investigations: "split work" [12], and "distance counting" [13]. The mechanisms that we have developed use a variety of novel not-trivial techniques, and are the first to overcome the severe limitation imposed by the lack of the FIFO assumption (available instead in all previous investigations with whiteboards or tokens).

The paper is organized as follows. We first present our techniques, prove their properties and analyze their complexity in the case of ring networks (Section 3). Then, in Section 4, we show how to modify and enhance those techniques so to obtain the same bounds also in the case of arbitrary graphs.

Due to space limitations, some proofs and the code of the algorithms are omitted; the interested reader can find them in [14].

2 Terminology and Definitions

Let $G = (V, E)$ be a simple biconnected[2] graph with $n = |V|$ nodes. At each node x, there is a distinct label from a totally ordered set associated to each of its incident links. We shall denote by (G, λ) the resulting edge-labelled graph.

Operating in (G, λ) is a team of identical autonomous mobile agents (or robots). All agents enter the system from the same node, called homebase. The agents have computing capabilities, computational storage (polynomially bounded by the size of the graph), and a map of (G, λ) with the indication of the homebase; they can move from node to neighbouring node, and obey the same set of behavioral rules (the *algorithm*). Every agent has a *pebble*; all pebbles are identical. A pebble can be carried, put down at a node if no other pebble is already there, and picked up from a node by an agent without pebbles.

When an agent enters a node, it can see if there is a pebble dropped there; it might be however unable to see other agents there or to determine whether they are carrying a pebble with them.

The system is *asynchronous* in the sense that (i) each agent can enter the system at an arbitrary time; (ii) traveling to a node other than the black hole takes a finite but otherwise unpredictable amount of time; and (iii) an agent might be idle at a node for a finite but unpredictable amount of time. The basic computational step of an agent (executed either when the agent arrives to a node, or upon wake-up) is to look for the presence of a pebble, drop or pick up the pebble if wanted, and leave the node through some chosen port (or terminate). The whole computational step is performed in local mutual exclusion as an atomic action, i.e. as if it took no time to execute it. Links are *not* FIFO: two agents moving on the same link in the same direction at the same time might arrive at destination in an arbitrary order.

To simplify the model, we can assume without loss of generality that the transition between two states of the agent at a node plus the corresponding move are instantaneous. In other words, the waiting due to asynchrony only

[2] Note that biconnectivity is necessary for BHS to be solvable [11].

occurs after the move of the agent. Furthermore we can assume that also the actions of agents at different nodes occur at different instants.

A *black hole* is a node that destroys any incoming agent; no observable trace of such a destruction will be evident to the other agents. The location of the black hole is unknown to the agents. The *Black Hole Search* problem is to find the location of the black hole. More precisely, the problem is solved if at least one agent survives, and all surviving agents know the location of the black hole.

The two measures of complexity of a solution protocol are the number of agents used to locate the black hole and the total number of moves performed by the agents.

3 Black Hole Search in Rings

3.1 Preliminaries

Without loss of generality, we can assume that the clockwise direction is the same for both agents: for example, the direction implied by the link with the smallest label at the homebase. In the following, going right (resp. left) means going in the clockwise (resp. counterclockwise) direction. An agent exploring to the right (resp. left) is said to be a *right* (resp. *left*) agent. Using this definition, an agent *changes role* if it was a left agent and becomes a right agent or vice versa. For $i \geq 0$, the node at distance i to the right, resp. to the left, of the home base will be called node i, resp. node -i. Hence node i and $i - n$ represent the same node, for $0 \leq i \leq n$.

In the algorithm the agents obey the two following metarules:

1. An agent always ensures that a pebble is lying at u before traversing an unknown edge $\{u, v\}$ from u to v (i.e. an edge that it does not know to be safe).
2. An agent never traverses an unknown edge $\{u, v\}$ from u to v if a pebble lies at u and the pebble was not dropped there by this agent.

These metarules imply that the two agents never enter the black hole from the same edge. Moreover, each agent keeps track of its progress by storing the number of the most-right, resp. most-left, node in a variable `Last_Right`, resp. `Last_Left`, used to detect termination: when only one node remains unexplored, this node is the black hole and the agent can stop.

A (right) agent is said to traverse an edge $\{u, v\}$ from u to v using *cautious walk* if it has one pebble, it drops it at u, traverses the edge (in state `Explore-Right`), comes back to u (in state `Pick-Up-Right`), retrieves the pebble and goes again to v (in state `Ping-Right`). A (left) agent is said to traverse an edge $\{u, v\}$ from u to v using *double cautious walk* if it has one pebble and the other is at u, it goes to v (in state `Explore-Left`) carrying one pebble, the other pebble staying at u, drops the pebble at node v, comes back to u (in state `Pick-Up-Left`), retrieves the other pebble and goes again to v (in state `Ping-Left`). We will see later that double cautious walk is employed only by left agents. Note that these two cautious explorations obey the first metarule.

3.2 The Algorithm

Our algorithm is based on a novel coordination and interaction technique for agents using simple tokens, *Ping-Pong*. The idea at the basis of this technique is the following: one agent explores the "right" side and one the "left" side (the side assigned to an agent changes dynamically, due to the non-FIFO nature of the links). However, only one agent at a time is allowed to explore; the agent willing to do so must first "steal" the pebble of the other, and then can proceed to explore its allowed side. When an agent discovers that its pebble has been stolen, it goes to find it and steal the other pebble as well. This generate a "ping-pong" movements of the agents on the ring. The actual Ping-Pong technique based on this idea must however take into account the non-FIFO nature of the links, which creates a large variety of additional situations and scenarios (e.g., an agent moving to steal the pebble of the other, might "jump over" the other agent).

Algorithm `EnhancedPingPong` is divided in two phases, each one further divided into stages. The first phase is the Ping-Pong technique. The second phase, whose function is to ensure that the costs are kept low, in some cases may not be executed at all. Inside a phase, a *stage* is a maximal period during which no agent changes role.

In the first phase, exploration to the right is always done using cautious walk, while exploration to the left is always done using double cautious walk (i.e., after stealing a pebble). Note that, since an agent exploring to the right uses one pebble and an agent exploring to the left uses two pebbles, the agents cannot make progress simultaneously in two different directions because there are only two pebbles in total. This also implies that while an agent is exploring new nodes it knows all the nodes that have already been explored, as well as the position of the only unexplored node where the other agent possibly died. This prevents the agents from exploring the same node and thus from dying in the black hole from two different directions.

Phase 1. Initially both agents explore to the right. Since links are not FIFO, an agent may pass the other and take the lead without any of the two noticing it. Nevertheless, it eventually happens that one agent L finds the pebble of the other agent R, say at node p (at the latest it happens when one agent locates or dies in the black hole). When this happens L drops its pebble at node $p-1$ (if its pebble is not already there) and steals R's pebble. Having control on the two pebbles, L starts to explore left using double cautious walk. The stage has now an even number. When/if R comes back to p to retrieve its pebble, it does not find it. It then goes left in state `Pong-Right` until it finds a pebble. Agent R does eventually find a pebble because at the beginning of the stage there is a pebble at its left (at node $p-1$), and Agent L never removes a pebble before putting the other pebble further to the left. At this point R retrieves the pebble and goes right again in state `Ping-Right` and explores to the right. When/if L realizes that one of its pebble has been stolen, it changes role (and the stage changes) and explores to the right using its remaining pebble. At this point, both agents explore to the right. Again, one agent will find and steal the pebble of the other. To ensure progress in exploration, a right agent puts down its pebble only when it reaches the last visited node to the right it knows (using its variable `Last_Right`). Consequently the stealing at the end of an odd stage always occurs at least one node further to the right from two stages before. Hence the algorithm of Phase 1 is in fact correct by itself but the number of moves can be $\Theta(n^2)$ in the worst case (one explored node every $O(n)$ moves). To decrease the worst case number of moves to $O(n \log n)$, the agents switch to Phase 2 as soon as at least two nodes have been explored to the right.

Phase 2. Phase 2 uses the *halving* technique, based on an idea of [12], but highly complicated by the absence of whiteboards and by the lack of FIFO. The idea is to regularly divide the workload (the unexplored part) in two. One agent has the left half to explore (using variable `Goal_Left`), while the second agent explores the right half (using variable `Goal_Right`). These explorations are performed concurrently by using (simple) cautious walk (for a right agent, in states `Halving-Explore-Right`, `Halving-Pick-Up-Right` and `Halving-Ping-Right`). After finite time, exactly one agent finishes its part and joins the other in exploring the other part, changing role and thus changing the stage number. At some point, one agent A will see the other agent's pebble. A steals the pebble and moves it by one position to indicate a change of stage to the second agent B. It then computes the new workload, divide it into two parts (using the function `Update_Goal_Left` or `Update_Goal_Right`), and goes and explores its newly assigned part, changing role again by switching to state `Halving-From-Left-To-Right` or `Halving-From-Right-To-Left`. This can happen several times (if B remains blocked by the asynchronous adversary or if it is dead in the black hole). When/if agent B comes back to retrieve its pebble, it does not find it. It further goes back to retrieve its pebble in state `Halving-Pong-Right` (if it is a right agent). The number of moves it has to perform to find the pebble indicates how many halvings (pair of stages) it misses. Knowing that, it can compute what is the current unexplored part and what is its current workload. It then starts to explore its part. Since there are at most $O(\log n)$ stages of $O(n)$ moves each, this leads to a total number of moves of $O(n \log n)$.

The algorithm starts with a few stages of Phase 1 because Phase 2 needs some safe nodes to put the pebble that is used as a message to indicate the current partition of the workload.

Several other technical details and precautions have to be taken because of asynchrony and lack of FIFO. Due to space restrictions, the code describing all the details of the state transitions can be found in [14].

3.3 Correctness and Complexity

As explained before the algorithm consists of up to two *phases*. The first one corresponds to the case where both agents are in one of the eight states `Ping-Right`, `Ping-Left`, `Explore-Right`, `Explore-Left`, `Pick-Up-Right`, `Pick-Up-Left`, `Pong-Right`, `Put-Pebble-Right`. If this is not the case, we say that the algorithm is in its second phase. (Note that this phase may not exist in all possible executions.) An agent is said to be a *right*, resp. *left*, agent if its state ends with -Right, resp. -Left. Using this definition, an agent *changes role* if it was a left agent and becomes a right agent or vice versa. Finally, inside a phase, a *stage* is a maximal period during which no agent changes role.

For the purpose of the proofs of the main theorems, we will use the three following properties.

Property. $\mathcal{P}(p)$, with $p \in \{0, 1\}$: There is a left agent L and a right agent R. The agent L is waiting at node $p-1$, where one pebble is located. Agent L is carrying the other pebble and is in state Ping-Left. Moreover, its variable Last_Right has value p. Agent R, empty-handed, is in one of the following situations:

- it is dead in the black hole located at node $p+1$;
- it is at node $p+1$ in state Explore-Right and its variable Last_Right has value p;
- it is already back from node $p+1$ at node p in state Pick-Up-Right and its variable Last_Right has value $p+1$.

Moreover, the termination condition of agent L is not satisfied, and in the last two cases, the value Last_Left is the same for each agent.

Property. $\mathcal{P}'_L(p, q)$, with $p \geq 2$, $q \leq 0$ and $p - q < n - 2$: There is a left agent L and a right agent R. There exists some $k \geq 0$, with $p - k - 1 > q$, such that L is waiting at node $p-k-1$ where one pebble is located. Agent L is carrying the other pebble and is in state Halving-From-Right-To-Left. Moreover, its variable Last_Right, resp. Last_Left, has value p, resp. q. Its variable Goal_Left has value Update_Goal_Left$(p, q, 1)$. (Its variable Goal_Right has value Goal_Left$+ n - 1$.) Agent R, empty-handed, is in one of the following situations:

- it is dead in the black hole located at node $p+1$; - it is waiting at $p+1$ in state Explore-Right and its variable Last_Right has value p;
- it is already back from node $p+1$ at node p in state Pick-Up-Right and its variable Last_Right has value $p+1$;
- it is waiting at node $p+1$ in state Halving-Explore-Right and its variable Last_Right has value p;
- it is already back from node $p+1$ at node p in state Halving-Pick-Up-Right and its variable Last_Right has value $p+1$;
- it is waiting at node $p-i$, $1 \leq i \leq k$, in state Halving-Pong-Right, its variable Last_Right has value $p+1$ and its variable Counter has value $i-1$.

Moreover, in the second and third cases, the value Goal_Left of Agent L is equal to Update_Goal_Left $(p, q', k+1)$, where q' is the value Last_Left of Agent R. In the last three cases, the value Goal_Left of Agent L is equal to Update_Goal_Left$(p, q', k+1-$Counter$)$, where q' equals Goal_Left of Agent R.

Property. $\mathcal{P}'_R(p, q)$, with $p \geq 2$, $q \leq 0$ and $p - q < n - 2$: There is a left agent L and a right agent R. There exists some $k \geq 0$, with $q + k + 1 < p$, such that R is waiting at node $q + k + 1$ where one pebble is located. Agent R is carrying the other pebble and is in state Halving-From-Left-To-Right. Moreover, its variable `Last_Left`, resp. `Last_Right`, has value q, resp. p. Its variable `Goal_Right` has value `Update_Goal_Right`$(p, q, 1)$. (Its variable `Goal_Left` has value `Goal_Right`$-n+1$.) Agent L, empty-handed, is in one of the following situations:

- it is dead in the black hole located at node $q-1$;
- it is waiting at $q-1$ in state `Halving-Explore-Left` and its variable `Last_Left` has value q;
- it is already back from node $q-1$ at node q in state `Halving-Pick-Up-Left` and its variable `Last_Left` has value $q-1$;
- it is waiting at node $q+i$, for some $1 \leq i \leq k$ in state `Halving-Pong-Left`, its variable `Last_Left` has value $q-1$ and its variable `Counter` has value $i-1$.

Moreover, in the last three cases, the value `Goal_Right` of Agent R is equal to `Update_Goal_Right` $(p', q, k+1-$`Counter`$)$, where p' is `Goal_Right` of Agent L.

Lemma 1. *Consider a n-node ring containing a homebase and a black hole, and two agents running Algorithm `EnhancedPingPong` from the homebase. After finite time, one of the following situations occurs:*

- Stage 2 of Phase 1 begins and Property $\mathcal{P}(p)$ holds for some $p \in \{0, 1\}$;
- Phase 2 begins and Property $\mathcal{P}'_L(p, 0)$ holds for some integer p such that $2 \leq p \leq n-2$;
- all agents of the non-empty set of surviving agents have terminated and located the black hole.

Moreover, at that time, each edge has been traversed at most a constant number of times since the beginning of the algorithm.

Lemma 2. *Consider a n-node ring containing a homebase and a black hole, and two agents running Algorithm `EnhancedPingPong` from the homebase. Assume that at some time t a Phase-1 stage of even number i begins and that Property $\mathcal{P}(p)$ holds for some $p \in \{0, 1\}$. Then at some time $t' > t$ one of the following situations occurs:*

- Stage $i+2$ of Phase 1 begins and Property $\mathcal{P}(p')$ holds for some integer p' such that $p < p' \leq 1$ (thus $p' = 1$);
- Phase 2 begins and Property $\mathcal{P}'_L(p', q)$ holds for some integers p' and q such that $p' \geq 2$, $q \leq 0$ and $p' - q < n-2$;
- all agents of the non-empty set of surviving agents have terminated and located the black hole.

Moreover, each edge has been traversed at most a constant number of times between times t and t'.

Lemma 3. *Consider a n-node ring containing a homebase and a black hole, and two agents running Algorithm `EnhancedPingPong` from the homebase. Assume that at some time t a Phase-2 stage of odd number i begins and that either Property $\mathcal{P}'_L(p, q)$ or Property $\mathcal{P}'_R(p, q)$ holds for some integers p and q such that $p \geq 2$, $q \leq 0$ and $p - q < n-2$. Then at some time $t' > t$ one of the following situations occurs:*

- Stage $i+2$ of Phase 2 begins and either Property $\mathcal{P}'_L(p',q')$ or Property $\mathcal{P}'_R(p',q')$ holds for some integers p' and q' such that $p' \geq p$, $q' \leq q$ and $n-(p'-q'+1) \leq \lceil \frac{n(p-q+1)}{2} \rceil$;
- all agents of the non-empty set of surviving agents have terminated and located the black hole.

Moreover, each edge has been traversed at most a constant number of times between times t and t'.

Theorem 1. *Algorithm `EnhancedPingPong` is correct. More precisely, consider a n-node ring containing a homebase and a black hole, and two agents running Algorithm `EnhancedPingPong` from the home base. After finite time, there remains at least one surviving agent and all surviving agents have terminated and located the black hole.*

Proof. From Lemmas 1 and 2, we know that the first phase contains at most five stages, each one ending after finite time. Furthermore we know that after finite time, either the algorithm terminates correctly, or Property $\mathcal{P}'_L(p,q)$ or $\mathcal{P}'_R(p,q)$ holds, for some integers p and q such that $q \leq 0 < p$ and $0 < p-q < n-2$. From Lemma 3, we know that a stage of Phase 2 ends after finite time. We also know that if the algorithm does not terminate after two stages i, $i+1$ in Phase 2, then Property $\mathcal{P}'_L(p',q')$ or $\mathcal{P}'_R(p',q')$ holds, for some integers p' and q' such that the positive value $p'-q'$ is stricty less than $p-q$. Hence, after finite time, neither $\mathcal{P}_L(p,q)$ nor $\mathcal{P}'_R(p,q)$ can be satisfied and the algorithm terminates correctly.

Theorem 2. *The total number of moves performed by two agents running Algorithm `EnhancedPingPong` in a n-node ring is at most $O(n \log n)$.*

Proof. From Lemmas 1 and 2, there are at most five stages in Phase 1 and for each of them the number of edge traversals performed by each agent is at most $O(n)$. From Lemma 3, there are at most $O(\log n)$ stages in Phase 2 because the unexplored part is basically halved every two stages. From the same lemma, we have that for each Phase-2 stage the number of edge traversals performed by each agent is at most $O(n)$. Hence, overall, the total number of moves performed by two agents running Algorithm `EnhancedPingPong` in a n-node ring is at most $O(n \log n)$.

The optimality of the algorithm follows from the fact that, in a ring, the problem cannot be solved with less agents or (asymptotically) less moves [12], and clearly not with less pebbles.

4 Black Hole Search in Arbitrary Graphs

4.1 Preliminaries

In this section, both agents are provided with a map of the network containing all edge labels and a mark showing the position of the homebase in this network.

Thus, each node of the map can be uniquely identified (for example by a list of edge labels leading to it from the homebase). Therefore, each agent is able to know where it lies at any point of the execution of the algorithm. It also knows where each edge incident to its position leads.

The algorithm `GeneralizedEnhancedPingPong` we propose for arbitrary networks is an adaptation of the algorithm `EnhancedPingPong` that we described for rings. To be able to apply `EnhancedPingPong` in a general graph, each agent will maintain a partial mapping between the node numbers used in the algorithm and the actual nodes in the network (or its map), such that at any point in time an agent knows what means "go left" and "go right".

During the execution of the algorithm, each agent maintains two walks W_R and W_L, defined as two sequences $(r_0, r_1, \dots, r_P)$ and $(l_0, l_1, \dots, l_Q)$ of nodes of the network. The nodes r_0 and l_0 correspond to the homebase. Since W_R and W_L are walks, we have that $\{r_i, r_{i+1}\}$ and $\{l_j, l_{j+1}\}$ are edges of the graph, for all $0 \le i < P$ and $0 \le j < Q$.

From these two walks, we define recursively function σ as follows. First $\sigma(0) = 0$. Assume that σ is defined for all j such that $0 \le j \le i$, for some $i \ge 0$. Then if there exists an element r_K in W_R such that $r_K \notin \{r_{\sigma(0)}, r_{\sigma(1)}, \dots, r_{\sigma(i)}\}$ but for all $k < K$, $r_k \in \{r_{\sigma(0)}, r_{\sigma(1)}, \dots, r_{\sigma(i)}\}$, then $\sigma(i+1) = K$, otherwise $\sigma(i+1)$ is not defined. Similarly, assume that σ is defined for all j such that $i \le j \le 0$, for some $i \le 0$. Then if there exists an element l_K in W_L such that $l_K \notin \{l_{\sigma(0)}, l_{\sigma(1)}, \dots, l_{\sigma(i)}\}$ but for all $k < K$, $l_k \in \{l_{\sigma(0)}, l_{\sigma(1)}, \dots, l_{\sigma(i)}\}$, then $\sigma(i-1) = K$, otherwise $\sigma(i-1)$ is not defined.

Let us assume that an agent lies at some node i. If $i \ge 0$ (i.e., the agent is at the homebase or somewhere in the explored part to the right) going one step right from node i means following the sub-walk $(r_{\sigma(i)}, r_{\sigma(i)+1}, \dots, r_{\sigma(i+1)})$ of W_R. Going left from node $i+1$ to node i means following this sub-walk in reverse order. Similarly, if $i \le 0$ (i.e., the agent is at the homebase or in the explored part to the left) going one step left from node i means following the sub-walk $(l_{\sigma(i)}, l_{\sigma(i)+1}, \dots, l_{\sigma(i-1)})$ of W_L. Going right from node $i-1$ to node i means following this sub-walk in reverse order.

4.2 The Algorithm

We now describe the definitions of the walks W_R and W_L throughout the algorithm. First of all, node v_0 denotes the homebase. Node v_1 is the neighbor of node v_0 reachable by the smallest edge label while node v_{-1} is the neighbor of node v_0 reachable by the largest edge label.

At the beginning of the algorithm, let T_R be a tree spanning all nodes except for node v_{-1} and containing the edge $\{v_0, v_1\}$. Let W_R be a DFS traversal of T_R starting from node v_0 by the edge $\{v_0, v_1\}$. Let W_L be (v_0, v_{-1}). Clearly, nodes v_{-1}, v_0 and v_1 are the nodes -1, 0 and 1.

Assume that the stage changes from an odd number to an even number in Phase 1. Let p and q be the values `Last_Right` and `Last_Left` of the left agent. Then the new walk W_R consists of the first $\sigma(p+1)+1$ elements of the old W_R, that is the sequence $(r_0, r_1, \dots, r_{\sigma(p+1)})$. In addition, let T_L be a tree spanning all nodes except for node $p+1$. Let S_L be a DFS traversal of T_L starting from node $q-1$. Then the new walk W_L is the concatenation of the old W_L and of the sequence S_L. The left agent does these updates of the walks when changing role. The other agent does these updates when it finds out that its pebble has been stolen. More precisely, it updates its walks just before switching to state `Pong-Right`. Note that both agents agree on the new definition of the walks because they use the same values for p and for q (cf. Property $\mathcal{P}(p)$).

Similarly assume that the stage changes from an even number to an odd number in Phase 1. Let p and q be the values `Last_Right` -1 and `Last_Left` of the right agent. Then the new walk W_L consists of the first $\sigma(q-1)+1$ elements of the old W_L, that is the sequence $(l_0, l_1, \ldots, l_{\sigma(q-1)})$. In addition, let T_R be a tree spanning all nodes except node $q-1$. Let S_R be a DFS traversal of T_R starting at node $p+1$. Then the new walk W_R is the concatenation of the old W_R and of the sequence S_R. The rigth agent does these updates of the walks when it retrieves a pebble, just before switching from state `Pong-Right` to state `Ping-Right`. The other agent does these updates when it finds out that its pebble has been stolen. More precisely, it updates its walks just before changing role. Note that again both agents agree on the new definition of the walks because they use the same values for p and for q.

The walks are also updated at the beginning of each Phase-2 stage of odd number i. More precisely this is done by an agent each time and just after it updates its knowledge of the unexplored part and its goals. Assume w.l.o.g. that the stage is now i because a right agent became a left agent. Let p, q and g be the values, respectively, of `Last_Right`, `Last_Left` and `Goal_Right` just after the update of the goals. Let $\{V_{ex}, V_{uex}\}$ be a partition of the nodes of the graph such that V_{ex} is the set of nodes $\{q, q+1, \ldots, p-1, p\}$. From Lemma 5.2 in [11], V_{uex} can be partitioned into V_R and V_L such that $|V_R| = p-g$, the node $p+1$ is in V_R, and the graphs G_R and G_L induced by, respectively, $V_{ex} \cup V_R$ and $V_{ex} \cup V_L$ are connected. Let T_R and T_L be spanning trees of G_R and G_L. Let S_R be a DFS traversal of T_R starting at node $p+1$. Similarly let S_L be a DFS traversal of T_L starting at node q. Finally, let W'_R, resp. W'_L, consists of the first $\sigma(p+1)$, resp. $\sigma(q)$, elements of W_R, resp. W_L. Then the new walks W_R and W_L are respectively the concatenation of W'_R and S_R and the concatenation of W'_L and S_L. Note that both agents agree on the new definition of the walks because they use the same values for p, q and g (cf. Properties $\mathcal{P}'_R(p,q)$ and $\mathcal{P}'_L(p,q)$).

In some cases, it is possible to use (safe) shortcuts to decrease the number of moves. Indeed, always following the walks W_R and W_L to go right and left may lead to a total of $n \log^2 n$ moves. The algorithm is modified as follows. During Phase 2, each agent maintains an additional variable `Last_Seen_Pebble` that basically memorizes the last place where the agent has seen the other pebble. When an agent finishes a half and switches to state `Halving-Ping-Left`, resp. `Halving-Ping-Right`, it goes directly to node `Last_Seen_Pebble` and if there are no pebbles at this node, it then goes directly to node `Last_Left`, resp. `Last_Right`. This is done by traversing only nodes that are known to be safe.

4.3 Correctness and Complexity

Theorem 3. *Algorithm `GeneralizedEnhancedPingPong` is correct. More precisely, consider a n-node graph containing a homebase and a black hole, and two agents running Algorithm `GeneralizedEnhancedPingPong` from the homebase. After finite time, at least one agent survives and all surviving agents have terminated and located the black hole.*

Proof. As noticed in the description of the algorithm in the previous subsection, the two agents agree on the definition of the walks and thus of the node numbers. Moreover, one can easily check that the function σ defining the node numbers always gives the same number to the same node as soon as this node has been explored by at least one agent. Indeed, if a node $i \geq 0$ is explored, then the initial part $(r_0, r_1, \ldots, r_{\sigma(i)})$ of W_R is kept unchanged forever. A similar property holds for $i \leq 0$. Finally note that a node of the graph has at most one pre-image by σ.

To summarize, Algorithm `GeneralizedEnhancedPingPong` behaves exactly the same as Algorithm `EnhancedPingPong`. The only difference is that traversing an edge in the ring may correspond to the traversals of (finitely) many edges in an arbitrary graph. Nevertheless, since Algorithm `EnhancedPingPong` is correct, Algorithm `GeneralizedEnhancedPingPong` is correct as well.

Theorem 4. *The total number of moves performed by two agents running Algorithm `GeneralizedEnhancedPingPong` in a n-node graph is at most $O(n \log n)$.*

Proof. In this proof we call the number of edge traversals performed by an agent going from node i to node $i+1$ $(-n < i < n-1)$ the length of the virtual edge $\{i, i+1\}$. We now bound the total number of moves performed by each agent in each phase.

As in the case of the ring, the first phase consists of at most five stages. Moreover, each update of the walks increases their length by at most $2n$ because the path appended to a walk is a DFS traversal of a tree. Hence, the sum of all the lengths of the virtual edges traversed in the first phase is at most $10n$. From Lemmas 1 and 2, each edge of the network has been traversed at most a constant number of times during Phase 1. Hence, the total number of moves performed by two agents running Algorithm `GeneralizedEnhancedPingPong` is at most $O(n \log n)$ in the first phase.

From the lemmas 1, 2 and 3, either Property $\mathcal{P}'_R(p,q)$ or Property $\mathcal{P}'_L(p,q)$) holds, for some integers p and q, at the beginning of a Phase-2 stage of odd number. Let p_i and q_i be the two integers corresponding to the stage $2i+1$ of the second phase, for $0 \leq i \leq s$. Note that s is at most $O(\log n)$. Let p_{s+1}, resp. q_{s+1}, be the right, resp. left, neighbor of the black hole. By definition of the properties and from the lemmas, we have that $q_{s+1} \leq \cdots \leq q_0 \leq 0 \leq p_0 \leq \cdots \leq p_{s+1}$. Since the walks W_R and W_L are updated when and only when the goals are updated, and since a walk is always extended by a DFS traversal of a tree, we obtain that the sum of all the lengths of the virtual edges from node p_i to p_{i+1}, and from node q_i to q_{i+1}, is at most $O(n)$, for $0 \leq i \leq s$. Moreover, from the previous paragraph, the sum of all the lengths of the virtual edges from node q_0 to p_0 is at most $O(n)$.

Consider a stage $2i+1$ of the second phase, for $0 \leq i \leq s$. Let A be the agent that started the stage by changing role and let B the other agent. Without loss of generality, assume that A is a left agent. The total number of moves performed by A in this stage is at most $O(n)$ because A first goes directly (by a shortest safe path) to the beginning q_i of its workload, thus in at most n moves, and then stays in the final part of W_L that corresponds to the DFS traversal of a tree to explore its assigned workload, which incurs at most $O(n)$ moves (from Lemma 3). If it succeeds to explore its half, then it goes directly to the node where it left the other pebble (thanks to the variable `Last_Seen_Pebble`). If the pebble is not there anymore, it further goes directly to node p_i and starts to explore the right half. Hence, in any case, for the same reasons as before, A performs at most $O(n)$ moves in stage $2i+2$. Concerning B, if it retrieves its pebble in stage $2i+1$ or $2i+2$, it will perform at most $O(n)$ moves in these two stages, without counting the moves dones in state `Halving-Pong-Right`. Indeed, again, exploring a half or going directly to the beginning of it costs at most a linear number of moves.

It remains to bound the number of moves done while in one of the states `Halving-Pong-Right` or `Halving-Pong-Left`. This is done globally over the whole second phase. Each edge traversed in one of these two states may cost up to $O(n)$ moves. However, there are at most $O(\log n)$ such traversals because any of them corresponds to an update of the workloads, which happens only a logarithmic number of times in the entire algorithm.

One can now conclude that the total number of moves performed by two agents running Algorithm `GeneralizedEnhancedPingPong` in a n-node graph is at most $O(n \log n)$.

The optimality of the algorithm follows from the fact that, in an arbitrary graph, BHS cannot be solved with less agents or (asymptotically) less moves [11], and clearly not with less pebbles.

Acknowledgment. This work was done during the stay of David Ilcinkas at the University of Ottawa, as a postdoctoral fellow. Paola Flocchini was partially supported by the University Research Chair of the University of Ottawa. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Discovery grants.

References

1. Albers, S., Henzinger, M.: Exploring unknown environments. In: 29th ACM Symposium on Theory of Computing (STOC), pp. 416–425 (1997)
2. Bender, M.A., Fernández, A., Ron, D., Sahai, A., Vadhan, S.P.: The power of a pebble: Exploring and mapping directed graphs. Information and Computation 176(1), 1–21 (2002)
3. Cao, J., Das, S. (eds.): Mobile Agents in Networking and Distributed Computing. John Wiley, Chichester (2008)

4. Chalopin, J., Das, S., Santoro, N.: Rendezvous of mobile agents in unknown graphs with faulty links. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 108–122. Springer, Heidelberg (2007)
5. Cooper, C., Klasing, R., Radzik, T.: Searching for black-hole faults in a network using multiple agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 320–332. Springer, Heidelberg (2006)
6. Czyzowicz, J., Kowalski, D., Markou, E., Pelc, A.: Complexity of searching for a black hole. Fundamenta Informaticae 71(2-3), 229–242 (2006)
7. Czyzowicz, J., Kowalski, D., Markou, E., Pelc, A.: Searching for a black hole in synchronous tree networks. Combinatorics, Probability & Computing 16, 595–619 (2007)
8. Das, S., Flocchini, P., Kutten, S., Nayak, A., Santoro, N.: Map construction of unknown graphs by multiple agents. Theoretical Computer Science 385(1-3), 34–48 (2007)
9. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. J. Graph Theory 32(3), 265–297 (1999)
10. Dobrev, S., Flocchini, P., Kralovic, R., Santoro, N.: Exploring a dangerous unknown graph using tokens. In: 5th IFIP Int. Conf. on Theoretical Computer Science (TCS), pp. 131–150 (2006)
11. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Searching for a black hole in arbitrary networks: optimal mobile agents protocol. Distributed Computing 19(1), 1–19 (2006)
12. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Mobile search for a black hole in an anonymous ring. Algorithmica 48, 67–90 (2007)
13. Dobrev, S., Kralovic, R., Santoro, N., Shi, W.: Black hole search in asynchronous rings using tokens. In: Calamoneri, T., Finocchi, I., Italiano, G.F. (eds.) CIAC 2006. LNCS, vol. 3998, pp. 139–150. Springer, Heidelberg (2006)
14. Flocchini, P., Ilcinkas, D., Santoro, N.: Optimal black hole search with pure tokens, http://www.scs.carleton.ca/-santoro/Reports/PingPong.pdf
15. Flocchini, P., Santoro, N.: Distributed Security Algorithms For Mobile Agents. In: 3, ch. 5 (2008)
16. Fraigniaud, P., Gasieniec, L., Kowalski, D., Pelc, A.: Collective tree exploration. Networks 48(3), 166–177 (2006)
17. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph exploration by a finite automaton. Theoretical Computer Science 345(2-3), 331–344 (2005)
18. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Approximation bounds for black hole search problems. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 261–274. Springer, Heidelberg (2005)
19. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Hardness and approximation results for black hole search in arbitrary networks. Theoretical Computer Science 384(2-3), 201–221 (2007)

Deterministic Rendezvous in Trees with Little Memory

Pierre Fraigniaud[1,⋆] and Andrzej Pelc[2,⋆⋆]

[1] CNRS and University Paris Diderot
[2] Département d'informatique, Université du Québec en Outaouais

Abstract. We study the size of memory of mobile agents that permits to solve deterministically the rendezvous problem, i.e., the task of meeting at some node, for two identical agents moving from node to node along the edges of an unknown anonymous connected graph. The rendezvous problem is unsolvable in the class of arbitrary connected graphs, as witnessed by the example of the cycle. Hence we restrict attention to rendezvous in trees, where rendezvous is feasible if and only if the initial positions of the agents are not symmetric. We prove that the minimum memory size guaranteeing rendezvous in all trees of size at most n is $\Theta(\log n)$ bits. The upper bound is provided by an algorithm for abstract state machines accomplishing rendezvous in all trees, and using $O(\log n)$ bits of memory in trees of size at most n. The lower bound is a consequence of the need to distinguish between up to $n-1$ links incident to a node. Thus, in the second part of the paper, we focus on the potential existence of pairs of *finite* agents (i.e., finite automata) capable of accomplishing rendezvous in all *bounded degree* trees. We show that, as opposed to what has been proved for the graph exploration problem, there are no finite agents capable of accomplishing rendezvous in all bounded degree trees.

1 Introduction

Two identical mobile agents initially located in two nodes of a network, modeled as an undirected connected graph, have to meet at some node of the graph. This task is known in the literature as the *rendezvous* problem in graphs [4]. In this paper we study the *size of the memory* of the agents permitting to solve the rendezvous problem deterministically. Throughout the paper we consider only deterministic rendezvous. If nodes of the graph are labeled then agents can use the protocol to meet at a predetermined node and the rendezvous problem reduces to graph exploration. However, in many applications, when rendezvous is needed in an unknown environment, such unique labeling of nodes may not be available, nodes may refrain from revealing their labels to agents due to security

[⋆] This work was done during this author's visit at the Research Chair in Distributed Computing of the Université du Québec en Outaouais. Additional supports from Action COST 295 "DYNAMO".

[⋆⋆] Supported in part by NSERC discovery grant and by the Research Chair in Distributed Computing of the Université du Québec en Outaouais.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 242–256, 2008.

reasons, or limited sensory capabilities of the agents may prevent them from perceiving such labels. Hence it is important to design rendezvous algorithms for agents operating in *anonymous* graphs, i.e., graphs without unique labeling of nodes. Clearly, the agents have to be able to locally distinguish ports at a node since, otherwise, an agent may even be unable to visit all neighbors of a node of degree 3 (after visiting the second neighbor, the agent cannot distinguish the port leading to the first visited neighbor from the port leading to the unvisited one). Consequently, agents initially located at two nodes of degree 3 might never be able to meet. Hence we make a natural assumption that all ports at a node are locally labeled $1, ..., d$, where d is the degree of the node. No coherence between those local labelings is assumed, and we do not assume any a priori knowledge of the topology of the graph, or of its size.

The above described rendezvous problem is unsolvable in the class of arbitrary connected graphs, as witnessed by the example of the (oriented) cycle in which a pair of identical agents can never meet, regardless of their initial positions. Hence we restrict attention to rendezvous in trees. For this class of networks, rendezvous is feasible if and only if the initial positions of the agents are not symmetric (we mean here symmetry with respect to local orientation of ports at each node, see the precise definition in the next subsection).

1.1 The Model and Terminology

We consider trees whose nodes are unlabeled, and edges incident to a node v have distinct labels $1, \ldots, d$, where d is the degree of v. Thus every undirected edge $\{u, v\}$ has two labels, which are called its *port numbers* at u and at v. Port numbering is *local*, i.e., there is no relation between port numbers at u and at v. A pair of nodes u, v of a tree is called *symmetric* if there exists an automorphism of the tree preserving port numbering, that carries one node on the other. More precisely, u and v are symmetric if there exists a bijection $f : V \to V$, where V is the set of nodes of the tree, such that:

1. for any $w, w' \in V$, w is adjacent to w' if and only if $f(w)$ is adjacent to $f(w')$;
2. for any $w, w' \in V$, the port number corresponding to edge $\{w, w'\}$ at node w is equal to the port number corresponding to edge $\{f(w), f(w')\}$ at node $f(w)$;
3. $f(u) = v$.

We consider mobile agents traveling in trees with locally labeled ports. The tree and its size are a priori unknown to the agents. We first define precisely an individual agent. An agent is an abstract state machine $\mathcal{A} = (S, \pi, \lambda, s_0)$, where S is a set of states among which there is a specified state s_0 called the *initial* state, $\pi : S \times \mathbb{N}^2 \to S$, and $\lambda : S \to \mathbb{N}$. Initially the agent is at some node u_0 in the initial state $s_0 \in S$. The agent performs actions in rounds measured by its internal clock. Each action can be either a move to an adjacent node or a null move resulting in remaining in the currently occupied node. State s_0 determines a natural number $\lambda(s_0)$. If $\lambda(s_0) = 0$ then the agent makes a null move (i.e., remains at u_0). If $\lambda(s_0) > 0$ then the agent leaves u_0 by port $\lambda(s_0)$. When incoming to a node v in state $s \in S$, the behavior of the agent is as follows. It reads the number i of the port through which it entered v and the degree d of v. The pair $(i, d) \in \mathbb{N}^2$ is an input symbol that causes the transition from state s to state $s' = \pi(s, (i, d))$. If the previous move of the agent was null, (i.e., the agent stayed at node v in state s) then the pair $(0, d) \in \mathbb{N}^2$ is the input symbol read by the agent, that causes the transition from state s to state $s' = \pi(s, (0, d))$. In both cases s' determines an integer $\lambda(s')$, which is either 0, in which case the agent makes a null move, or a positive integer indicating a port number by which the agent leaves v. The agent continues moving in this way, possibly infinitely. A state $s \in S$ is called *final* if $\lambda(s) = 0$ and $\pi(s, (i, j)) = s$, for all integers i, j. Thus, after entering a final state, the agent never changes the state and never moves.

Since we consider the rendezvous problem for identical agents, we assume that agents are copies A and A' of the same abstract state machine $\mathcal{A}$, starting at two distinct nodes v_A and $v_{A'}$, called the *initial positions.* We will refer to such identical machines as a *pair of agents.* It is assumed that the internal clocks of a pair of agents tick at the same rate. The clock of each agent starts when the agent starts executing its actions. Agents start from their initial position with *delay* $\theta \geq 0$, controled by an adversary. This means that the later agent starts executing its actions θ rounds after the first agent. Agents do not know which of them is first and what is the value of θ.

Initial positions forming a symmetric pair of nodes are crucial for our considerations. Indeed, it is easy to see that if the initial position is not a symmetric pair then there exists a pair of agents that can meet in a given tree, for any delay θ, and if the initial position is symmetric then meeting is impossible for any pair of agents, for $\theta = 0$. The argument is as follows. A tree has either a central node or a central edge. If the tree has a central node, then no initial position is symmetric (recall that symmetry is considered with respect to port labelings), and a pair of agents *equipped with suitable memory* can meet in this central node, regardless of the initial position. If the tree has a central edge, then for non-symmetric initial positions one of the endpoints of the central edge can be distinguished from the other, and agents will meet there. Finally, for a symmetric initial position in a tree with a central edge, simultaneously starting agents will never meet, as all their actions will be symmetric with respect to the central edge. Note that in the latter case agents could still meet “accidentally” , e.g., if the delay θ is large enough and the program of the agents causes exploration of the entire tree; in this case the first agent will join the later one before the later starts moving. Since we are interested in rendezvous protocols that cause meeting of the agents for any delay θ, successful meeting can be guaranteed only for non-symmetric initial positions.

We assume that when the agents meet, i.e., when there is a round in which they are in the same node of the tree, they perceive this fact and stop. Hence we adopt the following definitions of two variations of the rendezvous task.

- **Rendezvous without termination:** If the initial positions are not symmetric then, in some round, both agents are in the same node of the tree and enter a final state s_{rv} (rendezvous).
- **Rendezvous with termination:** If the initial positions are not symmetric then, in some round, both agents are in the same node of the tree and enter a final state s_{rv}, else, i.e., if the initial positions are symmetric,
    - either in no round agents are in the same node of the tree and both agents eventually enter the final state s_{no_rv} (no rendezvous)
    - or in some round agents are in the same node of the tree and enter a final state s_{rv}.

Note that our positive result (the algorithm) will concern the stronger task of rendezvous with termination, and our impossibility result will hold even for the weaker task of rendezvous without termination. The difference between the two variations of rendezvous is the following. For rendezvous without termination it is not specified what happens if agents never meet: in this case they may wander in the tree indefinitely. For rendezvous with termination agents either eventually meet (one of the agents may join later the other that first stopped), or each of them stops in the final state s_{no_rv} (in different nodes, possibly in different rounds), i.e., they detect that rendezvous is impossible. Moreover, false alarms are excluded: an agent cannot stop in state s_{no_rv} and still be accidentally joined later by the other agent. As mentioned previously, meeting of agents starting from symmetric initial positions is possible only for positive delay θ.

1.2 Our Results

We study memory requirements of the agents for the feasibility of the rendezvous task in trees of size at most n, measured by the number of states used by the agents in such an environment, or, alternatively, by the number of memory bits required to encode these states.

The obvious lower bound on this memory size for rendezvous is $\Omega(\log n)$ bits. Indeed, if two identical agents A and A' use less than $\log n$ bits in trees of size at most $2n+1$, then they use less than n states in such trees. As a consequence, the function λ has less than n different outputs in trees of size at most $2n+1$. Consider the tree T with $2n+1$ nodes, consisting of two nodes u and v of degree n, both linked to a common node w, and to $n-1$ leaves. Since λ has less than n different outputs, there exists $p \in \{1,\dots,n\}$ that not belong to the outputs of λ. Label the port numbers of $\{u,w\}$ at u, and of $\{w,v\}$ at v, by p. Place A and A' at u and v, respectively. It follows that A will never traverse $\{u,w\}$, and v will never traverse $\{v,w\}$, and therefore A and A' will never meet, although u and v are not symmetric positions (in view of different port numbers at w). Our main positive result says that this lower bound can be achieved. More precisely, we show that there exists a pair of identical agents accomplishing rendezvous with termination in all trees, and using, for any integer n, $O(\log n)$ bits of memory in trees of size at most n. Our proof is constructive, i.e., we describe the protocol followed by the agents to achieve rendezvous with termination.

Table 1. Space complexities of Exploration and Rendezvous in trees using abstract state machines

		Trees	Bounded degree trees
Exploration	without termination	$\Omega(\log n)$	$O(1)$
	with termination	$O(\log n)$ [22]	$\Omega(\log\log\log n)$ [18]
Rendezvous	without termination	$\Omega(\log n)$	$\Omega(\log\log n)$ **[This paper]**
	with termination	$O(\log n)$ **[This paper]**	

We then consider the problem of the existence of a pair of *finite* agents (i.e., finite state machines using $O(1)$ states) capable of accomplishing rendezvous in trees. Due to the above lower bound $\Omega(\log n)$ on memory size, finite agents achieving rendezvous in the class of all trees cannot exist. However, the lower bound is due to the potential presence of nodes of large degree, and a pair of finite agents could potentially accomplish rendezvous in all trees of bounded degree. Hence we reformulate the problem by asking about the existence of finite agents for bounded degree trees, i.e., a pair of finite agents capable of accomplishing rendezvous in all such trees. We show that the answer to this question is negative, even for the less demanding task of rendezvous without termination. In fact, we show that, for any pair of identical finite agents, there is a line on which these agents cannot accomplish rendezvous, even without termination. As a function of the size of the trees, our impossibility result indicates a lower bound $\Omega(\log\log n)$ bits on the memory size for rendezvous in bounded degree trees of at most n nodes.

The latter result should be contrasted with the related task of tree *exploration*, consisting in traversing all edges of the tree by a single agent. See Table 1. Exploration without termination can be accomplished by a single agent that leaves a degree-d node by port $(i \bmod d)+1$ if it entered it by port i. Hence, while there exists a finite agent exploring all trees of bounded degree, there are no finite agents that can accomplish rendezvous in all trees of this class (even without termination). Thus, as a consequence, rendezvous without termination is a more memory demanding task than exploration without termination. On the other hand, it was shown in [22] that exploration with termination (even with return to the starting node) can be done using logarithmic memory (which is optimal). Our positive result shows that memory of logarithmic size is also sufficient for rendezvous with termination, hence (using the measure of required memory) rendezvous with termination is not harder than exploration with termination.

1.3 Related Work

The rendezvous problem has been introduced in [28]. The large body of literature on rendezvous (see the book [4] for a complete discussion and more references) can be divided into two classes: studies considering the geometric scenario (rendezvous in an interval of the real line, see, e.g., [11,12,21], or in the plane, see, e.g., [9,10]), and those discussing rendezvous in graphs, e.g., [2,5]. Most of the papers, e.g., [2,3,7,11,23] consider the probabilistic scenario: inputs and/or rendezvous strategies are random. In [23] randomized rendezvous strategies are applied to study self-stabilized token management schemes. Randomized rendezvous strategies use random walks in graphs, which have been widely studied and applied also, e.g., in graph traversing [1], on-line algorithms [14] and estimating volumes of convex bodies [19]. Tradeoffs between the expected time of rendezvous and the memory requirements of the agents can be found in [25] A natural extension of the rendezvous problem is that of gathering [20,23,27,29], when more than 2 agents have to meet in one location.

Deterministic rendezvous with anonymous agents working in unlabeled graphs but equipped with tokens used to mark nodes was considered, e.g., in [26]. In [30] the authors considered rendezvous of many agents with unique labels. Deterministic rendezvous of two agents with unique labels was discussed in [16,17,24]. To the best of our knowledge, deterministic rendezvous of identical agents that have no possibility of marking nodes has never been studied before.

The impact of memory size on the feasibility of the related task of tree exploration, for trees with unlabeled nodes, has been studied in [18,22]. In [18] the authors showed that no agent can explore with termination all trees of bounded degree and that memory of size $O(\log^2 n)$ is enough to explore all trees of size n and return to the starting node. In [22] it was shown that the latter task can be accomplished by an agent with memory of size $O(\log n)$.

2 Rendezvous with Logarithmic Memory

This section is devoted to our main positive result, that is the design of a LOGSPACE algorithm enabling two agents to achieve rendezvous in all trees. As we mentioned in Section 1.2, this result is optimal, as no pair of agents can accomplish rendezvous (even without termination) in all trees, using less than $\log n - 1$ bits of memory in trees of size at most n.

Theorem 1. *There is a pair of identical agents accomplishing rendezvous with termination in all trees, and using, for any integer n, $O(\log n)$ bits of memory in trees of size at most n.*

To establish the result, we describe an agent $\mathcal{A}$ such that two copies A and A' of agent $\mathcal{A}$ will rendezvous in any tree $T = (V, E)$ for any pair $(v_A, v_{A'}) \in V \times V$ of non symmetric starting positions. More specifically, T has a port-labeling associated to it, and if v_A and $v_{A'}$ are not in the same port-label-preserving automorphism class in T, then A and A' will eventually meet, both terminating in the state s_{rv}. Otherwise, both A and A' will eventually terminate in the same state s_{no_rv} if they indeed do not meet, or s_{rv} if they happen to meet. The result holds for any delay θ imposed by the adversary between the starting times of A and A'. We show that, if T has at most n nodes, then $\mathcal{A}$ requires only a memory of size $O(\log n)$ bits.

We start by describing agent $\mathcal{A}$. Figure 1 describes the skeleton of the protocol performed by $\mathcal{A}$. A basic exception rule is that, whenever meeting occurs, the agent terminates in state s_{rv}. Now, the protocol executed by $\mathcal{A}$ is decomposed into two phases called the bounding phase and the searching phase.

```
Begin
    /* Exception */
    Whenever meeting the other agent do terminate in state $s_{rv}$.
    /* Bounding phase */
    Compute an upper bound $m$ on the size of $T$;
    Compute the maximum degree $\Delta$ of $T$;
    /* Searching phase */
    Stay idle for $2m$ steps;
    For $i = 1$ to $m$ do
        Traverse $T$ partially to compute label $\lambda_i$;
        Execute $2\lambda_i$ traversals of $T$ of length $m$;
        Stay idle for $2m$ steps;
    /* Termination */
    If meeting occurs then terminate in state $s_{rv}$
    else terminate in state $s_{no\_rv}$;
End
```

Fig. 1. Protocol of agent $\mathcal{A}$

- The role of the bounding phase is to find an upper bound m on the number of nodes in T. A key property of this phase is that this upper bound does not depend on the starting position of $\mathcal{A}$ but only on T. The maximum degree Δ of T is also computed during the bounding phase.
- The role of the searching phase is to achieve rendezvous, whenever possible. It begins with $\mathcal{A}$ staying idle for $2m$ steps. Then the searching is decomposed in m sub-phases. In each sub-phase, agent $\mathcal{A}$ first performs some partial exploration of T and computes a label λ. It then performs 2λ traversals of T of length m, and finally remains idle for $2m$ steps before entering the next sub-phase. A key property of the searching phase is that the labels λ_i, $i = 1, \ldots, m$, that are computed during the sub-phases depend on the starting position of $\mathcal{A}$ in a way insuring the following: if the copies of $\mathcal{A}$ start from two non symmetric positions v_A and $v_{A'}$, then there is a subphase i such that the label λ_i computed for position v_A is different from the label λ'_i computed for position $v_{A'}$. We will show that the two copies of $\mathcal{A}$ will meet during sub-phase i.

We describe in detail the two phases in the following two subsections. The proof of Theorem 1 will follow in Section 2.3.

2.1 Bounding Phase

As said in Section 1.3, it was shown in [22] that there is an agent that can accomplish exploration of any tree, starting from any node of the tree, so that, in trees of size at most n, this agent uses a memory of size $O(\log n)$ bits. Moreover, when the exploration is achieved, the agent returns to its starting position. We use this result as a black box for the description of agent $\mathcal{A}$. More precisely, we denote by E&R(v) (for explore-and-return) the protocol from [22] consisting in starting from node v, exploring T, and returning to v. This task is accomplished in polynomial time (i.e., polynomial number of steps). However, the exact execution time of E&R(v) depends on v. The objective of the bounding phase is therefore to perform E&R(v) from *every* node v of T, in order to set m as the maximum execution time of E&R(v) among all nodes $v \in V$.

We describe how to perform E&R from every node of T using $O(\log n)$ bits of memory in trees of size at most n. Let α and β be two positive constants such that the protocol E&R performs in at most $n^\alpha + \beta$ steps in trees of at most n nodes. We recall the standard notion of *basic walk*. A basic walk of length $2k$, for $k \geq 0$, executed by agent $\mathcal{A}$ starting at node v_A consists in, first, leaving node v_A by port 1, and, then, whenever entering a degree-d node by the port numbered p, leaving that node by the port numbered $(p \bmod d) + 1$. Then, after k edges have been traversed, $\mathcal{A}$ starts moving backward by leaving the current node via the same port as the one used to enter the node, and executing the reverse strategy: whenever entering a degree-d node by the port numbered p, $\mathcal{A}$ leaves that node by the port numbered $p - 1$ if $p > 1$, and by port number d if $p = 1$. A basic walk of length $2k$ can be executed by an agent with $O(\log k)$ bits of memory, this memory being mostly used to encode a counter for counting up to k.

During the bounding phase, agent $\mathcal{A}$, starting at v_A, first calls E&R(v_A). This results in exploring the whole tree T and returning to v_A. This allows $\mathcal{A}$ to compute the maximum degree Δ of T. Let n_0 be the number of steps performed by E&R(v_A). Since all nodes of T were visited, we have $n_0 \geq |V|$. $\mathcal{A}$ uses the bound n_0 to perform a basic walk of length $2n_0$ interleaved with calls to E&R. More precisely, after each of the n_0 first steps of the basic walk, $\mathcal{A}$ executes E&R(v) where v is the current node. By doing so, $\mathcal{A}$ eventually computes $m = \max_{v \in V} n_v$ where n_v is the number of steps performed by E&R(v). Once n_0 steps of the basic walk have been performed, $\mathcal{A}$ completes the walk by performing n_0 steps backwards, to return to v_A. This completes the description of the bounding phase. The following lemma summarizes the main properties satisfied by the bounding phase (due to lack of space, the proofs of all lemmas are omitted).

Lemma 1. *The bounding phase can be executed by an agent with $O(\log n)$ bits of memory in trees of size at most n. At the end of the bounding phase, agent $\mathcal{A}$ has computed the maximum degree Δ of T, and an upper bound $m \leq |V|^\alpha + \beta$ on the number of nodes in T. This latter bound is independent of the starting position v_A of $\mathcal{A}$.*

Although the upper bound $m = \max_{v \in V} n_v$ does not depend on v_A, note that the number of steps used by $\mathcal{A}$ to compute this bound is $2n_0 + \sum_{i=1,\ldots,n_0} n_i$ where, for $1 \leq i \leq n_0$, n_i denotes the number of steps performed by E&R when called at the ith node of the basic walk of length $2n_0$ performed by $\mathcal{A}$. This sum depends on v_A for at least two reasons. First, n_0 depends on v_A, and, second, the ordered sequence of nodes visited by the basic walk depends on the starting node of the walk. Nevertheless, when $\mathcal{A}$ completes the bounding phase, it is back at v_A, and it knows a universal upper bound m on the number of nodes in T (universal in the sense that it does not depend on v_A).

2.2 Searching Phase

The searching phase is decomposed into a succession of m sub-phases, each of them beginning with the computation of a label (see Figure 1). The label λ_i is obtained by $\mathcal{A}$ in the ith sub-phase, $1 \leq i \leq m$, by performing a basic walk of length $2i$, visiting nodes $u_0, u_1, \ldots, u_{i-1}, u_i, u_{i-1}, \ldots, u_1, u_0$ where $u_0 = v_A$. The label λ_i is set according to the port number p_i by which the agent enters the node u_i reached after i steps of the walk (i.e., the node where $\mathcal{A}$ reverses the walk and starts going backwards to v_A), and according to the degree of the previous node u_{i-1} visited before moving to u_i. More precisely:

$$\lambda_i = (\deg(u_{i-1}) - 1)\Delta + (p_i - 1)$$

where Δ is the maximum degree of T. We have $\lambda_i \in [0, \Delta^2 - 1] \subseteq [0, n^2 - 1]$. Once λ_i has been obtained, $\mathcal{A}$ performs λ_i basic walks of length $2m$, for a total number of $2\lambda_i m$ steps. After that, $\mathcal{A}$ remains idle at v_A for $2m$ steps.

Lemma 2. *The searching phase can be executed by an agent with $O(\log n)$ bits of memory in a tree of size at most n.*

2.3 Proof of Theorem 1

Let us consider two copies A and A' of agent $\mathcal{A}$. By Lemmas 1 and 2, they have the specified memory size. We prove that A and A' accomplish rendezvous with termination.

First, we make an observation about the delay between the times when A and A' enter their searching phase. Assume, w.l.o.g., that A enters the searching phase at time t while A' enters the searching phase at time $t' = t + \delta$, where $\delta \geq 0$. This delay δ can be caused by various reasons. One is a possible difference θ between the starting times of the two agents (the original delay imposed by the adversary). Another is the completion time of the bounding phase (cf. the remark in the last paragraph of Section 2.1). Nevertheless, we prove that δ cannot be too large unless the two agents meet.

Lemma 3. *If $\delta > m$ then A and A' meet before A has even started the first sub-phase of its searching phase.*

By Lemma 3, we can assume that $0 \leq \delta \leq m$, A entering the searching phase at time t while A' enters the searching phase at time $t' = t + \delta$. During their searching phase, A and A' will eventually compute two sequences of labels: $\lambda_1, \ldots, \lambda_m$ and $\lambda'_1, \ldots, \lambda'_m$, respectively, unless they meet earlier. We make a crucial observation about these two sequences:

Lemma 4. *The two sequences $\lambda_1, \ldots, \lambda_m$ and $\lambda'_1, \ldots, \lambda'_m$ are identical if and only if the two initial positions v_A and $v_{A'}$ are symmetric.*

We are now ready to prove the theorem. Let $\lambda_1, \ldots, \lambda_m$ and $\lambda'_1, \ldots, \lambda'_m$ be the two sequences of labels computed by A and A', respectively. If they are identical, then, by Lemma 4, v_A and $v_{A'}$ are symmetric, and if the two agents have not met at the end of their searching phase, then they both terminate at their initial position in state s_{no_rv}. Assume now that the two sequences are not identical. Let i be the smallest index for which they differ. We claim that A and A' meet during their ith sub-phase (if they have not met before). To establish this claim, note that, since $\lambda_j = \lambda'_j$ for all $j < i$, the delay between the times A and A' enter their ith sub-phase is identical to the delay δ between the times A and A' enter their searching phase. We have assumed, w.l.o.g., that A is ahead of A', i.e., A enters its ith sub-phase at time t before A' enters its ith sub-phase at time $t' = t + \delta$, with $0 \le \delta \le m$.

During its ith sub-phase, agent A is performing its $2\lambda_i$ traversals during the time interval

$$I_{active} = [t + 2i,\ t + 2i + 2\lambda_i m],$$

and is idle during the time interval

$$I_{idle} = [t + 2i + 2\lambda_i m,\ t + 2i + 2(\lambda_i + 1)m].$$

Similarly, during its ith sub-phase, agent A' is performing its $2\lambda'_i$ traversals during the time interval

$$I'_{active} = [t + \delta + 2i,\ t + 2i + 2\lambda'_i m],$$

and is idle during the time interval

$$I'_{idle} = [t + 2i + \delta + 2\lambda'_i m,\ t + 2i + \delta + 2(\lambda'_i + 1)m].$$

We consider two cases:

- If $\lambda'_i \ge \lambda_i + 1$ then $|I_{idle} \cap I'_{active}| \ge 2m(\lambda'_i - \lambda_i) + \delta \ge 2m$ since $\delta \ge 0$. Thus A remains idle during a time-interval during which A' has time to perform a full traversal of T. Hence A' will meet A.
- If $\lambda'_i \le \lambda_i - 1$ then $|I'_{idle} \cap I_{active}| \ge 2m(\lambda_i - \lambda'_i) - \delta \ge m$ since $\delta \le m$. Thus A' remains idle during a time-interval during which A has time to perform a full traversal of T. Hence A will meet A'.

Hence in both cases, A and A' meet, which completes the proof of Theorem 1. □

3 The Limited Power of Finite Agents

The lower bound $\Omega(\log n)$ bits for agents accomplishing rendezvous in all trees of size at most n is caused by the need for the agents to distinguish potentially up to $n-1$ port numbers of a node of degree $n-1$. We thus raise the question of whether there exist *finite* agents (i.e., agents with memory of size $O(1)$) that could accomplish rendezvous in all bounded degree trees. The main result of this section is a negative answer to this question: for any pair of identical finite agents, there is a bounded degree tree on which these agents cannot accomplish rendezvous. In fact, the result holds even for the class of lines, and even if one does not require termination.

Theorem 2. *For any pair of identical finite agents, there is a line on which these agents cannot accomplish rendezvous, even without termination.*

The rest of the section is dedicated to the proof of Theorem 2. For proving the theorem, note that we can restrict ourselves to lines whose edges are properly colored 1 and 2, so that the port numbers at the two extremities of an edge colored i are set to i. In this setting, the transition function of an agent in a line is $\pi : S \times \{1,2\} \rightarrow S$ that describes the transition that occurs when an agent enters a node of degree $d \in \{1,2\}$ in state $s \in S$. In this situation, the agent changes its state to state $s' = \pi(s,d)$, and performs the action $\lambda(s')$. The fact that one does not need to specify the incoming port number is a consequence of the edge-coloring, which implies that whenever an agent leaves a node by port i, it enters the next node by port i too.

Let us fix two identical agents A and A', with *finite* state set S, and transition function π. Let $\pi' : S \rightarrow S$ be the transition function applied at nodes of degree 2 of the edge-colored line, i.e., $\pi'(s) = \pi(s,2)$ for any $s \in S$. To π' is associated its transition digraph, whose nodes are the states in S, and there is an arc from s to s' if and only if $s' = \pi'(s)$. This digraph is composed of a certain number of connected components, say r, each of them of a similar shape, that is a circuit with inward trees rooted at the nodes of the circuit. Let $C_1, \ldots, C_r$ be the r circuits corresponding to the r connected components of the transition digraph, and let γ be the least common multiple of the number of arcs of these circuits, i.e., $\gamma = \text{lcm}(|C_1|, \ldots, |C_r|)$. We prove that there is a line of length proportional to $2\gamma + |S|$ in which A and A' do not rendezvous.

First, observe that if A and A' cannot go at arbitrarily large distance from their starting positions, say they go at maximum distance D, then they cannot rendezvous in a line of length $4D+4$. Indeed, if the initial positions are two nodes at distance $2D+1$, and at distance at least $D+1$ from the extremities of the line, then the ranges of activity of the two agents are disjoint, and thus they cannot meet (one edge is added at one extremity of the line to break the symmetry of the initial configuration).

Thus from now on, we assume that both agents can go at arbitrarily large distance from their starting positions.

For the purpose of establishing our result, place the two agents A and A' on two adjacent nodes v_A and $v_{A'}$ of an infinite line (whose edges are properly colored). Let $e = \{v_A, v_{A'}\}$ be the edge linking these two nodes.

- Let t_0 be large enough so that A is at distance at least $2\gamma + |S|$ from its starting position after t_0 steps.

Since $t_0 > |S|$, agent A at time t_0 is in some state $s_i \in C_i$ for some $i \in \{1, \ldots, r\}$. In fact, since $|C_i|$ divides γ, agent A has fully executed C_i at least twice.

We define the notion of *extreme position* for a circuit C. Let $s, \pi'(s), \ldots, \pi'^{(k)}(s)$ be a circuit, with $s = \pi'^{(k)}(s)$. Assume that agent A starts in state s from node u_0 at distance at least $k+1$ from both extremities of the line. After having performed C exactly once, i.e., after k steps, agent A is at some node u_k, back in state s. Let $u_0, u_1, u_2, \ldots, u_k$ be the $k+1$ non necessarily distinct nodes visited by A while executing C. The *extreme position* for C starting in state s is the node u_j satisfying $\text{dist}(u_0, u_j) = \text{dist}(u_0, u_k) + \text{dist}(u_k, u_j)$ and $\text{dist}(u_0, u_j) = \max_{0 \leq \ell \leq k} \text{dist}(u_0, u_\ell)$. Let u_i be the extreme position for C_i starting in s_i, and let us define the following parameters:

- τ is the first time step among the $|C_i|$ steps after step t_0 at which A reaches u_i.
- x is the distance of agent A at time τ from its original position, i.e., $x = \text{dist}(u_i, v_A)$;
- $\tau' = \tau + 2\gamma$;
- x' is the distance of agent A' at time τ' from its original position $v_{A'}$.

Note that, by symmetry of the port labeling, and from the fact that A and A' are identical and operate in an infinite line, the two agents are on the two different sides of edge e at time τ. Note also that, between times τ and τ', agent A' keeps on going further away from its original position, by repeating the sequence of actions determined by the circuit C_i. Hence $x' \neq x$. Actually, we have $x' > x$. We can therefore consider the following construction.

Initial configuration of the agents. Let $\mathcal{L}$ be the properly 2-edge-colored line of length $x + x' + 1$, formed by x edges, followed by one edge called e, and followed by x' edges. The two agents A and A' are placed at the two extremities v_A and $v_{A'}$ of e, the same way they were placed at the two extremities of e in the infinite line used to define x and x'.

Since $x \neq x'$, the initial positions of agents are not symmetric. Nevertheless, we prove that the two agents never meet in $\mathcal{L}$, and thus rendezvous, even without termination, is not accomplished. The adversary imposes no delay between the starting times of the agents, i.e., they both start acting simultaneously from their respective initial positions.

One ingredient used for proving that the two agents do not rendezvous is the following general result, that we state as a lemma for further reference.

Parity Lemma. *Consider two (not necessarily identical) agents initially at odd distance in a tree T, that start acting simultaneously in T. Let $t \geq 1$. Assume that one agent stays idle q times in the time interval $[1, t]$, while the other one stays idle q' times in the same time interval. If $|q - q'|$ is even, then the two agents are at odd distance at step t.*

The Parity Lemma enables to establish the following.

Lemma 5. *The two agents A and A' do not meet during the first τ steps.*

At step τ, the behavior of the two agents becomes different. Indeed, agent A is reaching one extremity of $\mathcal{L}$, while A' is visiting a degree-2 node.

We analyze the states of the two agents when they reach extremities of $\mathcal{L}$ during the execution of their protocol. Assume that agent A reaches the extremities of $\mathcal{L}$ at least $k \geq 1$ times. Let σ_j be the state of agent A when it reaches any of the two extremities of $\mathcal{L}$ for the jth time, $1 \leq j \leq k$.

Lemma 6. *Agent A' reaches the extremities of $\mathcal{L}$ at least k times. Moreover, if σ'_j is the state of agent A' when it reaches any of the two extremities of $\mathcal{L}$ for the jth time, $1 \leq j \leq k$, then $\sigma'_j = \sigma_j$.*

After time τ the walks of the agents can be decomposed in two different types of subwalks. A *traversal period* for an agent is the subwalk between two consecutive hits of two different extremities of $\mathcal{L}$ by this agent. A *bouncing period* for an agent is a subwalk (possibly empty) performed between two consecutive traversal periods. Roughly, a bouncing period for an agent is a walk during which the agent starts from one extremity of $\mathcal{L}$ and repeats bouncing (i.e., leaving and going back) that extremity until it eventually starts the next traversal period.

Globally, an agent starts from its original position, performs some initial steps (τ for A, and τ' for A'), and then alternates between bouncing periods and traversal periods. These periods are not synchronous between the two agents because there is a delay of 2γ between them. Nevertheless, by Lemma 6, if one agent bounces at one extremity of $\mathcal{L}$ during its kth bouncing period, then the other agent bounces at the other extremity of $\mathcal{L}$ during its kth bouncing period. Similarly, if one agent traverses $\mathcal{L}$ during its kth traversal period, then the other agent traverses $\mathcal{L}$ in the opposite direction during its kth traversal period. In fact, Lemma 6 guarantees that the two agents perform symmetric actions with a delay of 2γ, alternating bouncing at the two different extremities of $\mathcal{L}$, and traversing $\mathcal{L}$ in two opposite directions.

The following lemma holds, by establishing that whenever one agent is in a bouncing period, the two agents are far apart.

Lemma 7. *The two agents A and A' do not meet whenever one of them is in a bouncing period.*

The following lemma holds, by using the fact that γ is the least common multiple of the circuit lengths in the transition digraph of the agents, and by applying the Parity Lemma.

Lemma 8. *The two agents A and A' do not meet when both of them are in a traversal period.*

Proof of Theorem 2. The two agents start an initial period that lasts τ steps. By Lemma 5 they do not meet during this period. Then the two agents alternate between bouncing periods and traversal periods. By Lemma 7, they do not meet when one of the two agents is in a bouncing period. When the two agents are in a traversal period, Lemma 8 guarantees that they do not meet. Hence the two agents never meet, in spite of starting from non-symmetric positions, and thus they do not rendezvous in $\mathcal{L}$. □

Remark. By construction of the line $\mathcal{L}$ and the setting of γ in the proof of Theorem 2, we get that $\mathcal{L}$ is of length $O(|S|^{|S|})$. Therefore, rendezvous in all bounded degree trees of size at most n requires agents with memory of size at least $\Omega(\log \log n)$ bits.

4 Conclusion

Table 1 summarizes the contributions of this paper concerning the memory size required for the rendezvous problem, in comparison with previous contributions concerning the memory size required for the exploration problem. A natural extension of this paper would be to investigate the gathering problem, in which a set of $k \geq 2$ agents scattered in a network have to gather at a same node. In particular, it would be interesting to check whether this problem can be solved by agents with $O(\log n)$ bits of memory, in trees of size at most n.

References

1. Aleliunas, R., Karp, R.M., Lipton, R.J., Lovász, L., Rackoff, C.: Random walks, universal traversal sequences, and the complexity of maze problems. In: Proc. 20th Annual Symposium on Foundations of Computer Science (FOCS 1979), pp. 218–223 (1979)
2. Alpern, S.: The rendezvous search problem. SIAM J. on Control and Optimization 33, 673–683 (1995)
3. Alpern, S.: Rendezvous search on labelled networks. Naval Reaserch Logistics 49, 256–274 (2002)
4. Alpern, S., Gal, S.: The theory of search games and rendezvous. Int. Series in Operations research and Management Science. Kluwer Academic Publisher, Dordrecht (2002)
5. Alpern, J., Baston, V., Essegaier, S.: Rendezvous search on a graph. Journal of Applied Probability 36, 223–231 (1999)
6. Alpern, S., Gal, S.: Rendezvous search on the line with distinguishable players. SIAM J. on Control and Optimization 33, 1270–1276 (1995)
7. Anderson, E., Weber, R.: The rendezvous problem on discrete locations. Journal of Applied Probability 28, 839–851 (1990)
8. Anderson, E., Essegaier, S.: Rendezvous search on the line with indistinguishable players. SIAM J. on Control and Optimization 33, 1637–1642 (1995)
9. Anderson, E., Fekete, S.: Asymmetric rendezvous on the plane. In: Proc. 14th Annual ACM Symp. on Computational Geometry (1998)
10. Anderson, E., Fekete, S.: Two-dimensional rendezvous search. Operations Research 49, 107–118 (2001)
11. Baston, V., Gal, S.: Rendezvous on the line when the players' initial distance is given by an unknown probability distribution. SIAM J. on Control and Optimization 36, 1880–1889 (1998)
12. Baston, V., Gal, S.: Rendezvous search when marks are left at the starting points. Naval Reaserch Logistics 48, 722–731 (2001)
13. Cook, S.A., McKenzie, P.: Problems complete for deterministic logarithmic space. Journal of Algorithms 8(5), 385–394 (1987)
14. Coppersmith, D., Doyle, P., Raghavan, P., Snir, M.: Random walks on weighted graphs, and applications to on-line algorithms. In: Proc. 22nd Annual ACM Symposium on Theory of Computing (STOC 1990), pp. 369–378 (1990)
15. Coppersmith, D., Tetali, P., Winkler, P.: Collisions among random walks on a graph. SIAM J. on Discrete Math. 6, 363–374 (1993)

16. De Marco, G., Gargano, L., Kranakis, E., Krizanc, D., Pelc, A., Vaccaro, U.: Asynchronous deterministic rendezvous in graphs. Theoretical Computer Science 355, 315–326 (2006)
17. Dessmark, A., Fraigniaud, P., Kowalski, D., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica 46, 69–96 (2006)
18. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. Journal of Algorithms 51, 38–63 (2004)
19. Dyer, M., Frieze, A., Kannan, R.: A random polynomial time algorithm for estimating volumes of convex bodies. In: Proc. 21st Annual ACM Symposium on Theory of Computing (STOC 1989), pp. 375–381 (1989)
20. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous oblivious robots with limited visibility. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 247–258. Springer, Heidelberg (2001)
21. Gal, S.: Rendezvous search on the line. Operations Research 47, 974–976 (1999)
22. Gasieniec, L., Pelc, A., Radzik, T., Zhang, X.: Tree exploration with logarithmic memory. In: Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007), pp. 585–594 (2007)
23. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self stabilizing mutual exclusion. In: Proc. 9th Annual ACM Symposium on Principles of Distributed Computing (PODC 1990), pp. 119–131 (1990)
24. Kowalski, D., Malinowski, A.: How to meet in anonymous network. In: Flocchini, P., Gasieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 44–58. Springer, Heidelberg (2006)
25. Kranakis, E., Krizanc, D., Morin, P.: Randomized Rendez-Vous with Limited Memory. In: Proc. 8th Latin American Theoretical INformatics (LATIN) (2008)
26. Kranakis, E., Krizanc, D., Santoro, N., Sawchuk, C.: Mobile agent rendezvous in a ring. In: Proc. 23rd International Conference on Distributed Computing Systems (ICDCS 2003), pp. 592–599 (2003)
27. Lim, W., Alpern, S.: Minimax rendezvous on the line. SIAM J. on Control and Optimization 34, 1650–1665 (1996)
28. Schelling, T.: The strategy of conflict. Oxford University Press, Oxford (1960)
29. Thomas, L.: Finding your kids when they are lost. Journal on Operational Res. Soc. 43, 637–639 (1992)
30. Yu, X., Yung, M.: Agent rendezvous: a dynamic symmetry-breaking problem. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 610–621. Springer, Heidelberg (1996)

Broadcasting in UDG Radio Networks with Missing and Inaccurate Information

Emanuele G. Fusco[1] and Andrzej Pelc[2,⋆]

[1] Computer Science Department, Sapienza, University of Rome, 00198 Rome, Italy
fusco@di.uniroma1.it

[2] Département d'informatique, Université du Québec en Outaouais,
Gatineau, Québec J8X 3X7, Canada
pelc@uqo.ca

Abstract. We study broadcasting time in radio networks, modeled as unit disk graphs (UDG). Emek et al. showed that broadcasting time depends on two parameters of the UDG network, namely, its diameter D (in hops) and its *granularity* g. The latter is the inverse of the *density* d of the network which is the minimum Euclidean distance between any two stations. They proved that the minimum broadcasting time is $\Theta\left(\min\left\{D+g^2, D\log g\right\}\right)$, assuming that each node knows the density of the network and knows exactly its own position in the plane.

In many situations these assumptions are unrealistic. Does removing them influence broadcasting time? The aim of this paper is to answer this question, hence we assume that density is unknown and nodes perceive their position with some unknown error margin ϵ. It turns out that this combination of missing and inaccurate information substantially changes the problem: the main new challenge becomes fast broadcasting in sparse networks (with constant density), when optimal time is $O(D)$. Nevertheless, under our very weak scenario, we construct a broadcasting algorithm that maintains optimal time $O\left(\min\left\{D+g^2, D\log g\right\}\right)$ for all networks *with at least 2 nodes*, of diameter D and granularity g, if each node perceives its position with error margin $\epsilon = \alpha d$, for any (unknown) constant $\alpha < 1/2$. Rather surprisingly, the minimum time of an algorithm stopping if the source is alone, turns out to be $\Theta(D+g^2)$. Thus, the mere stopping requirement for the special case of the lonely source causes an exponential increase in broadcasting time, for networks of any density and any small diameter. Finally, broadcasting is impossible if $\epsilon \geq d/2$.

1 Introduction

The model and the problem. A *radio network* consists of stations with transmitting and receiving capabilities. Stations have synchronized clocks showing the *round* number. In a given round each station can act either as a *transmitter* or as a *receiver*. The network is modeled as an undirected graph called a *unit disk graph*

[⋆] Partially supported by NSERC discovery grant and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 257–273, 2008.

(UDG) whose nodes are the stations. These nodes are represented as points in the plane. Two nodes are joined by an edge if their Euclidean distance is at most 1. Such nodes are called *neighbors*. It is assumed that transmitters of all stations have equal power which enables them to transmit at Euclidean distance 1. We also assume that the terrain is flat and without large obstacles. Hence an edge between two nodes indicates that transmissions of one of them can reach the other, i.e., they can communicate directly. We refer to radio networks modeled by unit disk graphs as UDG radio networks.

A node acting as a transmitter in a given round sends a message delivered to all of its neighbors in the same round. An important distinction from the receiver's point of view is between a message being just *delivered* and being *heard*, i.e., received successfully by a node. A node acting as a receiver in a given round *hears* a message, if and only if, a message from exactly one of its neighbors is delivered in this round. If two neighbors v and v' of u transmit simultaneously in a given round, none of the delivered messages is heard by u in this round. In this case we say that a *collision* occurred at u. It is assumed that the effect of a collision is the same as that of no message being delivered in this round, i.e., a node cannot distinguish a collision from silence.

The network topology is assumed to be *unknown*, namely, each node is unaware of the coordinates of any other node including its neighbors. Likewise, nodes do not know any bound on the size of the network or on its diameter. Such networks are often called *ad hoc* networks.

We consider *broadcasting*, which is one of the basic communication primitives. One distinguished node, called the *source*, has a message which has to be transmitted to all other nodes. Remote nodes get the source message via intermediate nodes, along paths in the network. We assume that all stations are awake since round 1, and may transmit control messages even before they heard the source message. In order for the broadcasting to be feasible, we assume that the unit disc graph underlying the UDG radio network is connected.

The above described scenario was adopted in [1] (under the name of the spontaneous wake up model) with two additional assumptions. It was assumed that the nodes are aware of (a linear lower bound on) the *density* d of the network, which is the minimum Euclidean distance between any two nodes, and that each node knows its *exact* position in the plane, i.e., is aware of the exact values of its Euclidean coordinates in a global coordinate system.

Each of these two assumptions may be difficult to satisfy in practical applications. If nodes of a wireless network, e.g., sensors equipped with transmitters, are dynamically added to a network at unknown times and locations, it may be difficult or impossible to estimate the density of the network at any given time. On the other hand, reading Euclidean coordinates of a position, e.g., using a GPS, seems to be inherently prone to inaccuracies. Hence it is important to determine to what extent these two additional assumptions influence the time of broadcasting, and in particular, whether optimal broadcasting time changes if these two assumptions are removed. Establishing this influence and constructing optimal algorithms that do not use the additional assumptions is the goal of the present paper.

We work in the above described model from [1] modified by removing the assumptions of density knowledge and of availability of exact positions of nodes. Instead, if the real position of a node is (x, y), we assume that this node perceives its position as being (x', y'), where (x', y') is chosen by an adversary. The distance between (x, y) and (x', y') is bounded by the *error margin* ϵ, a positive real parameter of the model. The point (x', y'), is called the *perceived position* of the node $v = (x, y)$ and is denoted by $P(v)$. It should be stressed that we do *not* assume the knowledge of ϵ either. All other characteristics of the model are as mentioned before, and as in [1]. In particular, the topology, size and diameter of the network are unknown to nodes. We consider only distributed deterministic broadcasting. Thus decisions made by a node on whether to transmit or to receive in a given round, and what message to transmit, if any, are based exclusively on its perceived position and on the messages it heard so far. The execution time of a broadcasting algorithm in a given radio network is the smallest round number after which all nodes of the network have heard the source message and no other messages are sent.

Our definition of time of a broadcasting algorithm is slightly different from that in [1] and the same as, e.g., in [2]. In [1], time was defined simply as the smallest round number t after which all nodes of the network have heard the source message. Hence the messages could be subsequently sent indefinitely by various nodes without any possibility of stopping and the algorithm would be still considered correct and running in a given time t. The difference between these two definitions is negligible, if parameters of the network (the density and the diameter) are known to nodes. In this case the time of informing all nodes can be precomputed and the source can send a stopping message after this time, without changing broadcasting complexity. However, if the density is unknown, the difference between an algorithm that stops and one that does not is not trivial, and so is the difference between the two above definitions of execution time. We will see in the sequel that the requirement of total silence after the end of the algorithm is crucially used in our lower bound on time. We consider the restriction to broadcasting algorithms that stop and the definition of time requiring silence after the last round to be more natural and useful: algorithms that do not stop require an external "help", e.g., user's intervention, to be implemented and thus cannot be considered to be autonomous algorithms. Hence we use this definition rather than that from [1]. However, it follows from our result that the algorithm from [1] can be converted to a stopping algorithm and it will have the same time complexity according to our more demanding definition. In view of this, we refer to the time of the algorithm in [1], without distinguishing between the two definitions.

It was proved in [1] that the optimal broadcasting time (with each node knowing the density and its own exact position) is $\Theta\left(\min\left\{D + g^2, D \log g\right\}\right)$, where D is the diameter of the network (in the number of hops), and g is the inverse of the density, called the *granularity* of the network. The aim of the present paper is to establish whether this optimal broadcasting time changes when the additional assumptions are removed.

Our results. It turns out that the combination of missing and inaccurate information (unknown density and unknown error in perceiving positions) substantially changes the problem with respect to the easier scenario from [1]. The main challenge in our setting becomes fast broadcasting in sparse networks (with constant density), when optimal time is $O(D)$. (This was an easy task in the previous scenario.) One new difficulty comes from the fact that with unknown constant density d and unknown error margin ϵ possibly close to $d/2$, nodes with arbitrarily close perceived positions may be unable to communicate with each other. This invalidates election techniques used in [1]. The second new difficulty is the stopping problem combined with ignorance of network parameters, in particular of the density and of the diameter. Not knowing these parameters makes it impossible to predict when the entire algorithm or its particular procedures will finish. Thus simple time-out conditions used in [1] are no more available and special care must be taken to explicitly stop the algorithm at the proper time.

Nevertheless, under our very weak scenario, we construct a broadcasting algorithm that maintains optimal time $O\left(\min\left\{D+g^2, D\log g\right\}\right)$ for all networks *with at least 2 nodes*, of diameter D and granularity $g = 1/d$, assuming that each node perceives its position with error margin $\epsilon = \alpha d$, for any unknown constant $\alpha < \frac{1}{2}$. Rather surprisingly, the minimum time of an algorithm working correctly for *all* networks, and hence stopping if the source is alone, turns out to be $\Theta(D+g^2)$. Thus, the mere stopping requirement for the special case of the lonely source causes an exponential increase in broadcasting time, for networks of any density and any small diameter, e.g., polylogarithmic in g. Finally, we observe that bounding the error margin below $d/2$ is necessary: indeed, if $\epsilon \geq d/2$, then broadcasting turns out to be impossible in many networks. Due to lack of space, proofs are omitted.

Related work. Algorithmic aspects of radio communication were mostly studied for arbitrary graphs, either assuming complete knowledge of the network or assuming only limited information about the topology, available to nodes. The present paper belongs to the second area.

The first paper to study deterministic broadcasting in radio networks, assuming complete knowledge of the topology, was [3]. The authors also defined the graph model of radio networks subsequently used in many other papers. In [4], an $O(D\log^2 n)$-time broadcasting algorithm was proposed for all n-node networks of diameter D. This time complexity was then improved in a sequence of papers [5, 6, 7], culminating with the $O(D+\log^2 n)$ algorithm from [8]. The latter complexity is optimal in view of the lower bound $\Omega(\log^2 n)$ from [9].

Investigation of deterministic distributed broadcasting in radio networks, whose nodes have only local knowledge of the topology, was initiated in [10]. Several authors [2, 11, 12, 13, 14, 15, 16] studied deterministic broadcasting in radio networks assuming that nodes know only their own label. Such networks are called ad hoc. In [2] the authors gave a broadcasting algorithm working in time $O(n)$ for all symmetric n-node networks, assuming that nodes can transmit spontaneously, before getting the source message. A matching lower bound $\Omega(n)$ for symmetric networks was proved in [16]. In [2, 12, 13, 15] arbitrary directed graphs were considered. The currently fastest deterministic broadcasting algorithms working for arbitrary ad hoc networks have running times $O(n \log^2 D)$ [15] and $O(n \log n \log \log n)$ [17]. In [14] an $\Omega(n \log D)$ lower bound was proved. Randomized broadcasting in radio networks was studied in [10, 15, 18].

Another model of radio networks is based on geometry. Stations are represented as points in the plane and the graph modeling the network is no more arbitrary. It may be a unit disk graph, or one of its generalizations. Broadcasting in such geometric radio networks was considered, e.g., in [1, 19, 20, 21]. Deterministic broadcasting in geometric radio networks with restricted knowledge of topology was studied in [19]. In the model corresponding to our present scenario, the authors showed a broadcasting algorithm which works in time linear in the number of nodes, assuming that nodes are labeled by consecutive integers. In a recent paper [1] the authors considered broadcasting in radio networks modeled by unit disk graphs. Some of their results were discussed above. Other papers studied tasks other than broadcasting (e.g., the maximum independent set problem [22] or the coloring problem [23]) in radio networks modeled by unit disk graphs and their generalizations.

2 Terminology and Preliminaries

We may assume that the (unknown) density d of the network is at most 1, otherwise all nodes would be isolated. The *granularity* of the network is $g = 1/d$. We assume that the error margin ϵ on the perception of node positions is at most αd, for some (unknown) constant $\alpha < 1/2$; this implies $\epsilon < 1/2$. First observe that bounding ϵ below $d/2$ is necessary.

Proposition 1. *For error margin $\epsilon \geq d/2$ there exists a 4-node UDG radio network of density d in which broadcasting is impossible.*

Algorithms in [1] are based on three types of grids, composed of atomic squares with generic name *boxes*. The first grid is composed of *tiles*, of side length $d/\sqrt{2}$, the second of *blocks*, of side length $1/\sqrt{2}$ and the third one of *5-blocks*, of side length $5/\sqrt{2}$. Grids are aligned with coordinate axes. Tiles are small enough to contain at most one node. Blocks have diameter 1, hence all nodes in a block are able to communicate. 5-blocks are used to avoid collisions.

In our setting a node can decide if it is in a region only based on its perceived position. We say that a node *inhabits* a given box if its perceived position belongs to this box. Two boxes are *potentially reachable* from one another, if they can be inhabited by a pair of nodes with real positions at distance at most 1. Two boxes are *reachable* from one another, if they are inhabited by such a pair of nodes. If d can be as large as 1, any constant side length of a box is too large to ensure the property that nodes which inhabit the same box have real positions at distance at most 1, and hence that they can communicate. Indeed, let $d = 1$ and let ℓ be an arbitrary constant. Take ϵ such that $1/2 > \epsilon > (1-\ell)/2$ and consider a pair of nodes u and v at distance δ where $1 < \delta < \ell+2\epsilon$. The distance between perceived positions of nodes u and v can be as small as $\delta - 2\epsilon < \ell$, hence they may inhabit a box of side length $\ell/\sqrt{2}$, but be unable to communicate. It follows that we cannot partition the plane into boxes guaranteeing full communication within a box. This is the reason of designing separate algorithms for d below some threshold, when such a communication is possible, and for d above this threshold, when full communication within a box will not be needed. Let $\Delta = \left(\sqrt{2}-1\right) / \left(2\sqrt{2}+2\right)$. We take 2Δ as the threshold for d. A network is *sparse* if $d \geq 2\Delta$ and *dense* otherwise.

Both in the algorithm for sparse networks and in algorithms for dense networks we will heavily use the concept of *multiplexing* of procedures. By multiplexing we mean that the execution of a procedure, described as a sequence of consecutive steps, will be interleaved with the execution of other procedures needed to complete the task. In general, only one step of each procedure is executed in a round robin fashion, unless explicitly stated. Divisions in blocks and (as we will see in the next section) assignment of colors are used to interleave the execution of the same task in different rounds (depending on colors, on blocks or on a combination of both) in order to avoid collisions. Multiplexing will also be used on a higher level, in order to interleave the execution of different algorithms developed below. Indeed, as we are unaware of the value of d, we are unable to determine in advance if the network is sparse or dense, thus we have to run concurrently the algorithms for dense and sparse networks. This allows us to always complete the task in optimal time, by stopping slower algorithms after completion of the one that is the best for a given network.

3 Broadcasting in Sparse Networks

In this section we describe Algorithm `Color&Transmit`, working correctly for sparse networks. We call *block* a box with side length 1. Blocks are used to build a grid in the same fashion as mentioned in Sect. 2. Another grid is composed of 5-*blocks*, i.e., boxes of side length 5. Nodes which inhabit a block must have real position within distance $\epsilon < 1/2$ from the block. It follows that any transmission made by a node inhabiting the central block of a 5-block can only be heard by nodes which inhabit the 5-block.

In sparse networks, $\gamma = \left\lceil \frac{\pi}{\sqrt{3}} \cdot \frac{\sqrt{2}+1}{\sqrt{2}-1} \cdot \left(2 + \frac{\sqrt{2}-1}{\sqrt{2}+1}\right)^2 \right\rceil$ is an upper bound on the number of nodes which can inhabit a block (we use $\pi/\sqrt{12}$ as the upper bound on the ratio of the sum of areas of pairwise disjoint circles of radius Δ in a square of side length $2+2\Delta$ to the area of this square). We reserve a total of 25γ distinct colors. Nodes in block B_i of a 5-block, $1 \leq i \leq 25$, can be colored with colors from set C_i, where $|C_i| = \gamma$ and sets C_i are pairwise disjoint.

In our algorithm we will use a *grid refinement* process in order to perform various tasks. Nodes participating in grid refinement are selected among inhabitants of a box. As it is impossible to guarantee full communication in a box with any constant side length, we use a distinguished node, called *witness*, which coordinates the process and determines the set of participants, depending on the condition whether they are able to communicate with the witness or not.

The grid refinement process proceeds in phases. In the first phase, the whole box is divided in 4 square tiles, numbered from 1 to 4 proceeding left to right, top to bottom. In successive phases, the side length of tiles is halved, thus quadrupling the number of tiles in the grid. Tiles are allotted rounds in a round robin fashion. A participating node with perceived position in the i-th tile, transmits during rounds allotted to its tile. Rounds allotted to the witness are interleaved with those of the tiles. If only one participating node inhabits a tile, its transmissions will be heard by the witness (which thus learns the perceived position of the participating node). The witness sends a confirmation to the node immediately after. If more than one participating node inhabits a tile, transmissions collide, thus the witness hears silence and does not send any confirmation. At the end of each phase, it may be needed to check if the grid refinement has correctly terminated. If needed, such a check is performed as follows. First, one distinguished node u within communication range of the witness must be provided (such a node will be always explicitly defined whenever needed). Once node u is fixed, it transmits together with all participating nodes that did not receive a confirmation during the current phase. The witness is thus able to distinguish whether the process is completed or not; if it hears the message from u, this means that the grid is fine enough to have at most one participating node in each tile and the process is complete. Otherwise, the grid is further refined by halving the side length of tiles, and a new phase begins. Notice that only a constant number of phases are needed to complete the process because $\epsilon \leq \alpha d$, for some constant $\alpha < 1/2$, and hence the distance between perceived positions of distinct nodes is lower-bounded by a positive constant. Upon completion of grid refinement, the witness and all participating nodes know the complete set of participants (participating nodes learn it through confirmations of the witness).

Now we are able to describe Algorithm `Color&Transmit`. Broadcasting is based on a coloring of nodes in the network. The coloring algorithm defines a spanning tree of the network, whose height is in $O(D)$. The *parent* and *child* relations will always refer to this tree. The coloring satisfies the following conditions: for every pair (v_1, v_2) of nodes having the same color, the set of children of v_1 in the spanning tree does not contain any neighbor of v_2 (i.e., v_2 is at distance greater than 1 from any children of v_1). Moreover, the parent of v_1 (respectively v_2) is not adjacent to v_2 (respectively v_1). Siblings in the spanning tree have different colors and each node has a color different from the one of its parent. Notice that we do not enforce to assign different colors to neighbors in the graph.

Once a coloring satisfying the above conditions is available, it is possible to ensure that transmissions from a parent (child) reach all its children (its parent) without collisions, by allotting distinguished rounds to nodes of different colors. The bounded height of the spanning tree ensures termination of broadcasting in time $O(D)$, provided that the total number of colors is bounded by a constant.

Procedure. `Assign-Color` [Input: a pair of blocks B_i, B_j in a 5-block, and a color c in C_i.]

The procedure assigns, in constant time, colors from set C_j to those nodes in B_j that are yet uncolored and have a neighbor with color c in block B_i, respecting the coloring conditions.

We say that a node is *out of the tree* if it does not know its parent. During the coloring process we maintain the invariant that nodes with the same color do not share neighbors out of the tree.

In the first round of Procedure `Assign-Color`, all nodes with color c in block B_i transmit. Next, out of the tree neighbors of each node w with color c in block B_i, inhabiting block B_j, start a grid refinement process with w as a witness. (Thus, many grid refinement processes may be simultaneously active.) Node w becomes the parent of participants of the grid refinement process. In order to check termination of each grid refinement process, we need one more distinguished node u: this node is the parent of the corresponding node w. Notice that each node w has a different parent, as siblings do not get the same color. Consider two nodes, w and w', with color c in block B_i. Let p be the parent of w and p' be the parent of w'. The coloring must satisfy that w is not a neighbor of p', thus transmissions made by p and p' cannot collide at w. In constant time, all children of w, inhabiting block B_j (and w itself) know the full list L of perceived positions of children of w inhabiting block B_j. (Recall that children of w inhabiting block B_j may be unable to hear each other directly, so they rely on w to learn the list L.)

Subroutine. `Conflict-Detection` [Input: a quadruple (B_i, B_j, c, c') where B_i and B_j are blocks and $c \in C_i$, $c' \in C_j$ are colors.]

The subroutine allows each node v in block B_j seeking color c', whose parent w in block B_i has color c, to distinguish among three possible outcomes: node v can either *win* color c', *lose* it, or *make a draw* on it. All nodes seeking color c' are called *competitors*.

Subroutine `Conflict-Detection` works in 4 *consecutive* rounds (the usual interleaving of procedures is not respected for subroutine `Conflict-Detection` as interleaving different runs of the subroutine could cause unexpected behavior. As the number of rounds used by Subroutine `Conflict-Detection` is 4, the whole execution can be allotted a segment of 4 consecutive rounds, increasing the delay in the execution of other procedures by a constant factor only.). In the first round of Subroutine `Conflict-Detection` each node v that has not yet lost color c' transmits, claiming color c'. In the second round, all parents w of competing nodes transmit. At the same time, all non-competing nodes that did not hear any claim for color c' in the previous round transmit. If a competitor v heard the message from its parent w in the second round, no node v' sharing a non-competing neighbor with v is competing for color c'.

In the third round, all nodes w in B_i with color c transmit together with all non-competing nodes which know about a previous winner of color c'. If node v heard the message from its parent w in the third round, it means that none of its non-competing neighbors knows about a previous winner.

Node v wins color c', if it heard the message from its parent w in the second and third rounds. Node v loses color c', if it did not hear the message from its parent w in the third round (color c' was already won by a node sharing a non-competing neighbor with v). Node v makes a draw on color c', if it did not hear its parent w in the second round, but heard it in the third round.

Notice that children of v will be selected among out of the tree nodes, and thus are non-competing. Parents of nodes competing with v during the same execution of Procedure `Assign-Color`, have the same color as the parent of v. It follows that they cannot be adjacent to v, as v would have been an out of the tree node when they were assigned a color, and thus either they or the parent of v would have lost the color.

In round 4, each competitor v announces the result: win, lose or draw, informing all its non-competing neighbors in case of victory and at least its parent w in other cases. ♣

Once the list L is known to a parent w with color c in block B_i, the first node v in lexicographic order of perceived positions in the list L starts competing for the first available color $c' \in C_j$ (a color is available if the parent does not know about a previous winner). This is done by calling Subroutine `Conflict-Detection`. If v wins, Procedure `Assign-Color` removes v from the list L and color c' from available colors for all remaining nodes in L. Then, the first node in list L starts competing for the first available color. If v loses, Procedure `Assign-Color` removes color c' from available colors for all nodes in L and v starts competing for the next available color. If v makes a draw, it still needs to compete for color c'. When a draw occurs, nodes are in *conflict*; conflicts are resolved by Subroutine `Conflict-Resolution`.

Each competing node participating in Subroutine `Conflict-Resolution` will either win or lose the color it was competing for. Multiple runs of Subroutine `Conflict-Resolution` can be executed at the same time with the same block arguments, if more than one parent w is present in block B_i. Resolution of conflicts is achieved by ensuring that each of the conflicting nodes becomes the only competitor in a run of Procedure `Conflict-Detection`, within constant time from the first draw. The latter is enough to ensure there is no draw.

This result is achieved by using a grid refinement process that delays successive executions of Subroutine `Conflict-Detection` by increasing amounts of time, as described below.

Subroutine. `Conflict-Resolution` [Input: a quadruple (B_i, B_j, u, v), where $u \in B_i$, $v \in B_j$ and u is the parent of v.]

Subroutine `Conflict-Resolution` proceeds in consecutive phases. Consider phase i for a conflicting node v. Let $p_i(v)$ be the number of the tile inhabited by node v in the i-th grid of the grid refinement process of block B_j. Clearly $1 \leq p_i(v) \leq 4^i$. Let S_v be the time when node v started participating in Procedure `Conflict-Resolution`. In step $S_v + w_i + 4^{i+1} + p_{i+1}(v)$, where $w_i = \sum_{j=1}^{i} 2 \cdot 4^i$ and $w_0 = 0$, node v starts participating in Procedure `Conflict-Detection` for the $(i+1)$-th time. If the outcome of this procedure for node v is a draw, the grid is further refined and a new phase begins. ♣

After resolution of a conflict, node v either wins or loses color c'. The actions of Procedure `Assign-Color` were already described in both these cases. ∎

Lemma 1. *Subroutine* `Conflict-Resolution` *ends in constant time.*

In order to complete Algorithm `Color&Transmit`, we need to describe how to initialize the whole process starting from the source s. s is precolored with the first color available for the block it inhabits. Using a grid refinement process, s can elect a distinguished node in its neighborhood in constant time and assign it a color. (If there is no neighbor of the source, i.e., the source is the only node in the network, the source transmits only once.) This distinguished node allows the source to check the correct termination of the grid refinement process. From now on, using Procedure `Assign-Color`, the whole network can be colored with 25γ colors. Notice that the time taken to assign a color to a node is constant. Once a node is colored, all its neighbors are colored after a constant time. It follows that the coloring ends in time $O(D)$ and thus the height of the induced spanning tree is in $O(D)$. After getting the source message, a node waits until the first round assigned to its color and transmits the message, informing all its children. Confirmation messages are sent back to the source along this tree, starting from the leaves, again using colors to avoid collisions on parent nodes. Each transmission is delayed by a constant time only, and the whole process is completed in time $O(D)$. Confirmation will be used in the main algorithms.

Theorem 1. *There exists a deterministic algorithm that completes broadcast in time $O(D)$ in any UDG radio network of unknown diameter D and unknown density $d \geq 2\Delta$.*

4 Broadcasting in Dense Networks

In this section we assume $d < 2\Delta$. We call *Δ-block* a box with side length $(1-2\Delta)/\sqrt{2}$. Δ-blocks are used to build a grid in the same fashion as before. One more grid is composed of *5Δ-blocks* , i.e., boxes of side length $(1-2\Delta)\,5/\sqrt{2}$. Under the current assumption on d, the distance between the actual positions of two nodes in a Δ-block is < 1.

Lemma 2. *If $\epsilon < \Delta$, transmissions made by nodes inhabiting the central Δ-block of a 5Δ-block can only reach nodes inhabiting the 5Δ-block.*

It follows that there are 24 Δ-blocks potentially reachable from any Δ-block.

An $O\left(D+g^2\right)$-time algorithm. The algorithm `Elect&Transmit` proposed in [1] is based both on the perfect knowledge of positions and on the knowledge of the density of the network. This algorithm elects a pair of adjacent nodes called *ambassadors* for each pair of neighboring blocks in a preprocessing phase of time complexity $O(g^2)$. When d is known, multiplexing allows each node to transmit alone and reveal its position, in $25g^2$ rounds. Then knowledge acquired by each node is spread (in the same fashion) to all its neighbors. As the time taken to complete transmissions in each block is identical and known in advance when d is known, this is enough to elect a pair of ambassadors for each pair of neighboring blocks, e.g., by choosing the first pair in lexicographic order. Once the pairs of ambassadors are defined, broadcasting can be completed in time $O(D)$.

In our setting d is unknown and we only have inaccurate knowledge of positions, hence it is impossible to use algorithm `Elect&Transmit` from [1]. In what follows, we provide a new algorithm working in time $O\left(D + g^2\right)$, called Algorithm `Dense-1`. Recall, from Sect. 3, the description of the grid refinement process. Taking advantage of full communication within a Δ-block, we can mimic the grid refinement process in dense networks without having a predefined witness, as the role of the witness can be played by the first node that is able to transmit alone. Indeed, such a node is heard by all participants in the Δ-block; its perceived position is then appended to all the subsequent messages sent during the execution of the process, thus informing it of its role of witness as soon as a second node is able to transmit alone. In any Δ-block containing at least two nodes the grid refinement process ends in time $O(g^2)$ (the second node transmitting alone is used to check termination). Nodes that are alone in their Δ-block, would be involved in the grid refinement process forever, unless external help allows them to stop. (In our algorithm such help will be, of course, provided.)

In [1] the authors developed a procedure called `Conquer`. This procedure elects a pair of neighboring nodes in two blocks in time $O(\log g)$. The input of Procedure `Conquer` consists of two blocks, B_1 and B_2, and a node b_1 in B_1. Procedure `Conquer` can either be successful or unsuccessful, depending on whether a pair of adjacent nodes exists in the two blocks or not. Termination of Procedure `Conquer` was based on the assumption that d is known. Nodes elected by procedure `Conquer` can be used as ambassadors to spread information from one block to another, but in our setting we need to take additional care in order to guarantee termination. Logarithmic time is achieved in Procedure `Conquer` thanks to the ability of electing a neighbor of a given node c in a region R (of diameter at most 1) in time $O(\log g)$. This is based on procedure `Echo` from [24], which can be used to perform elections of a node in a set in logarithmic time, using a halving process which exponentially decreases the area of the region where the node can be elected. The knowledge of d allows the node c to precompute the duration of an unsuccessful election (i.e., an attempt to elect a node from the empty set). In our setting this is impossible, thus we need to avoid involving nodes in unsuccessful halving processes while electing ambassadors for neighboring Δ-blocks.

In our algorithm, election of ambassadors is performed using Procedure `Safe-Conquer`. Procedure `Safe-Conquer` uses an additional distinguished node with respect to Procedure `Conquer`, that allows to test whether an election based on halving would be successful or not, before attempting it. The procedure elects a pair (u, v) of adjacent nodes such that $u \in B_1$ and $v \in B_2$, in time $O(\log g)$, whenever such a pair exists; otherwise it *stops* in constant time. Now we describe Algorithm `Dense-1`.

In the first round, the source transmits alone. The source will not transmit any other message unless it hears a message from another node, allowing the protocol to end in time 1 when the whole network consists only of the source.

Starting from round 2, all Δ-blocks start the grid refinement process. Let B be a Δ-block in which the process ends, and let a and b be the first two nodes that transmitted alone in B. For any such block, Procedure `Safe-Conquer` is applied using as parameters B, B', a, and b, for every Δ-block B' potentially reachable from B (there are at most 24 such Δ-blocks).

As soon as the source s hears a message (which happens after time $O\left(g^2\right)$ in networks with more than one node), it starts participating in two tasks. The first task is the grid refinement in its own Δ-block. The second task is the election of a node b among those informed during the first round, based on halving. This election is performed for each of the 25 Δ-blocks potentially reachable from the source, including its own Δ-block (using multiplexing). Election is successful in one of these Δ-blocks and it is completed in time $O(\log g)$. Let S be the Δ-block inhabited by s. As soon as node b is elected, Procedure `Safe-Conquer` is applied using as parameters S, T, s and b, for any of the 24 Δ-blocks T potentially reachable from S.

For any Δ-block with at least 2 nodes (and for the Δ-block inhabited by the source), election of all ambassadors is completed in time $O\left(g^2 + \log g\right) = O\left(g^2\right)$. Whenever a pair of ambassadors (u, v) for a pair of blocks (B, B') is elected, if B' is a Δ-block where grid refinement is still running, node v verifies if it is alone in B'. This is done by reserving a round where each node in B', except v, transmits together with u. If v can hear u, it is alone in B' and it stops executing grid refinement, otherwise other nodes are present and grid refinement will eventually terminate. It follows that after $O\left(g^2\right)$ time, the only Δ-blocks that are still running grid refinement are those Δ-blocks containing only one node that have no neighbors in Δ-blocks with 2 or more nodes. For any such Δ-block B, the election of ambassadors for the 24 pairs of blocks (B, B'), where B' is potentially reachable from B, can be completed in constant time using Procedure `Safe-Conquer`. In each case, the parameters of this procedure will be B, B', the unique node u inhabiting B, and the node v that first informed u.

The source message is then passed through ambassadors. Let (u, v) be a pair of ambassadors for Δ-blocks B, B'. If v is newly informed by u, it informs B' and informs u that B' has been newly informed by B. In this case we say that (u, v) is the *informing couple* of B'. Otherwise v does not transmit. A Δ-block is a leaf if it does not newly inform any other Δ-block. Confirmation of broadcast completion proceeds from leaf Δ-blocks to the source, again using ambassadors.

Theorem 2. *There exists a deterministic algorithm that completes broadcast in time $O\left(D + g^2\right)$ in any UDG radio network of unknown diameter D and unknown density $d = 1/g < 2\Delta$.*

An $O\left(D \log g\right)$-time algorithm. In order to broadcast in time $O\left(D \log g\right)$ we restrict attention to networks with at least two nodes. In Sect. 6 we show that this restriction is necessary.

Algorithm Dense-2, is designed for networks of size at least 2 and works in time $O\left(D\log g\right)$. In such networks, we are guaranteed that there exists a node b adjacent to the source s, thus we can elect such a node b in logarithmic time in the same fashion as we did in Algorithm Dense-1, without waiting for the source to receive any message. Procedure Safe-Conquer is then used to elect pairs of ambassadors for any pair of Δ-blocks (B, B'), where B is inhabited by the source and B' is reachable from B. In general, fix a Δ-block B_1 not reachable from B. Consider the informing couple (u, v) of ambassadors for B_1. Nodes u and v are then used as parameters of Procedure Safe-Conquer for any Δ-block B_2 potentially reachable from B_1.

Election of ambassadors and spreading of the source message proceeds by a wave originating from the Δ-block inhabited by the source. Confirmation is done as previously. Since each election takes time $O\left(\log g\right)$, broadcasting is completed in time $O\left(D\log g\right)$.

Theorem 3. *There exists a deterministic algorithm that completes broadcast in time $O\left(D\log g\right)$ in any UDG radio network with at least two nodes, unknown diameter D and unknown density $d = 1/g < 2\Delta$.*

5 The Main Algorithms

Algorithms Dense-1 and Dense-2 developed for dense networks, may fail on a sparse network, as the assumption of having full communication inside a Δ-block may not hold. On the other hand, Algorithm Color&Transmit, developed for sparse networks, can fail on dense networks, as it can run out of colors when there are more than γ nodes in a block. In order to develop an algorithm that is suitable for all networks, we multiplex the execution of algorithms conceived for dense and for sparse networks, and use confirmation in order to stop the execution of the slower ones after the completion of the fastest. (Such stopping is crucial because grid refinement for dense networks could run forever on sparse networks.) The only nodes sending confirmation to the source in algorithms for dense networks are ambassadors. We need to ensure that whenever an algorithm for dense networks fails, at least one of the ambassadors becomes aware of the error, thus not sending confirmation back to the source. On the other hand, in algorithm Color&Transmit all nodes send confirmation, and thus failures are readily detected.

Procedure. Error-Detection [Input: a Δ-block B.]

Let (u, v) be the informing couple of B. (If B is the Δ-block inhabited by the source s, the informing couple is replaced by the pair (b, s), where b is the elected neighbor of s.) The procedure flags an error whenever a node at distance (in hops) at most 2 from v is not informed.

In the first round, v transmits together with all uninformed nodes. Let x be an informed node adjacent to v. x hears v in the first round, if and only if, it has no neighbors that are not informed. In the second round of the procedure, u transmits together with all neighbors of v that did not hear v in the previous round. The ambassador v flags an error (and does not send confirmation), if it does not hear u in the second round. ■

Procedure `Error-Detection` is called by algorithms `Dense-1` and `Dense-2` in any Δ-block B as soon as the following two conditions are satisfied: (1) confirmation from all Δ-blocks newly informed by B was obtained; (2) if B_1 is a Δ-block reachable from B and B_2 is a Δ-block reachable from B_1, then B_2 was informed.

Lemma 3. *Let A be either Algorithm* `Dense-1` *or* `Dense-2`*. If A fails to inform all nodes, then A does not provide confirmation to the source. If A is executed in a dense network, then A provides confirmation to the source.*

We propose two main algorithms: Algorithm `Universal Broadcast` that works for all networks, and Algorithm `Company-Aware Broadcast` that works for all networks of size at least 2. Algorithm `Universal Broadcast` consists of multiplexing Algorithm `Color&Transmit` with Algorithm `Dense-1`, while Algorithm `Company-Aware Broadcast` consists of multiplexing algorithms `Color&Transmit`, `Dense-1`, and `Dense-2`.

The running time of Algorithm `Universal Broadcast` is $O\left(D+g^2\right)$. Indeed, if confirmation is provided to the source by Algorithm `Dense-1`, the source can stop the execution of Algorithm `Color&Transmit`, in time $O(D)$, using the already elected ambassadors to spread the stopping message to the whole network. On the other hand, if confirmation is provided to the source by Algorithm `Color&Transmit`, the execution of Algorithm `Dense-1` can be stopped, again in time $O(D)$, using the already defined coloring to avoid collisions while spreading the stopping message from the source to the whole network. Algorithm `Universal Broadcast` is successful on any network, and its running time is bounded by the minimum between the running time of Algorithm `Color&Transmit` and Algorithm `Dense-1`, thus it is always $O\left(D+g^2\right)$. If the network contains only the source, the source will never receive any confirmation. Nevertheless, neither Algorithm `Color&Transmit` nor Algorithm `Dense-1` would use the source to transmit more than once in this case, thus allowing the combined algorithm to end in constant time. Hence we have the following result, which is proved optimal in Sect. 6.

Theorem 4. *There exists a deterministic algorithm that completes broadcast in time $O\left(D+g^2\right)$ in any UDG radio network of unknown diameter D and unknown density $d = 1/g$.*

If we neglect networks with only one node, Algorithm `Company-Aware Broadcast` can be used. Stopping slower component algorithms by the fastest is done as previously, guaranteeing time $O\left(\min\left\{D+g^2, D\log g\right\}\right)$. However, Algorithm `Company-Aware Broadcast` runs forever on a network containing the source only, hence it is not correct for all networks. Since the lower bound $\Omega\left(\min\left\{D+g^2, D\log g\right\}\right)$ from [1] also holds in our case, Algorithm `Company-Aware Broadcast` is optimal whenever it is correct. Hence we have the following result.

Theorem 5. *The optimal time of a broadcasting algorithm working correctly on all UDG radio networks with at least two nodes, unknown diameter D and unknown density $d = 1/g$, is $\Theta\left(\min\left(D + g^2, D \log g\right)\right)$.*

6 Lower Bound on Universal Broadcasting Time

We now establish optimal time of a broadcasting algorithm working correctly for *all* networks. In particular, this algorithm must stop after some fixed time when the source is the only node in the network. We show that this forces a lower bound $\Omega(g^2)$ on broadcasting time for some networks of constant diameter, and from there we derive the lower bound $\Omega(D + g^2)$ on broadcasting time for the class of UDG radio networks with unknown diameter D and unknown density d. This matches the time $O(D + g^2)$ of Algorithm `Universal Broadcast` from Sect. 5, thus establishing $\Theta(D + g^2)$ as optimal broadcasting time for the class of arbitrary networks.

Consider two squares: A with corners $(\frac{3}{5}, -\frac{1}{10}), (\frac{3}{5}, \frac{1}{10}), (\frac{4}{5}, -\frac{1}{10}), (\frac{4}{5}, \frac{1}{10})$ and B with corners $(\frac{6}{5}, -\frac{1}{10}), (\frac{6}{5}, \frac{1}{10}), (\frac{7}{5}, -\frac{1}{10}), (\frac{7}{5}, \frac{1}{10})$. For any positive constant d, consider a d-grid inside each of those squares, defined as the set of points in A (respectively, in B), including the lower-left corner and forming a grid with square side d. Let S be the d-grid in A, and T the d-grid in B. Note that, for any d, the size $\sigma(d)$ of both sets S and T is the same and it is $\Omega(g^2)$. For any $S' \subseteq S$ and any $T' \subseteq T$ define a UDG radio network $N(S', T')$ whose set of stations is $\{(0,0)\} \cup S' \cup T'$ and whose source is the point $(0,0)$. Let $\mathcal{N}$ be the class of all networks $N(S', T')$, for all $d = 1/g$, where g is an integer greater than 4, and for all $S' \subseteq S$ and $T' \subseteq T$. Note that networks of the class $\mathcal{N}$ have a simple structure: they are composed of two cliques, one on the set $\{(0,0)\} \cup S'$ and the other on the set T', with all possible edges between sets S' and T'. A network obtained for a given d has density d and diameter 2.

Theorem 6. *For any broadcasting algorithm $\mathcal{A}$ working correctly on all networks of the class $\mathcal{N}$, and for every $g > 4$, there exists a network of density $d = 1/g$ in the class $\mathcal{N}$, for which the algorithm $\mathcal{A}$ uses time $\Omega(g^2)$.*

It is easy to generalize the above lower bound to UDG radio networks of arbitrary diameter D and density d. Instead of networks of the class $\mathcal{N}$ that consist of a source $(0,0)$ and d-grids in squares A and B, we take the source $(0,0)$ followed by $D - 1$ squares of side $1/5$, arranged in a line with distances $3/5$ between consecutive square centers. In each of the squares we insert d-grids as before. It is easy to see that the $\Omega(g^2)$ lower bound can be proved as above, while the diameter D is a trivial lower bound on broadcasting time. Hence we get the following corollary, which holds even for error margin 0.

Corollary 1. *For any broadcasting algorithm $\mathcal{A}$ working correctly on all UDG radio networks, there exist networks of diameter D and density $d = 1/g$, for arbitrary D and g, for which algorithm $\mathcal{A}$ uses time $\Omega(D + g^2)$.*

Theorem 4 and Corollary 1 imply:

Corollary 2. *The optimal time of a broadcasting algorithm working correctly on all UDG radio networks with unknown diameter D and unknown density $d = 1/g$ is $\Theta(D + g^2)$.*

References

1. Emek, Y., Gasieniec, L., Kantor, E., Pelc, A., Peleg, D., Su, C.: Broadcasting in UDG radio networks with unknown topology. In: PODC, pp. 195–204 (2007)
2. Chlebus, B.S., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in ad hoc radio networks. Dist. Computing 15(1), 27–38 (2002)
3. Chlamtac, I., Kutten, S.: On broadcasting in radio networks - problem analysis and protocol design. IEEE Trans. on Communications 33, 1240–1246 (1985)
4. Chlamtac, I., Weinstein, O.: The wave expansion approach to broadcasting in multihop radio networks. IEEE Trans. on Communications 39, 426–433 (1991)
5. Gaber, I., Mansour, Y.: Centralized broadcast in multihop radio networks. J. Algorithms 46(1), 1–20 (2003)
6. Elkin, M., Kortsarz, G.: Improved schedule for radio broadcast. In: SODA, pp. 222–231 (2005)
7. Gasieniec, L., Peleg, D., Xin, Q.: Faster communication in known topology radio networks. In: PODC, pp. 129–137 (2005)
8. Kowalski, D.R., Pelc, A.: Optimal deterministic broadcasting in known topology radio networks. Dist. Computing 19(3), 185–195 (2007)
9. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A lower bound for radio broadcast. J. Comput. Syst. Sci. 43(2), 290–298 (1991)
10. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. J. Comput. Syst. Sci. 45(1), 104–126 (1992)
11. Bruschi, D., Pinto, M.D.: Lower bounds for the broadcast problem in mobile radio networks. Distrib. Comput. 10(3), 129–135 (1997)
12. Chlebus, B.S., Gasieniec, L., Östlin, A., Robson, J.M.: Deterministic radio broadcasting. In: ICALP, pp. 717–728 (2000)
13. Chrobak, M., Gasieniec, L., Rytter, W.: Fast broadcasting and gossiping in radio networks. In: FOCS, pp. 575–581 (2000)
14. Clementi, A.E.F., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: SODA, pp. 709–718 (2001)
15. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. In: FOCS, pp. 492–501 (2003)
16. Kowalski, D.R., Pelc, A.: Time complexity of radio broadcasting: adaptiveness vs. obliviousness and randomization vs. determinism. Theor. Comput. Sci. 333(3), 355–371 (2005)
17. Marco, G.D.: Distributed broadcast in unknown radio networks. In: SODA, pp. 208–217 (2008)
18. Kushilevitz, E., Mansour, Y.: An omega(log $(/d)$) lower bound for broadcast in radio networks. SIAM J. Comput. 27(3), 702–712 (1998)
19. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. J. Discrete Algorithms 5(1), 187–201 (2007)

20. Diks, K., Kranakis, E., Krizanc, D., Pelc, A.: The impact of information on broadcasting time in linear radio networks. Theor. Comput. Sci. 287(2), 449–471 (2002)
21. Sen, A., Huson, M.L.: A new model for scheduling packet radio networks. In: INFOCOM, pp. 1116–1124 (1996)
22. Moscibroda, T., Wattenhofer, R.: Maximal independent sets in radio networks. In: PODC, pp. 148–157 (2005)
23. Moscibroda, T., Wattenhofer, R.: Coloring unstructured radio networks. In: SPAA, pp. 39–48 (2005)
24. Kowalski, D.R., Pelc, A.: Time of deterministic broadcasting in radio networks with local knowledge. SIAM J. Comput. 33(4), 870–891 (2004)

Efficient Broadcasting in Known Geometric Radio Networks with Non-uniform Ranges*

Leszek Gąsieniec[1], Dariusz R. Kowalski[1], Andrzej Lingas[2], and Martin Wahlen[2]

[1] Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
{L.A.Gasieniec,D.Kowalski}@liverpool.ac.uk
[2] Department of Computer Science, Lund University, 22100 Lund, Sweden
{Andrzej.Lingas,Martin.Wahlen}@cs.lth.se

Abstract. We study here deterministic broadcasting in *geometric radio networks* (GRN) whose nodes have complete knowledge of the network. Nodes of a GRN are deployed in the Euclidean plane (R^2) and each of them can transmit within some range r assigned to it. We adopt model in which ranges of nodes are non-uniform and they are drawn from the predefined interval $0 \leq r_{min} \leq r_{max}$. All our results are in the *conflict-embodied model* where a receiving node must be in the range of exactly one transmitting node in order to receive the message.

We derive several lower and upper bounds on the time of deterministic broadcasting in GRNs in terms of the number of nodes n, a distribution of nodes ranges, and the eccentricity D of the source node (i.e., the maximum length of a shortest directed path from the source node to another node in the network). In particular:

(1) We show that $D + \Omega(\log(n - D))$ rounds are required to accomplish broadcasting in some GRN where each node has the transmission range set either to 1 or to 0. We also prove that the bound $D + \Omega(\log(n - D))$ is almost tight providing a broadcasting procedure that works in this type of GRN in time $D + O(\log n)$.

(2) In GRNs with a wider choice of positive node ranges from $r_{min}, ..., r_{max}$, we show that broadcasting requires $D + \Omega(\min\{\log \frac{r_{max}}{r_{min}}, \log(n - D)\})$ rounds and that it can be accomplished in $O(D \log^2 \frac{r_{max}}{r_{min}})$ rounds subsuming the best currently known upper bound $O(D(\frac{r_{max}}{r_{min}})^4)$ provided in [15].

(3) We also study the problem of simulation of minimum energy broadcasting in arbitrary GRNs. We show that energy optimal broadcasting that can be completed in h rounds in a *conflict-free model* may require up to $h/2$ additional rounds in the conflict-embodied model. This lower bound should be seen as a separation result between conflict-free and conflict-embodied geometric radio networks. Finally, we also prove that any h-hop broadcasting algorithm with the energy consumption $\mathcal{E}$ in a GRN can be simulated within $O(h \log \psi)$ rounds in the conflict-embodied model using energy $O(\mathcal{E})$, where ψ is the ratio between the largest and the shortest Euclidean distance between a pair of nodes in the network.

* Research supported in part by VR grant 621-2005-4085 and The Royal Society International Joint Project, IJP - 2006/R2.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 274–288, 2008.

1 Introduction

Wireless networks, including general radio and more specialized sensor networks, are amongst the most popular and challenging types of multi-hop networks [33]. They consist of a system of nodes without any wired infrastructure. Each node is assigned its *transmission range*, i.e., it can transmit solely to the nodes located within its transmission range. The range of a node depends on the energy supplied to it. We shall assume here the standard geometric setting where the nodes correspond to points in the Euclidean plane and the range of a node is bounded by the α^{th} root of the energy assigned to it, where $\alpha \geq 2$ is a given constant. In synchronous networks, considered here, network nodes communicate in synchronous *rounds*, also known as *time steps*. In each round, every node acts either as a transmitter or as a receiver, i.e., nodes cannot transmit and receive messages simultaneously. A node acting as a transmitter sends a message to all its neighbors, i.e., all nodes within its transmission range. Note here that the set of message recipients cannot be chosen arbitrarily. Also the message sent by a node v and addressed (as one of the co-recipients) to a node w is delivered reliably only if v is a unique transmitting neighbor of w during the current round. Otherwise, due to the potential *collisions* between several messages the node w may receive no message during this round. We consider (and compare our results in) two previously studied GRN models. In the first *conflict-free* model used extensively, e.g., in construction of energy efficient broadcasting trees, see [3,11], one assumes that collisions caused by simultaneous transmissions do not affect successful message deliveries. The second (more realistic) *conflict-embodied* model was proposed in [7] and later widely used in the context of arbitrary unknown [4,5,8,9,10,13,14,30] and known [1,16,24,28] radio networks. Recent developments in the field include work on energy efficient broadcasting [23] as well as broadcasting in unknown GRNs [17,18,20] with limited density of nodes. Other related problems considered in the context of GRNs include computing maximal independent sets [31] and asynchronous wake up [32]. An interesting study on communication in quasi unit disk graphs that motivated further work on GRNs with non-uniform ranges can be found in [6,29].

One of the most commonly studied communication problems in networks is *broadcasting*, i.e., the problem of distributing a broadcast message from a distinguished source node to all remaining nodes. The broadcasting problem in geometric ad-hoc wireless networks has been studied in the conflict-embodied model in a number of papers including [36,37]. Scheduling of an optimal broadcasting is known to be NP-hard even for geometric networks [37]. The authors of [15] presented several lower and upper bounds on the number of rounds necessary to accomplish broadcasting in geometric ad-hoc wireless networks in terms of the number of nodes, the eccentricity of the source node, and the so called radius of knowledge.

Ad hoc wireless networks are usually powered by very limited electricity resources, e.g., batteries. Therefore low power consumption has become a crucial issue in the design of communication algorithms for these networks. In particular, the problem of *minimum-energy broadcasting* in ad hoc wireless networks has gained a lot of attention recently [2, 11, 12, 19, 27, 34]. It consists in finding an assignment of the energy to the nodes of the network such that (1) the graph induced by the resulting node ranges includes a spanning tree rooted at the source node containing the message to broadcast; (2) the total energy assigned is minimized. Note that if the depth of the spanning tree is bounded by h and it is not required that a receiving node is in the range of exactly one transmitting node then the message from the source node can reach all nodes in h rounds (hops). A special variant of the minimum energy broadcasting problem is *minimum energy h-hop broadcasting* where the spanning tree is additionally required to be of depth at most h.

The problem of minimum energy broadcasting is known to be NP-hard both in its general graph version [22] and in its geometric version [11]. Geometric minimum energy h-hop broadcasting is known to be solvable for $h = 2$ in polynomial time and to admit a polynomial-time approximation scheme for any fixed $h > 2$ [3]. Recently, several further algorithmic, approximation and complexity results on the problem of minimum energy broadcasting and multicasting in geometric ad-hoc wireless networks have been reported [2,3,11,12,19,27,34,35]. Surprisingly, all these results are typically obtained under the simplified transmission-reception model discarding transmission-reception conflicts, i.e., in the conflict-free model. This situation contrasts with the literature on minimum time broadcasting and multi-casting where the aforementioned conflicts are very much taken into account by using the conflict-embodied model, see [1,7,8,15,17,23,24,28].

1.1 Our Contribution

In the first part of our paper, we derive lower and upper bounds on the time of deterministic broadcasting in GRN in the conflict-embodied model in terms of the number of nodes n and the eccentricity D of the source node (i.e., the maximum length of a shortest directed path from the source node to another node in the network). In particular, we show that $D + \Omega(\log(n - D))$ rounds are required to accomplish broadcasting in some GRNs where each node has either range 1 (with full routing capacity) or range 0 (understood as terminal nodes). We complement this result showing that broadcasting in such GRNs can be always implemented in time $D + O(\log n)$ rounds. Furthermore, we prove that in a GRN with positive ranges drawn from the interval $[r_{min}, r_{max}]$, broadcasting requires time $D + \Omega(\min\{\log \frac{r_{max}}{r_{min}}, \log n\})$, and can be accomplished in $O(D \log^2 \frac{r_{max}}{r_{min}})$ steps which improves the best previously known upper bound $O(D(\frac{r_{max}}{r_{min}})^4)$, due to Dessmark and Pelc [15].

In the second part, we study the problem of simulation of minimum energy broadcasting in GRNs in the conflict-embodied model. In particular, we show that an energy optimal h-hop broadcasting in a GRN may require up to $h/2$ additional rounds in this model. We also show that any h-hop broadcasting in a GRN using energy $\mathcal{E}$ can be simulated within $O(h \log \psi)$ rounds in the conflict-embodied model using energy $O(\mathcal{E})$, where ψ is the ratio between the largest and the shortest distance between a pair of nodes in the network.

2 New Results on Broadcasting in GRN with Non-uniform Ranges

2.1 The Lower Bound

We say that a node with the transmission range r located at position $(x, y) \in R^2$ defines a *disk* of *radius* r and the *center* (x, y). Let $\mathcal{D}_1$, $\mathcal{D}_2$, ...,$\mathcal{D}_k$ be a sequence of disks of radii 1 such that the center of disk $\mathcal{D}_i$ is at $(\frac{(i-1)}{2k},\ 0)$.

Lemma 1. *For a fixed k and any pair $i, j \in \{1, 2, ..., k\}$, s.t., $j > i+1$, the subsequence $\mathcal{D}_{i+1}, ..., \mathcal{D}_{j-1}$ has a non-empty intersection above the X-axis outside all the remaining disks.*

Proof. Note that $\mathcal{D}_{i+1}$ and $\mathcal{D}_{j-1}$ have a non-empty intersection I above the X-axis outside $\mathcal{D}_i$ and $\mathcal{D}_j$, hence outside of all $\mathcal{D}_q$, where $q \notin \{i+1, ..., j-1\}$, see Figure 1. It is sufficient to observe that I must be included in any $\mathcal{D}_{q'}$, for $q' \in \{i+2, ..., j-2\}$. □

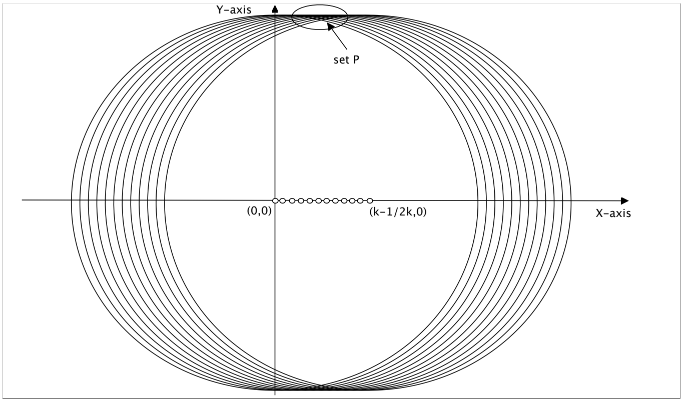


Fig. 1. Disks $\mathcal{D}_1$, $\mathcal{D}_2$, ...,$\mathcal{D}_k$

Consider again the sequence $\mathcal{D}_1$, $\mathcal{D}_2$, ...,$\mathcal{D}_k$. In each face of the arrangement defined by perimeters of exactly four disks and located above the X-axis insert a single point. Let P be the set of the inserted points. Note that $|P| = \Theta(k^2)$.

Lemma 2. *Assume that the centers of the disks $\mathcal{D}_1$, $\mathcal{D}_2$, ...,$\mathcal{D}_k$ are already informed, i.e., they contain the source message. Assume also that none of the points in P is informed yet and the range of each point in P is set to zero. In this case distribution of the source message to all points in P requires at least $\log_3 k$ rounds.*

Proof. We prove by induction on $j - i$, where $i,\ j \in \{1, ..., k\}$ and $i < j$, that at least $\log_3(j - i + 1)$ conflict-free rounds are required to distribute the message from the centers of the disks $\mathcal{D}_i, ..., \mathcal{D}_j$, to the points in P outside the disks $\mathcal{D}_q$, for all $q \notin \{i, ..., j\}$. Let $P_{i,j}$ be the set of all such points. The inductive assumption is true for $j - i \leq 2$. Suppose $j - i > 2$. By Lemma 1 and the definition of $P_{i,j}$, there exists a point in $P_{i,j}$ that belongs to the intersection of the disks $\mathcal{D}_i, ..., \mathcal{D}_j$. Hence, there must be a round in which only one of the centers of the disks $\mathcal{D}_i, ..., \mathcal{D}_j$, say of the disk $\mathcal{D}_{j'}$, transmits. We may assume, w.l.o.g., that $j - j' \geq j' - i$. It follows that the subsequence $\mathcal{D}_{j'+1}, ..., \mathcal{D}_j$ includes at least $(j - i + 1)/3$ disks. Recall that $P_{j'+1,j}$ denotes the subset of $P_{i,j}$ covered by the disks $\mathcal{D}_{j'+1}, ..., \mathcal{D}_j$. By the inductive assumption, at least $\log_3(j - j' + 1)$ rounds are needed to inform the points in $P_{j'+1,j}$. Hence, totally $1 + \log_3(j - j' + 1) \geq 1 + \log_3((j - i + 1)/3) \geq \log_3(j - i + 1)$ rounds are necessary to inform $P_{i,j}$. □

Theorem 1. *For any integers $n > D + 1 \geq 0$, there exists a GRN with n nodes, the (source node) eccentricity D and the node ranges in $\{0, 1\}$, in which broadcasting requires $D + \Omega(\log(n - D))$ rounds in the conflict-embodied model.*

Proof. Let $k = \lfloor \sqrt{n - D + 1}/\sqrt{6} \rfloor$. The network comprises k nodes (with unit transmission ranges forming disks $\mathcal{D}_1$,..., $\mathcal{D}_k$) located on the X-axis in points $p_i = (\frac{i-1}{2k}, 0)$, for $i = 1, .., k$. This is accompanied by $\Theta(k^2)$ nodes (with ranges set to 0) located in the set P and some extra $D - 1$ nodes (with unit ranges) located on the vertical line $x = \frac{k-1}{4k}$ such that the top point on this line is below X-axis and exactly at distance 1 from both p_1 and p_k. The remaining $D - 2$ points are placed one by one, separated by the unit distance, towards the lower end of the vertical line. The lowest point on the line acts as the source node. Note that the eccentricity of the network is D.

By the Euler's formula for planar graphs (expressing the relation $n - m + f = 2$, where n refers to the number of vertices, m is the number of edges and f is the number of faces in a planar embedding of the graph) the cardinality of P can be bounded by $6k^2$ since two unit disks can intersect in at most two points (in order to generate exactly n nodes, we may have to duplicate some of the nodes in P and move them slightly apart). By the construction, the centers $p_1, .., p_k$ of the disks $\mathcal{D}_1$,..., $\mathcal{D}_k$ are at the distance $D - 1$ from the source node and they require at least $D - 1$ conflict-free rounds to be informed. Also when the top node on the vertical line transmits it informs all centers $p_1, .., p_k$. Since all nodes in P are out of reach for the top node on the vertical line after this node transmits, by Lemma 2, at least $\log_3 k$ additional rounds are required to inform the points in P. □

Further we generalize the lower bounds from Lemma 2 and Theorem 1 to include the case where all node ranges are positive. We adopt a configuration composed of the set P and the collection of k disks $\mathcal{D}_1$,..., $\mathcal{D}_k$. In this configuration we replace the unit disks $\mathcal{D}_i$ by their k large counterparts with the radius r_{max} and the nodes with the range zero in P by $\Theta(k^2)$ small disks with the radius $r_{min} > 0$. We also prune the location of large and small disks so that each small disk is entirely contained in face of arrangements formed by perimeters of exactly four large disks.

Theorem 2. *There is a constant $c > 0$ such that for any integers $D \geq 4$ and $n \geq D - 2 + (2\frac{r_{min}}{cr_{max}} - \frac{{r_{min}}^2}{(cr_{max})^2})^{-1}$, where $r_{max} > r_{min} > 0$, there exists a GRN with n nodes, the (source node) eccentricity D and the node ranges in $[r_{min}, r_{max}]$, in which broadcasting requires $D + \Omega(\min\{\log \frac{r_{max}}{r_{min}}, \log(n - D)\})$ rounds in the conflict-embodied model.*

Proof. We may assume, w.l.o.g., that $r_{max} = 1$ (radii of disks can be scaled down to achieve this assumption). We first prove the lower bound $\Omega(\min\{\log \frac{r_{max}}{r_{min}}, \log(n -$

$D)\})$. Consider the sequence $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k$ of disks of the radius r_{max} as defined in Lemma 1, where k is to be set later. In the arrangement small disks can be fit into faces formed by perimeters of four different large disks of the radius $r = r_{max}$. By an immediate trigonometric argument one can argue that the radius of small disks is at least as large as $cr_{max}(1 - \sqrt{1 - \frac{1}{k^2}})$, for some positive constant c.

Now we solve the equation $r_{min} = cr_{max}(1 - \sqrt{1 - \frac{1}{k^2}})$, and set k to the largest integer not exceeding the minimum of $\sqrt{n - D + 2}$ and the value of the solution, both divided by $\sqrt{6}$, i.e.,

$$k = \lfloor \min\{\sqrt{n - D + 2}, (2\frac{r_{min}}{cr_{max}} - \frac{{r_{min}}^2}{c^2 r_{max}^2})^{-\frac{1}{2}}\}/\sqrt{6}\rfloor.$$

Further, we assume that the centers of the disks $\mathcal{D}_1$, $\mathcal{D}_2$, ...,$\mathcal{D}_k$ are informed nodes of range $r = r_{max}$. Note that by the definition of k, we can place in each of the at most $6k^2$ faces of the arrangement formed by the intersections of four disks (over the horizontal line induced by the centers of the disks), a node of range r_{min} so the whole disk of radii r_{min} around it is entirely contained in the face. Analogously to the proof of Lemma 1 one can show that at least $\log_3 k$ conflict-free rounds are required to inform the nodes with the smaller range r_{min}. Also by the specification of n and k and straightforward calculations, we obtain $\log_3 k = \Omega(\min\{\log \frac{r_{max}}{r_{min}}, \log(n - D)\})$.

Using an analogous argument as in the proof of Theorem 1 we extend the lower bound $\Omega(\min\{\log \frac{r_{max}}{r_{min}}, \log(n - D)\})$ to $D + \Omega(\min\{\log \frac{r_{max}}{r_{min}}, \log(n - D)\})$ for networks with the eccentricity D. □

2.2 The Upper Bounds

In this section we present broadcasting algorithms for GRNs with two types of ranges. First we consider the (extreme) model with ranges of size 0 and 1. Later we provide an efficient solution to broadcasting in the model with arbitrary non-uniform ranges.

The main idea of the first algorithm is to split the broadcasting process into two stages. During the first stage we focus on the communication in a sub-network with bidirectional edges, i.e., based on nodes with transmission ranges set to 1. We show that our algorithm accomplishes broadcasting in this sub-network in optimal (in view of the lower bound) time $D+O(\log n)$. Note that nodes with the transmission ranges set to 0 take only passive part in the broadcasting process, i.e., the source message is routed only at nodes with the transmission range set to 1. Thus after the first stage each node with the transmission range 0 is within the neighborhood of some (and possibly many) informed node with the transmission range 1. Thus in the second stage we focus solely on transmissions towards the nodes with ranges set to 0. We show that this process lasts $O(\log n)$ steps.

We start presentation of our broadcasting procedure from its second stage due to the similarity of the reasoning used in this stage to the proof of the lower bound from the previous section. Note, that if the informed nodes (with the transmission range 1) are placed on a horizontal line there is simple logarithmic (in a number of uninformed nodes) broadcasting mechanism. And indeed, if there is an informed node that covers a constant fraction of the uninformed nodes we perform broadcasting from this node reducing the number of uninformed nodes by a constant fraction in a single round. Otherwise, we can always pick some central informed node v whose range, and in particular, an independent transmission from it separates uninformed targets belonging to ranges to the left and to the right of v in a balanced way. In other words, the numbers of uninformed nodes in ranges located to the left and to the right of v differ only be a constant fraction. Thus after the transmission from v further communication to the left of v and to its right can be performed independently in parallel. We show that similar reasoning can be also applied in more general 2-dimensional case reflecting our needs.

For a positive real c, we denote by $M(c)$ a rectilinear grid which divides the 2-dimensional space into square shaped regions, called *cells*, of size $c \times c$. The point $(0, 0)$is shared by corners of four adjacent cells with indices $[0, 0], [-1, 0], [0, -1]$ and $[-1, -1]$. We also consider an *ordinary partition* $P(c) = P(c, 1) \cup P(c, 2) \cup ..P(c, 25)$ of cells in $M(c)$ where two cells of $M(c)$ belong to the same $P(c, i)$, for some $i = 1, .., 25$, if and only if the indices of their rows and columns in $M(c)$ are equal modulo 5 respectively.

Lemma 3. *The informed nodes in any cell A of the rectilinear grid $M(\frac{1}{\sqrt{2}})$ can inform all uninformed nodes within their unit ranges in time $O(\log n)$.*

Proof. Let $K = \{v_1, ..., v_k\}$ be the set of informed nodes in A. Note that all other nodes in A can be informed by any node from K instantly. Thus we must consider only the 20 cells (addressed later as B cells) around A whose nodes are potentially reachable directly from the set K. We assume, w.l.o.g., that $k \geq 3$ (otherwise two communication rounds suffice) and K forms a *minimal covering set* ([25]) defined as for each $v_i \in K$ there is an uninformed node u_i which is in the range of v_i and outside the ranges of the other nodes in K. Note that K can be always pruned greedily to possess the minimal covering set property. Further, it is also convenient to assume, w.l.o.g., that for $i < j$ the X-coordinate of u_i is not greater than that of u_j.

For $i = 1, ..., k$, let B_i be the region of B which is within the range of v_i and outside the ranges of the remaining vertices in K. Due to the property of unit disk graphs no range of any informer in K can split the range of another informer in two or more parts within any cell B from the neighborhood of A. In consequence, the regions B_i are one-connected and have the same horizontal order as the nodes u_i.

Now our broadcasting procedure is as in the aforementioned linear case. If there is a node v_i in K whose range contains at least one fourth of the uninformed nodes in B then we perform broadcasting from this node and remove it from K. Otherwise, we can always find a node v_j such that there are at least one fourth of uninformed nodes in B on one side of the range of v_j as well as on its other side. Thus indeed we can first perform broadcasting from v_j and later apply our procedure recursively in parallel to inform independently the nodes to the left and to the right of the range of v_j. □

Lemma 4. *Let G be a known GRN with n nodes equipped with ranges 0 or 1, where all nodes with the range 1 are already informed. The not yet informed nodes with the range 0 can be informed in $O(\log n)$ rounds.*

Proof. Consider the rectilinear grid $M(\frac{1}{\sqrt{2}})$ and its ordinary partition $P(\frac{1}{\sqrt{2}})$. For any pair of nodes located in two different cells of any $P(\frac{1}{\sqrt{2}}, i)$ their ranges are disjoint.

Therefore, we can apply Lemma 3 first to all cells in $P(\frac{1}{\sqrt{2}}, 1)$, further to all cells in $P(\frac{1}{\sqrt{2}}, 2)$, etc, to inform all not yet informed nodes in G. The whole process takes time $25 \cdot O(\log n) = O(\log n)$ and the thesis of the lemma follows. □

This completes the description and analysis of the second stage in the broadcasting procedure. In what follows we provide an optimal solution to the broadcasting task considered in the first stage.

Lemma 5. *Let G be a known GRN with n nodes equipped with ranges 0 or 1 and the (source) eccentricity D. All nodes of range 1 can be informed in $D + O(\log n)$ rounds.*

Proof. Consider the maximal subgraph H of a given GRN network G induced by nodes with the transmission range 1. The subgraph H forms a *unit disk graph* in which all edges in H are bidirectional. Let T be a *gathering-broadcasting spanning tree* of H as defined in [24], that is, T is a BFS spanning tree of H rooted at the source node and satisfying the following properties:

(1) The nodes of T are ranked. All leaves obtain rank 1. Consider any internal node $v \in T$ where the ranks of all its children are already established and ρ is the maximal rank among its children. The rank of v is set to ρ if ρ is attributed to a unique child of v, otherwise (there is more than one child with the rank ρ) the node v obtains rank $\rho + 1$. (Note that the largest rank in a tree of size n is bounded by $\log n$ [21]).

(2) For any four nodes v, w, v' and w' with the same rank in T such that v and w are located at the same BFS level and v and w are parents of v' and w' respectively, there are no *crossing edges* (v, w') and (w, v') in G. (In other words, simultaneous radio transmissions from v and w cause no conflict at v' and at w', and vice versa).

A gathering-broadcasting spanning tree can be constructed in time polynomial in the number of nodes in the graph, for details see [24].

The second important component of our radio broadcasting algorithm is a procedure that manages to propagate the broadcast message from one side of a bipartite unit disk graph (UDG for short) to another in constant time. More precisely, let $H' = (V_1, V_2, \mathcal{E}')$ be a bipartite subgraph of H, where sets V_1 and V_2 form a partition of nodes in H'. Assume that all nodes in V_1 are already informed (possess the source message). The goal is to inform all nodes in V_2 in constant time using radio transmissions along the edges in $\mathcal{E}'$. A procedure with required property can be found, e.g., in [15] (version for UDGs). We refer to it as BiB(H'). Finally β stands for the (constant) upper bound on the number of rounds required by the procedure BiB over all possible bipartite subgraphs H' of H.

We are now ready to present a broadcasting schedule for the whole graph H. Each node transmits at most $\beta + 1$ times. The rounds when a node v transmits are defined as follows. If v is in the lowest layer of T then it does not transmit at all. Otherwise, let L_i be the layer at which v is located in T, $\rho = \rho_v$ be its rank in T, and $H_{i,\rho}$ be the subgraph of H induced by nodes in L_i with the rank ρ and their neighbors in L_{i+1} with ranks smaller than ρ. Let $\xi_v \subseteq \{1, \ldots, \beta\}$ be the set of rounds in which node v transmits during execution of the procedure BiB$(H_{i,\rho})$. This execution is considered only for the purpose of computing sets ξ and it does not form a part of the constructed broadcasting schedule. The actual set of rounds in the broadcast schedule during which the node v transmits is defined as follows

$$\left\{(i+1)+3\cdot[(\beta+1)\cdot(\log n-\rho)+x] : x\in\{0\}\cup\xi_v\right\}. \qquad (1)$$

The sets of transmission rounds satisfy two important properties:

(a) if during a round two nodes in the same layer transmit simultaneously, they must have the same rank. This property is enforced by the product $(\beta+1)(\log n-\rho)$ in Equation 1;

(b) if during a round two nodes from different layers transmit simultaneously, they must be located in layers at distance at least 3. This property is guaranteed by the presence of factor 3 in Equation 1.

It remains to prove, by induction on the layer number i, that each node w in layer L_i receives the broadcast message prior to its first transmission round, that is, before round $(i+1)+3\cdot[(\beta+1)\cdot(\log n-\rho_w)]$.

A node in the first layer clearly receives the message in round $1 < 2+3\cdot[(\beta+1)\cdot(\log n-\rho)]$. Assume now that all nodes in layers L_i, for $1\le i<D$, receive the broadcast message prior to their first transmission rounds. We show that the same property holds for all nodes in L_{i+1}. Consider a node $w\in L_{i+1}$ and one of its neighbors $v\in L_i$ of rank $\rho=\rho_v\ge\rho_w$ (there must be at least one such node v, e.g., the parent of w in the gathering-broadcasting spanning tree). By the inductive assumption, all nodes in L_i of rank ρ receive the broadcast message before round $(i+1)+3\cdot[(\beta+1)\cdot(\log n-\rho)]$.

In case $\rho_v>\rho_w$ consider the set of transmission rounds defined by BiB$(H_{i,\rho})$. In this case the node w receives the broadcast message from one of its neighbors v with rank ρ in layer L_i in graph H in some round $(i+1)+3\cdot[(\beta+1)\cdot(\log n-\rho)+x]$ based on the layer index i, the rank ρ and some $1\le x\le\beta$. There are no conflicts caused by transmissions coming from other nodes in L_i with different ranks (property (a)). Also transmissions from other layers are too distant (property (b)). Thus the node w must be informed by some of its neighbors v at level L_i with rank ρ due to broadcasting ability of the procedure BiB. Finally, by the inequality $\rho=\rho_v>\rho_w$, we obtain $(i+1)+3\cdot[(\beta+1)\cdot(\log n-\rho)+x]<(i+2)+3\cdot[(\beta+1)\cdot(\log n-\rho_w)]$ for any $0\le x\le\beta$. Otherwise (i.e., $\rho=\rho_v=\rho_w$), and this happens when v is the parent of w, the node w receives the broadcast message in round $(i+1)+3\cdot[(\beta+1)\cdot(\log n-\rho)]$, since during this time every node in L_i with this rank transmits for the first time. Note that there are no conflicts at w, i.e., the broadcast message reaches w safely. This is due to the fact that at level L_i only nodes with rank ρ transmit during this time (property (a)) and there are no crossing edges outside of the tree property (1)). Moreover the transmissions at other levels do not interfere since these layers must be at distance 3 or larger (property (b)). The inductive proof is completed. The thesis of the theorem follows from

$$(i+1)+3\cdot[(\beta+1)\cdot(\log n-\rho)]=D+O(\log n). \qquad \square$$

By combining Lemmas 4 and 5, we obtain:

Theorem 3. *Let G be a known GRN with n nodes equipped with ranges 0 or 1. Broadcasting in G can be accomplished in $D + O(\log n)$ rounds in the conflict-embodied model.*

We are ready to prove that there exists an efficient broadcasting algorithm for GRNs with arbitrary non-uniform ranges that requires $O(D \log^2 \frac{r_{max}}{r_{min}})$ rounds, where the ranges of nodes are drawn from the segment $[r_{min}, r_{max}]$.

Theorem 4. *Let G be a known GRN with n nodes equipped with ranges taken from the segment $[r_{min}, r_{max}]$ and the eccentricity D. Broadcasting in G can be accomplished in $O(D \log^2 \frac{r_{max}}{r_{min}})$ rounds in the conflict-embodied model.*

Proof. The broadcast schedule proceeds in D phases that correspond to D layers in the network defined with respect to the source node in the broadcasting process. Each phase h, for $h = 0, .., D-1$, consists of two parts (1) and (2). During part (1) the informers from the layer h transmit to their neighbours in the layer $h+1$. On the conclusion of this part, for each cell of $M(\frac{r_{min}}{\sqrt{2}})$ containing nodes from the layer $h+1$, at least one node gets the broadcast message. During part (2) all other nodes in the layer $h+1$ receive the broadcast message.

During part (1) in each cell of $M(\frac{r_{min}}{\sqrt{2}})$ containing nodes from the BFS level $h+1$ we determine one (arbitrary) node called the leader of the cell. We also consider another grid $M(\frac{r_{max}}{\sqrt{2}})$ and the associated ordinary partitions $P(\frac{r_{min}}{\sqrt{2}})$ and $P(\frac{r_{max}}{\sqrt{2}})$. Recall that transmissions coming from different cells in any set $P(\frac{r_{max}}{\sqrt{2}}, i)$ do not conflict one another. Part (1) proceeds in 25 stages where during stage i informed nodes in $P(\frac{r_{max}}{\sqrt{2}}, i)$ transmit the broadcast message to the leaders in all reachable cells in $M(\frac{r_{min}}{\sqrt{2}})$. Note that nodes in each cell A belonging to $P(\frac{r_{max}}{\sqrt{2}}, i)$ can reach leaders in at most $O((\frac{r_{max}}{r_{min}})^2)$ cells of $M(\frac{r_{min}}{\sqrt{2}})$. It follows that one can select (via minimal covering set, see the proof of Lemma 3) at most $O((\frac{r_{max}}{r_{min}})^2)$ nodes in A that are directly connected with the leaders of cells in $M(\frac{r_{min}}{\sqrt{2}})$. Thus part (1) reduces to broadcasting in a bipartite graph of size $O((\frac{r_{max}}{r_{min}})^2)$. It is well known, see [8], that this can be done in an arbitrary bipartite n-node graph in time $O(\log^2 n)$, thus in our case in time $O(\log^2 \frac{r_{max}}{r_{min}})$.

During part (2) all yet uninformed nodes in the layer $h+1$ receive the broadcast message. Part (2) proceeds in $\log \frac{r_{max}}{r_{min}}$ stages. The stage $j = 0, .., \log \frac{r_{max}}{r_{min}} - 1$ uses the grid $M(\frac{2^j r_{min}}{\sqrt{2}})$ and the associated ordinary partition $P(\frac{2^j r_{min}}{\sqrt{2}})$. In each cell A of $M(\frac{2^j r_{min}}{\sqrt{2}})$ we get a *superleader* among the leaders (with ranges in $[2^j r_{min}, 2^{j+1} r_{min}]$) in all cells of $M(\frac{r_{min}}{\sqrt{2}})$ that are included in A. In consecutive steps $i = 1, ..., 25$ the superleader of any cell A of $M(\frac{2^j r_{min}}{\sqrt{2}})$ included in $P(\frac{2^j r_{min}}{\sqrt{2}}, i)$ transmits to all uninformed nodes in A. These transmissions are conflict free due to the structure of the ordinary partition $P(\frac{2^j r_{min}}{\sqrt{2}})$ and sufficient due to the size of ranges of the superleaders. Since each stage finishes in 25 steps, the whole Part (2) is executed in time $O(\log \frac{r_{max}}{r_{min}})$. □

3 Time Bounds on Energy Efficient Broadcasting in GRN

In this section we consider energy efficient broadcasting where the energy used by a transmitter is proportional to its range raised to the power $\alpha \geq 2$. We consider and compare our results in the *conflict-free* and *conflict-embodied* geometric radio networks. It is assumed that in both models during single execution of a minimum energy $h-$hop broadcasting procedure each node of GRN transmits at most once (i.e., the range of each node contributes to the total energy consumption at most once).

3.1 The Lower Bounds

We show that for any hop number $h > 1$, there is a GRN for which an energy optimal h-hop broadcasting requires at least $h/2$ *additional* rounds.

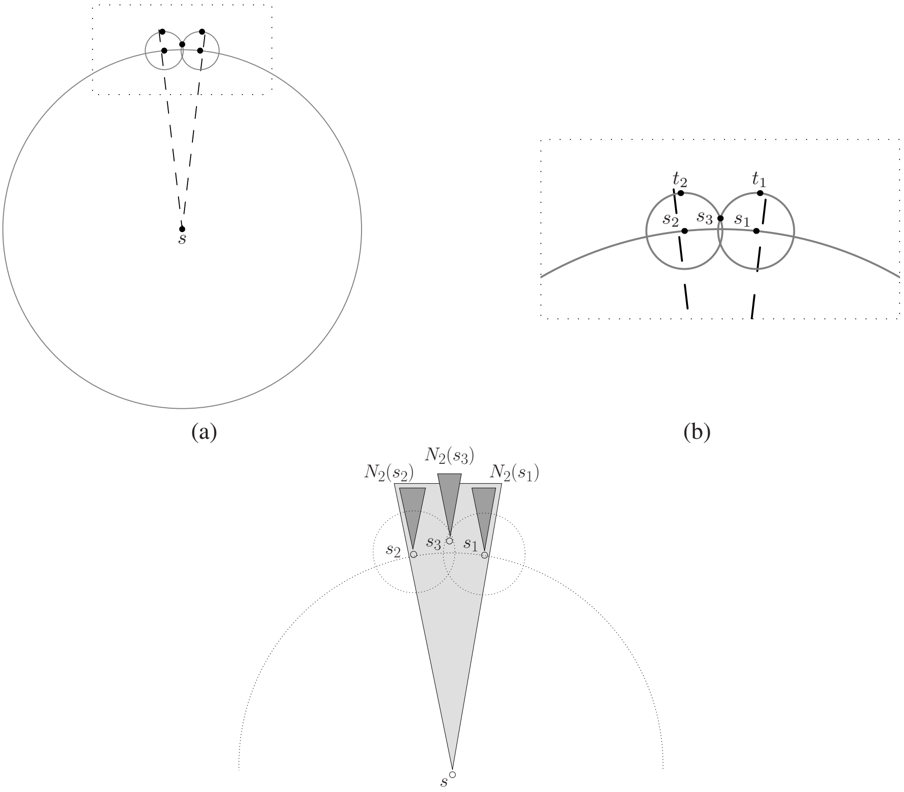
Theorem 5. *There is a sequence of GRNs $\{N_h\}_{h>1}$, for which energy optimal h-hop broadcasting in the conflict-free model requires $1.5h$ rounds in the conflict-embodied model.*

Proof. Assume first that $h = 2$. Apart from the source point s, consider two additional points s_1 and s_2 located above s on the perimeter of the unit disk $\mathcal{D}$ centered at s. More precisely the points s_1 and s_2 are symmetrically located on two sides of the Y-axis that goes through the source point s, where the distance between s_1 and s_2 is 0.2. Let $\delta = 0.106$. Also let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two disks of radius δ centered at s_1 and s_2, respectively. In each of regions $\mathcal{D}_1 \cap \mathcal{D}_2$ and $\mathcal{D}_1 \setminus \mathcal{D}_2, \mathcal{D}_2 \setminus \mathcal{D}_1$ insert a point as far as possible from the source s. The new inserted points in the three regions are named s_3 and t_1, t_2 respectively, see Figures 2(a) and 2(b). The configuration of six points has the following three properties: (1) it has an axis of symmetry that contains the segment (s, s_3); (2) all points belong to a $\pi/12$-angular slice of a disc with the radius $1 + \delta$, centered at s; (3) all points (excepts the source itself) are at distance at least one from s. In this configuration there is exactly one range assignment that leads to energy optimal two-hop broadcasting. One can prove that the range of s must be set to 1 and the ranges of s_1 and s_2 must be set to δ, where the total energy consumption is $1 + 2\delta^\alpha$. Because of the conflict in the point s_3 the two-hop broadcasting requires three rounds in the conflict-embodied model, i.e., we need two separate rounds for transmissions from s_1 and s_2. This proves the theorem for $h = 2$.

We now generalize the idea of the proof for the 6-point configuration to encompass all cases with $h > 2$. In the recursive construction, we use the following invariant on h.

The invariant: *All points lay symmetrically above the source on two sides of the Y-axis containing the source. All points belong to a $\pi/12$-angular slice of a disc with the radius $\frac{1-\delta^h}{1-\delta}$, centered at s and they (apart from s) are located at distance at least 1 from the source s. Moreover the points require at least $1.5h$ rounds in conflict-embodied model to simulate h-hop broadcasting with the minimum energy consumption in the conflict-free model.*

Let N_i denote such a configuration for $2 \leq i < h$ hops where for the completeness of the proof a network N_1 comprises a node s with the transmission range 1 and a node t located on the same X-axis as s at distance 1 from s. Recall that N_2 satisfies the



(a)

(b)

(c)

Fig. 2. The sequence of networks N_h: $h = 2$ (a,b) and $h > 2$ (c)

invariant for $h = 2$. We start the construction of N_h on the basis of N_2 in which points t_1 and t_2 are removed. Let $N_{h-1}(s_1)$ denote a copy of the configuration N_{h-1} that is scaled down by the multiplicative factor δ. The configuration $N_{h-1}(s_1)$ is aligned with N_2 (reduced by t_1 and t_2) such that the source point in $N_{h-1}(s_1)$ corresponds with the point s_1 in N_2 and the vertical line containing point s_1 is the axis of symmetry of $N_{h-1}(s_1)$. Similarly let $N_{h-1}(s_2)$ be the second copy of N_{h-1} embedded in the analogous way but with respect to the point s_2 (instead of s_1). Finally, we scale down the configuration N_{h-2} by the factor δ^2 and place in the analogous way but this time with respect to the point s_3, see Figure 2(c).

Note that due to the scaling process controlled by the inductive assumption in relation to $N_{h-1}(s_1), N_{h-1}(s_2)$, $N_{h-2}(s_3)$, all points in N_h lay above point s symmetrically on both sides of the vertical line containing s and s_3, their distance from the source s is at least 1 but at most $1 + \delta \cdot \frac{1-\delta^{h-1}}{1-\delta} = \frac{1-\delta^h}{1-\delta}$, and the angle of the slice is $\pi/12$. The unique assignment of ranges in minimum energy h-hop broadcasting for the resulting configuration N_h is determined as follows. The source node s has the range 1 and the remaining nodes have respectively scaled down ranges by the multiplicative factor δ in

$N_{h-1}(s_1)$ and $N_{h-1}(s_2)$ and the factor δ^2 in $N_{h-2}(s_3)$. Firstly, the optimal energy $\mathcal{E}_h$ for h-hop broadcast is at most $1 + 2\delta^\alpha \mathcal{E}_{h-1} + \delta^{2\alpha}\mathcal{E}_{h-2} < 1 + 2\delta^\alpha + \sum_{j=2}^{h-1}(3\delta^\alpha)^j$, where $\mathcal{E}_i$ denotes the optimal energy for i-hop broadcast in the configuration N_i, for $i \leq h$ (the inductive proof is straightforward). It follows that the source s must have the range 1 to transmit in a schedule with the energy $\mathcal{E}_h$ (if it informs any other node in N_h, it requires energy at least $(1+\delta/4)^\alpha > 1 + \delta/2 > 1 + 2\delta^\alpha + \sum_{j=2}^{h-1}(3\delta^\alpha)^j$). Secondly, a distance between two points within two different configurations among $N_{h-1}(s_1), N_{h-1}(s_2)$ and $N_{h-2}(s_3)$ is at least $\delta + \delta^2$. It follows that if any node in one of these configurations informs a node in another configuration (except for the nodes s_1, s_2 informing node s_3) the informer would have to use extra energy of size at least $(\delta + \delta^2)^\alpha > \delta^\alpha(1 + 2\delta) > \delta^\alpha + 10\delta^{2\alpha}$. This however, supplemented with energy 1 used by the source and at least δ^α energy used by one of the nodes from $\{s_1, s_2\}$ (at least one of them must transmit, since the range of the source is 1) would give the total energy larger than $1 + 2\delta^\alpha + \sum_{j=2}^{h-1}(3\delta^\alpha)^j > \mathcal{E}_h$.

Now we argue that simulation of h-hop broadcasting with energy $\mathcal{E}_h$ requires at least $1.5h$ rounds in the conflict-embodied model. In the first round, the source delivers the message to nodes s_1, s_2. In the second round, either both s_1, s_2 transmit, or only one of them, say, w.l.o.g., s_1, does. In the first case, none of the nodes in the configuration $N_{h-2}(s_3)$ is informed, and thus an extra $1 + 1.5(h-2)$ rounds are necessary by the fact that the node s_3 needs to be informed and by the invariant for $h-2$, which gives $3 + 1.5(h-2) = 1.5h$ rounds in total. In the latter case, an extra $1.5(h-1)$ rounds are necessary to inform all nodes in $N_{h-1}(s_2)$, which gives in total $2 + 1.5(h-1) \geq 1.5h$ rounds. □

3.2 The Upper Bound

The following upper bound can be seen as a generalization of Theorem 4 with respect to minimum energy h-hop broadcasting. Let ψ be the ratio between the largest and the shortest distance between a pair of nodes in the network.

Theorem 6. *Any h-hop broadcasting with energy $\mathcal{E}$ in a geometric radio network, can be simulated in conflict-embodied model in $O(h \log \psi)$ rounds and energy $O(\mathcal{E})$.*

Proof. Consider a configuration of points on the plane, and h-hop broadcasting with energy $\mathcal{E}$ performed by them in the conflict-free model. First note that the ratio $\frac{r_{max}}{r_{min}}$, where r_{max}, r_{min} are the maximum and the minimum ranges respectively in h-hop broadcasting, is at most ψ. This is because there is no benefit for nodes to transmit within smaller range than the minimum pairwise euclidean distance between points (no point receives the message), and similarly there is no need to transmit with ratio larger than the greatest pairwise distance between nodes (no additional point gets informed). Therefore, having the range assignment for the h-hop broadcasting with energy $\mathcal{E}$ (it corresponds to some GRN), we run the same broadcasting algorithm in the conflict-embodied model as presented in the proof of Theorem 4. It completes broadcasting in time $O(h \log \frac{r_{max}}{r_{min}}) \leq O(h \log \psi)$. It remains to prove that the total energy used is $O(\mathcal{E})$. Indeed, each node v transmits only constant number of times, specifically: during execution of the constant-time procedure LR-BiB corresponding to the range of node v in the phase corresponding to the layer containing the node v. □

References

1. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A lower bound for radio broadcast. Journal of Computer and System Sciences 43, 290–298 (1991)
2. Ambühl, C.: An optimal bound for the MST algorithm to compute energy efficient broadcast trees in wireless networks. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1139–1150. Springer, Heidelberg (2005)
3. Ambühl, C., Clementi, A.E.F., Di Ianni, M., Lev-Tov, N., Monti, A., Peleg, D., Rossi, G., Silvestri, R.: Efficient Algorithms for Low-Energy Bounded-Hop Broadcast in Ad-Hoc Wireless Networks. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 418–427. Springer, Heidelberg (2004)
4. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time complexity of broadcast in radio networks: an exponential gap between determinism and randomization. J. of Computer and System Sciences 45, 104–126 (1992)
5. Bruschi, D., Del Pinto, M.: Lower bounds for the broadcast problem in mobile radio networks. Distributed Computing 10, 129–135 (1997)
6. Chen, J., Jiang, A., Kanj, I.A., Xia, G., Zhang, F.: Separability and Topology Control of Quasi Unit Disk Graphs. In: Proc. 26th IEEE International Conference on Computer Communications INFOCOM 2007, pp. 2225–2233 (2007)
7. Chlamtac, I., Kutten, S.: On broadcasting in radio networks - problem analysis and protocol design. IEEE Transactions on Communication 33, 1240–1246 (1985)
8. Chlamtac, I., Weinstein, O.: The wave expansion approach to broadcasting in multihop radio networks. IEEE Transactions on Communication 39, 426–433 (1991)
9. Chlebus, B.S., Gąsieniec, L., Gibbons, A.M., Pelc, A., Rytter, W.: Deterministic broadcasting in unknown radio networks. Distributed Computing 15, 27–38 (2002)
10. Chlebus, B.S., Gąsieniec, L., Östlin, A., Robson, J.M.: Deterministic radio broadcasting. In: Welzl, E., Montanari, U., Rolim, J. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 717–728. Springer, Heidelberg (2000)
11. Clementi, A.E.F., Crescenzi, P., Penna, P., Rossi, G., Vocca, P.: On the complexity of computing minimum energy consumption broadcast subgraphs. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 121–131. Springer, Heidelberg (2001)
12. Clementi, A.E.F., Huiban, G., Penna, P., Rossi, G., Verhoeven, Y.C.: Some Recent Theoretical Advances and Open Questions on Energy Consumption in Ad-Hoc Wireless Networks. In: Proc. 3rd Workshop on Approximation and Randomization Algorithms in Communication Networks ARACNE 2002, pp. 23–38 (2002)
13. Clementi, A.E.F., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: Proc. 12th Ann. ACM-SIAM Symp. on Discrete Algorithms, SODA 2001, pp. 709–718 (2001)
14. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. In: Proc. 44th Symp. on Foundations of Computer Science, FOCS 2003, pp. 492–501 (2003)
15. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. J. of Discrete Algorithms 5(1), 187–201 (2007)
16. Elkin, M., Kortsarz, G.: An improved algorithm for radio broadcast. ACM Trans. on Algorithms 3(1), 1–21 (2007)
17. Emek, Y., Gąsieniec, L., Kantor, E., Pelc, A., Peleg, D., Su, C.: Broadcasting in UDG radio networks with unknown topology. In: Proc. 26th Annual ACM Symposium on Principles of Distributed Computing PODC 2007, pp. 195–204 (2007)
18. Emek, Y., Kantor, E., Peleg, D.: On the effect of the deployment setting on broadcasting in Euclidean radio networks. In: Proc. 27th Annual ACM Symposium on Principles of Distributed Computing PODC 2008 (to appear, 2008)

19. Flammini, M., Klasing, R., Navarra, A., Perennes, S.: Improved approximation results for the minimum energy broadcasting problem. In: Proc. Joint Work. on Foundations of Mobile Computing DIALM-POMC 2004, pp. 85–91. ACM Press, New York (2004)
20. Fusco, E., Pelc, A.: Broadcasting in UDG radio networks with missing and inaccurate information. In: Proc. 22nd International Symposium on Distributed Computing, DISC 2008 (to appear, 2008)
21. Gaber, I., Mansour, Y.: Centralized broadcast in multihop radio networks. Journal of Algorithms 46, 1–20 (2003)
22. Garey, M.R., Johnson, D.S.: Computers and Intractability: a Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York (1979)
23. Gąsieniec, L., Kantor, E., Kowalski, D.R., Peleg, D., Su, C.: Energy and Time Efficient Broadcasting in Known Topology Radio Networks. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 253–267. Springer, Heidelberg (2007)
24. Gąsieniec, L., Peleg, D., Xin, Q.: Faster communication in known topology radio networks. Distributed Computing 19(4), 289–300 (2007)
25. Gąsieniec, L., Potapov, I., Xin, Q.: Time efficient centralized gossiping in radio networks. Theoretical Computer Science 383(1), 45–58 (2007)
26. Kirousis, L.M., Kranakis, E., Krizanc, D., Pelc, A.: Power Consumption in Packet Radio Networks. Theoretical Computer Science 243, 289–305 (2000)
27. Klasing, R., Navarra, A., Papadopoulos, A., Perennes, S.: Adaptive broadcast consumption (ABC), a new heuristic and new bounds for the minimum energy broadcast routing problem. In: Mitrou, N.M., Kontovasilis, K., Rouskas, G.N., Iliadis, I., Merakos, L. (eds.) NETWORKING 2004. LNCS, vol. 3042, pp. 866–877. Springer, Heidelberg (2004)
28. Kowalski, D.R., Pelc, A.: Optimal deterministic broadcasting in known topology radio networks. Distributed Computing 19(3), 185–195 (2007)
29. Kuhn, F., Zollinger, A.: Ad-hoc networks beyond unit disk graphs. In: Proc. DIALM-POMC Joint Workshop on Foundations of Mobile Computing 2003, pp. 69–78 (2003)
30. Kushilevitz, E., Mansour, Y.: An Ω (D log(N/D)) lower bound for broadcast in radio networks. SIAM J. on Computing 27, 702–712 (1998)
31. Moscibroda, T., Wattenhofer, R.: Maximal independent sets in radio networks. In: Proc. 24th ACM Symp. on Principles of Distributed Computing PODC 2005, pp. 148–157 (2005)
32. Moscibroda, T., Wattenhofer, R.: Coloring unstructured radio networks. In: Proc. 17th ACM Symp. on Parallel Algorithms SPAA 2005, pp. 39–48 (2005)
33. Lauer, G.S.: Packet radio routing. In: Streenstrup, M. (ed.) Routing in communication networks, ch. 11, pp. 351–396. Prentice-Hall, Englewood Cliffs (1995)
34. Navarra, A.: Tighter bounds for the minimum energy broadcasting problem. In: Proc. 3rd International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks WiOpt 2005, pp. 313–322. IEEE CS, Los Alamitos (2005)
35. Peleg, D.: Recent Advances on Approximation Algorithms for Minimum Energy Range Assignment Problems in Ad-Hoc Wireless Networks. In: Erlebach, T. (ed.) CAAN 2006. LNCS, vol. 4235, pp. 1–4. Springer, Heidelberg (2006)
36. Ravishankar, K., Singh, S.: Broadcasting on [0,L]. Discrete Applied Mathematics 53, 299–319 (1994)
37. Sen, A., Huson, M.L.: A new model for Scheduling Packet Radio Networks. In: Proc. 15th Annual Joint Conference of the IEEE Computer and Communication Societies INFOCOM 1996, pp. 1116–1124 (1996)

On the Robustness of (Semi) Fast Quorum-Based Implementations of Atomic Shared Memory*

Chryssis Georgiou[1,**], Nicolas C. Nicolaou[2], and Alexander A. Shvartsman[2,3]

[1] Department of Computer Science, University of Cyprus, Nicosia, Cyprus
[2] Department of Computer Science and Engineering, University of Connecticut, Storrs, USA
[3] Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139, USA

Abstract. This paper studies a trade-off between fault-tolerance and latency in implementations of atomic read/write objects in message-passing systems. In particular, considering *fast* or *semifast quorum-based* implementations, that is, implementations where *all* or respectively *most* read and write operations complete in a single communication round-trip, it is shown that such implementations are *not robust* due to the fact that they necessarily require a quorum system with a common intersection between its quorums.

To trade speed for fault-tolerance, the notion of *weak-semifast* implementations is introduced. Here more than a single complete slow (two round-trip) read operation is allowed for each write operation (semifast implementations allow only one such slow read). A quorum-based algorithm is given next and it is formally shown that it constitutes a weak-semifast implementation of atomic registers. The algorithm uses the notion of *Quorum Views* to facilitate the characterization of all possible object timestamp distributions that a read operation may witness during its first communication round-trip. Noteworthy is that the algorithm allows fast read operations even if they are concurrent with other read and write operations. Finally, experimental results were gathered by simulating the algorithm using the NS-2 network simulator. The results show that under realistic conditions, less than 13% of read operations are slow, thus the overwhelming majority of operations take a single communication round-trip.

1 Introduction

Motivation and Prior Work. Atomic (linearizable) read/write memory is one of the fundamental abstractions in distributed computing. Fault-tolerant implementations of atomic objects in message-passing systems allow processes to share information with precise consistency guarantees in the presence of asynchrony and failures. A seminal implementation of atomic memory of Attiya *et al.* [1] gives a single-writer, multiple reader (SWMR) solution where each data object is replicated at n message-passing nodes. In that solution, memory access operations are guaranteed to terminate as long as the number of crashed nodes is less than $n/2$, i.e., the solution tolerates crashes of any minority of the nodes. The write protocol involves a single round-trip communication

[*] This work is supported in part by the NSF Grants 9988304, 0121277, and 0311368.

[**] The work of this author is supported in part by research funds at the University of Cyprus.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 289–304, 2008.

stage, while the read protocol involves two round-trip stages, where the second stage essentially performs the write of the value obtained in the first stage. Following this development, a folklore belief developed that in messaging-passing atomic memory implementations "atomic reads must write."

However, recent work by Dutta *et al.* [3] established that if the number of readers R is appropriately constrained with respect to the number of replicas S and the maximum number of crash-failures t in the system ($R < \frac{S}{t} - 2$), then single communication round-trip implementations of reads are possible. Such an implementation given in [3] is called *fast*. Subsequently, Georgiou *et al.* [9] relaxed the constraint in [3], and proposed *semifast* implementations with unbounded number of readers, where under realistic conditions most reads need only a single communication round-trip to complete. Their approach groups collections of readers into *virtual nodes*. Semifast behavior of their algorithm is preserved as long as the number of virtual nodes is appropriately restricted under $\frac{S}{t} - 2$.

Quorum systems are well-known mathematical tools that provide means for achieving coordination between processors in distributed systems [6, 11, 21, 22]. Given that the approach of Attiya *et al.* [1] is readily generalized from majorities to quorums (e.g., [20]), and that the algorithms in [3] and [9] rely on intersections in specific sets of responding servers, one may ask: *Can we characterize the conditions enabling fast implementations in a general quorum-based framework?*

This is what we establish in this work. We consider quorum-based implementations of atomic memory, and investigate the properties needed to achieve fast and semifast atomic memory implementations. Interestingly, when examining unconstrained — in terms of quorum construction and reader participation — quorum-based implementations, we discover that a *common* intersection among *all* quorum sets is necessary. This renders such implementations non-fault-tolerant, since the common intersection introduces a single point of failure. Then a natural question arises: *Was a common intersection implied in [3] and [9]?* The answer is "no", because (a) the constraint on the number of readers in [3] or virtual nodes in [9], and (b) the knowledge of the number of failures t, has the implication that the intersections of the replying sets of servers is guaranteed to consist of non-faulty processors. So, our new findings introduce complementary knowledge: *One cannot have fast or semifast implementations without common intersection unless one imposes additional constraints on the system.*

Based on this new understanding, we posed the question of whether one can avoid restrictions, such as the constraint on the number of readers or the common intersection among quorum sets, and still obtain practical and robust implementations. We show that this is indeed possible if some speed is traded for robustness. We introduce *weak-semifast* implementations that allow a greater proportion of slow reads, and develop a new algorithm that uses a predicate tool, called *quorum views*, also defined in this paper. We simulate our algorithm using the NS-2 simulator and we gather experimental results that demonstrate the practicality of our algorithm.

Related Work. Previous works extended the approach in [1] and used quorums to provide atomicity in the *multiple writer multiple reader* (MWMR) model [4, 5, 13, 20]. The work in [5] (similar to [1]) shows that the read operations must write to as many replicas as the maximum number of failures allowed. A dynamic atomic memory implementation using reconfigurable quorums is given in [18], where the sets of object replicas can arbitrarily change over time as processes join and leave the system. Refinements of the dynamic algorithm further improved its performance in practical implementations [7, 8, 12]. When the set of replicas is not being reconfigured, the read and write protocols involve two communication round-trips. Retargeting this work to ad-hoc mobile networks, Dolev *et al.* [2] formulated the GeoQuorums approach where replicas are implemented by stationary *focal points* that in turn are implemented by mobile nodes; here quorums are composed of focal points. Interestingly, in this work some reads involve a single communication round-trip when it is confirmed that the corresponding write operation has completed.

A recent work by Guerraoui and Vukolić [15] presented a powerful notion of *Refined Quorum Systems* (RQS), where quorum members are classified in three categories, called *quorum classes*, according to their intersection size with other quorums; the first class contains quorums of large intersection, the second of smaller intersection, and the third class corresponds to traditional quorums. The authors specify the properties that the members of each quorum class must possess and use RQSs to develop an efficient *Byzantine-resilient* SWMR atomic object implementation and a solution to the consensus problem. In synchronous failure-free runs their implementation allows single communication round-trip (fast) operations. That was an improvement over a previous result from the same authors, [14], that provided bounds and imposed system restrictions to achieve robust *safe* and *regular* storage implementations in the presence of Byzantine failures. In that work they showed that *two* communication round-trips are necessary for each read operation in both safe and regular implementations even though more than $2t+2b+1$ servers are used, where t the maximum number of crash and b the maximum number of byzantine failures. Our work complements the work in [15] by specifying the exact properties that a *general* quorum system must possess in order to achieve single round-trip operations under *crash failures* and *asynchrony*. Furthermore we do not use quorum formation constraints such as the categories mentioned above, rather we deal with the usual quorum systems. In our implementations we only rely on client side prediction tools we call *Quorum Views*.

Malkhi and Reiter in [21] studied constructions that improve fault-tolerance of quorum systems for *byzantine failures*. They organized their constructions according to the properties quorum sets satisfy and the degree of fault-tolerance they achieve. Using such constructions they provided *safe* and *regular* ([16]) read/write register implementations. Regularity was also studied for the MWMR model in [23]; the authors presented regular implementations with single round trip operations. Peleg and Wool in [22], investigated different families of quorum systems and presented their performance in terms of *process load*, *quorum availability* (failure tolerance) and *message complexity* (quorum size). A quorum construction was then proposed that achieves high performance in comparison with prior constructions.

Our Contributions. In this paper we study the properties of quorum systems that enable communication-efficient quorum-based implementations of atomic read/write registers. In particular, we study the efficiency of *general* quorum constructions deployed in implementations with *unconstrained* number of readers. We say that an atomic SWMR implementation is *fast* if all the read and write operations complete in a single communication round-trip. A *semifast* implementation as defined in [9] allows one complete slow read operation for each write, while the rest of read and write operations are required to be fast. In this paper we say that an implementation is *not robust* if it has a single point of failure. We consider implementations that use quorum systems $\mathbb{Q} = \{Q_i\}$, where for any two quorum sets in $\mathbb{Q}$ we have $Q_i \cap Q_j \neq \emptyset$. The contributions presented in this paper are as follows.

1. We show that fast quorum-based implementations that allow arbitrary number of readers must use certain quorum systems that necessarily render the implementations not robust. In particular we prove that a quorum-based fast implementation is possible *if and only if* the following property is satisfied by $\mathbb{Q}$: $\bigcap_{Q \in \mathbb{Q}} Q \neq \emptyset$.

In other words, there must be a *common intersection* among all quorum sets of $\mathbb{Q}$. Since a single failure in the common intersection disables the quorum system, we conclude that *robust* fast quorum-based implementations are *impossible*.

2. We then pose the natural question whether *semifast* ([9]) quorum-based implementations can be robust. We give a negative answer to this question as well: we show that *robust* semifast quorum-based implementations are also *impossible*. In particular we show that a certain property of the semifast definition ([9], Property 3) is violated when using quorum systems without a common intersection. This property states that only a *single complete* read operation is required to perform a second communication round-trip for every write operation. We prove that requiring a single complete slow read is impossible to satisfy using quorum-based implementations without a common intersection.

3. Consequently we seek implementations that enable fast reads, but permit multiple slow reads per write. We call such implementations *weak-semifast*. As a tool used in our development, we introduce the notion of *Quorum Views* that is used to characterize every possible timestamp distribution that a read operation may witness in a quorum set during its first communication round-trip. A quorum view may provide "sufficient" information on whether or not a write operation is complete. If so, then the read operation can be "fast." Otherwise the reader performs a second communication round-trip and the read operation is "slow." We define quorum views and we present an algorithm, called SLIQ (Semifast Like Implementation for Quorum systems), that makes use of the latter idea and we prove its correctness. The algorithm departs from the classic approach that implements all reads as slow, and also from the approach that allows fast reads after the completion of the write operation. In our algorithm fast reads are allowed even in the case of concurrent read and write operations.

4. We simulate our algorithm using the NS-2 network simulator and we observe that in common cases only less than 13% of the read operations need to perform a second communication round-trip, thus the overwhelming number of operations are fast.

Paper Organization. In Section 2 we present our model assumptions and definitions. In Section 3 we present the quorum system properties that are necessary to achieve fast and semifast quorum-based implementations and we show their non-robustness. The notion of quorum views and algorithm SLIQ are presented in Section 4 along with the algorithm's correctness. The results of our simulations are depicted in Section 5. We conclude in Section 6. For full proofs and additional discussion we refer the reader to [10].

2 Model and Definitions

We consider implementations for the single writer, multiple reader (SWMR) in the asynchronous message-passing model. Our system consists of three distinct sets of processes: a distinguished process w is the writer, the set of R readers with unique ids from the set $\mathcal{R} = \{r_1, \ldots, r_R\}$, and the set of S servers (where the object replicas are maintained) with unique ids from the set $\mathcal{S} = \{s_1, \ldots, s_S\}$.

A *quorum system* is a collection of sets of processes, known as *quorums*, such that every pair of such sets intersects. We define a quorum system $\mathbb{Q}$ over the set of servers $\mathcal{S}$ as follows: $\mathbb{Q} = \{Q_i : Q_i \subseteq \mathcal{S}\}$ such that for any two quorums $Q_i, Q_j \in \mathbb{Q}$, $Q_i \cap Q_j \neq \emptyset$. We assume that every process in the system is aware of $\mathbb{Q}$.

Our distributed system is modeled in terms of *I/O automata* [17, 19] where A_p represents the automaton A assigned to process p. Our system is composed of four kinds of automata: the writer $Writer_w$, servers $Server_{s_i}$, readers $Reader_{r_i}$, and $Channel_{p,q}$. Automata $Channel_{p,q}$ and $Channel_{q,p}$ are assumed to implement a reliable communication channels between processes $p \in \{w\} \cup \mathcal{R}$ and processes $q \in \mathcal{S}$. Each I/O automaton A_p consists of a set of states $states(A_p)$ that includes the initial state(s) of A_p, and a signature $sig(A_p)$ that specifies input, output, and internal actions that can be performed by A_p. For an action α, the tuple $\langle state, \alpha, state' \rangle$ represents the *transition* of A_p from state $state$ to $state'$ as the result of α. Such a tuple is also called a *step*, of A_p. An *execution fragment* φ of A_p is a finite or an infinite sequence $state_0, \alpha_1, state_1, \alpha_2, \ldots, \alpha_r, state_r, \ldots$ of alternating states and actions of A_p such that every $state_k, \alpha_{k+1}, state_{k+1}$ is a step of A_p. If an execution fragment begins with an initial state of A_p then it is called an *execution*. We say that an execution fragment φ' of A_p, *extends* a finite execution fragment φ of A_p if the first state of φ' is equal to the last state of φ. The concatenation of of φ and φ' is the result of the extension of φ by φ' where the duplicate occurrence of the last state of φ is eliminated. Such concatenation yields an execution fragment of A_p.

A process p *crashes* at any step $\langle state_k, \alpha_{k+1}, state_{k+1} \rangle$ in an execution ξ, if this is the last step of A_p in ξ. A process p is *faulty* in an execution ξ if p crashes in ξ; otherwise p is *correct*. A quorum $Q \in \mathbb{Q}$ is non-faulty if $\forall p \in Q$, p is correct; otherwise Q is faulty. We assume that any subset of readers, the writer, and all but one quorum in quorum system $\mathbb{Q}$ may be faulty at any execution.

2.1 Atomicity

Our goal is to implement a read/write atomic object in a message passing system by replicating the value of the object among the servers in the system. Each replica consists of a value v and an associated timestamp ts.

The client at process p may request a read operation ρ on the atomic register x by performing a $\mathsf{read}_{x,p}$ action if $p \in \mathcal{R}$. Similarly the client requests a write operation $\omega(*)$ by performing $\mathsf{write}(*)_{x,p}$ if process p is the writer. The step that includes the read or write action is called *invocation* step and the step that contains a $\mathsf{read\text{-}ack}(*)_{x,p}$

or a $\mathsf{write\text{-}ack}_{x,p}$ action is called a *response* step. An operation π is *incomplete* in an execution ξ, if ξ contains the invocation step of π but does not contain the associated response step for π; otherwise we say that π is *complete*. We assume that the requests of a client are *well-formed* meaning that the client does not request a $read$ or $write$ action on an object x, before receiving a $\mathsf{read\text{-}ack}$ or $\mathsf{write\text{-}ack}$ from a previously invoked action on x. From this point onward we assume a single register object. By composing multiple single register implementations, one may obtain the complete atomic shared-memory [17].

In an execution we say that an operation (read or write) π_1 *precedes* another operation π_2, or π_2 *succeeds* π_1, if the response step for π_1 precedes the invocation step of π_2; this is denoted by $\pi_1 \rightarrow \pi_2$. Two operations are *concurrent* if neither precedes the other.

Correctness of an implementation of an atomic object is defined in terms of the *termination* and *atomicity* properties. The termination property requires that any operation invoked by a correct process eventually completes, provided that the failures obey our failure model. Atomicity is defined as follows [17]: Consider the set Π of all complete operations in any well-formed execution. Then there exists an irreflexive partial ordering $\prec$ on operations in Π, satisfying the following: (1) For any operation $\pi \in \Pi$, there are finitely many operations π' such that $\pi' \prec \pi$. (2) If operation π_1 precedes the operation π_2 in Π, then it cannot be the case that $\pi_2 \prec \pi_1$. (3) If π is a write operation and π' is any operation in Π, then either $\pi \prec \pi'$ or $\pi' \prec \pi$. (4) The value returned by a read operation is the value written by the last preceding write operation according to $\prec$ (or $\perp$ if there is no such write).

2.2 Fast, Semifast and Weak-Semifast Implementations

We use the definition of *fast* implementation given by Dutta et al. [3], in particular we say that a read or write operation is ***fast*** if it completes in a single communication round-trip (or round for short). A fast implementation contains only fast operations in any execution. We define a communication round as follows:

Definition 1. *A process p performs a communication round during operation π if all of the following hold:*

(1) *p sends read or write messages for π to a subset of processes,*
(2) *any process p' that receives a message from p for operation π, replies to the message with a read or write acknowledgment respectively before receiving any other message*[1],
(3) *when process p receives enough replies for π it responds to the client.*

The results of [3] show that in fast implementations the number of readers must be constrained with respect to the number of servers. To relax such constraints, [9] proposed *semifast* implementations where some read operations are allowed to perform two communication rounds. The definition of semifast implementations is given below, where $\mathfrak{R}(\rho)$ denotes the unique write operation that wrote the value returned by ρ:

[1] Intuitively this property is used to stress the fact that processes do not need to wait for other messages before replying to p.

Definition 2. *An implementation I of an atomic object is* ***semifast*** *if the following are satisfied:*

P1. *Every* write *operation is fast.*
P2. *Any complete* read *operation performs one or two communication rounds between the invocation and response.*
P3. *For any execution ξ of I, if ρ_1 is a two-round read operation, then any read operation ρ_2 with $\mathfrak{R}(\rho_1) = \mathfrak{R}(\rho_2)$, such that $\rho_1 \rightarrow \rho_2$ or $\rho_2 \rightarrow \rho_1$, must be fast.*
P4. *There exists an execution ξ of I that contains at least one write operation ω and at least one read operation ρ_1 with $\mathfrak{R}(\rho_1) = \omega$, such that all read operations ρ with $\mathfrak{R}(\rho) = \omega$ (including ρ_1) are fast.*

We define a new class of implementations that we call ***weak-semifast*** implementations. This class is defined in terms of properties **P1**, **P2**, and **P4** of the semifast implementations, and it does not include property **P3**. In other words, weak-semifast implementations allow multiple "slow" complete read operations for every write operation in contrast with property **P3** that allows a single such read operation. Thus the two classes are distinct.

Given that any subset of readers and the writer may crash, then termination is guaranteed only if no operation waits for replies from any reader or writer processes. Moreover our failure assumptions on the quorum system imply that no operation can wait for more than a single quorum to reply. Finally, as shown in [3, 9], fast and semifast implementations require that server processes cannot wait for more messages before replying to a read or write operation. Notice that since weak-semifast implementations share the same communication scheme in terms of communication rounds as the semifast implementations, they also follow the rules presented in this paragraph.

2.3 Quorum-Based Algorithms

The results in the next two sections pertain to atomic register implementations that have the following characteristics: (1) they use a quorum system to group the object replicas, (2) participants are aware of the quorum system construction and the operation protocol and (3) in every execution there is at least one non-faulty quorum.

Characteristic (3) describes the failure model of the algorithms we consider. Observe that: (i) according to the pairwise intersection property of quorums it suffices to obtain replies from a single quorum, and (ii) according to (3) only a single quorum may be alive in any execution of the algorithm, and thus waiting for more than one quorum before replying may affect operation termination. Thus we assume that any operation waits for exactly one quorum to reply.

3 Quorum Properties and Fast/Semifast Impossibility

A process p, that invokes an operation π, is said to *contact* a subset of servers $\mathcal{G} \subseteq \mathcal{S}$, denoted by $cnt(\mathcal{G})_{p,\pi}$, if for every server $s_i \in \mathcal{G}$: (a) s_i receives the messages sent by p within the operation π, (b) s_i replies to p, and (c) p receives the reply from s_i. If $cnt(\mathcal{G})_{p,\pi}$ and additionally no other server (i.e., $s_i \notin \mathcal{G}$) receives any message from p within the operation π then we say that p *strictly contacts* $\mathcal{G}$, and is denoted by $scnt(\mathcal{G})_{p,\pi}$. Let $maxTS$ denote the maximum timestamp that a read operation ρ_i witnesses after $cnt(\mathcal{G})_{*,\rho_i}$ or $scnt(\mathcal{G})_{*,\rho_i}$, for some $\mathcal{G} \subseteq \mathcal{S}$, during its first round.

Below we discuss our results regarding quorum-based fast and semifast implementations. More detailed analysis can be found in [10].

Fast Implementations. We now state the quorum property that is both necessary and sufficient to obtain fast quorum-based implementations.

Theorem 1. *A fast quorum-based implementation I of a read/write atomic register is possible iff the underlying quorum system $\mathbb{Q}$ satisfies: $\bigcap_{Q \in \mathbb{Q}} Q \neq \emptyset$.*

Proof (Sketch). We prove the two directions of the theorem separately. We first show that if we want fast implementations it is necessary to have common intersection, and then we show that having a common intersection it is sufficient to build fast implementations.

The first part of the proof relies on an execution construction. The construction involves an execution ξ_0 that contains a complete write operation which $scnt(Q_i)_{*,\omega}$ and a series of executions $\xi_1, \ldots, \xi_{n-2}$ ($n = |\mathbb{Q}|$). Starting from ξ_0 we extend each execution ξ_i with $i + 2$ read operations such that each of them strictly contacts a different quorum. Using an induction on the number of read operations, it can be shown that atomicity is preserved only if the last read operation, say ρ_k in the execution ξ_{k-2}, may witness the maximum timestamp. Assuming that ρ_k $scnt(Q_z)_{*,\rho_k}$ and the maximum timestamp is introduced only in a quorum intersection $\mathcal{Q}_{inter}$ then ρ_k may witness the maximum timestamp only if $\mathcal{Q}_{inter} \cap Q_z \neq \emptyset$. Generalizing to the full quorum system the common intersection follows.

The fact that the common intersection is sufficient for fast implementations follows from a trivial implementation: each read/write operation contacts (only) the servers in the common intersection and returns the maximum timestamp observed in the first communication round. Notice here that according to our failure model, all the servers of the common intersection must remain alive during the execution. Thus atomicity is not violated since every read/write operation will gather all the servers in the common intersection and furthermore all operations complete in a single communication round.

Theorem 1 leads to the following result.

Theorem 2. *Fast quorum-based implementations are not robust.*

Proof. Theorem 1 requires a common intersection between the quorum sets of the quorum system $\mathbb{Q}$. If any member node s_i of the common intersection fails, then all quorum members of the quorum system are faulty since $\forall Q \in \mathbb{Q}$, $s_i \in Q$. Hence it follows that the quorum system $\mathbb{Q}$ fails. Therefore the quorum system suffers from a single point of failure and as a sequel it is not robust. As a result, any implementation that relies on such a quorum system is also not robust.

Semifast implementations. Since fast implementations are not possible if common intersection property is not satisfied by the quorum system, a natural question arise

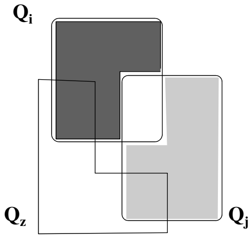


Fig. 1. Intersections of three quorums Q_i, Q_j, Q_z

whether robust *semifast* implementations can be achieved. We show that robust semifast implementations are also impossible if the common intersection between the quorums is not preserved. We use the properties of the semifast implementations as presented in Definition 2.

We first prove a lemma that specifies when a read operation is necessary to perform a second communication round. The lemma is general and algorithm-independent.

Lemma 1. *A read operation ρ from a reader r that $scnt(Q_i)_{r,\rho}, Q_i \in \mathbb{Q}$ cannot be fast if $\exists s \in Q_i : s.ts < maxTS$ and $\exists \mathcal{Q} \subset \mathbb{Q}$ s.t. $Q_i \cap \left(\bigcap_{q \in \mathcal{Q}} q\right) \neq \emptyset$ and $\forall s \in Q_i \cap \left(\bigcap_{q \in \mathcal{Q}} q\right)$, $s.ts = maxTS$.*

We can then derive the following result.

Theorem 3. *No quorum-based semifast implementation is possible if $\bigcap_{Q \in \mathbb{Q}} Q = \emptyset$.*

Proof (Sketch). The proof is build upon execution constructions that exploit a basic quorum system similar to the one presented in Figure 1 with no common intersection. Based on that figure, consider an execution that contains a write operation ω and three read operations ρ_1, ρ_2 and ρ_3. Assume that the write is incomplete and $scnt(Q_j \cap Q_i)_{w,\omega}$. Moreover the ρ_1 operation $scnt(Q_i)_{*,\rho_1}$ during its first communication round. According to Lemma 1, ρ_1 needs to perform a second communication round and suppose it $scnt(Q_i \cap Q_j)_{*,\rho_1}$ before the first communication round of ρ_2. Thus when ρ_2 is executed, and $scnt(Q_j)_{*,\rho_2}$, observes that ρ_1 performed a second communication round but it cannot distinguish whether ρ_1 is completed or not. Here is where the key idea of the proof lies: if ρ_2 is fast (s.t. Property 3 of the semifast definition holds), then if ρ_3 $scnt(Q_z)_{*,\rho_3}$, ρ_3 will not witness the maximum timestamp and thus will return an older value violating atomicity. So to preserve atomicity ρ_2 has to proceed to a second communication round. Since, however, ρ_2 cannot distinguish between a complete second communication round of ρ_1 that $scnt(Q_i)_{*,\rho_1}$ and the incomplete second communication round of ρ_1 that $scnt(Q_i \cap Q_j)_{*,\rho_1}$, then ρ_1 and ρ_2 may not be concurrent but yet be both slow. That however violates Property 3 of the semifast definition.

We conclude that robust quorum-based semifast implementations are not possible.

Corollary 1. *Semifast quorum-based implementations are not robust.*

Remark 1. Observe that the robustness of fast quorum-based implementations can be improved by the following techniques: (i) relaxing the failure model and requiring more than a single quorum to reply at any read/write operation, and (ii) impose restrictions on the number of reader participants and on the construction of the quorum system they deploy. This however will negatively affect the performance of the quorum system and will introduce strong assumptions for its maintenance, making eventually the use of quorums impractical. Thus in this work we avoid making such assumptions and we prefer to trade operation performance for higher fault-tolerance and applicability.

The following example help us visualize the application of the second technique presented in the above remark. Assume the following setting under [3]: $S = \{1, 2, 3, 4, 5\}, R = 2$ and $t = 1$. Any operation may receive replies from $S - t$ servers and from one of the sets: $Q_1 = \{1, 2, 3, 4\}, Q_2 = \{1, 3, 4, 5\}, Q_3 = \{1, 2, 4, 5\}, Q_4 = \{1, 2, 3, 5\}, Q_5 = \{2, 3, 4, 5\}$. Observe that the intersection of any three sets contains two servers $(t + 1)$. Since there are two readers and one writer then a written value may be disseminated by contacting at most three different sets in the worst case (a different set per read/write operation). So restricting the number of readers allows the concentration of the common intersection between a subset of quorum sets, which serves as a "hot spot" to ensure consistency between the operations.

4 Weak-Semifast Implementations

In the previous section we have established that no robust fast or semifast quorum-based implementations are possible. We therefore now consider weak-semifast implementations. In this section we introduce the notion of *Quorum Views* that describe certain knowledge that a read operation may witness during its first communication round. We then present an algorithm, called SLIQ, for atomic registers, and we reason, based on the knowledge in quorum views, about read operations needing to perform one or two communication rounds to complete. We deviate from the restrictive quorums properties presented in Section 3 and we allow our implementation to use an *arbitrary* quorum system construction.

4.1 Quorum Views

A *quorum view* refers to the distribution of the maximum timestamp that a read operation ρ_i witnesses after its first communication round. Consider that the read operation ρ_i strictly contacts quorum Q_i during its first communication round, denoted by $scnt(Q_i)_{*,\rho_i}$. Each member $s \in Q_i$ replies with a timestamp $s.ts$ to ρ_i. We define quorum views in terms of the following three possible cases for ρ_i:

1. $[qView(1)]$ $\forall s \in Q_i : s.ts = maxTS$,
2. $[qView(2)]$ $\forall Q_j \in \mathbb{Q}, i \neq j, \exists A \subseteq Q_i \cap Q_j$, s.t. $A \neq \emptyset$ and $\forall s \in A : s.ts < maxTS$,

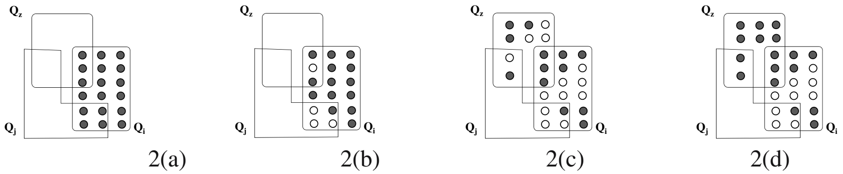


Fig. 2. (a) $qView(1)$, (b) $qView(2)$, (c) $qView(3)$ with incomplete write, (d) $qView(3)$ with complete write

3. $[qView(3)]$ $\exists s' \in Q_i : s'.ts < maxTS$ and $\exists Q_j \in \mathbb{Q}, i \neq j$ s.t. $\forall s \in Q_i \cap Q_j : s.ts = maxTS$.

Analyzing these three types of quorum views we can derive conclusions on the state of the write operation (complete or incomplete) that tries to propagate a value with the $maxTS$ in the system. Figure 2 illustrates those quorum views assuming that the read operation ρ, $scnt(Q_i)_{*,\rho}$. The dark nodes maintain the maximum timestamp of the system and white nodes or "empty" quorums maintain an older timestamp. Recall that it follows from our failure model that no operation (read or write) can wait for more than one quorum to reply. Thus having a full quorum reporting the same $maxTS$, as seen in Fig. 2(a), implies the possible completion of the write operation (in the case of Figure 2(a) the complete write operation strictly contacts Q_i).

Observe that if a full quorum contains $maxTS$ then the members of any intersection of that quorum contain $maxTS$. So witnessing a subset of members of each intersection of Q_i (as seen in Fig. 2(b) the representation of $qView(2)$) to maintain an older timestamp, implies directly that the write operation which propagates $maxTS$ is not yet complete.

Finally, $qView(3)$, provides insufficient information regarding the state of the write operation. Observe Figures 2(c) and 2(d). In the former an incomplete write operation propagates the $maxTS$ in the dark nodes and in the latter it completes by receiving replies from Q_z. Notice that if a read operation ρ strictly contacts Q_i (i.e., $scnt(Q_i)_{*,\rho}$) in the two executions, it won't be able to distinguish 2(c) from 2(d). So, more formally, if an operation witnesses some intersection $Q_i \cap Q_z$ that contains $maxTS$ in all of its members, then a write operation might: (i) have been completed and contacted Q_z or (ii) be incomplete and contacted a subset of servers B such that $Q_i \cap Q_z \subseteq B$ and $\forall Q_j \in \mathbb{Q}, Q_j \not\subseteq B$.

4.2 Algorithm Sliq

Our implementation includes, automaton $Writer_w$ that handles the write operations for the writer process w, automaton $Reader_{r_i}$ that handles the reading for each $r_i \in \mathcal{R}$, and automaton $Server_{s_i}$ that handles the read and write requests on the atomic register for each $s_i \in \mathcal{S}$. These automata use reliable asynchronous process-to-process channels $Channel_{p,q}$ to communicate.

Algorithm Description. Due to space limitations we only provide a high level description of our algorithm. A more technical description and formal specification can be found in [10].

Writer. The write protocol involves the propagation of a write message to all the servers. Once the writer receives replies from a full quorum it increments its timestamp and the operation completes.

Readers. The read protocol also requires that a reader propagates a read message to all the servers. Once the reader receives replies from a full quorum it examines the maximum timestamp ($maxTS$) distribution within that quorum, which in turn characterizes a quorum view. If the view is either $qView(1)$ or $qView(2)$ then the reader terminates in the first communication round and returns $maxTS$ or $maxTS - 1$ respectively. If the view is $qView(3)$ then the reader proceeds to the second communication round where it propagates the maximum timestamp to a full quorum in a similar manner as the writer. Once the reader gets replies from a full quorum, the operation completes, returning $maxTS$.

Servers. The servers maintain a passive role; they just receive messages, update their replica value according to the message contents and reply to those messages.

Algorithm Correctness. We prove that algorithm SLIQ follows its specifications and preserves atomicity; specifically we show that each atomicity property of Section 2.1 holds for any execution of the algorithm. The main theorem is the following:

Theorem 4. *Algorithm* SLIQ *implements a SWMR atomic read/write register.*

Proof (Sketch). Let $ts_p(\pi)$ to denote the timestamp of the read or write operation π from a process p, after the completion event of π. Summarizing the atomicity properties we want to show: (i) The timestamp at each process is monotonically increasing, (ii) If a read operation ρ succeeds a write operation ω then $ts_*(\rho) \geq ts_w(\omega)$ and, (iii) If ρ_1 and ρ_2 are two read operations such that $\rho_1 \rightarrow \rho_2$ then $ts_*(\rho_2) \geq ts_*(\rho_1)$. The monotonicity on the timestamps (property (i)) for every process can be easily derived from the algorithm. For property (ii) we can observe that since the write operation is completed then ρ will witness a timestamp ts greater or equal to $ts_w(\omega)$. If $ts > ts_w(\omega)$ then $ts_*(\rho) \geq ts_w(\omega)$ since $ts_*(\rho) = ts$ or $ts_*(\rho) = ts - 1$. If $ts = ts_w(\omega)$ then ρ will witness either $qView(1)$ or $qView(3)$. In either case $ts_*(\rho) = ts$. Finally for property (iii) we investigate all the possible quorum views. We need to show that if ρ_1 is fast then $ts_*(\rho_2)$ is at least equal to $ts_*(\rho_1)$. Notice that ρ_1 is fast only if a $qView(1)$ or $qView(2)$ is observed. If $qView(1)$ is witnessed by ρ_1 then ρ_2 witnesses an intersection with timestamps greater or equal to $ts_*(\rho_1)$. If $qView(2)$ is witnessed by ρ_1 then a timestamp $ts = ts_*(\rho_1) + 1$ has already introduced in the system and thus the write operation that wrote a timestamp equal to $ts_*(\rho_1)$ is already completed. Thus, by Property (ii) ρ_2 returns $ts_*(\rho_2) \geq ts_*(\rho_1)$, and hence property (iii) follows.

Note that it is straightforward to verify that SLIQ belongs in the class of weak-semifast implementations (that is, it satisfies properties **P1, P2** and **P4** of Definition 2).

5 Simulation Results

To practically evaluate our findings, we simulated our algorithm using the the NS-2 network simulator. The detailed testbed and discussion regarding the simulation appears in [10]. According to our setting, only the messages between the invoking processes and the servers, and the replies from the servers are delivered (no messages are exchanged between any servers or among the invoking processes).

We have evaluated our approach over multiple quorum systems (majorities $\mathbb{Q}_m$, matrix quorums $\mathbb{Q}_x$ and crumbling walls $\mathbb{Q}_c$), but due to space limitations we only present here some of the plots we obtained exploiting crumbling walls (see [22]). The quorum system is generated apriori and is distributed to each participant node via an external service (out of the scope of this work). No dynamic quorums are assumed, so the configuration of the quorum system remains the same throughout the execution of the simulation. We model server failures by choosing the non-faulty quorum and allowing any server that is not a member of that quorum to fail by crashing. Note that the non-faulty quorum is not known to any of the participants. The positive time parameter $cInt$ is used, to model the failure frequency or reliability of every server s_i.

We use the positive time parameters $rInt$ and $wInt$ (both greater than 1 sec) to model the time intervals between any two successive read operations and any two successive write operations respectively. We considered three simulation scenarios corresponding to the following parameters: (i) $rInt < wInt$: this models frequent reads and infrequent writes, (ii) $rInt = wInt$: this models evenly spaced reads and writes, (iii) $rInt > wInt$: this models infrequent reads and frequent writes.

Furthermore for each one of the above scenarios we consider two settings:

(a) *Stochastic setting*: the read/write intervals vary randomly within $[0 \ldots rInt]$ and $[0 \ldots wInt]$ respectively.

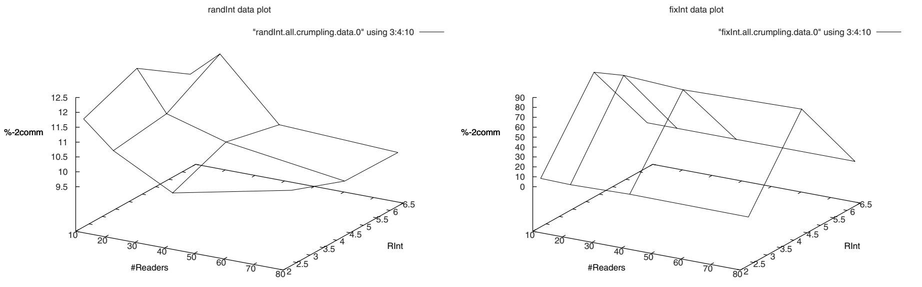
(b) *Fixed setting*: the read/write intervals are fixed to the value of $rInt$ and $wInt$ respectively.

We can summarize our simulations testbed for each class of quorums and for the settings presented above, as follows:

(1) **Simple Runs: ($\mathbb{Q}_c,\ \mathbb{Q}_x,\ \mathbb{Q}_m$)** $|\mathcal{S}| = 25$ ($\mathbb{Q}_c,\ \mathbb{Q}_x$) or $|\mathcal{S}| = 10$ ($\mathbb{Q}_m$), $cInt = 0$ (failure check for every reply) and $|\mathcal{R}| \in [10, 20, 40, 80]$. Here we want to demonstrate the performance of the algorithm under similar environments (quorum,failures) but with different read load.

(2) **Quorum Diversity Runs: ($\mathbb{Q}_c,\ \mathbb{Q}_x$)** $|\mathcal{S}| \in [11, 25, 49]$ ($\mathbb{Q}_c$) and $|\mathcal{S}| \in [11, 25, 49]$ ($\mathbb{Q}_x$), $cInt = 0$ and $|\mathcal{R}| \in [10, 20, 40, 80]$. These runs demonstrate the performance of the algorithm in different quorum systems with varying quorum membership. Each quorum is tested in variable read load.

(3) **Failure Diversity Runs: ($\mathbb{Q}_c,\ \mathbb{Q}_x$)** $|\mathcal{S}| = 25$, $cInt \in [10 \ldots 50]$ with steps of 10 and $|\mathcal{R}| \in [10, 20, 40, 80]$. These runs tested the durability of the algorithm to failures. Notice that the smaller the crash interval the faster we diverge to the non-faulty quorum. As the crash interval becomes bigger, less servers fail and thus more quorums "survive" in the quorum system. For this class of runs we tested both the cases when the servers get the crash interval randomly from $[0 \ldots cInt]$ and $[10 \ldots 10 + cInt]$.



Setting a: Stochastic simulations Setting b: Fixed interval simulations

Fig. 3. Simple runs using Crumbling Walls

Figure 3 illustrates the results obtained when we assumed simple runs and exploiting crumbling walls quorum. The Z axis presents the percentage of the read operations that performed two communication rounds, the X axis corresponds to the number of reader participants and the Y axis represents time and in particular the $rInt$ interval. In the stochastic environment (Figure 3.a) we observe that the percentage of slow reads drops as the number of readers increases, regardless of the value of $rInt$. This behavior can be explained from the fact that the concurrency between the operations is minimized and thus the maximum timestamp is propagated (by both the writer and the readers) to enough servers that favor the fast behavior. Since the convergence point is similar regardless the number of readers, then increasing the readers, increases the number of fast reads and decreases the percentage of slow reads. Similar behavior is observed in the fixed interval environment (Figure 3.b) whenever there is no strict concurrency between the reads and the writes. The worst case is observed at the point where all operations are invoked concurrently.

Our results (including the ones given in [10]) reveal that in realistic cases (i.e. stochastic settings), the percentage of two communication round reads does not exceed 13%. The only case that requires more than 85% of the reads to be slow is the worst case scenario were the read and write intervals are fixed to the same value. Notice however that this scenario is unlikely to appear in practical settings. Comparing our results with the ones obtained in [9] one can observe that the difference in the random scenarios does not exceed 6%.

6 Conclusions

In this paper we have shown that no robust fast or semifast quorum-based implementations of atomic read/write objects are possible in the presence of crashes. We thus introduced the notion of weak-semifast implementations, reasoned that this notion is meaningful, and showed that robust weak-semifast quorum-based implementations exist. As a tool, we introduced the notion of a Quorum View that we used in the design and analysis of our robust algorithm. We formally proved the correctness of the algorithm and we obtained simulation results that demonstrate that under realistic conditions the overwhelming number of read operations are fast.

The algorithm does not explicitly provide any guarantees on the relative frequency of slow and fast read operations. Thus it would be interesting to examine ways to reduce the number of slow operations either by imposing a supplemental communication scheme or by using a special form of quorum systems. An interesting direction is to investigate whether combining quorum views with refined quorum systems [15] can lead to more efficient implementations. In another direction, dynamic membership of quorum systems can also further improve the flexibility and fault tolerance of quorum-based implementations. Given the results in [4, 18] a quorum reconfiguration requires some communication overhead. So a natural question arises regarding the communication efficiency of such dynamic systems and their impact on the performance of a weak-semifast implementation.

References

1. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message passing systems. Journal of the ACM 42(1), 124–142 (1996)
2. Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., Welch, J.: Geoquorums: Implementing atomic memory in mobile ad hoc networks. In: Proceedings of the 17th International Symposium on Distributed Computing (DISC), pp. 306–320 (2003)
3. Dutta, P., Guerraoui, R., Levy, R.R., Chakraborty, A.: How fast can a distributed atomic read be? In: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC), pp. 236–245 (2004)
4. Englert, B., Shvartsman, A.A.: Graceful quorum reconfiguration in a robust emulation of shared memory. In: Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS), pp. 454–463 (2000)
5. Fan, R., Lynch, N.: Efficient replication of large data objects. In: Proceedings of the 17th International Symposium on Distributed Computing (DISC), pp. 75–91 (2003)
6. Garcia-Molina, H., Barbara, D.: How to assign votes in a distributed system. Journal of the ACM 32(4), 841–860 (1985)
7. Georgiou, C., Musial, P.M., Shvartsman, A.A.: Developing a consistent domain-oriented distributed object service. In: Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA), pp. 149–158 (2005)
8. Georgiou, C., Musial, P.M., Shvartsman, A.A.: Long-lived RAMBO: Trading knowledge for communication. Theoretical Computer Science 383(1), 59–85 (2007)
9. Georgiou, C., Nicolaou, N., Shvartsman, A.: Fault-tolerant semifast implementations for atomic read/write registers. Journal of Parallel and Distributed Computing (accepted, 2008); A preliminary version appears in SPAA 2006, pp. 281–290 (2006)
10. Georgiou, C., Nicolaou, N., Shvartsman, A.: On the robustness of (semi) fast quorum-based implementations of atomic shared memory(2008), http://www.cse.uconn.edu/~ncn03001/pubs/TRs/GNS08.pdf
11. Gifford, D.K.: Weighted voting for replicated data. In: Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP), pp. 150–162 (1979)
12. Gilbert, S., Lynch, N., Shvartsman, A.: RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In: Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN), pp. 259–268 (2003)

13. Gramoli, V., Anceaume, E., Virgillito, A.: SQUARE: scalable quorum-based atomic memory with local reconfiguration. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), pp. 574–579 (2007)
14. Guerraoui, R., Vukolić, M.: How fast can a very robust read be? In: Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC), pp. 248–257 (2006)
15. Guerraoui, R., Vukolić, M.: Refined quorum systems. In: Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC), pp. 119–128 (2007)
16. Lamport, L.: On interprocess communication, part I: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
17. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco (1996)
18. Lynch, N., Shvartsman, A.: RAMBO: A reconfigurable atomic memory service for dynamic networks. In: Proceedings of the 16th International Symposium on Distributed Computing (DISC), pp. 173–190 (2002)
19. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI-Quarterly, 219–246 (1989)
20. Lynch, N.A., Shvartsman, A.A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS), pp. 272–281 (1997)
21. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11(4), 203–213 (1998)
22. Peleg, D., Wool, A.: Crumbling walls: A class of high availability quorum systems. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC), pp. 120–129 (1995)
23. Shao, C., Pierce, E., Welch, J.L.: Multi-writer consistency conditions for shared memory objects. In: Proceedings of the 17th International Symposium on Distributed Computing (DISC), pp. 106–120 (2003)

Permissiveness in Transactional Memories[⋆]

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh

EPFL, Switzerland

Abstract. We introduce the notion of permissiveness in transactional memories (TM). Intuitively, a TM is permissive if it never aborts a transaction when it need not. More specifically, a TM is permissive with respect to a safety property *p* if the TM accepts every history that satisfies *p*. Permissiveness, like safety and liveness, can be used as a metric to compare TMs. We illustrate that it is impractical to achieve permissiveness deterministically, and then show how randomization can be used to achieve permissiveness efficiently. We introduce Adaptive Validation STM (AVSTM), which is probabilistically permissive with respect to opacity; that is, every opaque history is accepted by AVSTM with positive probability. Moreover, AVSTM guarantees lock freedom. Owing to its permissiveness, AVSTM outperforms other STMs by up to 40% in read dominated workloads in high contention scenarios. But, in low contention scenarios, the book-keeping done by AVSTM to achieve permissiveness makes AVSTM, on average, 20-30% worse than existing STMs.

1 Introduction

Transactional memory (TM) tries to maximize concurrency in an implementation while providing the illusion of sequentiality to the programmer. It holds the promise to exploit the computational power of modern multi-processor architectures within the security afforded by a simple, non-concurrent programming model. A transaction is an atomic program that can commit its actions to memory, or abort without changing the memory. An abort can be caused by the programmer (say, if some exception is raised), or by the TM itself, if there is a risk to violate the correctness of the memory. Typically, this correctness is expressed by some form of serialization for transactions; that is, a transaction can commit only if the state of the memory could have been generated by some sequential execution of the transactions so far.

At first glance, one would expect from a TM that it never aborts a transaction when it need not, i.e., when there is no risk of violating correctness. It turns out, however, that proposed TMs have certain scenarios where a transaction is aborted even if it could have committed without violating correctness. In other words, these TMs do not enable the maximal amount of possible concurrency among a set of transactions. This observation naturally raises the question of whether one can devise an ideal, maximal TM, which never aborts a transaction unless necessary for correctness. We call such a TM *permissive*.

[⋆] This research was supported by the Swiss National Science Foundation.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 305–319, 2008.

In this paper, we formalize the notion of permissiveness and discuss why the existing TM implementations are not permissive. We argue that permissiveness is expensive to achieve in a deterministic TM. We then present a randomized permissive TM. We show in particular that using randomization in choosing the serialization point of every transaction creates an efficient permissive STM.

Formally, a deterministic TM is an online algorithm that is given a sequence of statements and decides for each statement, based on the statements so far, whether or not to accept the statement. A deterministic TM is *permissive* with respect to a given safety property (e.g., serializability) if every history (finite sequence of statements) that satisfies the safety property is accepted by the TM. In Section 2, we show that existing TMs, like TL2 [1], WSTM [3], and DSTM [9], are not permissive with respect to serializability or opacity, a strong form of serializability that arguably corresponds to what should be expected from a TM [7,9]. Opacity captures the practical notion in TM that transactions execute serially, and even aborting transactions do not view inconsistent state. To our knowledge, the only deterministic TM permissive with respect to serializability or opacity occurs in our recent work [5]. This TM is built using the notion of conflict graphs [10]. But the conflict graph changes globally with every statement. Capturing this change incurs a high cost per statement, and the feasibility of a practical deterministic permissive TM remains questionable.

For randomized TMs, it is natural to consider weaker, probabilistic notions of permissiveness. Formally, a randomized TM is an online algorithm that is given a sequence of statements and decides for each statement, based on a random coin toss, whether to serialize the transaction at the current statement, and based on the statements so far, whether or not to accept the statement. We say that a randomized TM is permissive with respect to a safety property if every history that satisfies the safety property is accepted by the TM with probability 1. Moreover, we say that a randomized TM is *probabilistically permissive* with respect to a safety property if every history that satisfies the safety property is accepted by the TM with positive probability. We do not know of any existing randomized TM that is permissive, or probabilistically permissive, with respect to serializability or opacity. We present Adaptive Validation STM (AVSTM), which can be configured to be probabilistically permissive for strict serializability (SS-AVSTM) or opacity (OP-AVSTM). AVSTM uses randomization to determine an ordering (serialization) point during the life-time of each transaction. We have designed AVSTM in such a manner that it guarantees lock freedom; that is, infinitely many transactions commit in every infinite history produced by AVSTM.

Typically, the efficiency of a TM is measured by the number of transactions that commit per time unit. So, in theory, putting the book-keeping aside, a more permissive TM should also be more efficient, as it aborts less often. We evaluate this claim in practice by implementing an AVSTM prototype and comparing its performance to existing TMs. Our evaluation on a multi-processor architecture (4 processor dual-core running Linux) shows that, indeed, AVSTM outperforms existing TMs (such as DSTM, WSTM, TL2) by upto 40% in high-contention scenarios, where many processes are accessing a small set of variables. In low-contention scenarios, AVSTM does not outperform the most efficient TMs, and suffers performance by 20-30%. This is due to the amount of book-keeping used by AVSTM, which turns out to be more expensive than a few unnecessary aborts in low-contention scenarios. We present a simple scheme to compose TL2 with AVSTM obtaining the advantages of both algorithms. In short, the processes run by default TL2 and dynamically switch to AVSTM when contention increases.

Related work. Many STMs [1,3,8,9,11] have been proposed in the literature. Most of these guarantee opacity, but none of them is opacity-permissive. Existing STMs guarantee different levels of liveness. DSTM [9] guarantees obstruction freedom. Many contention managers [6,12] have been proposed to boost obstruction freedom. But, in our knowledge, there is no contention manager that boosts obstruction freedom to yield lock freedom. WSTM [3] guarantees lock freedom.

2 Framework

We formalize the notion of safety and permissiveness in transactional memories.

Preliminaries. Let V be a set $\{1, \ldots, k\}$ of k *variables*. Let $C = \{\mathsf{commit}\} \cup \{\mathsf{abort}\} \cup (\{\mathsf{read}, \mathsf{write}\} \times V)$ be the set of *commands* on the variables in V. For our formalism, we treat these commands as atomic. Let $P = \{1, \ldots, n\}$ be the set of *processes*. Let $\Sigma = C \times P$ be a finite set of *statements*. A *history* $h \in \Sigma^*$ is a finite sequence of statements. Given a history h, we define the *projection* $h|_p$ of h on process $p \in P$ as the longest subsequence h' of h such that every statement in h' is in $C \times \{p\}$. Given a projection $h|_p = \sigma_0 \ldots \sigma_m$ of a history h, a statement σ_i is *finishing* in $h|_p$ if it is a commit or an abort. A statement σ_i is *initiating* in $h|_p$ if it is the first statement in $h|_p$, or the previous statement σ_{i-1} is a finishing statement.

Given a projection $h|_p$ of a history h on process p, a consecutive subsequence $x = \sigma_0 \ldots \sigma_m$ of $h|_p$ is a *transaction* of process p in h if (i) σ_0 is initiating in $h|_p$, and (ii) σ_m is either finishing in $h|_p$, or σ_m is the last statement in $h|_p$, and (iii) no other statement in x is finishing in $h|_p$. The transaction x is *committing* in h if σ_m is a commit statement. Given a history h and two transactions x and y in h (possibly of different processes), we say that $x <_h y$ if the finishing statement of x occurs before the initiating statement of y in h. A history h is *sequential* if for every pair (x, y) of transactions in h, either $x <_h y$ or $y <_h x$.

We define a function $com : \Sigma^* \to \Sigma^*$ such that for all histories $h \in \Sigma^*$, the history $com(h)$ is the longest subsequence h' of h such that every statement in h' is part of a committing transaction in h. Thus, $com(h)$ consists of all statements of all committing transactions in h. A statement $\sigma = ((\mathsf{read}, v), p)$ in x is a *global read* of the variable v if there is no write to v before σ in x.

Safety properties. Strict serializability [10] is a commonly used correctness criterion for concurrent systems and, in particular, for transactional systems. In the scope of STMs, a stronger notion of correctness, referred to as *opacity* has been suggested [7,9] to avoid unexpected side effects, like infinite loops, or array bound violations due to inconsistent state seen by aborting transactions. Opacity requires that a history is strictly serializable, and that the aborting transactions do not see inconsistent values. Transactional memories use direct update semantics (every transaction modifies the shared variables in place and restores them in case of abort), or deferred update semantics (every transaction modifies a local copy, and changes the shared copy upon a commit). We define the notion of a conflict under the deferred update semantics. A statement σ_1 of transaction x and a statement σ_2 of transaction y $(x \neq y)$ *conflict* in a history h if (i) σ_1 is a global read of variable v and σ_2 is a commit and y writes to v, or (ii) σ_1 and σ_2 are both commits, and x and y write to v. Note that the definition of a conflict would be different with direct update semantics.

A history $h = \sigma_0 \ldots \sigma_m$ is *strictly equivalent* to a history h' if (i) $h|_p = h'|_p$ for all processes $p \in P$, and (ii) for every pair σ_i, σ_j of statements in h, if σ_i and σ_j conflict and $i < j$, then σ_i occurs before σ_j in h', and (iii) for every pair x, y of transactions in h, if $x <_h y$ then it is not the case that $y <_{h'} x$. A history $h \in \Sigma^*$ is *strictly serializable* if there exists a sequential history h' such that h' is strictly equivalent to $com(h)$. Furthermore, we define that a history h is *opaque* if there exists a sequential history h' such that h' is strictly equivalent to h. (Note that h may contain unfinished transactions.) We note that if a history h is opaque, then h is strictly serializable.

We define the safety property *strict serializability* $\pi_{ss} \subseteq \Sigma^*$ as the set of all strictly serializable histories, and the safety property *opacity* $\pi_{op} \subseteq \Sigma^*$ as the set of all opaque histories.

Transactional memories. We model transactional memories as transition systems, that consist of a set of states, an initial state, an alphabet of statements, and a transition relation between the states.

We define a *TM* $A = \langle Q, q_{init}, \Sigma, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, Σ is the set of statements, and $\delta \subseteq Q \times \Sigma \times \Gamma \times Q$ is the transition relation, where $\Gamma = (0, 1]$ represents the probability of a transition. For all $q \in Q$ and $\sigma \in \Sigma$, if there are m outgoing transitions $(q, \sigma, \gamma_i, q') \in \delta$ with $1 \leq i \leq m$, then we have $\sum_i \gamma_i = 1$. A transition relation δ is *deterministic* if for all $q \in Q$ and $\sigma \in \Sigma$, if $(q, \sigma, \gamma_1, q_1) \in \delta$ and $(q, \sigma, \gamma_2, q_2) \in \delta$, then $q_1 = q_2$ and $\gamma_1 = \gamma_2$. Given a TM A, a sequence $q_0 \ldots q_m$ of states is a *run* of A for a history $h = \sigma_0 \ldots \sigma_m$ if (i) $q_0 = q_{init}$, and (ii) for all i such that $0 \leq i \leq m$, we have $(q_i, \sigma_i, \gamma_i, q_{i+1}) \in \delta$ where γ is positive. The outcome of a TM captures the probability of the histories accepted by the TM. The *outcome* O_A of the TM A is a function $O_A : \Sigma^* \rightarrow [0, 1]$. Given a history h and a TM A, the outcome $O_A(h) = \gamma$ if there exists a set $\rho_1, \ldots, \rho_m$ of runs for h with probabilities $\gamma_1 \ldots \gamma_m$ such that $\sum_{0 \leq i \leq m} \gamma_i = \gamma$.

Safety and permissiveness of TM. We formalize the safety and permissiveness properties of TM, assuming that the commands in Σ occur atomically. A TM A is π-*safe* for a safety property $\pi \subseteq \Sigma^*$, if for every history $h \in \Sigma^*$ such that $O_A(h) > 0$, the history $h \in \pi$. In other words, a TM is safe with respect to a property if the outcome of the TM is positive only for histories that satisfy the property.

A TM A is *π-permissive* if for every history $h \in \pi$, we have $O_A(com(h)) = 1$. A TM A is *probabilistically π-permissive* if for every history $h \in \pi$, we have $O_A(com(h)) > 0$. Note that a deterministic TM is probabilistically permissive with respect to a property π if and only if it is permissive with respect to π. On the other hand, a randomized TM may not be π-permissive, while being probabilistically π-permissive.

We now show an example why the existing STMs are not permissive. Consider the history $h = ((\mathsf{write}, v_1), p_1), ((\mathsf{read}, v_1), p_2), ((\mathsf{write}, v_2), p_2)\ (\mathsf{commit}, p_1), (\mathsf{commit}, p_2)$. The history h is opaque, but its outcome is 0 for STMs like DSTM, TL2, and WSTM. In fact, it is easy to see that any TM that checks at the time of commit that the current values of the read variables are equal to the values read earlier (that is, validates the read set), cannot be permissive with respect to opacity. On the other hand, most of the existing TMs, for reasons of good overall performance, do exploit such a validation strategy to ensure safety. We now give algorithms that guarantee permissiveness in STMs.

3 Permissive Transactional Memories

We start with motivating the notion of permissiveness in TMs. TMs are online algorithms. That is, a TM decides whether to accept a statement or abort the corresponding transaction, only based on the statements seen so far. Let h be the history seen by the TM so far. Let x be the unfinished transaction of the process p in history h, and $h' = h \cdot (c, p)$. A TM A may decide to abort the transaction x in three scenarios:

- *Correctness.* The history h' is not opaque. In this case, any TM safe with respect to opacity needs to abort x. Thus, even a opacity-permissive TM aborts x.
- *Performance.* The history h' is opaque, but A is not sure whether $h \cdot (c, p)$ is opaque. For efficiency, A decides to abort x and retry it. In this case, a permissive TM will find out that h' is opaque, and thus not abort x. It is crucial for a permissive TM to efficiently compute, given that h is opaque, whether h' is opaque or not.
- *Priority.* The history h' is opaque, but the unfinished transaction y of some process $p' \neq p$ has to abort in every history extension of h'. The TM A prioritizes p' and hence aborts x, so that y retains the possibility to commit. In this case, we argue that as a TM does not know the future input after h, even after the TM aborts x, it is possible that due to conflicts, the TM has to abort y too. We believe that the idea of permissiveness can well be integrated with the notion of prioritizing certain processes, by making a process wait (rather than abort) for the commit of another process with higher priority.

Thus, the key to an efficient permissive TM for a given safety property lies in minimizing the cost of book-keeping required to check on the fly, whether the history produced by the TM satisfies the property.

3.1 A Deterministic Permissive Transactional Memory

In recent work [5], we described transactional memory specifications for strict serializability and opacity. The algorithm to obtain these specifications is based on the idea of conflicts graphs [10]. We refer to this algorithm as Spec in this paper. Spec assumes that the commands read, write, commit, and abort execute atomically. Our assumption is justified as we only analyze Spec, and do not build a deterministic permissive TM implementation from it.

The central idea of Spec [5] is the prohibited read and write sets, which allow us to remove finished transactions from the conflict graph. This keeps the conflict graph finite. Spec can be configured to obtain specifications for strict serializability or opacity. As the algorithm provides TM specifications for strict serializability (resp. opacity), it has outcome 1 for all and only those histories which are safe with respect to strict serializability (resp. opacity). In other words, we get a TM which is safe and permissive with respect to strict serializability or opacity.

The cost of the read operation in Spec for n processes is $O(n^2)$. In a practical scenario, most of the statements are reads, which in existing TMs, have a cost of $O(n)$. Some highly performance oriented TMs, like TL2, just require $O(1)$ for a read statement. So, we believe that the high cost of a read operation in Spec makes it a poor choice for a practical implementation. Hence, we do not create a TM implementation from Spec. We open the question whether there exists a deterministic permissive TM where the read operation is at most linear in the number of processes. Here, we now describe a randomized STM, called Adaptive Validation STM (AVSTM), which is probabilistically permissive w.r.t. opacity, and at the same time, performs well practically. The algorithm derives its name from the fact that transactions adaptively validate themselves. All transactions maintain a possible interval to serialize themselves, and at the time of commit, randomly choose a serialization point within that interval.

3.2 A Randomized Permissive Transactional Memory

Algorithm 1 shows the algorithm for AVSTM, which can be used for either strict serializability (SS-AVSTM) or opacity (OP-AVSTM). We do not assume that the commands in Σ are atomic in AVSTM. Only the reads and writes of global variables, and the CAS operation are treated as atomic. This allows us to use AVSTM as a real TM implementation, which we discuss in Section 3.5. The idea of AVSTM is to *randomly* choose, at the time of commit, a possible serialization point for every transaction. In principle, this allows transactions to probabilistically commit in the past, or in the future. For example, if two transactions x and y access the same variable, where x writes and y reads – even if x commits, y may commit afterwards if the transaction x chooses a serialization point in the future.

We first describe the variables and functions used in AVSTM. The function $rs : P \to 2^V$ is the read set and $ws : P \to 2^V$ is the write set of the processes. The function $rv : V \to \mathbb{N}$ gives the read version number and $wv : V \to \mathbb{N}$ gives the write version number of the variables. The function $Global : V \to \mathbb{N}$ is the global valuation of V, and $Local : P \times V \to \mathbb{N}$ is the process-local valuation of V.

Moreover, the functions $\texttt{min\_ser\_point} : P \rightarrow \mathbb{N}$ and $\texttt{max\_ser\_point} : P \rightarrow \mathbb{N}$ denote the minimum and maximum possible serialization points for the processes respectively. The function $\texttt{ser\_point} : P \rightarrow \mathbb{N}$ represents the tentatively chosen serialization point for the processes. AVSTM uses a variable $\texttt{commit\_num} \in \mathbb{N}$ that represents the sequence number of the commit being performed, and a variable $\texttt{owner} \in P \cup \{\perp\}$ that denotes the process which owns the current commit. When no process is committing, then the owner is $\perp$. The commit sequence number and the owner process are treated as an atomic pair so that they can be atomically manipulated. For the implementation, we encode `commit_num` and `owner` within the same variable as $\texttt{commit\_num} \cdot (n + 1) + i$, where n is the number of processes, and $i = 0$ if $\texttt{owner} = \perp$ and $i = \texttt{owner}$ otherwise. Also, every process uses a local variable $\texttt{lcn} \in \mathbb{N}$. The read set and write set of all processes are initially empty. Version numbers of all variables and serialization points of all processes are initially set to 0. When a transaction is started by process p, the variable `commit_num` is first read into the local variable `lcn`. Then, `lcn` is written into $\texttt{ser\_point}(p)$ and $\texttt{min\_ser\_point}(p)$, and $\texttt{max\_ser\_point}(p)$ is initialized to infinity. We now give an informal description of the algorithm.

- *Upon read of variable v by process p:* If v is in the write set of the process, then p returns the local value of v. Otherwise, the global value of v is copied as the local value of v for p. The value $\texttt{ser\_point}(p)$ is set to the maximum of the write version of v and the previous value of $\texttt{ser\_point}(p)$. If some process p' is committing a transaction that writes to v, then p first helps p' to commit. Then, v is added to the read set of p. If the global value of v has not changed during this read operation, then the value read is returned. Otherwise, the read is performed again. To ensure opacity, p also needs to check whether it has a positive interval to commit.
- *Upon write of variable v with value* `val` *by process p:* The variable v is added to the write set. The local value of v in process p is set to `val`. The value $\texttt{ser\_point}(p)$ is set to the maximum of its previous value, the write version of v and the read version of v.
- *Upon commit of a transaction by process p:* For a read only transaction, committing a transaction simply requires to set the read set to empty. Otherwise, the process p first aims to gain ownership of the commit. Until then, it helps other processes which are committing. p increments $\texttt{ser\_point}(p)$ to ensure that for every variable v in the write set, $\texttt{ser\_point}(p) \geq rv(v)$. If there is no positive interval to commit, then the unfinished transaction of p is aborted. Once p obtains ownership of the commit ($\texttt{owner} = p$), the process p starts the *helpCommit* for itself. The commit of p may also be helped by other processes. Finally, the read and write sets of p are reset to empty.
- *Upon abort of a transaction by process p:* The read and write sets are reset to empty.

The procedure $\textit{helpCommit}(\texttt{lcn}, p)$ is shown in Algorithm 2. It allows a process to help a process p in committing a transaction as follows. First of all, the read version of all variables in $rs(p)$ is incremented to the $\texttt{ser\_point}(p)$. Then, the write version of all variables in $ws(p)$ is also incremented to $\texttt{ser\_point}(p)$.

Then, the global value of these variables is updated. Then, the maximum serialization point of all processes whose read set intersects with the write set of p is set to `ser_point`(p). Similarly, the minimum serialization point of all processes whose write set intersects with the write set of p is set to `ser_point`(p). Last of all, the commit is made unowned (`owner` set to $\perp$). We note that the version numbers increase monotonically. Also the epoch based storage management ensures that a pointer to a memory location is not freed if any process holds a reference to the location. Thus none of the CAS operations in AVSTM suffer from the ABA problem. We now analyze the safety, liveness, and permissiveness of AVSTM.

3.3 Safety and Permissiveness of AVSTM

We note that the order of statements in the read and *helpCommit* procedure is essential for the safety and permissiveness of AVSTM. Upon a read of a variable v, the value of v is read (line 3) before the version number of v is read (line 4). Also, the read is successful only if the global value of v observed at the end of the read procedure (line 10) is same as the one read at the beginning of the read procedure (line 3). In the procedure *helpCommit*, for the variables being written, the write version number is updated (line 18) before the value of the variable is updated (line 19). Moreover, if the variable read by a process p is being written by some process p', then p first helps p' to commit. Together, this ensures that the version number read in line 4 of the read procedure corresponds exactly to that of the value read in line 3.

We prove the following properties of OP-AVSTM: safety and probabilistic permissiveness with respect to opacity. Similar proofs can be obtained for SS-AVSTM with respect to strict serializability. We also give an example of how OP-AVSTM works on a given opaque history.

Theorem 1. *OP-AVSTM is safe with respect to opacity.*

Proof. The procedure *helpCommit* allows many processes to commit a transaction for a particular process. But, the values and version numbers are committed exactly once, as the version numbers increase monotonically. Thus, the CAS does not suffer from the ABA problem. Also, the transaction x of process p commits only if it has a positive interval at the time of commit. For the sake of opacity, it is also ensured that when a variable is read, then the transaction has a positive interval to commit. A transaction x of process p successfully reads a variable only if `max_ser_point` $>$ `min_ser_point`. Note The order of operations ensures that if a newer value of a variable is read, then its corresponding version number is also read. When a transaction x of a process p starts, the variable `min_ser_point` and `ser_point` are set to the serialization point of the last committed transaction. This ensures that for non-overlapping transactions x and y, if y finishes before x starts, then x serializes after y. Moreover, the transaction x of process p can commit only if `max_ser_point` $>$ `min_ser_point` at the time of commit. Also, for every variable $v \in rs(p)$, when v is read by transaction x, the write version $wv(v) <$ `min_ser_point`. Moreover, after v is read, no transaction that writes to

Algorithm 1. OP-AVSTM (SS-AVSTM obtained by removing line marked OP)

```
Upon read of variable $v$ by process $p$
  if $v \in ws(p)$ then return $Local(p, v)$
  do forever
    $Local(p, v) := Global(v)$
    $\texttt{local\_write\_version} := wv(v)$
    $\texttt{ser\_point}(p) := \max(\texttt{ser\_point}(p), \texttt{local\_write\_version})$
    $\langle \texttt{lcn}, p' \rangle := \langle \texttt{commit\_num}, \texttt{owner} \rangle$
    if $p' \neq \perp$ and $v \in ws(p')$ then $helpCommit(\texttt{lcn}, p')$
    if $\max(\texttt{min\_ser\_point}(p), \texttt{ser\_point}(p)) \geq \texttt{max\_ser\_point}(p)$ then abort (OP)
    if $Local(p, v) = Global(v)$ then break
  $rs(p) := rs(p) \cup \{v\}$
  return $Local(p, v)$

Upon write of variable $v$ with value $\texttt{val}$ by process $p$
  $\texttt{local\_write\_version} := wv(v)$
  $\texttt{local\_read\_version} := rv(v)$
  $\texttt{ser\_point}(p) := \max(\texttt{ser\_point}(p), \texttt{local\_write\_version}, \texttt{local\_read\_version})$
  $ws(p) := ws(p) \cup \{v\}$
  $Local(p, v) := \texttt{val}$

Upon commit by process $p$
  if $ws(p) = \emptyset$ then $rs(p) := \emptyset$;    return
  do forever
    $\langle \texttt{lcn}, p' \rangle := \langle \texttt{commit\_num}, \texttt{owner} \rangle$
    while $p' \neq \perp$ do
      $helpCommit(\texttt{lcn}, p')$
      $\langle \texttt{lcn}, p' \rangle := \langle \texttt{commit\_num}, \texttt{owner} \rangle$
    for each variable $v \in ws(p)$ do
      $\texttt{ser\_point}(p) := \max(\texttt{ser\_point}(p), rv(v))$
    $\texttt{ser\_point}(p) := \max(\texttt{min\_ser\_point}(p), \texttt{ser\_point}(p))$
    if $\texttt{ser\_point}(p) \geq \texttt{max\_ser\_point}(p)$ then abort
    $\texttt{ser\_point}(p) :=$ a random number between $\texttt{ser\_point}(p)$ and $\texttt{max\_ser\_point}(p)$
    $\texttt{new\_cp} := \texttt{lcn} + 1$
    if $CAS(\langle \texttt{commit\_num}, \texttt{owner} \rangle, \langle \texttt{lcn}, \perp \rangle, \langle \texttt{new\_cp}, p \rangle) = \langle \texttt{lcn}, \perp \rangle$ then break
  $helpCommit(\texttt{new\_cp}, p)$
  $rs(p) := \emptyset$;    $ws(p) := \emptyset$

Upon abort by process $p$
  $rs(p) := \emptyset$;    $ws(p) := \emptyset$
```

v commits with a serialization point less than max_ser_point. Similarly, the variables in $ws(p)$ have not been written later than min_ser_point. Thus, the transaction x sees a state of the variables, consistent in the interval min_ser_point to max_ser_point. As this holds for every committing, aborting, and unfinished transaction of every process, opacity is guaranteed by OP-AVSTM. □

Theorem 2. *OP-AVSTM is probabilistically permissive with respect to opacity.*

Algorithm 2. *helpCommit*$(\texttt{lcn}, p)$

```
local_ser := ser_point($p$)
local_read_set := $rs(p)$
local_write_set := $ws(p)$
if $\langle$lcn, $p\rangle \neq \langle$commit_num, owner$\rangle$ then return
for each variable $v \in$ local_read_set
  local_read_version := $rv(v)$
  if local_read_version < local_ser then
    $CAS(rv(v)$, local_read_version, local_ser$)$
for each variable $v \in$ local_write_set
  local_write_version := $wv(v)$
  local_read_version := $rv(v)$
  old_val := $Global(v)$
  new_val := $Local(p, v)$
  if $\langle$lcn, $p\rangle \neq \langle$commit_num, owner$\rangle$ then return
  if local_read_version < local_ser then
    $CAS(rv(v)$, local_read_version, local_ser$)$
  if local_write_version < local_ser then
    $CAS(wv(v)$, local_write_version, local_ser$)$
  $CAS(Global(v)$, old_val, new_val$)$
for all processes $p' \neq p$ do
  for each variable $v \in rs(p') \cap ws(p)$ do
    local_max := max_ser_point($p'$)
    if local_ser < local_max then
      $CAS($max_ser_point($p'$), local_max, local_ser$)$
  for each variable $v \in ws(p') \cap ws(p)$ do
    local_min := min_ser_point($p'$)
    if local_ser > local_min then
    $CAS($min_ser_point($p'$), local_min, local_ser$)$
$CAS(\langle$commit_num, owner$\rangle, \langle$lcn, $p\rangle, \langle$lcn, $\perp\rangle)$
```

Proof sketch. For any opaque history h, every transaction serializes at some point within its lifetime. We can thus mark the serialization point of every transaction in h. As the length of h is finite, there is positive probability that every transaction in h chooses the required serialization point in OP-AVSTM. Hence, h is accepted by OP-AVSTM with positive probability. □

Example of probabilistic permissiveness of OP-AVSTM. Consider an opaque history h = ((write, v_1), p_1), ((write, v_2), p_2), ((read, v_1), p_2) (commit, p_1), (commit, p_2). First, the process p_1 writes to v_1. Then, the process p_2 writes to v_2. Then p_2 reads the variable v_1. When p_1 commits, let the chosen serialization point be c. The write version of v_1 is seen as the initial value 0 by p_2 as p_2 reads before p_1 updates the write version number of v_1 to c. In this case, it is ensured that the maximum serialization point of p_2 is also set to c. Now, when p_2 commits, it can choose a serialization point less than c and successfully commit. Even if the commands in Σ are not considered to be atomic, a similar argument can be made about the probabilistic permissiveness of AVSTM.

3.4 Liveness of AVSTM

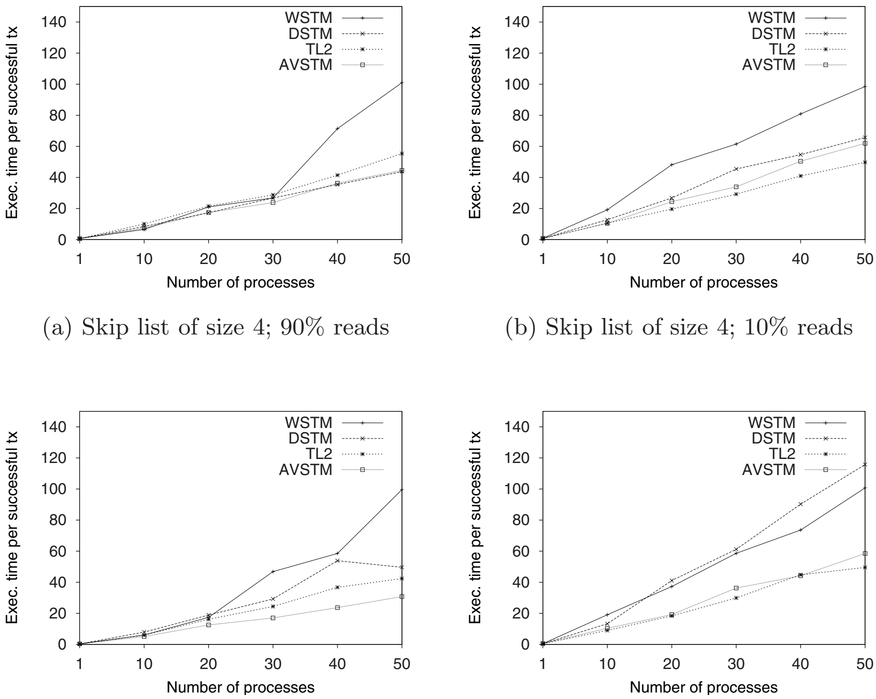
Different notions of liveness have been proposed for transactional memories. A TM is *obstruction free* if for every infinite history h produced by the TM, if some process $p \in P$ takes infinitely many steps in isolation in h, then p commits infinitely often. A TM is *lock free* if every infinite history h produced by the TM contains infinitely many commits. A TM is *wait free* if every infinite history h produced by the TM contains infinitely many commits for every process $p \in P$.

Theorem 3. *The algorithm AVSTM is lock free.*

Proof. We prove the theorem by contradiction. Let there exist a time t after which no process commits a transaction. We first note that no operation in AVSTM is blocking. Thus, every process, when scheduled, executes a statement of the algorithm. Also, transactions are of finite size. We note that a process can loop in the read or commit operation only if some other process performs a successful commit. So, if there is no commit of any transaction after time t, then there must be an infinite number of aborts of transactions after t. A transaction aborts only if it cannot find an interval to commit. As the maximum serialization point is initially ∞ when a transaction starts, the only way that a transaction does not find an interval to commit is when the maximum serialization point is set to a finite value. This occurs only within the procedure *helpCommit*. We also note that the maximum serialization point is changed in the procedure *helpCommit* at most k times, where k is the number of variables. Moreover, the procedure *helpCommit* for a particular transaction can be executed at most once by every process. Thus, there exists a time $t' \geq t$ such that the maximum serialization point of any process does not change after t'. Thus, there is no abort after t'. This contradicts our assumption. Hence, AVSTM guarantees lock freedom. □

4 Implementation and Experiments

To evaluate the practical importance of the notion of permissiveness, we implemented AVSTM within the LibLTX package [3]. The LibLTX package includes an implementation of DSTM with the Polka contention manager (which typically gives best performance results to DSTM [12]) and WSTM [3]. We also implemented an STM based on TL2 [1] within the LibLTX package. We integrated the storage management of AVSTM with the epoch-based garbage collector [2], where a memory pointer is not freed if any process holds a reference to it. This allows the simple use of CAS in the procedure *helpCommit*, without causing any ABA problem. We compare the performance of AVSTM configured for opacity, with DSTM, WSTM, and TL2 in high contention scenarios, that is, when many processes are accessing a small set of shared variables. We experimented on a quad dual-core (8 processors) 2.8 GHz server with 16 GB RAM. Executing a large number of processes on a small number of processors creates a practical scenario, where a process holding a lock may get descheduled, and the liveness property of lock freedom becomes critical for performance. We use two different



(a) Skip list of size 4; 90% reads

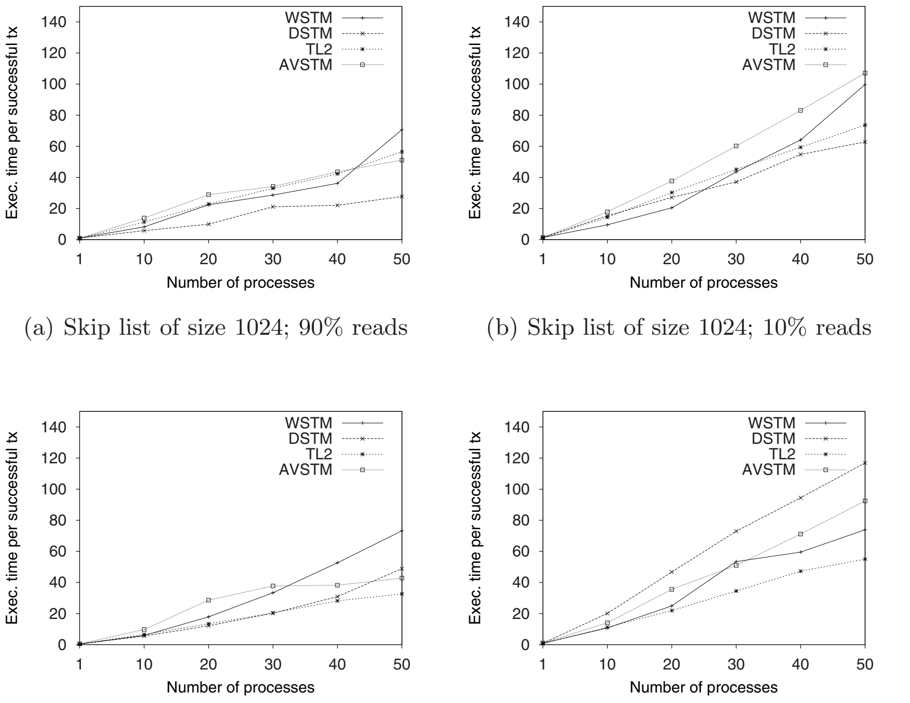
(b) Skip list of size 4; 10% reads

(c) Red-black tree of size 4; 90% reads

(d) Red-black tree of size 4; 10% reads

Fig. 1. Performance results on benchmarks in high contention. The execution time is measured in micro-seconds.

benchmarks: skip lists and red-black trees. For both of these benchmarks, we experiment with a data size of 4 items, and with two different types of workloads, one is read dominated with 90% reads, and other is write dominated with 10% reads. The results (Figure 1) show that AVSTM always outperforms DSTM and WSTM by upto 40% in high contention scenarios. Also, in our experiments, AVSTM outperforms TL2 in high contention when the workload is read dominated. We admit that the official implementation of TL2 will perform better than our version of TL2, but our experiments do show that AVSTM is comparable to the existing TM algorithms in the literature. We also note that AVSTM is lock free while TL2 is not. TL2 performs better than AVSTM in high contention when the workload is write dominated. This, we believe, has an interesting theoretical explanation. When the workload is write dominated, even AVSTM has to abort very often in order to be safe. This gives TL2 an advantage over AVSTM as TL2 uses a simpler invalidation scheme. On the other hand, when the workload is read dominated, AVSTM has to abort less often than TL2. This is because TL2 aborts many opaque histories, whereas the book-keeping of AVSTM helps it to avoid some redundant aborts.



(a) Skip list of size 1024; 90% reads

(b) Skip list of size 1024; 10% reads

(c) Red-black tree of size 1024; 90% reads

(d) Red-black tree of size 1024; 10% reads

Fig. 2. Performance results on benchmarks in low contention. The execution time is measured in micro-seconds.

Permissiveness in AVSTM does come with a price. As we saw in the algorithm for AVSTM, different processes access the global data concurrently. Thus, AVSTM incurs high overhead due to poor cache performance. To evaluate this overhead, we evaluate the performance of different STMs in a low contention scenario of 1024 data items. The results are shown in Figure 2. We find that although AVSTM is not as good as other TMs, it does not yet pay an overwhelming penalty for the book-keeping needed to achieve permissiveness. On average, AVSTM performs 20-30% worse than existing TM algorithms. We now discuss how AVSTM can be used as a fallback mechanism with TL2 to boost performance and progress guarantees in high contention scenarios.

Combining AVSTM with TL2

Our experiments gave us a clue on how we can make the best use of AVSTM, which has excellent performance in high contention scenarios. AVSTM also gives the progress guarantee of lock freedom. On the other hand, AVSTM is not a good performer in low contention scenarios. We propose to use AVSTM as a fallback mechanism, that is, it be combined with other STMs to get good performance and guaranteed progress when high contention brings both performance and progress at risk. We discuss here, as an example, how AVSTM can be combined with TL2.

A TM that is running in low contention uses the TL2 mode. A process that faces a series of aborts changes the mode from TL2 to AVSTM. Now, other processes which are not in the final phase of committing (once the locks have been acquired and the read set has been validated) can safely change their mode to AVSTM by observing the change of mode, for example, during the validation phase in the commit. A process which is in the final phase of commit has to be dealt with properly. When a process p_1 running in AVSTM mode tries to access a variable which is locked in TL2 mode by process p_2, following cases may occur:

- If the process p_1 observes that the process p_2 is not yet in the final phase of commit (by reading a flag for example), then p_1 can safely assume that p_2 will not write to the variable.
- If the process p_1 observes that p_2 is in the final phase of commit, then p_1 helps p_2 to commit. For this to work properly, processes running in TL2 mode should commit using a compare and swap (CAS). As the write set is generally small, this introduces negligible overhead.

5 Concluding Remarks

We presented a notion of permissiveness in TMs. As liveness guarantees are hard to provide in TMs, we believe that permissiveness can be an interesting, complementary metric while evaluating TMs theoretically. We discussed the high performance cost of a deterministic permissive STM due to the overhead of bookkeeping. We presented a randomized STM, AVSTM, that is probabilistically permissive for strict serializability and opacity. The randomization allows probabilistic decisions, and hence lowers the cost. We showed the practical importance of permissiveness by experiments that demonstrate how AVSTM outperforms existing STMs in high-contention scenarios. We also provided a strategy to use the randomized permissive STM in combination with TL2 to boost performance and progress guarantees.

Future work. We look ahead to prove a lower bound on the time complexity of an opacity-permissive deterministic TM, supporting the intuition that a practical deterministic TM cannot be opacity-permissive. We also plan to extend the formalism of permissiveness to *quantify* the amount of permissiveness of a TM. For example, a TM is k-permissive (where $0 \leq k \leq 1$) with respect to opacity if on every opaque history, the ratio of unnecessarily aborting transactions to the total number of transactions is at most k. This allows us to compare even non-permissive TMs by their degree of permissiveness. Also, as in other STM formalizations [4,5,13], we assumed the atomicity of individual commands (read, write, commit). Generally, the commit is not atomic and it would be interesting to revisit the notion of permissiveness with a finer grained model in mind.

References

1. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: DISC, pp. 194–208. Springer, Heidelberg (2006)
2. Fraser, K.: Practical Lock Freedom. PhD thesis, Computer Laboratory, University of Cambridge (2003)
3. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Trans. Comput. Syst. (2007)
4. Guerraoui, R., Henzinger, T.A., Jobstmann, B., Singh, V.: Model checking transactional memories. In: PLDI, pp. 372–382. ACM Press, New York (2008)
5. Guerraoui, R., Henzinger, T.A., Singh, V.: Nondeterminism and completeness in transactional memories. In: CONCUR. Springer, Heidelberg (2008)
6. Guerraoui, R., Herlihy, M., Kapałka, M., Pochon, B.: Robust contention management in software transactional memory. In: SCOOL (October 2005)
7. Guerraoui, R., Kapałka, M.: On the correctness of transactional memory. In: PPoPP. ACM Press, New York (2008)
8. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA, pp. 388–402 (2003)
9. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101. ACM Press, New York (2003)
10. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM, 631–653 (1979)
11. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: DISC, pp. 284–298. Springer, Heidelberg (2006)
12. Scherer, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC, pp. 240–248. ACM Press, New York (2005)
13. Scott, M.L.: Sequential specification of transactional memory semantics. In: ACM SIGPLAN WTC (2006)

The Synchronization Power of Coalesced Memory Accesses

Phuong Hoai Ha[1], Philippas Tsigas[2], and Otto J. Anshus[1]

[1] University of Tromsø, Department of Computer Science, Faculty of Science,
NO-9037 Tromsø, Norway
{phuong,otto}@cs.uit.no

[2] Chalmers University of Technology, Department of Computer Science and
Engineering, SE-412 96 Göteborg, Sweden
tsigas@chalmers.se

Abstract. Multicore processor architectures have established themselves as the new generation of processor architectures. As part of the one core to many cores evolution, memory access mechanisms have advanced rapidly. Several new memory access mechanisms have been implemented in many modern commodity multicore processors. Memory access mechanisms, by devising how processing cores access the shared memory, directly influence the synchronization capabilities of the multicore processors. Therefore, it is crucial to investigate the synchronization power of these new memory access mechanisms.

This paper investigates the synchronization power of coalesced memory accesses, a family of memory access mechanisms introduced in recent large multicore architectures like the CUDA graphics processors. We first design three memory access models to capture the fundamental features of the new memory access mechanisms. Subsequently, we prove the exact synchronization power of these models in terms of their consensus numbers. These tight results show that the coalesced memory access mechanisms can facilitate strong synchronization between the threads of multicore processors, without the need of synchronization primitives other than reads and writes. In the case of the contemporary CUDA processors, our results imply that the coalesced memory access mechanisms have consensus numbers up to sixteen.

1 Introduction

One of the fastest evolving multicore architectures is the graphics processor one. The computational power of graphics processors (GPUs) doubles every ten months, surpassing the Moore's Law for traditional microprocessors [13]. Unlike previous GPU architectures, which are single-instruction multiple-data (SIMD), recent GPU architectures (e.g. Compute Unified Device Architecture (CUDA) [2]) are single-program multiple-data (SPMD). The latter consists of multiple SIMD multiprocessors of which each, at the same time, can execute a different instruction. This extends the set of applications on GPUs, which are no longer restricted

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 320–334, 2008.

to follow the SIMD-programming model. Consequently, GPUs are emerging as powerful computational co-processors for general-purpose computations.

Along with their advances in computational power, GPUs memory access mechanisms have also evolved rapidly. Several new memory access mechanisms have been implemented in current commodity graphics/media processors like the Compute Unified Device Architecture (CUDA) [2] and Cell BE architecture [1]. For instance, in CUDA, single-word write instructions can write to words of different size and their size (in bytes) is no longer restricted to be a power of two [2]. Another advanced memory access mechanism implemented in CUDA is the coalesced global memory access mechanism. The simultaneous global memory accesses by each thread of a SIMD multiprocessor, during the execution of a single read/write instruction, are coalesced into a *single* aligned memory access if the simultaneous accesses follow the coalescence constraint [2]. The access coalescence takes place even if some of the threads do not actually access memory. It is well-known that memory access mechanisms, by devising how processing cores access the shared memory, directly influence the synchronization capabilities of multicore processors. Therefore, it is crucial to investigate the synchronization power of the new memory access mechanisms.

Research on the synchronization power of memory access operations (or objects) in conventional architectures has received a great amount of attention in the literature. The synchronization power of memory access objects/mechanisms is conventionally determined by their consensus-solving ability, namely their consensus number [10]. The *consensus number* of an object type is either the maximum number of processes for which the consensus problem can be solved using only objects of this type and registers, or infinity if such a maximum does not exist. For hard real-time systems, it has been shown that any object with consensus number n is universal[1] for any numbers of processes running on n processors [14]. For systems that allow processes to simultaneously access m objects of type T in one atomic operation (or multi-object operation), upper and lower bounds on the consensus number of the multi-object called type T^m have been provided for the base type T with *consensus number greater than or equal to two* [4,11,16]. In the case of registers (which have consensus number one), the m-register assignment, which allows processes to write to m *arbitrary* registers atomically, has been proven to have consensus number $(2m-2)$, for $m > 1$ [10].

Note that the aforementioned CUDA coalesced memory accesses are neither the atomic m-register assignment [10] nor the multi-object types [4,11,16]. They are not the atomic m-register assignment since they do not allow processes to atomically write to m *arbitrary* memory words; instead, processes can atomically write to m memory words only if the m memory words are located within an aligned size-bounded memory portion (i.e. memory alignment restriction) (cf. Section 2). The CUDA coalesced memory accesses are not the multi-object type since their base object type T is the conventional memory word, which has *consensus number less than two.*

[1] An object is *universal* in a system of n processes iff it has a consensus number not lower than n.

This paper investigates the consensus number of the new memory access mechanisms implemented in current graphics processor architectures. We first design three new memory access models to capture the fundamental features of the new memory access mechanisms. Subsequently we prove the exact synchronization power of these models in terms of their respective consensus number. These tight results show that the new memory access mechanisms can facilitate strong synchronization between the threads of multicore processors, without the need of synchronization primitives other than reads and writes.

We first design a new memory access model, the *svword* model where *svword* stands for the *s*ize-*v*arying *word* access, the first of the two aforementioned advanced memory access mechanisms implemented in CUDA. Unlike single-word assignments in conventional processor architectures, the new single-word assignments can write to words of size b (in bytes), where b can vary from 1 to an upper bound B and b is no longer restricted to be a power of 2 (e.g. type *float3* in [2]). By carefully choosing b for the single-word assignments, we can *partly* overlap the bytes written by two assignments, namely each of the two assignments has some byte(s) that is not overwritten by the other overlapping assignment (cf. Figure 1(a) for an illustration). Note that words of different size must be aligned from the address base of the memory. This memory alignment constraint prevents single-word assignments in conventional architectures from partly overlapping each other since the word-size is restricted to be a power of two. On the other hand, since the new single-word assignment can write to a subset of bytes of a *big* word (e.g. up to 16 bytes) and leave the other bytes of the word intact, the size of values to be written becomes a significant factor. The assignment can atomically write B values of size 1 (instead of just one value of size B) to B consecutive memory locations. The observation has motivated us to develop the *svword* model.

Inspired by the coalesced memory accesses, the second of the aforementioned advanced memory access mechanisms, we design two other models, the *aiword* and *asvword* models, to capture the fundamental features of the mechanism. The mechanism coalesces simultaneous read/write instructions by each thread of a SIMD multiprocessor into a *single* aligned memory access even if some of the threads do not actually access memory [2]. This allows each SIMD multiprocessor (or process) to atomically write to an arbitrary subset of the aligned memory units that can be written by a single coalesced memory access. We generally model this mechanism as an *a*ligned-*i*nconsecutive-*word* access, *aiword*, in which the memory is aligned to A-unit words and a single-word assignment can write to an arbitrary non-empty subset of the A units of a word. Note that the single-*aiword* assignment is not the atomic m-register assignment [10] due to the memory alignment restriction[2]. Our third model, *asvword*, is an extension of the second model *aiword* in which *aiword*'s A memory units are now replaced by A *svword*s of the same size b. This model is inspired by the fact that the read/write instructions of different coalesced global memory accesses can access words of different size [2].

[2] In this paper, we use term "single" in *single-*word assignment* when we want to emphasize that the assignment is not the *multiple* assignment [10].

The contributions of this paper can be summarized as follows:

- We develop a general memory access model, the *svword* model, to capture the fundamental features of the size-varying word accesses. In this model, a single-word assignment can write to a word comprised of b *consecutive* memory units, where b can be any integer between 1 and an upper bound B. We prove that the single-*svword* assignment has consensus number 3, $\forall B \geq 5$, and that consensus number 3 is also the upper bound of consensus numbers of the single-*svword* assignment $\forall B \geq 2$. We also introduce a technique to minimize the size of (proposal) values in consensus algorithms, which allows a *single-word* assignment to write many values atomically and handle the consensus problem for several processes (cf. Section 3).
- We develop a general memory access model, the *aiword* model, to capture the fundamental features of the coalesced memory accesses. The second model is an aligned-inconsecutive-word access model in which the memory is aligned to A-unit words and a single-word assignment can write to an arbitrary non-empty subset of the A units of a word. We present a wait-free consensus algorithm for $N = \lfloor \frac{A+1}{2} \rfloor$ processes using only single-*aiword* assignments and subsequently prove that the single-*aiword* assignment has consensus number exactly $N = \lfloor \frac{A+1}{2} \rfloor$ (cf. Section 4).
- We develop a general memory access model, *asvword*, to capture the fundamental features of the *combination* of the size-varying word accesses and the coalesced memory accesses. The third model is an extension of the second model *aiword* in which *aiword*'s A units are A *svword*s of the same size $b, b \in \{1, B\}$ (cf. Section 5). We prove that the consensus number of the single-*asvword* assignment is exactly N, where

$$N = \begin{cases} \frac{AB}{2}, & \text{if } A = 2tB, t \in \mathbb{N}^* \text{ (positive integers)} \\ \frac{(A-B)B}{2} + 1, & \text{if } A = (2t+1)B, t \in \mathbb{N}^* \\ \lfloor \frac{A+1}{2} \rfloor, & \text{if } B = tA, t \in \mathbb{N}^* \end{cases} \quad (1)$$

  In the case of the contemporary CUDA processors (with compute capability up to 1.1) in which $A = 16$ and $B = 2$, the consensus number of the *asvword* model is sixteen.

The rest of this paper is organized as follows. Section 2 presents the three new memory access models. Sections 3, 4 and 5 present the exact consensus numbers of the first, second and third models, respectively.

Due to space limitations, we present here only intuitions behind the consensus number results. Complete proofs of the results can be found in the full version of this paper [9].

2 Models

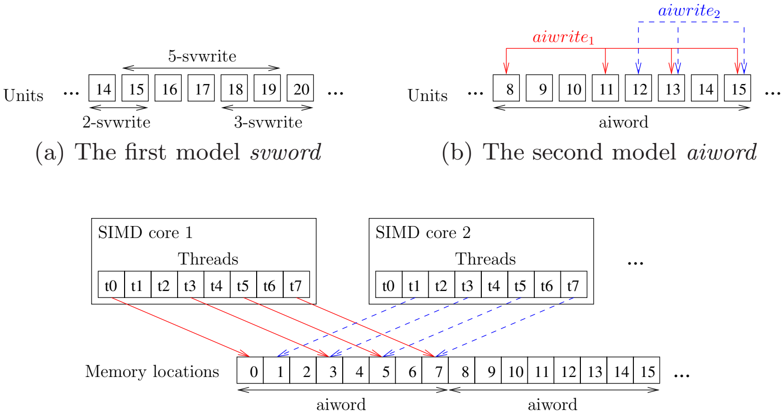
Before describing the details of each of the three new memory access models, we present the common properties of all these three models. The shared memory in the three new models is sequentially consistent [3,12], which is weaker than the linearizable one [5] assumed in most of the previous research on the synchronization power of the conventional memory access models [10]. Processes are asynchronous. The new models use the conventional 1-dimensional memory address space. In these models, one memory *unit* is a *minimum* number of consecutive bytes/bits which a basic read/write operation can atomically read from/write to (without overwriting other unintended bytes/bits). These memory models address individual memory units. Memory is organized so that a group of n consecutive memory units called *word* can be stored or retrieved in a *single* basic write or read operation, respectively, and n is called *word size*. Words of size n must always start at addresses that are multiples of n, which is called *alignment restriction* as defined in the conventional computer architecture.

The first model is a *s*ize-*v*arying-*word* access model (*svword*) in which a single read/write operation can atomically read from/write to a word consisting of b *consecutive* memory units, where b can be any integer between 1 and an upper bound B and is called *svword size*. The upper bound B is the maximum number of *consecutive* units which a basic read/write operation can atomically read from/write to. *Svword*s of size b must always start at addresses that are multiples of b due to the memory alignment restriction. We denote b-*svword* to be an *svword* consisting of b units, b-*svwrite* to be a b-*svword* assignment and b-*svread* to be a b-*svword* read operation. Reading a unit U is denoted by 1-*svread*(U) or just by U for short. This model is inspired by the CUDA graphics processor architecture in which basic read/write operations can atomically read from/write to words of different size (cf. types *float1, float2, float3* and *float4* in [2], Section 4.3.1.1). Figure 1(a) illustrates how 2-*svwrite*, 3-*svwrite* and 5-*svwrite* can partly overlap their units with addresses from 14 to 20, with respect to the memory alignment restriction.

The second model is an *a*ligned-*i*nconsecutive-*word* access model (*aiword*) in which the memory is aligned to A-unit words and a single read/write operation can atomically read from/write to an arbitrary non-empty subset of the A units of a word, where A is a constant. *Aiword*s must always start at addresses that are multiples of A due to the memory alignment restriction. We denote A-*aiword* to be an *aiword* consisting of A units, A-*aiwrite* to be an A-*aiword* assignment and A-*airead* to be an A-*aiword* read operation. Reading only one unit U (using *airead*) is denoted by U for short. In the *aiword* model, an *aiwrite* operation executed by a process cannot *atomically* write to units located in different *aiword*s due to the memory alignment restriction.

Figure 1(b) illustrates the *aiword* model with $A = 8$ in which the *aiword* consists of eight consecutive units with addresses from 8 to 15. Unlike in the *svword* model, the assignment in the *aiword* model can atomically write to *inconsecutive* units of the eight units: *aiwrite*$_1$ atomically writes to four units $8, 11, 13$ and 15; *aiwrite*$_2$ writes to three units $12, 13$ and 15.

This model is inspired by the coalesced global memory accesses in the CUDA architecture [2]. The CUDA architecture can be generalized to an abstract model



(a) The first model *svword*

(b) The second model *aiword*

(c) The coalesced memory access

Fig. 1. Illustrations for the first model, *s*ize-*v*arying-*word* access (*svword*), the second model, *a*ligned-*i*nconsecutive-*word* access (*aiword*) and the coalesced memory access

of a MIMD[3] chip with multiple SIMD cores sharing memory. Each core can process A threads simultaneously in a SIMD manner, but different cores can simultaneously execute different instructions. The instance of a program that is being sequentially executed by one SIMD core is called *process*. Namely, each process consists of A parallel threads that are running in SIMD manner. The process accesses the shared memory using the CUDA memory access models. In CUDA, the simultaneous global memory accesses by each thread of a SIMD core during the execution of a single read/write instruction can be coalesced into a *single* aligned memory access. The coalescence happens even if some of the threads do not actually access memory (cf. [2], Figure 5-1). This allows a SIMD core (or a process consisting of A parallel threads running in a SIMD manner) to atomically access multiple memory locations that are not at consecutive addresses.

Figure 1(c) illustrates the coalesced memory access, where $A = 8$. The left SIMD core can write atomically to four memory locations 0, 3, 5 and 7 by letting only four of its eight threads, $t0, t3, t5$ and $t7$, simultaneously execute a write operation (i.e. divergent threads). The right SIMD core can write atomically to its own memory location 1 and shared memory locations 3, 5 and 7 by letting only four threads $t1, t3, t5$ and $t7$ simultaneously execute a write operation. Note that the CUDA architecture allows threads from different SIMD cores to communicate through the global shared memory [7].

The third model is a coalesced memory access model (*asvword*), an extension of the second model *aiword* in which *aiword*'s A units are now replaced by A *svword*s of the same size $b, b \in [1, B]$. Namely, the second model *aiword* is a special case of the third model *asvword* where $B = 1$. This model is inspired by the fact

[3] MIMD: Multiple-Instruction-Multiple-Data.

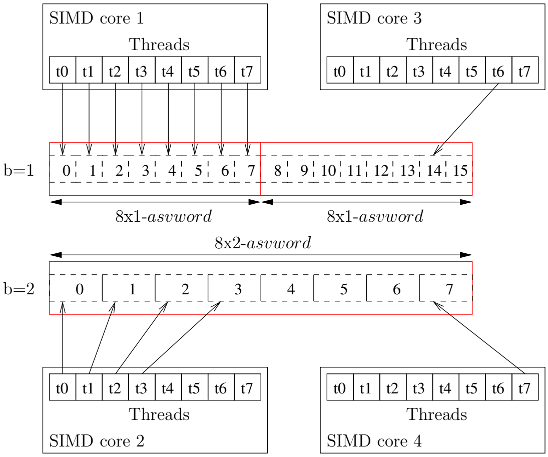


Fig. 2. An illustration for the *asvword* model

that in CUDA the read/write instructions of different coalesced global memory accesses can access words of different size. Let Axb-*asvword* be the *asvword* that is composed of A *svword*s of which each consists of b memory units. Axb-*asvword*s whose size is $A \cdot b$ must always start at addresses that are multiples of $A \cdot b$ due to the memory alignment restriction. We denote Axb-*asvwrite* to be an Axb-*asvword* assignment and Axb-*asvread* to be an Axb-*asvword* read operation. Reading only one unit U (using Ax1-*asvread*) is denoted by U for short. Due to the memory alignment restriction, an Axb-*asvwrite* operation cannot atomically write to b-*svword*s located in *different* Axb-*asvword*s. Since in reality A and B are a power of 2, in this model we assume that either $B = k \cdot A, k \in \mathbb{N}^*$ (in the case of $B \geq A$) or $A = k \cdot B, k \in \mathbb{N}^*$ (in the case of $B < A$). (At the moment, CUDA supports the *atomic* coalesced memory access to only words of size 4 and 8 bytes (i.e. only *svword*s consisting of 1 and 2 *units* in our definition), cf. Section 5.1.2.1 in [2]). For the sake of simplicity, we assume that $b \in \{1, B\}$ holds. A more general model with $b = 2^c, c = 0, 1, \cdots, \log_2 B$, can be established from this model. Since both Ax1-*asvword*s and AxB-*asvword*s are aligned from the address base of the memory space, any AxB-*asvword* can be aligned with B Ax1-*asvword*s as shown in Figure 2.

Figure 2 illustrates the *asvword* model in which each dash-dotted rectangle/square represents an *svword* and each red/solid rectangle represents an *asvword* composed of eight *svword*s (i.e. $A = 8$). The two rows show the memory alignment corresponding to the size b of *svword*s, where b is 1 or 2 (i.e. $B = 2$), on the same sixteen consecutive memory units with addresses from 0 to 15. An *asvwrite* operation can atomically write to some or all of the eight *svword*s of an *asvword*. Unlike the *aiwrite* assignment in the second model, which can atomically write to at most 8 units (or A units), the *asvwrite* assignment in the third model can atomically write to 16 units (or $A \cdot B$ units) using a single 8x2-*asvwrite* operation (i.e. write to the whole set of eight 2-*svword*s, cf. row $b = 2$). For an

8x1-*asvword* on row $b = 1$, there are two methods to update it atomically using the *asvwrite* operation: i) writing to the whole set of eight 1-*svword*s using a single 8x1-*asvwrite* (cf. SIMD core 1) or ii) writing to a subset consisting of four 2-*svword*s using a single 8x2-*asvwrite* (cf. SIMD core 2). However, if only one of the eight units of an 8x1-*asvword* (e.g. unit 14) needs to be updated and the other units (e.g. unit 15) must remain untouched, the only possible method is to write to the unit using a single 8x1-*asvwrite* (cf. SIMD core 3). The other method, which writes to one 2-*svword* using a single 8x2-*asvwrite*, will have to overwrite another unit that is required to stay untouched (cf. SIMD core 4).

Terminology. This paper uses the conventional terminology from bivalency arguments [8,10,15]. The *configuration* of an algorithm at a moment in its execution consists of the state of every shared object and the internal state of every process. A configuration is *univalent* if all executions continuing from this configuration yield the same consensus value and *multivalent* otherwise. A configuration is *critical* if the next operation op_i by any process p_i will carry the algorithm from a *multivalent* to a *univalent* configuration. The operations op_i are called *critical operations*. The *critical value* of a process is the value that would get decided if that process takes the next step after the critical configuration.

3 Consensus Number of the *Svword* Model

Before proving the consensus number of the single-*svword* assignment, we present the essential features of any wait-free consensus algorithm $\mathcal{ALG}$ for N processes using only single-**word* assignments and registers, where **word* can be *svword, aiword* or *asvword*. It has been proven that such an algorithm must have a critical configuration, C_0, and the next assignment op_i (i.e. the critical operation) by each process p_i must write to the same object $\mathcal{O}$ [10]. The object $\mathcal{O}$ consists of *memory units*.

Lemma 1. *The critical assignment op_i by each process p_i must atomically write to*

- *a "single-writer" unit (or 1W-unit for short) u_i written only by p_i and*
- *"two-writer" units (or 2W-units for short) $u_{i,j}$ written only by two processes p_i and p_j, where p_j's critical value is different from p_i's, $\forall j \neq i$.*

Proof. The proof is similar to the bivalency argument of Theorem 13 in [10]. □

In this section, we first present a wait-free consensus algorithm for 3 processes using only the single-*svword* assignment with $B \geq 5$ and registers. Then, we prove that we cannot construct any wait-free consensus algorithms for more than 3 processes using only the single-*svword* assignment and registers regardless of how large B is.

The new wait-free consensus algorithm SVW_Consensus is presented in Algorithm 1. The main idea of the algorithm is to utilize the size-variation feature of the *svwrite* operation. Since b-*svwrite* can atomically write b values of

Algorithm 1. SVW_CONSENSUS(buf_i: proposal) invoked by process $p_i, i \in \{0,1,2\}$

$PROPOSAL[0,1,2]$: contains proposals of 3 processes. $PROPOSAL[i]$ is only written by process p_i but can be read by all processes.
WR_1 = set $\{u_0, u_1, u_2\}$ of *units*: initialized to $Init$ and used in the first phase. $WR_1[0]$ and $WR_1[2]$ are 1W-units written only by p_0 and p_1, respectively. $WR_1[1]$ is a 2W-unit written by both processes
WR_2 = set $\{v_0, \cdots, v_4\}$ of *units*: initialized to $Init$ and used in the second phase. $WR_2[0]$, $WR_2[2]$ and $WR_2[4]$ are 1W-units written only by p_0, p_2 and p_1, respectively. $WR_2[1]$ and $WR_2[3]$ are 2W-units written by pairs $\{p_0, p_2\}$ and $\{p_2, p_1\}$, respectively.

```
Input: process $p_i$'s proposal value, $buf_i$.
Output: the value upon which all 3 processes (will) agree.
1V: $PROPOSAL[i] \leftarrow buf_i$; // Declare $p_i$'s proposal
    // Phase 1: Achieve an agreement between $p_0$ and $p_1$.
2V: if $i = 0$ or $i = 1$ then
3V:     $first \leftarrow$ SVW_FIRSTAGREEMENT($i$);
4V: end if
    // Phase II: Achieve an agreement between all three processes.
5V: $winner \leftarrow$ SVW_SECONDAGREEMENT($i, first_{ref}$); // $first_{ref}$ is the reference to $first$
6V: return $PROPOSAL[winner]$
```

Algorithm 2. SVW_FIRSTAGREEMENT(i: bit) invoked by process $p_i, i \in \{0,1\}$

```
Output: the preceding process of $\{p_0, p_1\}$
1SF: if $i = 0$ then
2SF:     SVWRITE($\{WR_1[0], WR_1[1]\}, \{Lower, Lower\}$); // atomically write to 2 units
3SF: else
4SF:     SVWRITE($\{WR_1[1], WR_1[2]\}, \{Higher, Higher\}$); // $i = 1$
5SF: end if
6SF: if $WR_1[(\neg i) * 2] = \perp$ then
7SF:     return $i$; // The other process hasn't written its value
8SF: else if ($WR_1[1] = Higher$ and $i = 0$) or ($WR[1] = Lower$ and $i = 1$) then
9SF:     return $i$; // The other process comes later and overwrites $p_i$'s value in $WR_1[1]$
10SF: else
11SF:     return $(\neg i)$;
12SF: end if
```

size 1 unit (instead of just one value of size b units) to b consecutive memory units, keeping the size of values to be atomically written as small as 1 unit will maximize the number of processes for which b-*svwrite*, together with registers, can solve the consensus problem. Unlike the seminal wait-free consensus algorithm using the m-word assignment by Herlihy [10], which requires the word size to be large enough to accommodate a proposal value, the new algorithm stores proposal values in shared memory and uses only two bits (or one unit) to determine the preceding order between two processes. This allows a single-*svword* assignment to write atomically up to B (or $\frac{B}{2}$ if units are single bits) ordering-related values. The new algorithm utilizes process unique identifiers, which are an implicit assumption in Herlihy's consensus model [6].

The SVW_CONSENSUS algorithm has two phases. In the first phase, two processes p_0 and p_1 will achieve an agreement on their proposal values (cf. Algorithm 2). The agreed value, $PROPOSAL[first]$, is the proposal value of the *preceding process*, whose SVWRITE (lines 2SF and 4SF) precedes that of the other process (lines 6SF-11SF).

Algorithm 3. SVW_SecondAgreement(i: index; $first_{ref}$: reference) invoked by process $p_i, i \in \{0, 1, 2\}$

```
1SS: if $i = 0$ then
2SS:     SVWrite($\{WR_2[0], WR_2[1]\}, \{Lower, Lower\}$);
3SS: else if $i = 1$ then
4SS:     SVWrite($\{WR_2[3], WR_2[4]\}, \{Lower, Lower\}$);
5SS: else
6SS:     SVWrite($\{WR_2[1], WR_2[2], WR_2[3]\}, \{Higher, Higher, Higher\}$);
7SS: end if
8SS: if (($WR_2[0] \neq \perp$ or $WR_2[4] \neq \perp$) and $WR_2[2] = \perp$) or // The predicates are checked in the
       writing order.
       ($WR_2[0] \neq \perp$ and $WR_2[1] = Higher$) or
       ($WR_2[4] \neq \perp$ and $WR_2[3] = Higher$) then
9SS:     return $first$; // $p_2$ is preceded by either $p_0$ or $p_1$. $first$ is obtained by dereferencing
           $first_{ref}$.
10SS: else
11SS:     return 2;
12SS: end if
```

Due to the memory alignment restriction, in order to be able to allocate memory for the WR_1 variable (cf. Algorithm 1) on which p_0's and p_1's SVWrites can partly overlap, p_0's and p_1's SVWrites are chosen as 2-*svwrite* and 3-*svwrite*, respectively. The WR_1 variable is located in a memory region consisting of 4 consecutive units $\{u_0, u_1, u_2, u_3\}$ of which u_0 is at an address multiple of 2 and u_1 at an address multiple of 3. This memory allocation allows p_0 and p_1 to write atomically to the first two units $\{u_0, u_1\}$ and the last 3 units $\{u_1, u_2, u_3\}$, respectively (cf. Figure 3(a)). The WR_1 variable is the set $\{u_0, u_1, u_2\}$ (cf. the solid squares in Figure 3(a)), namely p_1 ignores u_3 (cf. line 4SF in Algorithm 2).

Subsequently, the agreed value will be used as the critical value of both p_0 and p_1 in the second phase in order to achieve an agreement with the other process p_2 (cf. Algorithm 3). Let p_{first} be the preceding process of p_0 and p_1 in the first phase. The second phase returns p_{first}'s proposal value if either p_0 or p_1 precedes p_2 (line 9SS) and returns p_2's proposal value otherwise.

Units written by processes' SVWrite are illustrated in Figure 3(b). In order to be able to allocate memory for the WR_2 variable, process p_0's, p_1's and p_2's SVWrites are chosen as 2-*svwrite*, 3-*svwrite* and 5-*svwrite*, respectively. The WR_2 variable is located in a memory region consisting of 7 consecutive units $\{u_0, \cdots, u_6\}$ of which u_0 is at an address multiple of 2, u_4 at an address multiple of 3 and u_1 at an address multiple of 5. Since 2, 3 and 5 are prime numbers, we always can find such a memory region. For instance, if the memory address space starts from the unit with index 0, the memory region from unit 14 to unit 20 can be used for WR_2 (cf. Figure 1(a)). This memory allocation allows p_0, p_1 and p_2 to write atomically to the first two units $\{u_0, u_1\}$, the last three units $\{u_4, u_5, u_6\}$ and the five middle units $\{u_1, \cdots, u_5\}$, respectively. The WR_2 variable is the set $\{u_0, u_1, u_2, u_5, u_6\}$ (cf. the solid squares in Figure 3(b)).

Lemma 2. *The* SVW_SecondAgreement *procedure returns index 2 if p_2 precedes both p_0 and p_1. Otherwise, it returns index $first$.*

Lemma 3. *The* SVW_Consensus *algorithm is wait-free and solves the consensus problem for 3 processes.*

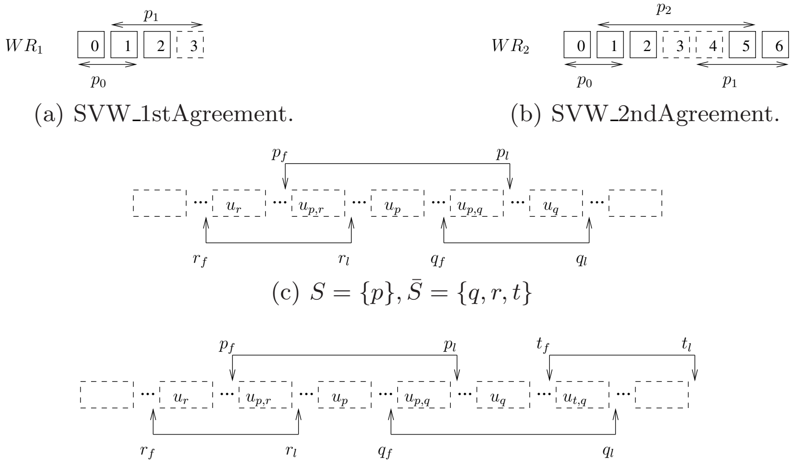
Proof. It is obvious from the pseudocode in Algorithms 1, 2 and 3 that the SVW_Consensus algorithm is wait-free.

From Lemma 2, the SVW_Consensus algorithm returns the same values for all invoking processes. The value is either $PROPOSAL[2]$ (if p_2 precedes both p_0 and p_1) or $PROPOSAL[first]$, $first \in \{0, 1\}$ (otherwise). □

Lemma 4. *The single-*svword *assignment has consensus number at least 3,* $\forall B \geq 5$*.*

Lemma 5. *The single-*svword *assignment has consensus number at most 3,* $\forall B \geq 2$*.*

Proof. (*Intuition; the full proof is in [9]*) We prove the lemma by contradiction. Assume that there is a wait-free consensus algorithm $\mathcal{ALG}$ for four processes p, q, r and t. At the critical configuration of the algorithm, we can always divide the set of the four processes into two non-empty subsets S and $\bar{S}$ where S consists of at most two processes with the same critical value called V and $\bar{S}$ consists of processes with critical values different from V (If three of the four processes have the same critical value, the other process is chosen as S). Since the *svwrite* operation writes to *consecutive* memory units in the conventional *1-dimensional* memory address space, let $[k_f, k_l]$ be the range of consecutive units to which a process $k \in \{p, q, r, t\}$ atomically writes using its critical operation op_k (cf. Lemma 1). For any pair of processes $\{h, k\}$, where h and k belong to different subsets S and $\bar{S}$, $[h_f, h_l]$ and $[k_f, k_l]$ must partly overlap (due to the second



(a) SVW_1stAgreement.

(b) SVW_2ndAgreement.

(c) $S = \{p\}, \bar{S} = \{q, r, t\}$

(d) $S = \{p, t\}, \bar{S} = \{q, r\}$

Fig. 3. Illustrations for the SVW_FirstAgreement, SVW_SecondAgreement and Lemma 5

requirement of Lemma 1) and none of them are completely covered by ranges $[v_f, v_l]$ of the other processes v (due to the first requirements of Lemma 1).

Figures 3(c) and 3(d) illustrate the proof when S consists of one and two processes, respectively. In Figure 3(c), the range $[t_f, t_l]$ of process t cannot partly overlap with that of process p without completely covering (or being covered by) the range of process r or q. In Figure 3(d), t and r belong to different subsets S and $\bar{S}$, respectively, but their ranges cannot partly overlap. □

Theorem 1. *The single-*svword *assignment has consensus number 3 when $B \geq 5$ and three is the upper bound of consensus numbers of single-*svword *assignments $\forall B \geq 2$.*

4 Consensus Number of the *Aiword* Model

In this section, we prove that the single-*aiword* assignment (or *aiwrite* for short) has consensus number exactly $\lfloor \frac{A+1}{2} \rfloor$. First, we prove that the *aiwrite* operation has consensus number at least $\lfloor \frac{A+1}{2} \rfloor$. We prove this by presenting a wait-free consensus algorithm AIW_Consensus for $N = \lfloor \frac{A+1}{2} \rfloor$ processes (cf. Algorithm 4) using only the *aiwrite* operation and registers. Subsequently, we prove that there is no wait-free consensus algorithm for $N + 1$ processes using only the *aiwrite* operation and registers.

The main idea of the AIW_Consensus algorithm is to gradually extend the set S of processes agreeing on the same value by one at a time. This is to minimize the number of 1W- and 2W-units that must be written atomically by the *aiword* operation (cf. Lemma 9). The algorithm consists of N rounds and a process $p_i, i \in [1, N]$, participates from round r_i to round r_N. A process p_i leaves a round $r_j, j \geq i$, and enters the next round r_{j+1} when it reads the value upon which all processes in the round r_j (will) agree. A round r_j starts with the first process that enters the round, and ends when all j processes $p_i, 1 \leq i \leq j$, have left the round. At the end of a round r_j, the set S consists of j processes $p_i, 1 \leq i \leq j$.

Lemma 6. *All correct processes*[4] *p_i agree on the same value in round r_j, where $1 \leq i \leq j \leq N$.*

With the assumption that AIWrite can atomically write to p_j's units at line 2I and p_i's units at line 11I, it follows directly from Lemma 6 that all the N processes will achieve an agreement in round r_N.

Lemma 7. *The* AIW_Consensus *algorithm is wait-free and can solve the consensus problem for $N = \lfloor \frac{A+1}{2} \rfloor$ processes.*

Proof. (*Intuition; the full proof is in [9]*) The time complexity for a process using AIW_Consensus to achieve an agreement among N processes is $O(N^2)$ due

[4] A *correct* process is a process that does not crash.

Algorithm 4. AIW_Consensus(buf_i: proposal) invoked by process $p_i, i \in [1, N]$

```
A^r[i]: p_i's agreed value in round r;
U^r_{i,j}: the 2W-unit written only by processes p_i and p_j in round r. U^r_i: the 1W-unit written only by process p_i in round r;
Input: process p_i's proposal value, buf_i.
Output: the value upon which all N processes (will) agree.
     // $p_i$ starts from round $i$
1I:  $A^i[i] \leftarrow buf_i$; // Initialized $p_i$'s agreed value for round $i$
2I:  AIWrite($\{U^i_i, U^i_{i,1}, \cdots, U^i_{i,i-1}\}$, $\{Higher, Higher, \cdots, Higher\}$) // Atomic assignment
3I:  for $k = 1$ to $(i-1)$ do
4I:      if $U^i_k \neq \perp$ and $U^i_{i,k} = Higher$ then
5I:          $A^i[i] \leftarrow A^i[k]$; // Update $p_i$'s agreed value to the set $S$'s agreed value
6I:          break;
7I:      end if
8I:  end for
     // Participate rounds from $(i+1)$ to $N$
9I:  for $j = i+1$ to $N$ do
10I:     $A^j[i] \leftarrow A^{j-1}[i]$; // Initialized $p_i$'s agreed value for round $j$
11I:     AIWrite($\{U^j_i, U^j_{j,i}\}$, $\{Lower, Lower\}$; // Atomic assignment
12I:     if $U^j_j \neq \perp$ and $U^j_{j,i} = Lower$ then
13I:         $WinnerIsJ \leftarrow$ true; // Check if $p_j$ precedes $p_k, \forall k < j$.
14I:         for $k = 1$ to $j-1$ do
15I:             if $U^j_k \neq \perp$ and $U^j_{j,k} = Higher$ then
16I:                 $WinnerIsJ \leftarrow$ false; // $p_k$ precedes $p_j$;
17I:                 break;
18I:             end if
19I:         end for
20I:         if $WinnerIsJ =$ true then
21I:             $A^j[i] \leftarrow A^j[j]$; // $p_j$ precedes $p_k, \forall k < j, \Rightarrow p_j$'s value is the agreed value in round $j$.
22I:         end if
23I:     end if
24I: end for
25I: return $A^N[i]$;
```

to the for-loops at lines 9I and 14I. Therefore, the AIW_Consensus algorithm is wait-free.

From Lemma 6, the AIW_Consensus algorithm can solve the consensus problem for $N = \lfloor \frac{A+1}{2} \rfloor$ processes if AIWrite can *atomically* write to p_j's units at line 2I and p_i's units at line 11I. Indeed, since $N = \lfloor \frac{A+1}{2} \rfloor$, an A-unit *aiword* (or A-*aiword* for short) can accommodate both $(N-1)$ 2W-units $U^N_{N,i}, 1 \leq i < N$, and N 1W-units $U^N_k, 1 \leq k \leq N$, used in round r_N. Since the single-*aiword* assignment AIWrite can atomically write to an arbitrary subset of the A units of an *aiword* and leave the other units untouched, each process $p_k, 1 \leq k \leq N$ can atomically write to *only*[5] its 1W and 2W units. □

Lemma 8. *The single-*aiword *assignment has consensus number at least* $\lfloor \frac{A+1}{2} \rfloor$.

Lemma 9. *The single-*aiword *assignment has consensus number at most* $\lfloor \frac{A+1}{2} \rfloor$.

Proof. (*Intuition; the full proof is in [9]*) We prove this lemma by contradiction. Assume that there is a wait-free consensus algorithm $\mathcal{ALG}$ for N processes where

[5] "Only" here means to leave other units untouched.

$N \geq \lfloor \frac{A+1}{2} \rfloor + 1$. At the critical configuration of the $\mathcal{ALG}$ algorithm, we divide N processes into two subsets S and $\bar{S}$ where S consists of processes with the same critical value called V and $\bar{S}$ consists of processes with critical values different from V. Let $|S|$ and $|\bar{S}|$ to be the sizes of the subsets, we have $|S| + |\bar{S}| = N$. Due to the memory alignment restriction, all the 1W-units and 2W-units used by critical assignments in the $\mathcal{ALG}$ algorithm must be located in the same A-*aiword* called AI. Let M be the number of the 1W-/2W-units, we have $M \leq A$.

Since $\mathcal{ALG}$ is a wait-free consensus algorithm for N processes, it follows from Lemma 1 that there are N 1W-units and $|S| \cdot |\bar{S}|$ 2W-units, i.e. $M = N + |S| \cdot |\bar{S}|$. Since $1 \leq |S| \leq (N-1)$, $M \geq (2N-1)$. Since $N \geq \lfloor \frac{A+1}{2} \rfloor + 1$ due to the hypothesis, $M \geq (A+1)$ must hold. This contradicts the requirement $M \leq A$. □

Theorem 2. *The single-*aiword *assignment has consensus number exactly* $\lfloor \frac{A+1}{2} \rfloor$.

5 Consensus Number of the *Asvword* Model

The intuition behind the higher consensus number of the *asvword* model compared with the *aiword* model (cf. Equation (1)) is that process p_N in Algorithm 4 can atomically write to $A \cdot B$ units using AxB-*asvwrite* instead of only A units using A-*aiwrite*. To prevent p_N from overwriting unintended units (as illustrated by SIMD core 4 in Figure 2), each B-*svword* located in $A_l, 1 \leq l \leq B$, contains either 1W-units or 2W-units but not both as illustrated in Figure 4, where B-*svword*s labeled "1W" contain only 1W-units and B-*svword*s labeled "2W" contain only 2W-units. This allows p_N to atomically write to only B-*svword*s with 2W-units $U^N_{N,i}$ (and keep 1W-unit $U^N_i, i \neq N$, untouched) using AxB-*asvwrite*. For each process $p_i, i \neq N$, its 1W-unit U^N_i and 2W-unit $U^N_{N,i}$ are located in two B-*svword*s labeled "1W" and "2W", respectively, that belong to the same A_l. This allows p_i to atomically write to only its two units using Ax1-*asvwrite*. A complete proof of the exact consensus number can be found in the full version of this paper [9].

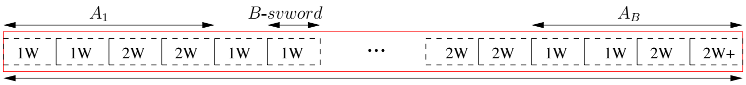


Fig. 4. An illustration for grouping units in the *asvword* model

Acknowledgements. The authors wish to thank the anonymous reviewers for their helpful and thorough comments on the earlier version of this paper. Phuong Ha's and Otto Anshus's work was supported by the Norwegian Research Council (grant numbers 159936/V30 and 155550/420). Philippas Tsigas's work was supported by the Swedish Research Council (VR) (grant number 37252706).

References

1. Cell Broadband Engine Architecture, version 1.01. IBM, Sony and Toshiba Corporations (2006)
2. NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.1. NVIDIA Corporation (2007)
3. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (1996)
4. Afek, Y., Merritt, M., Taubenfeld, G.: The power of multi-objects (extended abstract). In: PODC 1996: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, pp. 213–222 (1996)
5. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. John Wiley and Sons, Inc., Chichester (2004)
6. Buhrman, H., Panconesi, A., Silvestri, R., Vitanyi, P.: On the importance of having an identity or, is consensus really universal? Distrib. Comput. 18(3), 167–176 (2006)
7. Castano, I., Micikevicius, P.: Personal communication. NVIDIA (2008)
8. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
9. Ha, P.H., Tsigas, P., Anshus, O.J.: The synchronization power of coalesced memory accesses. Technical report CS:2008-68, University of Tromsø, Norway (2008)
10. Herlihy, M.: Wait-free synchronization. ACM Transaction on Programming and Systems 11(1), 124–149 (1991)
11. Jayanti, P., Khanna, S.: On the power of multi-objects. In: Mavronicolas, M. (ed.) WDAG 1997. LNCS, vol. 1320, pp. 320–332. Springer, Heidelberg (1997)
12. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess progranm. IEEE Trans. Comput. 28(9), 690–691 (1979)
13. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
14. Ramamurthy, S., Moir, M., Anderson, J.H.: Real-time object sharing with minimal system support. In: Proc. of Symp. on Principles of Distributed Computing (PODC), pp. 233–242 (1996)
15. Ruppert, E.: Determining consensus numbers. In: Proc. of Symp. on Principles of Distributed Computing (PODC), pp. 93–99 (1997)
16. Ruppert, E.: Consensus numbers of multi-objects. In: Proc. of Symp. on Principles of Distributed Computing (PODC), pp. 211–217 (1998)

Optimizing Threshold Protocols in Adversarial Structures

Maurice Herlihy[1], Flavio P. Junqueira[2], Keith Marzullo[3],
and Lucia Draque Penso[1]

[1] Brown University
[2] Yahoo! Research Barcelona
[3] UC San Diego

Abstract. Many replication protocols are using a *threshold model* in which failures are independent and identically distributed (IID). In this model, one assumes that no more than t out of n components can fail. In many real systems, however, failures are not IID, and a straightforward application of threshold protocols yields suboptimal results.

Here, we examine the problem of optimally transforming threshold protocols into survivor-set protocols tolerating dependent failures. In particular, we are interested in threshold protocols where the number of components n and the number of failures t are related by $n > k \cdot t$, where k is a positive integer constant k. We develop an *optimal transformation* that translates any such threshold protocol to a survivor-set dependent failure model, and hence, to adversarial structures. Our transformation does not require authentication, self-verification or encryption. We characterize *equivalence classes of adversarial structures*, regarding solvability, using certain hierarchical properties based on set intersection.

1 Introduction

Many replication protocols are expressed in terms of a *threshold model* in which failures are independent and identically distributed (IID). In this model, one assumes that no more than t out of n components can fail [21]. Lower bounds for problems are often stated in this model. For instance, it is widely known that consensus in a synchronous system with Byzantine failures requires $n > 3t$ processors without digital signatures [19].

In many real systems, however, failures are not necessarily IID, and a straightforward application of threshold protocols yields suboptimal results. Junqueira and Marzullo [13,14,15,17] have developed an alternative *dependent failure model* that captures the behavior of systems where failures are not necessarily IID. This model replaces thresholds with *survivor sets*, the unique collection of minimal sets of correct processors in any execution. The survivor set model is strictly more powerful than the threshold model: thresholds can be expressed in terms of survivor sets, but not vice-versa. Survivor sets also encapsulate two distinct abstractions: the availability of quorum systems and the failure patterns of adversarial structures. Hence, survivor sets provide at same time both a structure to use with protocols and an expressive way of describing failure patterns.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 335–349, 2008.

Junqueira and Marzullo rederive some classic lower bounds for consensus in this model, and develop protocols optimal with respect to these bounds. They demonstrate that there are cases in which it is possible to solve consensus using survivor sets, but not using threshold protocols.

In this paper we investigate how to optimally transform threshold protocols into survivor-set protocols. We focus on threshold protocols where the number of components n and the number of failures t are related by $n > k \cdot t$, where k is a positive integer constant k. We make two main contributions. First, to translate such threshold protocols to a survivor set dependent failure model, we provide an *optimal transformation* which does not require authentication, self-verification or encryption. Then we characterize *equivalence classes of adversarial structures*, regarding solvability, in terms of certain hierarchical properties based on set intersection.

Discovering automatically Byzantine consensus protocols has recently attracted lots of attention [24], as well as making smart usage of dependent failure models such as adversarial structures [9]. Furthermore, less general transformations for asymmetric dependent failure models, focusing on a restricted set of both problems and assumptions, have lately been an object of study by Warns, Freiling and Hasselbring [23].

2 System Model

A system is a set of processors $\Pi = \{p_1, p_2, \cdots, p_n\}$ that communicate by exchanging messages. Each processor is capable of executing multiple automata, as in the model described by Attiya and Welch for simulations [1]. The set of automata of a system is $\{sm_1, sm_2, \cdots, sm_m\}$, where each automaton sm_i has a state, code, and an identity which is used by other automata to send messages to sm_i. An automaton is deterministic, in that its current state is determined only from its initial state and the sequence of messages it has received. We model an automaton as a state machine [22]. Furthermore, there is a quasi-reliable channel between every pair of automata. A quasi-reliable channel guarantees that if a correct automaton sm_i sends a message m to a correct automaton sm_j, then m is eventually received by sm_j [6,8].

We assume that processors fail arbitrarily, though results also apply to benign failures. A faulty processor can crash, modify the content of messages arbitrarily, omit to send or receive messages, and behave in malicious ways. It is important to observe that the failure of a processor implies the failure of all automata in it. Thus, if a processor is arbitrarily faulty, then all of the automata running at that processor can exhibit arbitrary behavior. The converse is also true: if a processor is correct, then all automata running in it are also correct.

We do not assume a threshold on the number of processor failures. Instead, we characterize failure scenarios by providing the *survivor set system*, the unique collection of all minimal sets of correct processors, each set containing all correct processors of some execution [13,14,15,16]. Informally, elements of a survivor set system, called survivor sets, generalize subsets of size $n - t$ under the threshold model, where n is the total number of processor and t is again a threshold on the number of processor failures. Such elements are the bijective complement of elements belonging to fail-prone systems [14] and can describe a general adversarial structure [12] that comprises quorums [20]. Let ϕ be an execution from the set of all possible executions Φ for the protocol, and *Correct*(ϕ) the set of automata which never fail in ϕ, that is, which always remain correct in ϕ. More formally, we define a survivor set as follows:

Definition 1. *A subset $S \subseteq \Pi$ is a survivor set if and only if: 1) $\exists \phi \in \Phi$,* $\mathrm{Correct}(\phi) = S$*; 2) $\forall \phi \in \Phi, p_i \in S$,* $\mathrm{Correct}(\phi) \not\subseteq S/p_i$*.*

Henceforth, we refer to the survivor set system as the collection of survivor sets of Π as S_Π. Note that it is unique given a fail-prone system [14], that is, an exact configuration of possible failures. The inverse is also true.

A survivor set system is equivalent to a core system, and one defines uniquely the other. Informally, a core is a subset of processors that generalizes subsets of size $t + 1$ in the threshold model, where t is a threshold on the number of processor failures. Thus, in every execution of the system, there is at least one automaton in every core that is correct. From the set of cores, one can obtain the survivor set system by creating all minimal subsets of processors that intersect every core. More formally, we define cores as follows:

Definition 2. *A subset $C \subseteq \Pi$ is a core if and only if: 1) $\forall \phi \in \Phi$,* $\mathrm{Correct}(\phi) \cap C \neq \emptyset$*; 2) $\forall p_i \in C$, $\exists \phi \in \Phi$ such that $C/p_i \cap$* $\mathrm{Correct}(\phi) = \emptyset$*.*

Henceforth, we refer to the set of cores of Π as C_Π. Along with Π, both collections of sets constitute a *system profile*. We use the notation $\langle \Pi, S_\Pi, C_\Pi \rangle$ to refer to a system profile. Note that either S_Π or C_Π is sufficient to define a system profile as one can uniquely be obtained from the other, just as the fail-prone system representing the adversarial structure. We provide both for convenience. Additionally, we assume that for a system $\langle \Pi, C_\Pi, S_\Pi \rangle$ there is no processor $p_i \in \Pi$ in every survivor set of S_Π, *i.e.*, there is no automaton that never fails. The solution to many problems in distributed computing would be trivial if we assume such reliable processors.

It is important to observe that in this paper we focus on the transformation of threshold algorithms to dependent-failure algorithms, so we run automata of threshold algorithms on processors, and we model the failures of processors with cores and survivor sets. Cores and survivor sets, however, can also model failures of other common abstractions such as processes, or even automata. Thus, we constrain cores and survivor sets to processors only for clarity of presentation.

Equivalence. We now present a formal argument showing the equivalence between cores and survivor sets. A survivor set system and its respective core system can be generated from each other (without extra knowledge, such as about the set of executions), which also implies that they convey the very same information.

Let Π be some finite set, such as the set of processors. Let $\mathcal{S} \subseteq 2^\Pi$ (representing the survivor set system) be such that $S \not\subseteq T$ and $T \not\subseteq S$ for all $S, T \in \mathcal{S}$

with $S \neq T$. Let $\mathcal{C} \subseteq 2^{\Pi}$ (representing the core system) be the set of all subsets of Π which intersect all elements of $\mathcal{S}$ and are minimal with respect to inclusion subject to this property.

Similarly, let $\mathcal{S}' \subseteq 2^{\Pi}$ be the set of all subsets of Π which intersect all elements of $\mathcal{C}$ and are minimal with respect to inclusion subject to this property. Let $\mathcal{C}' \subseteq 2^{\Pi}$ be the set of all subsets of Π which intersect all elements of $\mathcal{S}'$ and are minimal with respect to inclusion subject to this property.

Theorem 1. $\mathcal{S} = \mathcal{S}', \mathcal{C} = \mathcal{C}'$.

Proof. First, we prove that $\mathcal{S} \subseteq \mathcal{S}'$. Therefore, let $S \in \mathcal{S}$. By the definition of $\mathcal{C}$, the set S intersects all elements of $\mathcal{C}$. Let $p \in S$. Since no set in $\mathcal{S} \setminus \{S\}$ is a subset of S, there is a set $C_p \in \mathcal{C}$ with $C_p \subseteq \Pi \setminus (S \setminus \{p\})$. Since C_p does not intersect $S \setminus \{p\}$, the set S intersects all elements of $\mathcal{C}$ and is minimal with respect to inclusion subject to this property, i.e. $S \in \mathcal{S}'$ and hence $\mathcal{S} \subseteq \mathcal{S}'$. Next, we prove that $\mathcal{S}' \subseteq \mathcal{S}$. Therefore, we assume the existence of some $S' \in \mathcal{S}' \setminus \mathcal{S}$. Since $\mathcal{S} \subseteq \mathcal{S}'$, no set in $\mathcal{S}$ is contained in S'. This implies the existence of a set $C' \in \mathcal{C}$ with $C' \subseteq \Pi \setminus S'$. Since C' does not intersect S', we obtain the contradiction $S' \notin \mathcal{S}'$ which completes the proof that $\mathcal{S} = \mathcal{S}'$. By analogy, $\mathcal{C} = \mathcal{C}'$. □

Examples. We now present two examples to motivate the use of survivor set systems. The first example illustrates the advantage of our approach when the probability of failure of distinct processors is not the same. In the second example, we look at a system in which failures of processors are partially correlated. In the following examples, we use target degree of reliability to denote the maximum probability acceptable for the failure probability of a subset of processors. That is, if the target degree of reliability is r, and the failure probability of a subset of processors is x, then x is negligible if only if $x \leq r$.

Example 1. Consider a system with five processors $\Pi{=}\{p_1, p_2, p_3, p_4, p_5\}$, where:

$$\begin{aligned} P(p_1 \text{ is faulty in an execution}) &= P(p_2 \text{ is faulty in an execution}) = \\ &= P(p_3 \text{ is faulty in an execution}) = 0.01 \\ P(p_4 \text{ is faulty in an execution}) &= P(p_5 \text{ is faulty in an execution}) = 0.001 \end{aligned}$$

This means that p_1, p_2, and p_3 are not as reliable as p_4 and p_5. Assuming both that these processors fail independently, and that target degree of reliability for this system is 0.0001, we can then infer the following survivor set system and core system:

$$S_{\Pi}{=}\{\{p_1, p_4, p_5\}, \{p_2, p_4, p_5\}, \{p_3, p_4, p_5\}, \{p_1, p_2, p_3, p_4\}, \{p_1, p_2, p_3, p_5\}\}$$
$$C_{\Pi}{=}\{\{p_1, p_2, p_3\}, \{p_1, p_4\}, \{p_1, p_5\}, \{p_2, p_4\}, \{p_2, p_5\}, \{p_3, p_4\}, \{p_3, p_5\}, \{p_4, p_5\}\}$$

This system satisfies Byzantine Intersection and Byzantine Partition [15], two equivalent properties necessary and sufficient to solve consensus in a synchronous system with arbitrarily faulty processors.

Example 2. Consider a system with six processors $\{p_1, p_2, p_3, p_4, p_5, p_6\}$, where:

- All processors have the same probability x of failure in an execution, $0 < x < 1$;
- There are two distinct groups: $A = \{p_1, p_2, p_3\}$ and $B = \{p_4, p_5, p_6\}$;
- Let $\phi \in \Phi$: $1 > P(p_i \in A \text{ is faulty in } \phi \mid p_j \in B \text{ is faulty in } \phi) = P(p_i \in A \text{ is faulty in } \phi) = x$, $i \neq j$;
- Let $\phi \in \Phi$: $1 > P(p_i \in \Psi \text{ is faulty in } \phi \mid p_j \in \Psi \text{ is faulty in } \phi) > P(p_i \in \Psi \text{ is faulty in } \phi) = x$, $i \neq j$ and $\Psi \in \{A, B\}$;
- Let $\phi \in \Phi$: $1 > P(p_i \in \Psi \text{ is faulty in } \phi \mid p_j, p_k \in \Psi \text{ are faulty in } \phi) < x^2$, $i \neq j, k$ and $\Psi \in \{A, B\}$;

Assuming that the target degree of reliability for this system is x^2, we can infer the survivor set system and core system:

$$\begin{aligned} S_\Pi = \{&\{p_1, p_2, p_3, p_4\}, \{p_1, p_2, p_3, p_5\}, \{p_1, p_2, p_3, p_6\}, \{p_1, p_4, p_5, p_6\},\\ &\{p_2, p_4, p_5, p_6\}, \{p_3, p_4, p_5, p_6\}\}\\ C_\Pi = \{&\{p_1, p_2, p_3\}, \{p_4, p_5, p_6\}, \{p_1, p_4\}, \{p_1, p_5\}, \{p_1, p_6\}, \{p_2, p_4\}, \{p_2, p_5\},\\ &\{p_2, p_6\}, \{p_3, p_4\}, \{p_3, p_5\}, \{p_3, p_6\}\} \end{aligned}$$

Consider an implementation of a fault-tolerant state machine that tolerates arbitrary failures, such as the one described by Castro and Liskov [4]. To make failures independent, one can use opportunistic n-version programming, as proposed by Castro *et al* in [5]. If only two implementations are available, then one could analyze the failure behavior of these implementations. If the properties of these two implementations satisfy those described above, then such a system can be used to solve the given problem, since one can be used in group A and the other in group B. Note that if there is a high probability that all processors executing the same implementation fail together, then this construction is not useful.

It is important to observe that our dependent failure model does not violate classic impossibility results, such as the minimum degree of replication necessary to solve consensus with arbitrary failures [19]. We observe, however, that under realistic assumptions [10,11], our approach is able, in several instances, to refine those impossibility results.

3 Replication with Cores and Survivor Sets

Junqueira and Marzullo [15] show two equivalent properties on replication that are necessary and sufficient to solve consensus assuming arbitrary process failures under the core and survivor set systems model. These properties, called *Byzantine Partition* and *Byzantine Intersection*, generalize the bound on replication based on a threshold $n > 3t$, where n is the number of processors in a system and t is a threshold on the number of processor failures. Based on these properties, we stated two parameterized properties (α, β)-Partition and

(α, β)-Intersection, for integers α, β and $\alpha > \beta \geq 1$, that generalizes a bound of the form $n > \lfloor \alpha t/\beta \rfloor$. An example of such a bound in the literature is the lower bound for primary-backup with receive-omission failures, which is $n > \lfloor 3t/2 \rfloor$ [3].

Here, we concentrate on the cases in which $\beta = 1$ and $\alpha = k \geq 2$, which is equivalent in the threshold model to a replication requirement of $n > k \cdot t$ for $k \geq 2$. This replication bound implies that if one constructs k subsets of the processes, then at least one of them will contain at least $n - t$ processes. Generalized to an expression on cores, we have:

Property 1. **k-Partition**
For every partition $\mathcal{A} = \{A_1, A_2, \cdots, A_k\}$ of Π, at least one of the sets A_i contains a core.

In the following sections, we use the equivalent $(k, 1)$-intersection property, which we call from this point on k-Intersection. This is a property of survivor sets rather than cores, and we use it with both our optimal automatic translation protocol and the definition of equivalence classes of adversarial structures. Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile such that there is no core in C_Π of size one. Then:

Property 2. **k-Intersection**
For every $\{S_1, S_2, \cdots, S_k\} \in S_\Pi$, $\cap_i S_i \neq \emptyset$.

Relating to adversary structures. Non-threshold protocols were also considered in the context of secure multi-party computation with adversary structures [12]. Adversary structures and survivor set systems differ in three fundamental ways. First, survivor set systems are sets of correct processors, whereas adversary structures contain sets of faulty processors, such as with fail-prone systems. Recall that fail-prone systems are the unique complement of survivor set systems. Second, adversary structures can represent more than one failure mode, *e.g.*, crash failures and arbitrary failures. Each failure mode is described with sets of possibly faulty processors (processors are referred to as *players* in this literature). Third, all sets of possibly faulty players are given. Using all possible sets of players that can deviate from the correct protocol behavior as opposed to only maximal sets as with fail-prone systems (or minimal sets of correct processors, as with survivor set systems) gives one more expressiveness in modeling system failures. Using survivor sets, however, is sufficient for establishing bounds on processor replication. Moreover, these bounds hold even for a more expressive model such as adversary structures. This is because we use properties about the intersections of sets of correct processors. If the intersection property holds for some minimal sets of processors A_1, A_2, ..., A_m then it holds for the sets of processes $A'_1 \supset A_1$, $A'_2 \supset A_2$, ..., $A'_m \supset A_m$. Hence, one only has to consider the minimal sets of correct processors in these intersection properties [17].

4 Constructing Protocols

In this section, we present ways to build protocols for our dependent failure model out of protocols designed for the threshold model. By assumption, protocol

Λ_t requires $n > k \cdot t$ replication for some positive integer value of k and some threshold on the number of failures $t > 0$. Let $\mathcal{A}_t$ be the set of automata in which Λ_t runs. The main idea is to allow a processor to run more than one automaton of Λ_t. At first glance, this may appear to be a fruitless approach, since automata executing at the same processor fail together. That is, if a processor p_i is faulty, so are all of automata of Λ_t that p_i executes. By choosing where and how to replicate automata, however, one can increase replication enough without also increasing t so that the replication requirement $n > k \cdot t$ is met.

Our goal is to provide a method for constructing a protocol Λ_{CS} for the core and survivor set systems model based on protocol Λ_t. More specifically, given a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfying k-Intersection, we provide a set $\mathcal{A}_{cs}$ and a mapping $\psi(\mathcal{A}_{cs})$ of these automata to processors.

First, we describe a procedure to determine a value of n and to assign automata of Λ_t to processors, such that in no execution more that t automata of Λ_t fail, for some value of t under the constraint that $n > k \cdot t$. Consider a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$. Let l_i be the fraction of automata of $\mathcal{A}_t$ that processor $p_i \in \Pi$ executes, where $0 \leq l_i < 1$. Moreover, in each execution, the fraction of correct automata is as follows:

$$\frac{n-t}{n} > \frac{(k \cdot t - t)}{k \cdot t} = \frac{k-1}{k}$$

These observations lead us to the following set of constraints:

$$\sum_{p_i \in \Pi} l_i = 1$$
$$\forall s \in S_\Pi : \sum_{p_i \in s} l_i > \frac{k-1}{k} \qquad (1)$$

These equations imply that every automaton is executed by exactly one processor, and in no execution there are more than $\lfloor n/k \rfloor$ faulty automata. If we solve this system of linear equations, and choose a large enough value of n such that $n \cdot l_i$ is an integer for every i, then we have a solution for our problem. We can then simply choose the smallest value of n for which this condition holds. That is:

$$\min_n \{\forall l_i : n \cdot l_i \text{ is an integer}\} \qquad (2)$$

Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile and $\mathcal{L}$ be a set of values l_i, $0 \leq l_i < 1$, with a value l_i for each processor of Π. We say that $\mathcal{L}$ is a valid solution for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and degree of replication k if these values satisfy Constraints 1 for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k. The following theorem states that there being a valid solution is sufficient for a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ to satisfy k-Intersection.

Theorem 2. *Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile and k be a degree of replication. There is a valid solution $\mathcal{L}$ for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k only if $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfies k-Intersection.*

Proof. We prove this theorem by contradiction. Suppose that there is a system with profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ that does not satisfy k-Intersection and there is a valid array of values l_i for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k. From the first assumption, there is a subset $\{S_1, S_2, \cdots, S_k\}$ in S_Π such that $\cap_i S_i = \emptyset$. From the second assumption we have the following:

$$\forall i \in \{1, \cdots, k\} : \sum_{p_j \in S_i} l_j > \frac{k-1}{k}$$

Summing together these k equations, we have the following:

$$\left(\sum_{p_i \in S_1} l_i \right) + \left(\sum_{p_i \in S_2} l_i \right) + \cdots + \left(\sum_{p_i \in S_k} l_i \right) > (k-1) \tag{3}$$

Since no processor is in all k survivor sets by assumption and $k \geq 2$ we also have that:

$$\left(\sum_{p_i \in S_1} l_i \right) + \left(\sum_{p_i \in S_2} l_i \right) + \cdots + \left(\sum_{p_i \in S_k} l_i \right) \leq \sum_{p_i \in \Pi} l_i \leq 1 \leq (k-1) \tag{4}$$

Equations 3 and 4, however, cannot both hold, giving us our contradiction.

Using the value of n provided by (2), we can determine the set of automata $\mathcal{A}_t$. Assuming a valid solution $\mathcal{L}$, we make $\mathcal{A}_{cs} \leftarrow \mathcal{A}_t$ and build $\psi(\mathcal{A}_{cs})$ as follows:

$$\begin{aligned} &\forall sm \in \mathcal{A}_{cs} : \\ &\qquad \psi(\mathcal{A}_{cs}) \leftarrow [\psi(\mathcal{A}_{cs}) \mid sm_i \rightarrow p_j], (p_j \in \Pi) \wedge (|(\psi(\mathcal{A}_{cs}))^{-1}(p_j)| \leq n \cdot l_j) \end{aligned}$$

The following theorem states that in every execution of Λ_{CS} there are at most t faulty automata, where $t = n - \lfloor \frac{k \cdot (n+1) - n}{k} \rfloor$.

Theorem 3. *Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile and k a degree of replication. Given a valid solution $\mathcal{L}$ for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k, there are at most t faulty automata in any execution $\phi \in \Phi$ of the protocol Λ_{CS}, for $t = n - \lfloor \frac{k \cdot (n+1) - n}{k} \rfloor$.*

Proof. We first show that in every execution, there are at least $n - t$ correct automata, for some value of t. By assumption, for every execution $\phi \in \Phi$ there is at least one survivor set $S \in S_\Pi$ containing only correct processors. Given that$\mathcal{L}$ is a valid solution, we have:

$$\sum_S n \cdot l_i = n \cdot \sum_S l_i > \frac{n \cdot (k-1)}{k} \geq \left\lfloor \frac{n \cdot (k-1)}{k} + 1 \right\rfloor \tag{5}$$

It also must be the case that for all $S \in S_\Pi$, the sum of t and $\sum_S n \cdot l_i =$ is greater or equal to n. Otherwise, there is at least one execution in which there are more than t faulty automata. Expressed more formally,

$$\forall S \in S_\Pi : n \cdot \sum_S l_i + t \geq n$$

$$\Rightarrow \qquad \forall S \in S_\Pi : t \geq n - n \cdot \sum_S l_i \tag{6}$$

$$\Rightarrow \qquad t \geq n - \min_{S \in S_\Pi} \{n \cdot \sum_S l_i\} \tag{7}$$

By equation 5, we have that the value of the previous sum is bounded from below by $\left\lfloor \frac{k\cdot(n+1)-n}{k} \right\rfloor$. That is:

$$\min_S \{n \cdot \sum_S l_i\} \geq \left\lfloor \frac{k \cdot (n+1) - n}{k} \right\rfloor \tag{8}$$

If we then choose t as:

$$t = n - \left\lfloor \frac{k \cdot (n+1) - n}{k} \right\rfloor \tag{9}$$

We have:

$$\forall S \in S_\Pi : n - n \cdot \sum_S l_i \leq n - \left\lfloor \frac{k \cdot (n+1) - n}{k} \right\rfloor = t \tag{10}$$

From Equation 10, we conclude that there is no execution in which more that t automata fail, where t is given by Equation 9.

With this construction, automata behave as in the original protocol, sending and receiving messages from each other. The only difference is that some automata may run in the same processor, and consequently these automata fail together. From the previous theorem, however, our construction provides a threshold on the number of faulty automata that does not violate the replication requirement for protocol Λ_t, thereby guaranteeing the correct execution of Λ_{CS}.

A problem is that, in some instances, even if a system profile satisfies k-Intersection, for some value of k, there is no valid solution for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k. We show this with the following theorem, and explain in the sequence how to circumvent optimally this impossibility result.

Theorem 4. *For every value of $k > 1$, there is a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfying k-Intersection such that there is no valid solution $\mathcal{L}$ for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k.*

Proof. The case $k = 2$ follows directly from Theorem 3.1 in [7]. We show that it holds for $k \geq 3$.

We construct a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ for which the proposition holds. Suppose that $|\Pi| = (k-1) \cdot k$. Now, partition Π into $k-1$ disjoint sets $A = (A_1, A_2, \cdots, A_{k-1})$, each of size k, and let C_Π be as follows:

$$C_\Pi = \{A_1, A_2, \cdots, A_{k-1}\} \cup \{\{p_i, p_j\} \mid p_i \in A_x, p_j \in A_y, x \neq y\} \tag{11}$$

From this set of cores, we can build the set of survivor sets as follows:

$$S_{\Pi} = \{A_x \cup \{p_i\} \mid p_i \in A_y, x \neq y\} \tag{12}$$

This system clearly satisfies k-partition, since any partition of Π into k subsets will result in at least one subset containing either all of some A_x or two automata from different subsets A_x and A_y. We now show that there is no set of values l_i, one for each processor $p_i \in \Pi$, satisfying equations in 1.

The set of linear equations for our system is as follows. For each A_x and $p \in A_x$:

$$\left(\sum_{A_y \in A / \{A_x\}} \sum_{p_i \in A_y} l_i \right) + l_p > \frac{k-1}{k} \tag{13}$$

From 13, there are $k \cdot (k-1)$ equations, where each l_i appears on the left side of exactly $k \cdot (k-2) + 1 = (k-1)^2$ equations. Summing up each side these equations, we get:

$$\begin{aligned} & (k-1)^2 \cdot (l_1 + l_2 + \cdots + l_{|\Pi|}) > k \cdot (k-1) \cdot \frac{k-1}{k} \\ \Rightarrow & (k-1)^2 \cdot (l_1 + l_2 + \cdots + l_{|\Pi|}) > (k-1)^2 \\ \Rightarrow & (l_1 + l_2 + \cdots + l_{|\Pi|}) > 1 & (14) \\ & & (15) \end{aligned}$$

We conclude that the first equation of Constraints 1 cannot be fulfilled as we wanted to show.

To illustrate this construction, consider the system profile of Example 1. Recall that in that system there are five processors. There is one core that contains three processors, and all of other cores are of size two. Given $k = 3$, we have the following solution for this system:

- $l_1 = \frac{1}{7}$, $l_2 = \frac{1}{7}$, $l_3 = \frac{1}{7}$, $l_4 = \frac{2}{7}$, $l_5 = \frac{2}{7}$;
- We choose $n = 7$.
- Let $\mathcal{A}_t = \{sm_1, sm_2, sm_3, sm_4, sm_5, sm_6, sm_7\}$. A possible mapping $\psi(\mathcal{A}_{cs})$ is as follows: $\{sm_1 \rightarrow p_1, sm_2 \rightarrow p_2, sm_3 \rightarrow p_3, sm_4 \rightarrow p_4, sm_5 \rightarrow p_4, sm_6 \rightarrow p_5, sm_7 \rightarrow p_5\}$;
- $t = n - \lfloor \frac{k \cdot (n+1) - n}{k} \rfloor = 7 - 5 = 2$;

In example 2, however, there is no solution satisfying the set of constraints 1. This example is actually the case presented in the proof of Theorem 4 for $k = 3$.

However, to circumvent the impossibility result, it suffices to replicate automata in such a way that it guarantees that there is enough replicas to emulate a threshold environment, and to deliver messages atomically to the replicas of an automaton, assuming the automata are deterministic. This is equivalent to implementing a replicated state machine using atomic broadcast (such as in [4] for $k \geq 3$, Paxos [18] otherwise). Having byzantine-tolerant state machine replication in a timely manner [4,18,22] is in fact equivalent to having byzantine consensus [2,15].

```
**Constants:**
$k$: Degree of replication
$\mathcal{A}_t$: Set of automata identifiers of $\Lambda_t$

**Global variables:** all initially empty
$\mathcal{A}_{cs}$: Set of automata identifiers of $\Lambda_{CS}$
$\psi(\mathcal{A}_{cs})$: $\mathcal{A}_{cs} \rightarrow \Pi$
$Rep$: $\Pi \rightarrow 2^{\mathcal{A}_t}$
$Label$: $\mathcal{A}_{cs} \rightarrow 2^{S_\Pi}$
**Procedure Bin**
**Input:**
$\mathcal{S} \subset S_\Pi$: A set of survivor sets
$w \in (\mathcal{A}_t)^*$: A sequence of automata id's

**Main:**
        Let $\{B_1, B_2, \ldots, B_{k+1}\}$ be a set of initially empty bins
        $Label \leftarrow [Label \mid w \circ v_1 \mapsto B_1, w \circ v_2 \mapsto B_2, \ldots, w \circ v_{k+1} \mapsto B_{k+1}]$
        Populate $B_1, B_2, \ldots, B_{k+1}$ as follows:
                For every $S \in \mathcal{S}$: $S$ is in exactly $k$ bins $B_i$
                For each bin $B_i$: $(B_i \neq \emptyset) \wedge (B_i \neq \mathcal{S})$
        For each bin $B_i$
                if $\exists p_l \in \Pi$ such that $\forall S \in B_i$, $p \in B_i$
                  Let $wi = jw'$, where $v_j \in \mathcal{A}_t$ and $w' \in (\mathcal{A}_t)^*$
                  If $v_j \notin Rep(p_l)$
                        id $\leftarrow$ generate-id($v_j$, $p_l$)
                        $\mathcal{A}_{cs} \leftarrow \mathcal{A}_{cs} \cup \{\text{id}\}$
                        $\psi(\mathcal{A}_{cs}) \leftarrow \left[\psi(\mathcal{A}_c s) \mid \text{id} \mapsto p_l\right]$
                        $Rep \leftarrow [Rep \mid p_l \mapsto (Rep(p_l) \cup \{v_j\})]$
                else Bin($B_i$, $wi$)
        Return
```

Fig. 1. How to place automata in processors optimally

Figure 1 shows how to optimally place automata in processors, so that our transformation remains optimal not only with respect to (maximal) failure tolerance, but also to (minimal) automata set size and (minimal) k-intersection between survivor sets of automata.

To place automata in processors, we use a procedure that recursively splits the set of survivor sets until it reaches a subset $\mathcal{S}$ that mutually intersect. At this point, a processor in the intersection is chosen to execute one of the replicas for an automaton in $\mathcal{A}_{cs}$. Note that this processor is correct whenever one of the survivor sets in $\mathcal{S}$ contains only correct automata. From the definition of a survivor set, for each survivor set s in $\mathcal{S}$, there is at least one execution in which s contains only correct automata.

The procedure in Figure 1 creates $k+1$ bins. It first splits the set of survivor sets into $k+1$ bins such that each survivor set is in exactly k bins. As each survivor set is a minimal set of automata that, for some run, do not fail, any set of failures will result in k bins that contain a survivor set that does not fail. If there is a processor that is in all of the survivor sets of bin B_i, then a replica of the automaton can be run on that processor and the procedure stops splitting bin B_i. If not, then the procedure is recursively invoked with the set of survivor sets associated with bin B_i. Eventually, this recursive procedure terminates, assuming that the initial set of survivor sets satisfy k-Intersection (see Theorem 5 below).

We can label each bin constructed recursively by a sequence of automaton identifiers, e.g. $[a_1 a_1 a_2]$, indicating a bin that was created from B_1 on the first call, having originated $k+1$ bins under B_1. The first bin of this second level was split again, and the label refers to the second bin from this second split. Once a processor p is in each survivor set associated with some bin B with label $[a_i \ldots]$, we associate a replica for automaton a_i is associated with p. Note that this construction may repeatedly assign a_i to p while recursively constructing Λ_{CS}. Only one copy of a_i needs to run on p, though. As an example, we consider the survivor set system of Example 2 in the following:

Example 3. Let $S_1 = \{p_1, p_2, p_3, p_4\}$, $S_2 = \{p_1, p_2, p_3, p_5\}$, $S_3 = \{p_1, p_2, p_3, p_6\}$, $S_4 = \{p_1, p_4, p_5, p_6\}$, $S_5 = \{p_2, p_4, p_5, p_6\}$, $S_6 = \{p_3, p_4, p_5, p_6\}$.

$B_1 = \{S_1,S_2, S_3, S_4, S_5\}$: Implements automaton v_1

$B_{11} = \{S_1, S_2, S_3\}$: p_1 is in the intersection

$B_{12} = \{S_1,S_3, S_4, S_5\}$

$B_{121} = \{S_1, S_3, S_5\}$: p_2 is in the intersection

$B_{122} = \{S_1, S_3, S_4\}$: p_1 is in the intersection

$B_{123} = \{S_1, S_4, S_5\}$: p_4 is in the intersection

$B_{124} = \{S_3, S_4, S_5\}$: p_6 is in the intersection

$B_{13} = \{S_1, S_2, S_4, S_5\}$

$B_{131} = \{S_1, S_2, S_5\}$: p_2 is in the intersection

$B_{132} = \{S_1, S_2, S_4\}$: p_1 is in the intersection

$B_{133} = \{S_1, S_4, S_5\}$: p_4 is in the intersection

$B_{134} = \{S_2, S_4, S_5\}$: p_5 is in the intersection

$B_{14} = \{S_2, S_3, S_4, S_5\}$

$B_{141} = \{S_2, S_3, S_5\}$: p_2 is in the intersection

$B_{142} = \{S_2, S_3, S_4\}$: p_1 is in the intersection

$B_{143} = \{S_3, S_4, S_5\}$: p_6 is in the intersection

$B_{144} = \{S_2, S_4, S_5\}$: p_5 is in the intersection

$B_2 = \{S_1, S_2, S_3, S_5, S_6\}$: Implements automaton v_2

$B_{21} = \{S_1, S_2, S_3\}$: p_1 is in the intersection

$B_{22} = \{S_1, S_3, S_5, S_6\}$

$B_{221} = \{S_1, S_3, S_5\}$: p_2 is in the intersection

$B_{222} = \{S_1, S_3, S_6\}$: p_3 is in the intersection

$B_{223} = \{S_1, S_5, S_6\}$: p_4 is in the intersection

$B_{224} = \{S_3, S_5, S_6\}$: p_6 is in the intersection

$B_{23} = \{S_1, S_2, S_5, S_6\}$

$B_{231} = \{S_1, S_2, S_5\}$: p_2 is in the intersection

$B_{232} = \{S_1, S_2, S_6\}$: p_3 is in the intersection

$B_{233} = \{S_1, S_5, S_6\}$: p_4 is in the intersection
$B_{234} = \{S_2, S_5, S_6\}$: p_5 is in the intersection
$B_{24} = \{S_2, S_3, S_5, S_6\}$
$B_{241} = \{S_2, S_3, S_5\}$: p_2 is in the intersection
$B_{242} = \{S_2, S_3, S_6\}$: p_3 is in the intersection
$B_{243} = \{S_3, S_5, S_6\}$: p_6 is in the intersection
$B_{244} = \{S_2, S_5, S_6\}$: p_5 is in the intersection

$B_3 = \{S_1, S_3, S_4, S_6\}$: Implements automaton v_3
$B_{31} = \{S_1, S_4, S_6\}$: p_4 is in the intersection
$B_{32} = \{S_1, S_3, S_4\}$: p_1 is in the intersection
$B_{33} = \{S_1, S_3, S_6\}$: p_3 is in the intersection
$B_{34} = \{S_3, S_4, S_6\}$: p_6 is in the intersection
$B_4 = \{S_2, S_4, S_5, S_6\}$: p_5 is in the intersection; Implements automaton v_4

This construction resembles a set of trees, one for each automaton. In fact, we can use these trees to recursively vote on values. For example, with a distributed register, when reading its value from a set of automata, we can read the values of each of the corresponding automaton replicas and vote recursively using its tree. Also, as we mentioned previously, if we have an atomic broadcast object available, then we can use this object to vote on messages of the different replicas of an automaton to generate one single sequence of messages to each replica (automata in principle is not aware of replication).

Theorem 5. *Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile and $k > 1$ a degree of replication. The transformation terminates only if $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfies k-Intersection.*

Proof. We prove this theorem by contradiction. Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile that does not satisfy k-Intersection, and that the bin procedure from Figure 3 that places automata in processors executes successfully. For such a system, there is a set $\{S_1, S_2, \cdots, S_k\} \subseteq S_\Pi$ such that $\cap_i S_i = \emptyset$. If $|S_\Pi| < k$, then the procedure from Figure 1 fails because in placing every survivor set in exactly k bins, we finish with at least one containing k survivor sets. More specifically, we have that the every survivor set in S_Π must be in exactly k bins. This gives us $(|S_\Pi| - 1) \cdot (k + 1)$ valid positions for placing the survivor sets, but we need $(|S_\Pi| \cdot k)$, and $(|S_\Pi| - 1) \cdot (k + 1) < (|S_\Pi| \cdot k)$ for $|S_\Pi| < k$. This violates the rules for populating the bins. Thus, S_Π must be such that $|S_\Pi| \geq k$. In splitting a subset $\mathcal{S}$ of S_Π of size $k' \geq k$ into $k + 1$ bins following the rules for populating these bins, we have that for every combination C of k survivor sets in $\mathcal{S}$, there is a bin B_i such that all survivor sets in C are in B_i. To see this, we observe that every k combinations of $k + 1$ elements must intersect in at least one element. With simple induction on the size of $\mathcal{S}$, we can show that there is a call to the procedure from Figure 1 with arguments $\mathcal{S}$ and w, such that $|\mathcal{S}| = k$ and $\{S_1, S_2, \cdots, S_k\} \subseteq \mathcal{S}$. In this call, we have that each bin B_i must have exactly $k - 1$ survivor sets, by the rules of the procedure for populating, and these survivor sets do not intersect in any element by assumption. The subsequent calls to the procedure from Figure 1 for each B_i are such that the argument $\mathcal{S} = B_i$ and $|\mathcal{S}| < k$. This case, however, corresponds to the first case we analyzed. The execution of the procedure from Figure 1 fails, contradicting our initial assumption.

5 Equivalence Classes of Adversarial Structures

To end, we would like to point out that, directly from previous section, the hierarchy of k-intersection properties defines then, under given constraints and assumptions, equivalence classes of adversarial structures regarding solvability.

Corollary 1. *A protocol requiring $n > kt$ replication is solvable for a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ if and only if $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfies k-intersection.*

6 Conclusions

We introduced a Byzantine-resilient transformation that translates any threshold protocol to work in a dependent failure model. We also defined *equivalence classes of adversarial structures*, regarding solvability, by making use of a particular set of hierarchical properties based on set intersection.

The importance of our results is both theoretical and practical. As we transform algorithms from the threshold model into algorithms of a dependent failure model, we are showing the equivalence of these two models, under the constraints of our theorems and constructions. In practice, one of the potential advantages of such a translation is being able to execute with fewer than n automata, thus saving compute power. Two important practical questions are how efficient the transformed algorithms are and if there are more efficient transformations.

References

1. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. McGraw-Hill, New York (1998)
2. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for t-resilient asynchronous computations. In: Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing (STOC 1993), pp. 91–100. ACM Press, New York (1993)
3. Budhiraja, N., Marzullo, K., Schneider, F., Toueg, S.: Optimal primary-backup protocols. In: Proceedings of the Sixth International Workshop on Distributed Algorithms (WDAG 1997), pp. 362–378 (November 1992)
4. Castro, M., Liskov, B.: Practical byzantine fault-tolerance and proactive recovery. ACM Transactions on Computer Systems 20, 398–461 (2002)
5. Castro, M., Rodrigues, R., Liskov, B.: BASE: Using abstraction to improve fault tolerance. ACM Transactions on Computer Systems 21, 236–269 (2003)
6. Ekwall, R., Urban, P., Schiper, A.: Robust TCP connections for fault tolerant computing. In: Proceedings of the Ninth IEEE International Conference on Parallel and Distributed Systems, pp. 501–508. ACM Press, New York (2002)
7. Garcia-Molina, H., Barbara, D.: How to assign votes in a distributed system. Journal of the ACM 32(4), 841–860 (1985)

8. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer, Heidelberg (2006)
9. Guerraoui, R., Vukolic, M.: Refined quorum systems. In: Proceedings of the Twenty-Sixth ACM Symposium on Principles of Distributed Computing (PODC 2007), pp. 119–128. Springer, Heidelberg (2007)
10. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
11. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM 46(6), 858–923 (1999)
12. Hirt, M., Maurer, U.: Complete characterization of adversaries tolerable in secure multi-party computation. In: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1997), pp. 25–34 (August 1997)
13. Junqueira, F.: Coping with Dependent Failures in Distributed Systems. Number 0737 in CS2003. Ph.D. Thesis, UC San Diego (September 2002)
14. Junqueira, F., Marzullo, K.: Designing algorithms for dependent process failures. Future Directions in Distributed Computing 2584, 24–28 (2003)
15. Junqueira, F., Marzullo, K.: Synchronous consensus for dependent process failures. In: Proceedings of the Conference on Distributed Computing Systems (ICDCS 2003), pp. 274–283. Springer, Heidelberg (2003)
16. Junqueira, F., Marzullo, K.: Replication predicates for dependent-failures algorithms. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 617–632. Springer, Heidelberg (2005)
17. Junqueira, F., Marzullo, K.: A framework for the design of dependent-failure algorithms. Concurrency and Computation: Practice and Experience 19(17), 2255–2269 (2007)
18. Lamport, L.: Fast Paxos. Distributed Computing 19, 79–103 (2006)
19. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
20. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11(4) (October/June 1998)
21. Neumann, P.G.: Computer Related Risks. ACM Press, New York (1995)
22. Schneider, F.: Implementing fault-tolerant services using the state-machine approach: a tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
23. Warns, T., Freiling, F.C., Hasselbring, W.: Consensus using structural failure models. In: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006), pp. 212–224. Springer, Heidelberg (2006)
24. Zieliński, P.: Automatic verification and discovery of Byzantine consensus protocols. In: The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), pp. 25–28. IEEE Computer Society, Los Alamitos (2007)

Hopscotch Hashing

Maurice Herlihy[1], Nir Shavit[2,3], and Moran Tzafrir[3]

[1] Brown University, Providence, RI
[2] Sun Microsystems, Burlington, MA
[3] Tel-Aviv University, Tel-Aviv, Israel

Abstract. We present a new class of resizable sequential and concurrent hash map algorithms directed at both uni-processor and multicore machines. The new hopscotch algorithms are based on a novel *hopscotch* multi-phased probing and displacement technique that has the flavors of chaining, cuckoo hashing, and linear probing, all put together, yet avoids the limitations and overheads of these former approaches. The resulting algorithms provide tables with very low synchronization overheads and high cache hit ratios.

In a series of benchmarks on a state-of-the-art 64-way Niagara II multicore machine, a concurrent version of hopscotch proves to be highly scalable, delivering in some cases 2 or even 3 times the throughput of today's most efficient concurrent hash algorithm, Lea's `ConcurrentHashMap` from *java.concurr.util*. Moreover, in tests on both Intel and Sun uni-processor machines, a sequential version of hopscotch consistently outperforms the most effective sequential hash table algorithms including cuckoo hashing and bounded linear probing.

The most interesting feature of the new class of hopscotch algorithms is that they continue to deliver good performance when the hash table is more than 90% full, increasing their advantage over other algorithms as the table density grows.

1 Introduction

Hash tables are one of the most widely used data structures in computer science. They are also one of the most thoroughly researched, because any improvement in their performance can benefit millions of applications and systems.

A typical resizable hash table is a continuously resized array of buckets, each holding an expected constant number of elements, and thus requiring an expected constant time for add(), remove(), and contains() method calls [1]. Typical usage patterns for hash tables have an overwhelming fraction of contains() calls [2], and so optimizing this operation has been a target of many hash table algorithms.

This paper introduces *hopscotch hashing*, a new class of open addressed resizable hash tables that are directed at today's cache-sensitive machines.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 350–364, 2008.

1.1 Background

Chained hashing [3] is closed address hash table scheme consisting of an array of buckets each of which holds a linked list of items. Though closed addressing is superior to other approaches in terms of the time to find an item, its use of dynamic memory allocation and the indirection makes for poor cache performance [4]. It is even less appealing for a concurrent environment as dynamic memory allocation typically requires a thread-safe memory manager or garbage collector, adding overhead in a concurrent environment.

Linear probing [3] is an open-addressed hash scheme in which items are kept in a contiguous array, each entry of which is a bucket for one item. A new item is inserted by hashing the item to an array bucket, and scanning forward from that bucket until an empty bucket is found. Because the array is accessed sequentially, it has good cache locality, as each cache line holds multiple array entries. Unfortunately, linear probing has inherent limitations: because every contains() call searches linearly for the key, performance degrades as the table fills up (when the table is 90% full, the expected number of locations to be searched until a free one is found is about 50 [3]), and clustering of keys may cause a large variance in performance. After a period of use, a phenomenon called *contamination* [5], caused by deleted items, degrades the efficiency of unsuccessful contains() calls.

Cuckoo hashing [4] is an open-addressed hashing technique that unlike linear probing requires only a deterministic constant number of steps to locate an item. Cuckoo hashing uses two hash functions. A new item x is inserted by hashing the item to two array indexes. If either slot is empty, x is added there. If both are full, one of the occupants is displaced by the new item. The displaced item is then re-inserted using its other hash function, possibly displacing another item, and so on. If the chain of displacements grows too long, the table is resized. A disadvantage of cuckoo hashing is the need to access sequences of unrelated locations on different cache lines. A bigger disadvantage is that Cuckoo hashing tends to perform poorly when the table is more than 50% full because displacement sequences become too long, and the table needs to be resized.

Lea's algorithm [6] from *java.util.concurrent*, the Java™ Concurrency Package, is probably the most efficient known concurrent resizable hashing algorithm. It is a closed address hash algorithm that uses a small number of high-level locks rather than a lock per bucket. Shalev and Shavit [7] present another high-performance lock-free closed address resizable scheme. Purcell and Harris [8] were the first to suggest a nonresizable open-addressed concurrent hash table based on *linear probing.* A concurrent version of cuckoo hashing can be found in [9].

2 The Hopscotch Hashing Approach

Hopscotch hashing algorithms are open addressed algorithms that combine elements of cuckoo hashing, linear probing, and chaining, in a novel way. Let us begin by describing a simple variation of the hopscotch approach, later to be refined as we present our actual implementations.

The hash table is an array of buckets. The key characteristic of the hopscotch approach is the notion of a *neighborhood* of buckets around any given items bucket. This neighborhood has the property that the cost of finding the desired item in any of the buckets in the neighborhood is the same or very close to the cost of finding it in the bucket itself. Hopscotch hashing algorithms will then attempt to bring an item into its neighborhood, possibly at the cost of displacing other items.

Here is how our simple algorithm works. There is a single hash function h. The item hashed to an entry will always be found either in that entry, or in one of the next $H-1$ neighboring entries, where H is a constant (H could for example be 32, the standard machine word size). In other words, the neighborhood is a "virtual" bucket that has fixed size and overlaps with the next $H-1$ buckets. Each entry includes a *hop-information* word, an H-bit bitmap that indicates which of the next $H-1$ entries contain items that hashed to the current entry's virtual bucket. In this way, an item can be found quickly by looking at the word to see which entries belong to the bucket, and then scanning through the constant number of entries (on most machines this requires at most two loads of cache lines).

Here is how to add item x where $h(x) = i$:

- Starting at i, use linear probing to find an empty entry at index j.
- If the empty entry's index j is within $H-1$ of i, place x there and return.
- Otherwise, j is too far from i. To create an empty entry closer to i, find an item y whose hash value lies between i and j, but within $H-1$ of j, and whose entry lies below j. Displacing y to j creates a new empty slot closer to i. Repeat. If no such item exists, or if the bucket already i contains H items, resize and rehash the table.

In other words, the idea is that hopscotch "moves the empty slot towards the desired bucket" instead of leaving it where it was found as in linear probing, or moving an item out of the desired bucket and only then trying to find it a new place as in cuckoo hashing. The cuckoo hashing sequence of displacements can be cyclic, so implementations typically abort and resize if the chain of displacements becomes too long. As a result, cuckoo hashing works best when the table is less than 50% full. In hopscotch hashing, by contrast, the sequence of displacements cannot be cyclic: either the empty slot moves closer to the new item's hash value, or no such move is possible. As a result, hopscotch hashing supports significantly higher loads (see Section 5). Moreover, unlike in cuckoo hashing, the chances of a successful displacement do not depend on the hash function h, so it can be a simple function that is easily shown to be close to universal.

The hopscotch neighborhood is a virtual bucket with multiple items, but unlike in chaining these items have great locality since typically items can be located in adjacent memory locations and even in the same cache lines. Items are inserted in constant expected time as in linear probing, but without fear of clustering and contamination.

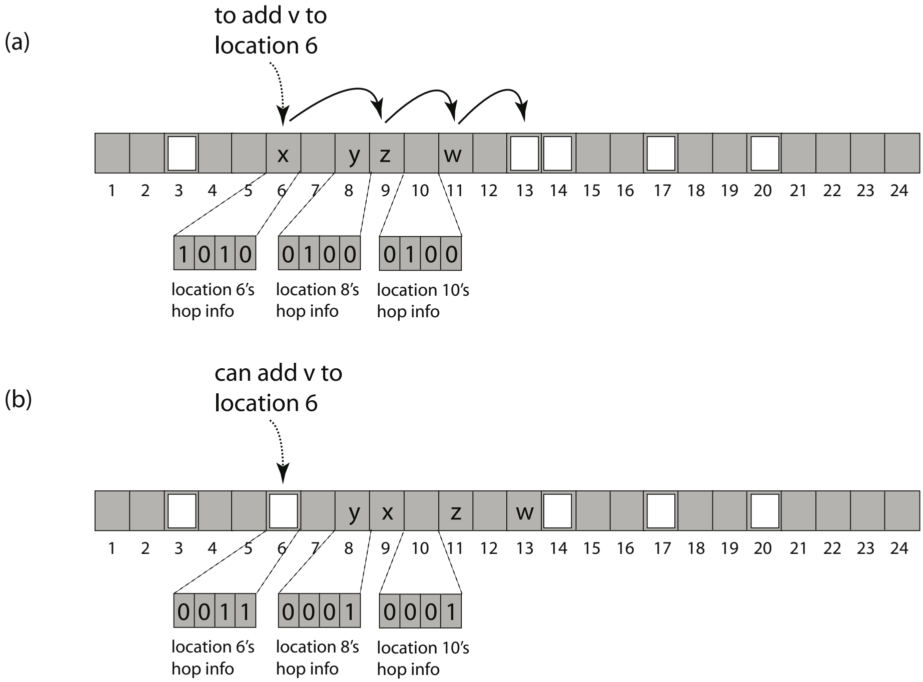


Fig. 1. The blank entries are empty, all others contain items. Here, H is 4. In part (a), we add item v with hash value 6. A linear probe finds entry 13 is empty. Because 13 is more than 4 entries away from 6, we look for an earlier entry to swap with 13. The first place to look is $H - 1 = 3$ entries before, at entry 10. That entry's hop information bit-map indicates that w at entry 11 can be displaced to 13, which we do. Entry 11 is still too far from entry 6, so we examine entry 8. The hop information bit-map indicates that z at entry 9 can be moved to entry 11. Finally, x at entry is moved to entry 9. Part (b) shows the table state just before adding v.

Finally, notice that in our simple algorithm, if more than a constant number of items are hashed by h into a given bucket, the table needs to be resized. Luckily, for a universal hash function h, the probability of this type of resize happening for a given H is $\Theta(1/H!)$. [1] Figure 1 shows an example execution of the simple hopscotch algorithm.

The concurrent version of the simple hopscotch algorithm maps a bucket to each lock. This lock controls access to all table entries that hold items of that bucket. The contains() calls are obstruction-free [10], while the add() and remove() acquire the bucket lock before applying modifications to the data.

In the next section we describe in detail a cache aware version of the hopscotch approach.

[1] In our implementation, as we describe in the next section, we can actually use a pointer scheme in the *hop-information* word instead of the easier to explain bitmap, and so buckets can actually grow H dynamically.

3 A Hopscotch Hashing Algorithm

We now present the more complex algorithm in the hopscotch class, the one used in our benchmarked implementations. The high level ideas behind this algorithm, in both the sequential and concurrent case, are as follows.

The table consists of segments of buckets, and in the concurrent case each segment is protected by a lock in a manner similar to that used in [6]. Instead of a bitmap representation of the *hop-information* word. We represent keys in each "virtual" bucket as a linked list, starting at the key's original bucket, and distributed throughout other buckets reachable via a sequence of pointers from the original one. Each pointer can point a distance *hop_range* to the next item.

The add() method in Figure 3 first tries to add it to some free bucket in its original bucket's cache line (line 8). If that fails, it probes linearly to find an empty slot within hop range of the key's original bucket, and then points to it from the last bucket in key's virtual bucket, that is, the last bucket in the list of buckets originating in the key's original bucket. If it could not find such an empty bucket (line 18), it applies a sequence of hopscotch displacements, trying to displace the free-bucket to reside within hop_range of the original bucket of

```
template <typename _tKey, typename _tData, typename _tHash, typename _tEqual,
          _tKey _empty_key, _tData _empty_data, typename _tLock>
class ConcurrentHopscotchHashMap {
  static const short _null_delta = SHRT_MIN;
  //inner classes
  struct Bucket{short volatile _first_delta ; short volatile _next_delta ;
                _tKey volatile _key;          _tData volatile _data;};
  struct Segment {_tLock      _lock ;     Bucket*     _table ;
                  int volatile _timestamp; int volatile _count;
                  Bucket*      _last_bucket ;};
  // fields
  const int      _segment_shift ;
  const int      _segment_mask;
  Segment* const _segments;
  int            _bucket_mask;
  const int      _cache_mask;
  const bool      _is_cacheline_alignment ;
  cont int       _hop_range;
public:
  ConcurrentHopscotchHashMap(int initial_capacity , int concurrency_level ,
                  int cache_line_size , bool is_optimize_cacheline , int hop_range);
  virtual ~ConcurrentHopscotchHashMap();
  virtual bool contains(const _tKey key);
  virtual _tData get(const _tKey key);
  virtual _tData add(const _tKey key, const _tData data);
  virtual _tData remove(const _tKey key);
};
```

Fig. 2. The hopscotch class

```
virtual _tData add(const _tKey key, const _tData data) {
  const int       hash          (_tHash::Calc(key));
  Segment&        segment       (getSegment(hash));
  Bucket* const start_bucket (getBucket(segment, hash));
  segment._lock.aquire();
  if(contains(key)) {segment._lock.release(); return keys data;}
  Bucket* free_bucket=freeBucketInCacheline(segment, start_bucket);
  if(null != free_bucket) {
    addKeyBeginingList(start_bucket, free_bucket, hash, key, data);
    ++(segment._count);
    segment._lock.release();
    return null;
  }
  free_bucket=nearestFreeBucket(segment, start_bucket);
  if(null != free_bucket) {
    int free_dist =(free_bucket - start_bucket);
    if( free_dist > _hop_range || free_dist < -_hop_range)
    free_bucket=displaceFreeRange(segment, start_bucket, free_bucket);
    if(null != free_bucket) {
      addKeyEndList(start_bucket, free_max_bucket, hash, key, data, last_bucket);
      ++(segment._count);
      segment._lock.release();
      return null;
    }
  }
  segment._lock.release();
  resize();
  return add(key, data);
}
```

Fig. 3. add() pseudo-code

the key. If such a sequence of displacements cannot be performed then the table is resized.

The remove() method in Figure 4 removes a key and tries to optimize the cache line alignment of keys belonging to the neighborhood of the emptied bucket. The idea is to find a key that will, once moved into the emptied bucket, reside in its associated bucket's cache line, thus improving cache performance. In the concurrent version of remove(), while traversing the bucket's list, the method locks segments as it proceeds. The remove() operations will modify the _timestamp field, since concurrent contains() calls need to be failed by concurrent removes.

Notice that it may be the case that once a key x is removed, the preceding key pointing to the x in its associated bucket's list, may find that the item following x in the bucket's list is outside its *hop_range*. In this case, the remove will find the last item y in the list belonging to x's bucket and displace it to x's empty bucket. The new empty bucket of the removal of x will thus be the bucket of the displaced y.

```
virtual _tData remove(const _tKey key) {
  const int       hash          (_tHash::Calc(key));
  Segment&        segment       (getSegment(hash));
  Bucket* const   start_bucket  (getBucket(segment,hash));
  Bucket*         last_bucket   ( null );
  Bucket*         curr_bucket   ( start_bucket );
  segment._lock.aquire();
  short next_delta (curr_bucket->_first_delta );
  do {
     if ( _null_delta ==next_delta) {segment._lock.release(); return null;}

     curr_bucket += next_delta;
     if (_tEqual::IsEqual(key, curr_bucket->_key)) {
        _tData const found_key_data(curr_bucket->_data);
        removeKey(segment, start_bucket, curr_bucket, last_bucket, hash);
        if ( _is_cacheline_alignment ) cacheLineAlignment(segment, curr_bucket);
        segment._lock.release ();
        return found_key_data;
     }
     last_bucket = curr_bucket;
     next_delta = curr_bucket->_next_delta;
  } while(true);
  return null;
}
```

Fig. 4. remove() pseudo-code

```
virtual bool contains(const _tKey key) {
  const int hash          (_tHash::Calc(key));
  Segment& segment        (getSegment(hash));
  Bucket*  curr_bucket    (getBucket(segment, hash));
  int start_timestamp;
  do {
     start_timestamp = segment._timestamp[hash & _timestamp_mask];
     short next_delta (curr_bucket->_first_delta );
     while ( _null_delta != next_delta) {
        curr_bucket += next_delta;
        if (_tEqual::IsEqual(key, curr_bucket->_key))
           return true;
        next_delta = curr_bucket->_next_delta;
     }
  } while(start_timestamp != segment._timestamp[hash & _timestamp_mask]);
  return false;
}
```

Fig. 5. contains() pseudo-code

To optimize the key's cache-line alignment (line 16), for each of the buckets in the cache line of the empty bucket, the method traverses the associated lists and moves the last item in the list into the empty bucket. This process can be performed recursively.

The contains() method appears in Figure 5. It traverses the buckets key list, and if a key is not found (line 15) there are two possible cases: (i) the key does not exist or (ii) a concurrent remove() (detected via a changed _timestamp in the associated segment) caused the contains() to miss the key, in which case the method looks for the key again.

4 Analysis

Unsuccessful add() and remove() methods are linearized respectively at the points where their internal contains() method calls are successful in finding the key (for add()) or unsuccessful (for remove()). A successful add() is linearized when it adds the new bucket to the list of buckets that hashed to the same place, either updating by _next_delta or _first_delta , depending on whether the bucket is in the cache line. A successful remove() is linearized when it overwrites the key from the table entry. A successful contains() is linearized when it finds the desired key in the array entry. An unsuccessful contains() is linearized when it reached the end of the list, and found the timestamp unchanged.

The add() and remove() methods use lockqs. They are deadlock-free but not livelock-free. The contains() method is obstruction-free.

We next analyze the complexity of the sequential and concurrent versions of the hopscotch algorithm. The most important property of a hash table is its expected constant time performance. We will assume that the hash function h is universal and follow the standard practice of modeling the hash function as a uniform distribution over keys [1]. As before, the constant H is the maximal number of keys a bucket can hold, n is the number of keys in the table, m is the table size, and $\alpha = n/m$ is the density or *load factor* of the table.

Lemma 1. *The expected number of items in a bucket is*

$$f(m,n) = 1 + \frac{1}{4}((1+\frac{2}{m})^n - 2\frac{n}{m}) \approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{4}$$

Proof. The expected number of items in a hopscotch bucket is the same as the expected number of items in a chained-hashing bucket. The result follows from Theorem 15 in Chapter 6.4 of [3].

In hopscotch hashing, as in chained hashing, in the common case there are very few items in a bucket.

Lemma 2. *The maximal expected number of items in a bucket is constant.*

Proof. Again, following Knuth [3], the function $f(m,n)$ is increasing, and the maximal value for n, the number of items, is m, so that the function's value tends to approximately 2.1.

This implies that that typically there is very little work to be performed when searching for an item in a bucket.

Lemma 3. *Calls to* remove() *and* contains() *complete in constant expected time when running in isolation.*

Proof. A complete proof will appear in the final paper. Suppose that there r threads calling remove(), a threads calling add(), and c threads calling contains(). Since the number of segments n is at least the number of threads, $(a+c+r) <= n$. The probability of having at least one segment with remove() and contains() assigned to it is $n\left(\frac{n^{c-1}n^{a}n^{r-1}}{n^n}\right) = \frac{1}{n}$. By Lemma 2, each iteration in contains() takes at most a constant expected number of steps. The expected number of iteration is:

$$\begin{aligned} 1 + \sum \frac{i}{n^i} = \frac{\frac{k(1+k)}{2}}{\frac{1-n^k}{1-n}} &= 1 + \frac{k(1+k)(1-n)}{2(1-n^k)} \\ &= 1 + \frac{k(1+k)(1-n)}{2(1-n)\sum 1^t n^{k-1-t}} \\ &= 1 + \frac{k(1+k)}{2\sum 1^t n^{k-1-t}} \\ &\rightarrow 1 \end{aligned}$$

It is known that the worst case number of items in a bucket, even when using a universal hash function, is not constant [3]. So what are the chances of a resize () due to overflow? It turns out that chances are low.

Lemma 4. *The probability of having more then H keys in a bucket is $1/H!$.*

Proof. The probability of having H keys in a bucket is equal to the probability of having a chain of length H in chained hashing. From section 3.3.10 of [5] it follows that

$$\begin{aligned} \Pr\{\text{H items in bucket}\} &= \binom{n}{H}\frac{(m-1)^{n-H}}{m^n} \\ &= \frac{n!}{H!(n-H)!}\frac{(m-1)^{n-H}}{m^n} \\ &= \frac{1}{H!}n(n-1)\ldots(n-H+1)\frac{(m-1)^{n-H}}{m^{n-H}m^H} \\ &\leq \frac{1}{H!} \end{aligned}$$

The last inequality follows by substituting m for n, where m is a monotonically increasing function's maximal value.

With uniform hashing, the probability of resizing due to having more than $H = 32$ items in a bucket is $1/32!$.

Lemma 5. *The probability of probing for an empty slot, with length bigger then H is α^{H-1}.*

The result follows from Theorem 11.6 of [1]. For $\alpha = 0.8$, $Pr(H = 32) \approx 0.000990$, $Pr(H = 64) \approx 7.846377 \cdot 10^{-7}$, and for $\alpha = 0.9$ $Pr(H = 128) \approx 1.390084 \cdot 10^{-6}$, $Pr(H = 256) \approx 7.846377 \cdot 10^{-7}$. This is true for universal hash functions. The experimental results show that using the *displacement-algorithm* the probability of failing to find an empty slot in range H is smaller. For example, for $\alpha = 0.8$ and $H = 32$, no resize occurred. The analysis of the probably to fail to find an empty slot in range H, when using the displacement algorithm is left for the full paper.

Lemma 6. *Calls to* add() *complete within expected constant time.*

Proof. From Knuth [3] for an open-address hash table that is $\alpha = n/m < 1$ full, the expected number of entries until an open slot is found is at most $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}^2\right)$ which is constant. Thus, the expected number of displacements, which is bounded by the number of entries tested until an open slot is found is at most constant.

For example, if the hash table is 50% full, the expected number of entries tested until an open slot is found is at most $1/2(1+(1/(1-0.5))^2) = 2.5$ and when it is 90% full, the expected number of entries tested is $1/2(1+(1/(1-0.9))^2) = 50$. In the full paper we will show that:

Lemma 7. *Calls to* resize() *complete within $O(n)$ time.*

5 Performance Evaluation

This section compares the performance of hopscotch hashing to the most effective prior algorithms in both concurrent (multicore) and sequential (uniprocessor) settings. We use micro-benchmarks similar to those used by recent papers in the area [4,7], but with significantly higher table densities (up to 99%).

We sampled each test point 10 times, and plotted the average. To make sure that the table does not fit into a cache-line, we use a table size of approximately 2^{23} items. Each test used the same set of keys for all the hash-maps. All tested hash-maps were implemented using C++, and were compiled using the same compiler and settings. Closed-address hash-maps, like chained-hashing, dynamically allocate list nodes, in contrast with open-address hash-maps like linear-probing. To ensure that memory-management costs do not skew our numbers, we show results for the closed-address hash-maps both with the mtmalloc multi-threaded malloc library, and with pre-allocated memory.

In all algorithms, each bucket encompasses pointers to the key and data (satellite key and data). This scheme is thus a general hash-map.

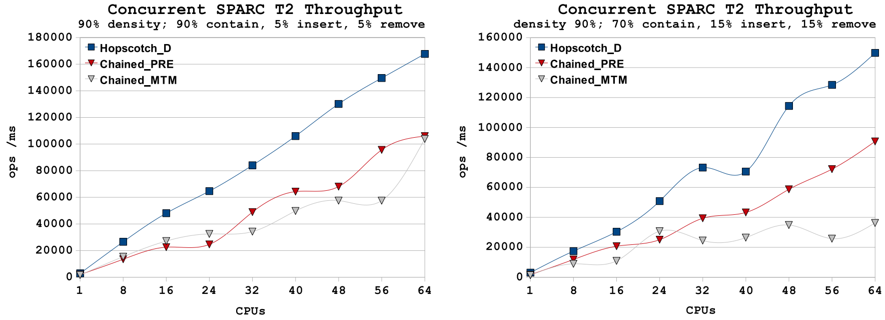


Fig. 6. Concurrent performance under high-loads

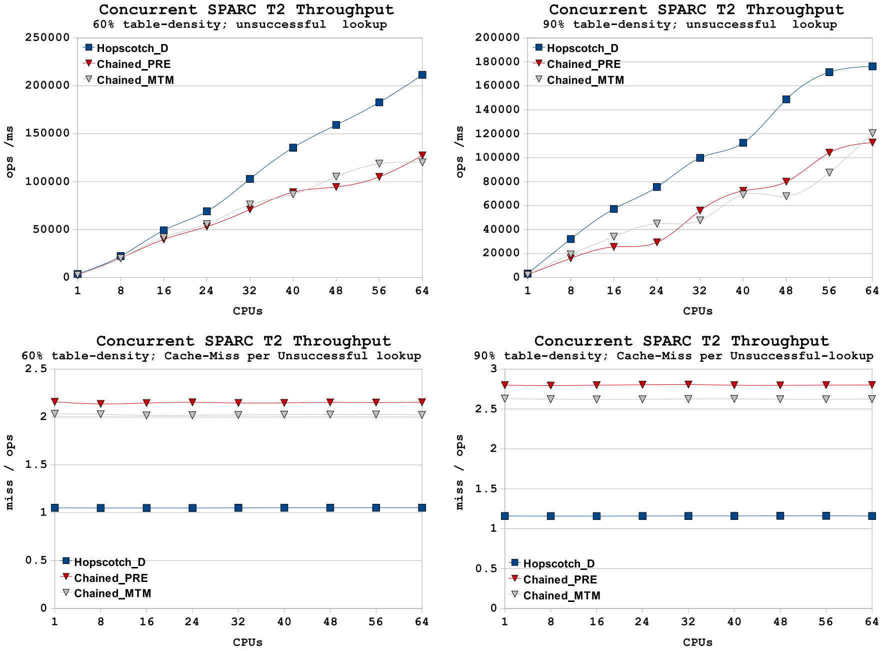
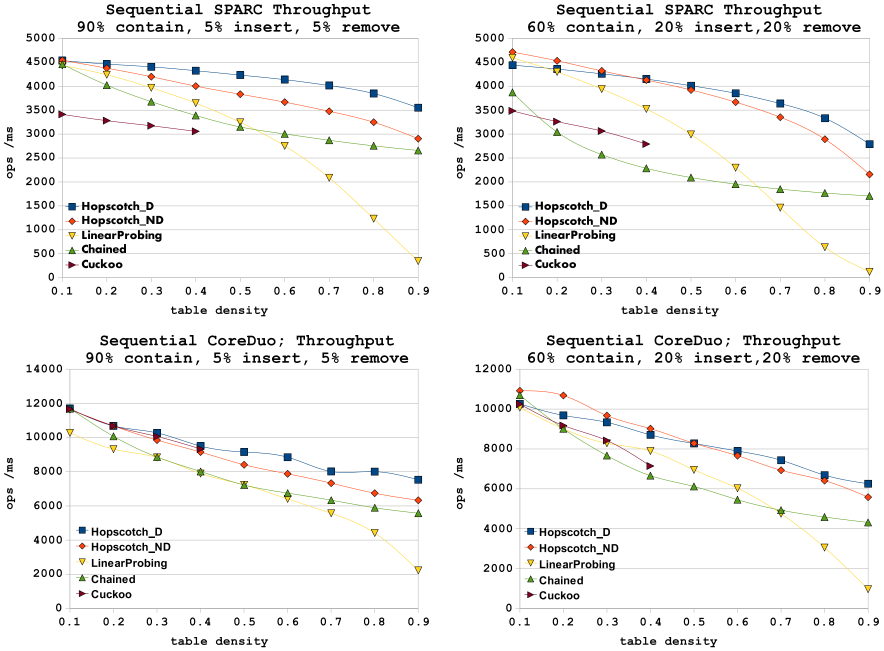


Fig. 7. Concurrent Benchmark: unsuccessful contains() throughput and cache-miss counts

5.1 Concurrent Hash-Maps on Multicores

We compared, on a 64-way Sun UltraSPARC^{TM} T2 multicore machine, two versions of the concurrent hopscotch hashing to the *Lock-based Chained* algorithm of Lea [6]: *Chained_PRE* is the pre-allocated memory version, and *Chained_MTM* is the mtmalloc library version, and the *New Hopscotch* algorithm: a variant of hopscotch that uses a an 16bit representation of the pointers in the *hop-information* word, providing an effectively unbounded range of 32767 locations.


Sequential SPARC Throughput 90% contain, 5% insert, 5% remove

Sequential SPARC Throughput 60% contain, 20% insert,20% remove

Sequential CoreDuo; Throughput 90% contain, 5% insert, 5% remove

Sequential CoreDuo; Throughput 60% contain, 20% insert,20% remove

Fig. 8. Sequential high-load benchmark

To neutralize any effects of the choice of locks, all lock-based concurrent algorithms use the same locks, one per memory segment.The number of segments is the concurrency level (number of threads). We also use the same hash function for all the algorithms (except *Cuckoo-Hashing*).

Figure 6 illustrates how the algorithms scale under high loads and a table density of 90%. As can be seen, hopscotch is about twice as fast as *Chained_PRE* and more than three times as fast as *Chained_MTM*.

Figure 7 compares the performance of the hash table's unsuccessful contains() calls which are more trying for all algorithms than successful ones. As can be seen, the hopscotch algorithm retains its advantage when only contains() operations are counted, implying that the difference in performance among the various algorithms is not due to any effect of the locking policy. The lower graphs explain the results: hopscotch suffers fewer cache misses per operation than the other algorithms. The benefit of using an open addressed table is further exemplified by the gap it opens from the chained algorithm once memory allocation costs are not discounted. Though we do not show it, the advantage in performance continues even as the table density reaches 99%, though with a smaller gap.

5.2 Sequential Hash-Maps

We selected the most effective known sequential hash-maps.

- *Linear-Probing*: we used an optimized version [3], that stores the maximal probe length in each bucket.

- *Chained*: We used an optimized version of [3] that stores the first element of each linked-list directly in the hash table.
- *Cuckoo*: Thanks to the kindness of the authors of [4], we obtained the original cuckoo hash map code.
- *New Hopscotch*: we used the sequential version. *Hopscotch_D* is a variant of hopscotch algorithm, that displaces keys to improve cache-line alignment. We contrast it with *Hopscotch_ND* that does not perform displacements to allow cache-line alignment.

We ran a series of benchmarks on two different uniprocessor architectures. A single core of a Sun UltraSPARC^{TM} T1 running at 1.20GHz, and an Intel^{R} Core^{TM} Duo CPU 3.50GHz.

The graphs in this section show throughput as function of the average table density. Tests cover high loads, extremely high table density, and contains() throughput. First, we contaminated the tables (e.g ran a long sequence of add() and calls). This preparation is important for a realistic analysis of open-address hash tables such as hopscotch or linear probing [3].

As can be seen, the performance of all the hash-maps diminishes with density. Nevertheless, the results show that the overhead associated with hopscotch's key-cache alignment and displacement does not diminish relative performance at high loads. As in the concurrent case, to ensure that memory allocation overhead

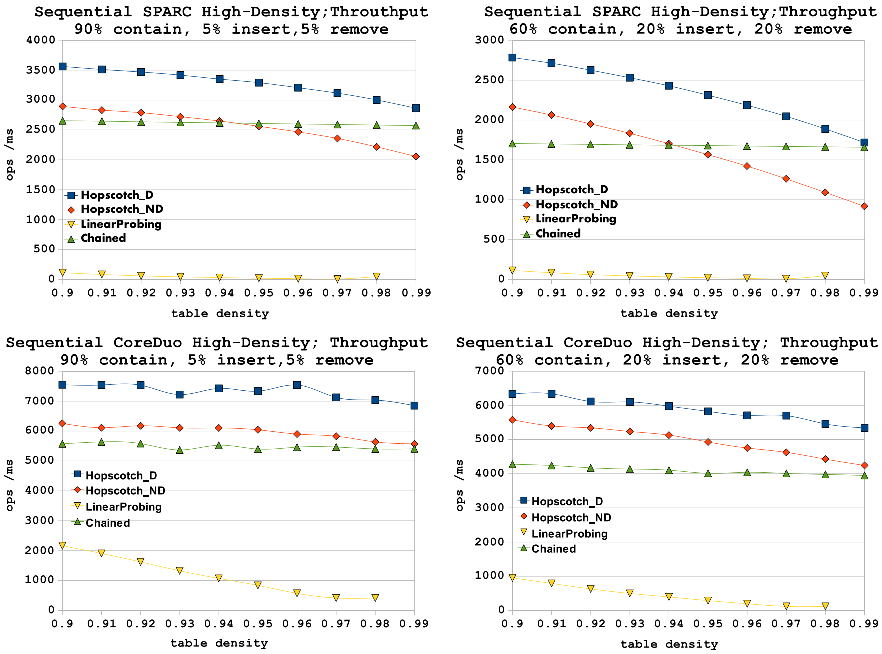


Fig. 9. Sequential extremely high table density benchmark under 60% load

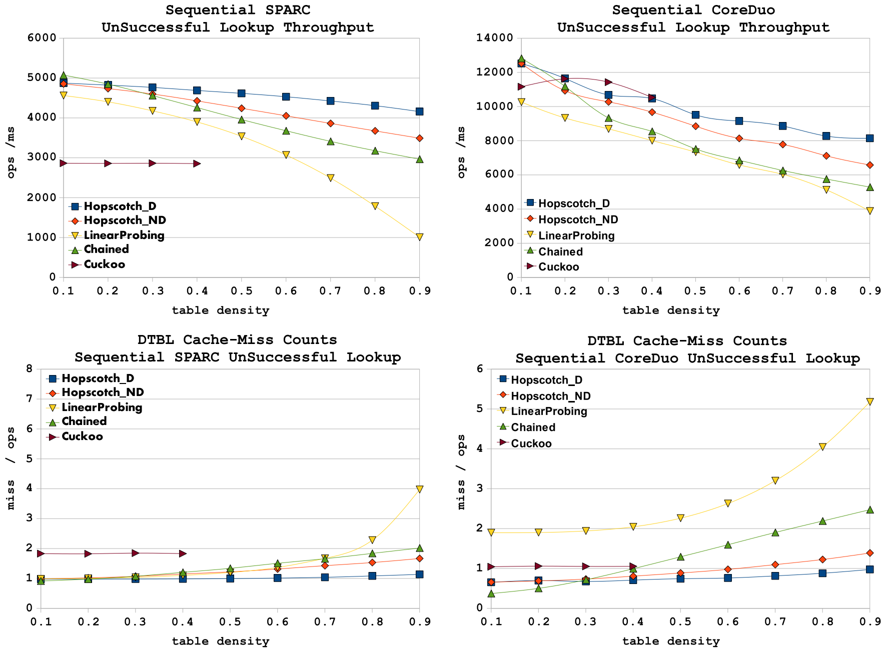


Fig. 10. Sequential contains() throughput with related cache miss counts below

does not distort the results, we use neutralized versions of the dynamic memory algorithms where all memory needed is preallocated.

Figure 9 shows the extremely high table density benchmark. As table density increases, so does the average number of keys per bucket, implying both longer CPU time for all hash-map operations, and the likelihood of more cache misses. As expected, cuckoo hashing does not carry beyond 50% table densities due to cycles in the displacement sequences. The open-addressed linear-probing does poorly compared to hopscotch. Note that the Hopscotch_D overhead of ensuring key's cache-line alignment and displacement proved beneficial even in this extreme setting.

Figure 10 shows the throughput of the unsuccessful contains() method calls, a measure dominated by memory access times. The lower graphs show how performance is completely correlated with cache miss rates and how the cache-line alignment of the hopscotch algorithm serves to allow it to dominate the other algorithms. Perhaps quite amazingly, hopscotch delivers less than one cache miss on average on the Intel architecture!

Acknowledgments

We thank Dave Dice and Doug Lea for their help throughout the writing of this paper.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
2. Lea, D.: Personal communication (January 2003)
3. Knuth, D.E.: The art of computer programming. In: Fundamental algorithms, 3rd edn. Addison Wesley Longman Publishing Co., Inc., Redwood City (1997)
4. Pagh, R., Rodler, F.F.: Cuckoo hashing. Journal of Algorithms 51(2), 122–144 (2004)
5. Gonnet, G.H., Baeza-Yates, R.: Handbook of algorithms and data structures: in Pascal and C, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1991)
6. Lea, D.: Hash table util.concurrent.concurrenthashmap in java.util.concurrent the Java Concurrency Package, `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/-jsr166/-src/main/java/util/concurrent/`
7. Shalev, O., Shavit, N.: Split-ordered lists: Lock-free extensible hash tables. Journal of the ACM 53(3), 379–405 (2006)
8. Purcell, C., Harris, T.: Non-blocking hashtables with open addressing. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 108–121. Springer, Heidelberg (2005)
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, NY (2008)
10. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS 2003: Proceedings of the 23rd International Conference on Distributed Computing Systems, p. 522. IEEE Computer Society, Washington (2003)

Computing Lightweight Spanners Locally

Iyad A. Kanj[1,⋆], Ljubomir Perković[1], and Ge Xia[2]

[1] School of Computing, DePaul University, 243 S. Wabash Ave., Chicago, IL 60604
[2] Department of Computer Science, Lafayette College, Easton, PA 18042

Abstract. We consider the problem of computing bounded-degree lightweight plane spanners of unit disk graphs in the local distributed model of computation. We are motivated by the hypothesis that such subgraphs can provide the underlying network topology for efficient unicasting and multicasting in wireless distributed systems. We present the *first* local distributed algorithm that computes a bounded-degree plane lightweight spanner of a given unit disk graph. The upper bounds on the degree, the stretch factor, and the weight of the spanner, are very small. For example, our results imply a local distributed algorithm that computes a plane spanner of a given unit disk graph U, whose degree is at most 14, stretch factor at most 8.81, and weight at most 8.81 times the weight of a Euclidean Minimum Spanning Tree of $V(U)$.

We show a wider application of our techniques by giving an $O(n \log n)$ time centralized algorithm that constructs bounded-degree plane lightweight spanners of unit disk graphs (which include Euclidean graphs), with the *best* upper bounds on the spanner degree, stretch factor, and weight.

1 Introduction

Efficiency, fault tolerance, scalability, and robustness are central goals in distributed computing. This is especially true for emerging wireless distributed systems such as ad-hoc, mesh, ubiquitous, and sensor networks. Efficiency is critical because wireless devices have typically very limited power. Fault tolerance is required because wireless communication is prone to many errors. Scalability is important because, in practice, wireless systems are often very large. Robustness is necessary to deal with the devices' mobility and the dynamic nature of wireless networks.

Most of the above goals can be achieved, to some extent, with algorithms developed under the *local* distributed computational model, as defined by Linial [15] and Peleg [16]. Assuming that the distributed system is modeled as a graph, a distributed algorithm is said to be *k-local* if, "intuitively", the computation at each point of the graph depends solely on the information about the points at

[⋆] The corresponding author. Email: ikanj@cs.depaul.edu. Supported in part by a DePaul University Competitive Research Grant.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 365–378, 2008.

distance (number of edges) at most k from the point (i.e., within k hops from the point). This notion can be formalized as follows [15,16,18]: a distributed algorithm is *k-local* if it runs in at most k synchronous communication rounds for some integer parameter $k > 0$. An algorithm is called *local* if it is k-local for some integer constant k. Efficient local distributed algorithms are naturally fault-tolerant and robust because faults and changes can be handled locally by such algorithms. These algorithms are also scalable because the computation performed by a device is not affected by the total size of the network. Therefore, it is natural to study what problems can or cannot be solved under this model, as did Kuhn, Moscibroda, and Wattenhofer in [12].

We focus our attention in this paper on developing efficient local algorithms for fundamental problems in emerging distributed systems technologies, such as wireless ad-hoc and sensor networks. For these applications, the network is often modeled as a *unit disk graph* (UDG) in the Euclidean plane: the points of the UDG correspond to the mobile wireless devices, and its edges connect pairs of points whose corresponding devices are in each other's transmission range equal to one unit.

The fundamental problem under consideration in this paper is the construction of *lightweight spanners* of a UDG U. The weight of each edge in U is defined to be its Euclidean distance, and the weight of a subgraph of U is the sum of the weights of its edges. It is well-known that a connected UDG contains a Euclidean Minimum Spanning Tree (EMST) of its point-set. A spanning subgraph of U is said to have *low* weight, or to be *lightweight*, if its weight is at most $c \cdot wt(\text{EMST})$ for some constant c. A subgraph H of U is said to be a *spanner* of U if there exists a constant ρ such that: for every two points $A, B \in U$, the weight of a shortest path between A and B in H is at most ρ times the weight of a shortest path between A and B in U. The constant ρ is called the *stretch factor* of H (with respect to U). Lightweight spanners of UDGs are fundamental to wireless distributed systems because they represent topologies that can be used for efficient unicasting *and* broadcasting. Lightweight spanners are also important in computational geometry, and much of the early work on lightweight spanners was done from that perspective under the centralized model of computation [1,2,5,6,7,8,13]. Additional requirements on spanners that have been considered are planarity and bounded degree [2,8,9,14,17]. These requirements are usually motivated by applications in wireless and sensor networks, whose devices have limited resources. For example, the planarity of the topology is often a requirement for efficient routing (see [3,9,11,14,17]).

The specific problem we are thus considering is the design of algorithms (in particular, local distributed algorithms) that construct bounded-degree plane lightweight spanners of unit disk graphs. Levcopoulos and Lingas [13] developed the first centralized algorithm for this problem on Euclidean graphs (i.e., the complete graph on n points in the plane), which are a special case of UDGs. Their $O(n \log n)$ time algorithm, given a rational $\lambda > 2$, produces a plane spanner with stretch factor $(\lambda - 1) \cdot C_{del}$ and total weight $(1 + \frac{2}{\lambda - 2}) \cdot wt(\text{EMST})$, where the constant $C_{del} \approx 2.42$ is the stretch factor of the Delaunay subgraph of the Euclidean graph. Althöfer et al. [1] gave a polynomial time greedy algorithm that constructs a lightweight plane spanner of a Euclidean graph having the same upper bound on the stretch factor and weight as the algorithm by Levcopoulos and Lingas [13]. The degree of the lightweight spanner in both [13] and [1], however, may be unbounded: it is not possible to bound the degree without worsening the stretch factor or the weight. A more recent $O(n \log n)$ time algorithm by Bose, Gudmundsson, and Smid [2] for Euclidean graphs, succeeded in bounding the degree of the plane spanner by 27 but at a large cost: the stretch factor of the obtained plane spanner is approximately 10.02, and its weight is $O(wt(\text{EMST}))$, where the hidden constant in the asymptotic notation is undetermined.

Our contribution with regard to this problem is a centralized algorithm for unit disk graphs, which include Euclidean graphs, that improves the above algorithms. We design a centralized algorithm that, for any integer constant $\Delta \geq 14$ and constant $\lambda > 2$, constructs a plane spanner of a unit disk graph (or a Euclidean graph) having degree at most Δ, stretch factor $(\lambda - 1) \cdot (1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$, and weight at most $(1 + \frac{2}{\lambda - 2}) \cdot wt(\text{EMST})$ (Theorem 3.1). We can compare our algorithm with the algorithm by Bose, Gudmundsson, and Smid [2] if we let $\Delta = 14$ and $\lambda \approx 2.475$ in Theorem 3.1: we obtain an $O(n \log n)$ time algorithm that, given a unit disk graph (or a Euclidean graph) on n points, computes a plane spanner of the given graph having degree at most 14, stretch factor at most 5.22, and weight at most $5.22 \cdot wt(\text{EMST})$.

We consider next the problem of computing bounded-degree plane lightweight spanners of unit disk graphs using a local distributed algorithm. To the best of our knowledge, the only distributed algorithm for this problem is the algorithm in [4]. While the distributed algorithm in [4] solves the problem for a generalization of unit disk graphs, called quasi-unit ball graphs, in higher dimensional Euclidean spaces, the algorithm is not local (it runs in a poly-logarithmic number of rounds), and the weight and the degree of the spanner are only bounded asymptotically. We note that distributed algorithms for computing lightweight spanners of general graphs have been extensively considered in the literature; see for example [16] for a survey on some of these results. In this paper we show that (Theorem 4.2): for any integer constant $\Delta \geq 14$ and constant $\lambda > 2$, there exists a k-local distributed algorithm, where $k = \lfloor (8/\pi) \cdot (\lambda + 1)^2 \rfloor$, that computes a plane spanner of a given unit disk graph containing a EMST on its point-set, of degree at most Δ, weight at most $(1 + \frac{2}{\lambda - 2}) \cdot wt(\text{EMST})$, and stretch factor $(\lambda - 1)^4 \cdot (1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$. This is the first local algorithm for this problem. If we set $\Delta = 14$ and $\lambda \approx 2.256$, we obtain a k-local algorithm with k at most 26, that computes a plane spanner of degree at most 14, stretch factor at most 8.81, and weight at most $8.81 \cdot wt(\text{EMST})$, of the given unit disk graph.

The remainder of this paper is organized as follows. We cover the preliminaries in Section 2. In Section 3 we present the centralized algorithm, and in Section 4 we present the local distributed algorithm. In Section 5 we give some further comparisons between our algorithm and the previous ones.

2 Preliminaries

Given a set of points S in the plane, the *Euclidean graph* E on S is defined to be the complete graph whose point-set is S. The *unit disk graph* U on S is the subgraph of E with the same point-set as E, and such that AB is an edge of U if and only if $|AB| \leq 1$, where $|AB|$ is the Euclidean length of edge AB. We assume in this paper that the unit disk graph U is connected. We define the *weight* of an edge AB to be the Euclidean distance between points A and B, that is $wt(AB) = |AB|$. For a subgraph $H \subseteq E$, we denote by $V(H)$ and $E(H)$ the set of vertices and the set of edges of H, respectively, and by $wt(H)$ the sum of the weights of all the edges in H, that is, $wt(H) = \sum_{XY \in E(H)} wt(XY)$. The *length* of a path P (resp. cycle C) in a subgraph $H \subseteq E$, denoted $|P|$ (resp. $|C|$), is the number of edges in P (resp. C). A point B is said to be a *k-neighbor* of A in a subgraph $H \subseteq E$, if there exists a path P from A to B in H satisfying $|P| \leq k$.

Each of the synchronous communication rounds in a local distributed algorithm consists of two phases: phase 1, in which every point receives messages sent to it in the preceding phase, and phase 2, in which every point sends messages to its neighbors. The local computation in a round occurs between the two phases. Since our focus is on wireless systems, we will assume that a message broadcast by a point in U will be received by all its neighbors.

The local distributed algorithm we develop in this paper constructs a subgraph of U and takes two steps. In the first step, all points learn about their k-hop neighbors using a local distributed algorithm. In the second step, each point runs a local computation to make a decision on what incident edges to select in the final spanner (no messages are exchanged in this step). A *k-local k-neighborhood algorithm* is a k-local algorithm in which each point learns about the coordinates of its k-hop neighbors. A basic k-local k-neighborhood algorithm runs as follows. In the first round, every point broadcasts its ID and coordinates to its neighbors in U. In the remaining $k-1$ rounds, every point broadcasts the ID and coordinates of every point it learned about in the previous round.

Let G be a plane graph and let T be a spanning tree of G. Call an edge $e \in E(T)$ a *tree edge* and an edge $e \in E(G) - T$ a *non-tree edge*. Every non-tree edge induces a unique cycle in the graph $T + e$ called the *fundamental cycle* of e. Since T is embedded in the plane, we can talk about the *fundamental region* of e, which is the closed region in the plane enclosed by the fundamental cycle of e (other than the outer face of $T + e$).

Definition 2.1. Define a relationship $\preceq$ on the set $E(G)$ as follows. For every edge e, $e \preceq e$. For two edges e and e' in $E(G)$, $e \preceq e'$ if and only if e is contained in the fundamental region of e'.

It is not difficult to verify that $\preceq$ is a partial order relation on $E(G)$, and hence $(E(G), \preceq)$ is a partially ordered set (POSET). Note that any two distinct tree edges are not comparable by $\preceq$, and that every tree edge is a minimal element in $(E(G), \preceq)$. Therefore, we can topologically sort the edges in $E(G)$ to form a list $\mathcal{L} = \langle e_1, \ldots, e_r \rangle$, in which no non-tree edge appears before a tree edge, and such that if $e_i \preceq e_j$ then e_i does not appear after e_j in $\mathcal{L}$.

Lemma 2.1. *Let e_i be a non-tree edge. Then there exists a unique face F_i in G such that every edge e_j of F_i satisfies $e_j \preceq e_i$.*

Proof. Let F_i be the face of G containing e_i and residing in the fundamental region of e_i, and let e_j be an edge on F_i. Since e_j is on F_i, e_j is contained in the fundamental region of e_i. By the definition of $\preceq$, we have $e_j \preceq e_i$. This shows the existence of such a face F_i.

To prove the uniqueness of F_i, suppose that there is another distinct face F_i' with the above properties. Since every edge e_j on F_i' satisfies $e_j \preceq e_i$, every edge on F_i' is contained in the fundamental region of e_i, and hence the whole face F_i' is contained in the fundamental region of e_i. This means that there are two distinct faces containing e_i that are enclosed within the fundamental cycle of e_i. This contradicts the planarity of G.

We will call the unique face associated with a non-tree edge e_i, described in Lemma 2.1, the *fundamental face* of e_i.

The following result is a consequence of the proof of Theorem 2 in [1]. A similar, but less general result, was also proved earlier by Levcopoulos and Lingas [13]. A different proof can also be found in [10].

Theorem 2.1. ([1])

(i) Let G be a connected weighted planar graph with nonnegative weights satisfying the following property: for every cycle C in G and every edge $e \in C$, $wt(C) \geq \lambda \cdot wt(e)$ for some constant $\lambda > 2$. Then $wt(G) \leq (1 + \frac{2}{\lambda-2}) \cdot wt(T)$, where T is a MST of G.

(ii) Let G be a connected weighted plane graph with nonnegative weights, and let T be a spanning tree in G. Let $\lambda > 2$ be a constant. Suppose that for every edge $e \in E(G) - T$ we have $wt(F_e) \geq \lambda \cdot wt(e)$, where F_e is the boundary cycle of the fundamental face of e in G. Then $wt(G) \leq (1 + \frac{2}{\lambda-2}) \cdot wt(T)$.

3 The Centralized Algorithm

In this section we present a centralized algorithm that constructs a bounded-degree plane lightweight spanner of U.

Kanj and Perlović [9] gave an $O(n \log n)$ time centralized algorithm that, given a Euclidean graph E on a set of n points in the plane, and an integer parameter $\Delta \geq 14$, constructs a plane spanner G' of E containing a EMST of $V(E)$, of degree at most Δ, and of stretch factor $\rho = (1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$, where $C_{del} \approx 2.42$ is the stretch factor of the Delaunay subgraph of E. This result can be extended to unit disk graphs:

Lemma 3.1. *For any $\Delta \geq 14$, the subgraph G_U' of the spanner G' described in [9], consisting of those edges in G' of weight at most 1, is a plane spanner* *of the unit disk graph U on $V(E)$ of degree at most Δ, and of stretch factor $\rho = (1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$ (with respect to U). Moreover, G'_U contains a EMST of $V(U)$.*

Proof. We need to verify that the subgraph G'_U of G' obtained by removing every edge of weight greater than 1 from G', is also a spanner of the unit disk graph U satisfying the same properties that G' satisfies with respect to Euclidean graph E.

It was shown in [9] that the spanner G' satisfies the property that, for every edge $AB \in E$ that is not in G', there exists a path P_{AB} from A to B in G' of weight at most $\rho \cdot wt(AB)$, and such that AB has maximum weight among all edges on P_{AB} (see Theorem 2.10 in [9]). Since the unit disk graph U is the subgraph of E consisting precisely of those edges in E of weight at most 1, by discarding from G' every edge of weight greater than 1, we obtain a subgraph G'_U of U that is plane and of degree at most Δ. Since U is connected, U contains a EMST of $V(U)$, and hence every edge in the EMST has weight at most 1. Since G' contains a EMST of $V(U)$, it follows from the preceding statement, and from the definition of G'_U, that G'_U contains a EMST of $V(U)$ as well. If an edge $AB \in U$ is not in G', from the properties of the spanner G' described above, there exists a path P_{AB} in G' whose weight is at most $\rho \cdot wt(AB)$, and on which AB is the edge of maximum weight. From the definition of G'_U, the path P_{AB} is also in G'_U. It follows that the same algorithm described in [9] computes a plane spanner G'_U of the unit disk graph U, of degree at most Δ, and of stretch factor $(1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$, for any integer parameter $\Delta \geq 14$. □

The spanner G'_U, however, may not be of light weight. Therefore, we need to discard edges from G'_U so that the resulting subgraph is of light weight, while at the same time not affecting the stretch factor of G'_U by much. To do so, since G'_U is a plane graph containing a EMST of $V(U)$, we would like to employ part (ii) of Theorem 2.1. However, there is one technical problem: the fundamental faces of G'_U may not satisfy the condition in part (ii) of Theorem 2.1, namely that the weight of every fundamental face F_e of a non-EMST edge e in G'_U satisfies $wt(F_e) \geq \lambda \cdot wt(e)$ ($\lambda > 2$ is a constant). We will show next how to prune the set of edges in G'_U so that this condition is satisfied.

Let T be a EMST of $V(U)$ contained in G'_U. As described in Section 2, we can order the non-tree edges in G'_U with respect to the partial order $\preceq$ described in Definition 2.1. Let $\mathcal{L}' = \langle e_1, e_2, \ldots, e_s \rangle$ be the sequence of non-tree edges in G'_U sorted in a non-decreasing order with respect to the partial order $\preceq$. Note that, by the definition of the partial order $\preceq$, if we add the edges in $\mathcal{L}'$ to T in the respective order they appear in $\mathcal{L}'$, once an edge e_i is added to form a fundamental face in the partially-grown graph, this fundamental face will remain a face in the resulting graph after all the edges in $\mathcal{L}'$ have been added to T. That is, the face will not be affected (i.e., changed/split) by the addition of any later edge in this sequence.

Given a constant $\lambda > 2$, to construct the desired lightweight spanner G, we first initialize G to the EMST T. We consider the non-tree edges of G'_U in the order that they appear in $\mathcal{L}'$. Inductively, suppose that we have processed the edges $e_1, \ldots, e_{i-1}$ in $\mathcal{L}'$. To process edge e_i, let F_i be the fundamental face of e_i in $G + e_i$. If $wt(F_i) > \lambda \cdot wt(e_i)$, we add e_i to G; otherwise, e_i is not added to G. This completes the description of the construction process. Let G be the resulting graph at the end of the construction process.

Lemma 3.2. *Given the set of n points $V(E)$ in the plane, the graph G can be constructed in $O(n \log n)$ time.*

Proof. We first describe how to compute the sequence $\mathcal{L}'$.

The bounded-degree plane spanner G' of E can be constructed in $O(n \log n)$ time [9], and obviously so can G'_U. Since every point in G'_U has bounded degree, and since G'_U is a geometric plane graph, in $O(1)$ time we can compute a rotation system for the points in G'_U (for example, for every point in G'_U, we can list its incident edges in clockwise order). Moreover, since G'_U has $O(n)$ edges, the EMST T contained in G'_U can be computed in $O(n \log n)$ time by a standard MST algorithm. Now using the rotation system of G'_U, we can traverse the edges on the boundary face of G'_U. As we traverse these edges, we remove them from the graph and push the non-tree (with respect to T) edges into a stack; we also remove any isolated points resulting from this process. Note that the non-tree edges on the outer face of G'_U are the maximal edges with respect to the ordering $\preceq$. We repeat this process until G'_U is empty, and at that point, the stack contains the sequence of non-tree edges, sorted according to the partial order $\preceq$; this stack constitutes the list $\mathcal{L}'$. Clearly, this process can be carried out in $O(n)$ time.

After computing $\mathcal{L}'$, we initialize G to the EMST T. As we consider the edges in $\mathcal{L}'$, when we add an edge e in $\mathcal{L}'$ to form a fundamental face F_e in $G + e$, we need to check whether the fundamental face F_e satisfies the condition $wt(F_e) > \lambda \cdot wt(e)$. To do so, we need to traverse the edges on F_e. If e is not subsequently added to G, we might need to traverse some edges on F_e multiple times when we later consider edges that are larger than e in the ordering $\preceq$. To avoid this problem, we can do the following. If we decide to add an edge to G, we add this edge and mark it as a "real" edge of G. On the other hand, if e is not to be added to G, we still add e to G but we mark it as a "virtual" edge of G, and assign it a weight equal to the weight of its fundamental face. The graph G will consist of the tree T plus the set of edges that were marked as real edges. This way each edge in G is traversed at most twice (as every edge appears in at most two faces), and the running time is kept $O(n)$.

It follows that G can be constructed in $O(n \log n)$ time, and the proof is complete. □

Theorem 3.1. *For any integer parameter $\Delta \geq 14$ and any constant $\lambda > 2$, the subgraph G of the unit disk graph U constructed above is a plane spanner of U containing a EMST of $V(U)$, whose degree is at most Δ, whose stretch factor is $(\lambda - 1) \cdot \rho$, where $\rho = (1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$, and whose weight is at most $(1 + \frac{2}{\lambda-2}) \cdot wt(EMST)$. Moreover, G can be constructed in $O(n \log n)$ time.*

Proof. The planarity and degree bound of G follow from the fact that G is a subgraph of G'_U. By construction, G contains a EMST of $V(U)$, and every fundamental face F_e of a non-tree edge e in G satisfies $wt(F_e) \geq \lambda \cdot wt(e)$. Therefore, by part (ii) of Theorem 2.1, we have $wt(G) \leq (1 + \frac{2}{\lambda-2}) \cdot wt(\text{EMST})$. Since by Lemma 3.2 G can be constructed in $O(n \log n)$ time, it suffices to show that the stretch factor of G with respect to U is $(\lambda - 1) \cdot \rho$.

Note that G'_U has stretch factor ρ with respect to U. If an edge e_i is in G'_U but not in G, then by the construction of G, when the edge e_i is considered, the fundamental face F_i of e_i in $G+e_i$ satisfies $wt(F_i) \leq \lambda \cdot wt(e_i)$ (otherwise, the edge e_i would have been added). Therefore, when edge e_i was considered, G contained a path between the endpoints of e_i whose weight is at most $(\lambda - 1) \cdot wt(e_i)$. This path will remain in G after all edges in $\mathcal{L}'$ have been considered. Therefore, every edge in $E(G'_U) - E(G)$ is stretched by a factor at most $\lambda - 1$. Since G'_U has stretch factor ρ with respect to U, it follows that the stretch factor of G with respect to U is $(\lambda - 1) \cdot \rho$. This completes the proof. □

Note that since a Euclidean graph is a unit disk graph with radius equal to ∞, the above theorem holds for Euclidean graphs as well.

4 The Local Distributed Algorithm

In this section we present a local distributed algorithm that constructs a bounded-degree plane lightweight spanner of U.

The same paper by Kanj and Perlović [9], described above, presents a 3-local distributed algorithm that, given a unit disk graph U and an integer parameter $\Delta \geq 14$, constructs a plane spanner G' of U containing a EMST of $V(U)$, of degree at most Δ and stretch factor $\rho = (1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$. Again, G' might not be of light weight, and we need to discard edges from G' so that the obtained subgraph is of light weight. Ultimately, we would like to be able to apply Theorem 2.1. However, a serious problem, which was not present previously in the centralized model, poses itself here in the local model: the removal of the edges from the spanner by different points in the graph needs to be coordinated. This problem was overcome in the centralized model by using a global ordering among the edges of the spanner. Clearly, no local distributed algorithm is capable of computing the global partial order described in Definition 2.1. To coordinate the removal of edges, we use an idea that at its core sits a clustering technique.

Fix an infinite rectilinear tiling $\mathcal{T}$ of the plane whose tiles are $\ell \times \ell$ squares, for some positive constant ℓ to be determined later. Assume, without loss of generality, that one of the tiles in $\mathcal{T}$ has its bottom-left corner coinciding with the origin $(0, 0)$, and that this fact is known to the points in U. Note that this assumption is justifiable in practice because an absolute reference system usually exists (a coordinates system, for example). Therefore, any point in U can determine (using simple arithmetic operations) which tile of $\mathcal{T}$ it resides in. We start with the following simple fact whose proof is easy to verify.

Fact 4.1. *Let C be a cycle of weight at most ℓ. The orthogonal projection* [1] *of C on any straight line has weight at most $\ell/2$.*

Let T_I be the translation with vector $(0, 0)$ (the identity translation), T_H the translation of vector $(\ell/2, 0)$ (horizontal translation), T_V the translation of vector $(0, \ell/2)$ (vertical translation), and T_D the translation of vector $(\ell/2, \ell/2)$ (diagonal translation). We have the following simple lemma.

Lemma 4.1. *Let C be any cycle of weight at most ℓ. There exists a translation T in $\{T_I, T_H, T_V, T_D\}$ such that the translate of C, $T(C)$, resides in a single tile of $\mathcal{T}$.*

Proof. (Sketch) If C resides within a single tile of $\mathcal{T}$ then clearly translation T_I serves the purpose. If C resides within exactly two horizontal (resp. vertical) tiles of $\mathcal{T}$, then these two tiles must be adjacent, and it is easy to verify using Fact 4.1 that translation T_H (resp. T_V) serves the purpose. Finally, if C resides within more than two tiles of $\mathcal{T}$, then again, using Fact 4.1, it can be easily verified that translation T_D serves the purpose. □

Even though a cycle of weight ℓ may not reside within a single tile of $\mathcal{T}$, Lemma 4.1 shows that by affecting some translation T in $\{T_I, T_H, T_V, T_D\}$, the translate of C under T will reside in a single tile. For each translation T in $\{T_I, T_H, T_V, T_D\}$, the points in G whose translates under T reside in a single tile will form a separate cluster. Then, these points will coordinate the detection and removal of the low-weight cycles residing in the cluster by applying a centralized algorithm to the cluster. Since the clusters do not overlap, and since each cluster works as a centralized unit, this maintains the stretch factor under control, while ensuring the removal of every low weight cycle. The centralized algorithm that we apply to each cluster is the standard greedy algorithm that has been extensively used (see for example [1]) to compute lightweight spanners. Given a graph H and a parameter $\alpha > 1$, this greedy algorithm sorts the edges in H in a non-decreasing order of their weight, and starts adding these edges to an empty graph in the sorted order. The algorithm adds an edge AB to the growing graph if and only if no path between A and B whose weight is at most $\alpha \cdot wt(AB)$ exists in the growing graph. We will call this algorithm **Centralized Greedy**. The following properties about this greedy algorithm are known:

Fact 4.2. *Let H be a subgraph of the Euclidean graph E, and let $\alpha > 1$ be a constant. Let H' be the subgraph of H constructed by the algorithm* **Centralized Greedy** *when applied to H with parameter α. Then:*

(i) H' is a spanner of H with stretch factor α.
(ii) H' contains a MST of H.
(iii) For any cycle C in H' and any edge e on C, $wt(C) > (1 + \alpha) \cdot wt(e)$.

[1] By the orthogonal projection of C on a given line we mean the set of points that are the orthogonal projections of the points in C on the given line. Note that, by the continuity of the curve C, this set of points is a straight line segment.

Lemma 4.2. *Let t_0 be a tile in $\mathcal{T}$, and let U_{t_0} be the subgraph of U induced by all the points of U residing in tile t_0. If A and B are two points in the same connected component of U_{t_0}, then A and B are $(\lfloor(8/\pi)\cdot(\ell+1)^2\rfloor)$-hop neighbors in U (i.e., A and B are at most $\lfloor(8/\pi)\cdot(\ell+1)^2\rfloor$ hops away from one another in U).*

Proof. Let $P_{min} = (A = p_0, p_1, \ldots, p_x = B)$ be a path between A and B in t_0 of minimum length. Let D_i, for $i = 0, \ldots, x$, be the disk centered at p_i and of radius $1/2$, and observe that all the disks D_i are contained within a bounding square-box B of dimensions $(\ell+1)\times(\ell+1)$, whose center coincides with the center of t_0. Observe also that the disks D_i, for even i, are mutually disjoint; that is, the points p_i, for even i, form an independent set in U (otherwise, P_{min} would not be a minimal-length path between A and B). Therefore, the area of the region R, denoted a, determined by the union of the disks D_i, for even i, is the sum of the areas determined by these individual disks. The value of a is precisely $(\pi/4)\cdot\lceil x/2\rceil$. Since the region R is contained in the bounding box B of area $(\ell+1)\times(\ell+1)$, we have $a \le (\ell+1)^2$. Consequently, $(\pi/4)\cdot\lceil x/2\rceil \le (\ell+1)^2$. Solving for the integer x in the previous equation we obtain $x \le \lfloor(8/\pi)\cdot(\ell+1)^2\rfloor$. This shows that the length of the path P_{min}, which is x, is bounded by $\lfloor(8/\pi)\cdot(\ell+1)^2\rfloor$, and the proof is complete. □

We now present the local distributed algorithm formally and prove that it constructs the desired lightweight spanner. The input to the algorithm is the spanner G' of U constructed in [9], and a constant $\lambda > 2$. We set $\ell = \lambda$ in the above tiling $\mathcal{T}$. We assume that each point in U has computed its $(\lfloor(8/\pi)\cdot(\lambda+1)^2\rfloor)$-hop neighbors in U by applying the k-local k-neighborhood algorithm described in Section 2, where $k = \lfloor(8/\pi)\cdot(\lambda+1)^2\rfloor$. By Lemma 4.2, this ensures that every point knows all the points in its connected component residing with it in the same tile under any translation.[2] After that, for every round $j \in \{I, H, V, D\}$, each point $p \in U$ executes the following algorithm **Local-LightSpanner**:

(i) p applies translation T_j to compute its virtual coordinates under T_j; Suppose that the translate of p under T_j, $T_j(p)$, resides in tile $t_0 \in \mathcal{T}$;
(ii) p determines the set $S_j(p)$ of all the points in the resulting subgraph of G' (prior to round j) whose translates under T_j reside in the same connected component as $T_j(p)$ in tile t_0;
(iii) p applies the algorithm **Centralized Greedy** to the subgraph $H_j(p)$ of the resulting graph of G' induced by $S_j(p)$ with parameter $\alpha = \lambda - 1$; **if** p decides to remove an edge (p, q) from $H_j(p)$ **then** p removes (p, q) from its adjacency list in G';

Note that since all the points whose translate reside in a single tile apply the same algorithm to the same subgraph during any round j, if a point p decides to remove an edge (p, q), then point q must reach the same decision of removing edge (p, q).

[2] Note that the subgraph of G' induced by the set of points in a single tile may not be connected.

Let G be the subgraph of G' consisting of the set of remaining edges in G' after each point $p \in G'$ applies the algorithm **Local-LightSpanner**.

Theorem 4.1. *The subgraph G of G' is a spanner of U containing a EMST of $V(U)$, with stretch factor $\rho \cdot (\lambda - 1)^4$, and satisfying $wt(G') \leq (1 + \frac{2}{\lambda-2}) \cdot wt(EMST)$, where ρ is the stretch factor of G'.*

Proof. We first show that G is of light weight. To do so, we need to show that G satisfies the conditions of part (i) in Theorem 2.1. We show first that G contains a EMST of $V(U)$.

Since G' contains a EMST of $V(U)$, it suffices to show that after each round of the algorithm **Local-LightSpanner**, the resulting graph still contains a EMST of $V(U)$. Fix a round $j \in \{I, H, V, D\}$, and let G'^{+} be the graph resulting from G' just before the execution of round j, and G'^{-} that resulting from G' after the execution of round j. Assume inductively that G'^{+} contains a EMST of $V(U)$. Note that any edge removed from G'^{+} in round j must have its translate contained within a single tile in $\mathcal{T}$. Let t_0 be a tile in $\mathcal{T}$. In round j, each point p whose translate $T_j(p)$ is in t_0, applies the algorithm **Centralized Greedy** to the subgraph of G'^{+}, $H_j(p)$, induced by the set of vertices $S_j(p)$ defined in the algorithm **Local-LightSpanner**. By part (ii) of Fact 4.2, this algorithm computes a spanner for $H_j(p)$ containing a"local" EMST τ_0 of $H_j(p)$. It is easy to see that an edge e in a EMST of G'^{+} whose translate $T_j(e)$ is in $H_j(p)$, its translate $T_j(e)$ is either an edge of τ_0, or is contained in a cycle whose edges other than e have the same weight as e and are in τ_0. Otherwise, by adding $T_j(e)$ to τ_0, we create a cycle on which $T_j(e)$ is the edge of maximum weight (if not, $T_j(e)$ could replace an edge of τ_0 of larger weight than e, contradicting the minimality of τ_0), and this means that $T_j(e)$ would be the edge of maximum weight on some cycle of G'; since a translation is an isometric transformation—and hence preserves length, this contradicts the fact that e is an edge in a EMST of G'^{+}. Therefore, if an edge in a EMST of G'^{+} is removed during round j, then G'^{-} will still contain a path between the endpoints of e all of whose edges have the same weight as e. Consequently, G'^{-} will still contain a EMST of $V(U)$. It follows that G contains a EMST of $V(U)$.

Now we show that for every cycle C in G, and for every edge e on C, we have $wt(C) \geq \lambda \cdot wt(C)$. Suppose not, and let cycle C and edge $e \in C$ be a counter example. Since every edge in U has weight at most 1, and $wt(C) < \lambda \cdot wt(e)$, it follows that $wt(C) < \lambda$, and by Lemma 4.1, there exists a round j in which the translate of C resides in a single tile t_0 of $\mathcal{T}$. By part (iii) of Fact 4.2, after the application of the algorithm **Centralized Greedy** to the connected component κ containing the translate of C in tile t_0 in round j, no cycle of weight smaller or equals to $(1+\alpha) \cdot wt(e) = (1+\lambda-1) \cdot wt(e) = \lambda \cdot wt(e)$ in the inverse translation of κ remains; in particular, the cycle C will no longer be present in the resulting graph. This is a contradiction. It follows that G satisfies the conditions of part (i) in Theorem 2.1, and $wt(G) \leq (1 + \frac{2}{\lambda-2}) \cdot wt(\text{EMST})$.

Finally, it remains to show that the stretch factor of G, with respect to U, is at most $\rho \cdot (\lambda - 1)^4$. Since G' has stretch factor ρ, it suffices to show that after each round of the algorithm **Local-LightSpanner**, the stretch factor of the resulting graph increases from the previous round by a multiplicative factor of at most $(\lambda - 1)$. Fix a round $j \in \{I, H, V, D\}$, and let G'^{+} and G'^{-} be as above. Suppose that an edge e is removed by the algorithm in round j. Then the translate of e in round j must reside in a single tile t_0 of $\mathcal{T}$. Since by part (i) of Fact 4.2 the algorithm **Centralized Greedy** has stretch factor $\alpha = \lambda - 1$, and since a translation is an isometric transformation, a path of weight at most $(\lambda - 1) \cdot wt(e)$ remains between the endpoints of e in G'^{-}. Therefore, the stretch factor of G'^{-} with respect to G'^{+} increases by a multiplicative factor of at most $(\lambda - 1)$ during round j. This completes the proof. □

We conclude with the following theorem:

Theorem 4.2. *Let U be a connected unit disk graph, $\Delta \geq 14$ be an integer constant, and $\lambda > 2$ be a constant. Then there exists a k-local distributed algorithm with $k = \lfloor (8/\pi) \cdot (\lambda + 1)^2 \rfloor$, that computes a plane spanner of U containing a EMST of $V(U)$, of degree at most Δ, weight at most $(1 + \frac{2}{\lambda - 2}) \cdot wt(EMST)$, and stretch factor $(\lambda - 1)^4 \cdot (1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}) \cdot C_{del}$, where $C_{del} \approx 2.42$.*

5 Conclusion

We have developed in this paper a robust, scalable, and efficient algorithm for a fundamental communication problem—constructing efficient topologies for broadcasting *and* unicasting—in systems modeled as unit disk graphs. The bounds on the parameters of the algorithm and the constructed topology are small, and suggest that the algorithm and the topology are practical, as the following discussion shows.

In table 1, we compare the centralized Euclidean graph lightweight spanner algorithms LL92 by Levcopoulos and Lingas [13], ADDJS93 by Althöfer et al. [1], and BGS05 by Bose, Gudmundsson, and Smid [2] with our centralized algorithm KPX08 and our local distributed algorithm KPXLoc08, both developed to compute lightweight spanners of the more general unit disk graphs. The table gives the bounds on the stretch factor, the weight factor (the constant c^* such that the weight of the spanner is at most $c^* \cdot wt(\text{EMST})$), the maximum degree and the running time. Note that the first two algorithms (LL92 and ADDJS93) do not guarantee an upper bound on the degree of the spanner. Our algorithms

Table 1. A comparison of lightweight spanner algorithms given the constant $\lambda > 2$ and the maximum degree bound Δ; the following notations are used: $\rho^* = (\lambda - 1) \cdot C_{del}$, $c^* = (1 + \frac{2}{\lambda - 2})$, and $a^* = 1 + 2\pi(\Delta \cos \frac{\pi}{\Delta})^{-1}$

Algorithm	LL92 [13]	ADDJS93 [1]	BGS05 [2]	KPX08	KPXLoc08
Stretch factor	ρ^*	ρ^*	10.02	$a^* \cdot \rho^*$	$a^* \cdot (\lambda - 1)^3 \cdot \rho^*$
Weight factor	c^*	c^*	$O(1)$	c^*	c^*
Max. degree	∞	∞	27	Δ	Δ
Running time	$O(n \log n)$	$O(n^2 \log n)$	$O(n \log n)$	$O(n \log n)$	N/A

Table 2. Comparison between algorithm BGS05 [2] and our algorithms KPX08 and KPXLoc08 for different values of Δ

$\Delta =$	14	27
BGS05	N/A	$\rho^* = 10.02, c^* = O(1)$
KPX08	$\rho^*, c^* = 5.22$	$\rho^*, c^* = 4.63$
KPXLoc08	$\rho^*, c^* = 8.81$	$\rho^*, c^* = 8.08$

match their bounds on the weight factor to provide a maximum degree bound at a small multiplicative cost in the stretch factor (a^* for our centralized algorithm and $(\lambda - 1)^3 \cdot a^*$ for our local distributed algorithm). For example, for a degree bound of 14, our upper bound on the stretch factor increases (with respect to [13] and [1]) by a multiplicative constant of 1.47 for the centralized algorithm, and of 2.92 (corresponding to $\lambda = 2.256$) for the local distributed algorithm. For larger values of Δ, the multiplicative factors are even smaller.

In table 2 we use some concrete values for Δ and λ in order to compare our algorithms with the algorithm BGS05 by Bose, Gudmundsson, and Smid [2]. Their algorithm only guarantees a maximum degree bound of 27. The listed bounds for stretch factor ρ^* and weight factor c^* for $\Delta = 27$ are obtained by setting $\lambda = 2.551$ in KPX08 and $\lambda = 2.282$ in KPXLoc08. The bounds for stretch factor ρ^* and weight factor c^* when $\Delta = 14$ are obtained by setting $\lambda = 2.475$ in KPX08 and $\lambda = 2.256$ in KPXLoc08.

References

1. Althöfer, I., Das, G., Dobkin, D., Joseph, D., Soares, J.: On sparse spanners of weighted graphs. Discrete & Computational Geometry 9, 81–100 (1993)
2. Bose, P., Gudmundsson, J., Smid, M.: Constructing plane spanners of bounded degree and low weight. Algorithmica 42(3-4), 249–264 (2005)
3. Bose, P., Morin, P., Stojmenovic, I., Urrutia, J.: Routing with guaranteed delivery in ad hoc wireless networks. wireless networks 7(6), 609–616 (2001)
4. Damian, M., Pandit, S., Pemmaraju, S.: Local approximation schemes for topology control. In: Proceedings of PODC, pp. 208–217 (2006)
5. Das, G., Heffernan, P., Narasimhan, G.: Optimally sparse spanners in 3-dimensional euclidean space. In: Proceedings of SoCG, pp. 53–62 (1993)
6. Das, G., Narasimhan, G.: A fast algorithm for constructing sparse euclidean spanners. In: Proceedings of SoCG, pp. 132–139 (1994)
7. Das, G., Narasimhan, G., Salowe, J.: A new way to weigh malnourished euclidean graphs. In: Proceedings of SODA, pp. 215–222 (1995)
8. Gudmundsson, J., Levcopoulos, C., Narasimhan, G.: Fast greedy algorithms for constructing sparse geometric spanners. SIAM J. Comput. 31(5), 1479–1500 (2002)
9. Kanj, I., Perković, L.: On geometric spanners of euclidean and unit disk graphs. In: Proceedings of STACS (2008)
10. Kanj, I., Perkovic, L., Xia, G.: Computing lightweight spanning subgraphs locally. Technical report # 08-002, `http://www.cdm.depaul.edu/research/Pages/TechnicalReports.aspx`

11. Kranakis, E., Singh, H., Urrutia, J.: Compass routing on geometric networks. In: Proceeding of CCCG, vol. 11, pp. 51–54 (2005)
12. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally! In: Proceedings of PODC, pp. 300–309 (2004)
13. Levcopoulos, C., Lingas, A.: There are planar graphs almost as good as the complete graphs and almost as cheap as minimum spanning trees. Algorithmica 8(3), 251–256 (1992)
14. Li, X.-Y., Calinescu, G., Wan, P.-J., Wang, Y.: Localized delaunay triangulation with application in ad hoc wireless networks. IEEE Trans. on Parallel and Distributed Systems. 14(10), 1035–1047 (2003)
15. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comput. 21(1), 193–201 (1992)
16. Peleg, D.: Distributed computing: A Locality-Sensitive Approach. SIAM Monographs on Discrete Mathematis and Applications (2000)
17. Wang, Y., Li, X.-Y.: Localized construction of bounded degree and planar spanner for wireless ad hoc networks. MONET 11(2), 161–175 (2006)
18. Wattenhofer, R.: Sensor networks: distributed algorithms reloaded - or revolutions? In: Proceedings of SIROCCO, pp. 24–28 (2006)

Dynamic Routing and Location Services in Metrics of Low Doubling Dimension
(Extended Abstract)[*]

Goran Konjevod, Andréa W. Richa, and Donglin Xia

Arizona State University, Tempe AZ 85287, USA
{goran,aricha,dxia}@asu.edu

Abstract. We consider dynamic compact routing in metrics of low doubling dimension. Given a set of nodes V in a metric space with nodes joining, leaving and moving, we show how to maintain a set of links E that allows compact routing on the graph $G(V, E)$. Given a constant $\epsilon \in (0, 1)$ and a dynamic node set V with normalized diameter Δ in a metric of doubling dimension $\alpha \in O(\text{loglog}\,\Delta)$, we achieve a dynamic graph $G(V, E)$ with maximum degree $2^{O(\alpha)} \log^2 \Delta$, and an optimal $(9 + \epsilon)$-stretch compact name-independent routing scheme on G with $(1/\epsilon)^{O(\alpha)} \log^4 \Delta$-bit storage at each node. Moreover, the amortized number of messages for a node joining, leaving and moving is polylogarithmic in the normalized diameter Δ; and the cost (total distance traversed by all messages generated) of a node move operation is proportional to the distance the node has traveled times a polylog factor. (We can also show similar bounds for a $(1 + \epsilon)$-stretch compact dynamic labeled routing scheme.)

One important application of our scheme is that it also provides a node location scheme for mobile ad-hoc networks with the same characteristics as our name-independent scheme above, namely optimal $(9 + \epsilon)$ stretch for lookup, polylogarithmic storage overhead (and degree) at the nodes, and locality-sensitive node move/join/leave operations. We also show how to extend our dynamic compact routing scheme to address the more general problem of devising locality-sensitive Distributed Hash Tables (DHTs) in dynamic networks of low doubling dimension. Our proposed DHT scheme also has optimal $(9 + \epsilon)$ stretch, polylogarithmic storage overhead (and degree) at the nodes, locality-sensitive publish/unpublish and node move/join/leave operations.

1 Introduction

A routing scheme on the graph $G = (V, E)$ is a distributed algorithm running at each node that allows any source node to send packets to any destination node along the links in E. A routing scheme on a metric space (M, d) builds a graph $G = (V, E)$ whose vertices correspond to points of M by distributedly selecting the edges (u, v) to be in E (the length of an edge (u, v) is given by $d(u, v)$), and routes packets along the selected edges in E. The *stretch* of a routing path is

[*] Work supported in part by NSF grant 0830791.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 379–393, 2008.

its length divided by the metric distance of its two endpoints. The *stretch* of a routing scheme is the maximum stretch of a routing path. The *space* requirement of a scheme is the maximum size of a routing table at any node. We call a routing scheme *compact* if the routing table and packet header size are both polylog($|V|$).

We differentiate between *labeled and name-independent routing. Labeled routing* allows the scheme designer to label the nodes with additional routing information. In *name-independent routing*, the scheme must use solely the (arbitrary) original naming.

Compact routing research has recently focused on graphs of low doubling dimension [2, 7, 11, 12, 14, 19, 20, 22] (the *doubling dimension* of a metric space is the minimum α such that any ball of radius r can be covered by at most 2^{α} balls of radius $r/2$). That is both due to lower bounds known for general graphs [3], and the fact that low doubling dimension seems to be a characteristic shared by many networks of interest, such as Internet-like networks. All of the existing schemes are static, i.e., they assume that the network is fixed. Moreover, with the exception of [20], they all assume a centralized pre-configuration procedure for building the routing tables. None of the previous work generalizes in a straightforward fashion to a dynamic metric scenario.

In this paper we finally cross the bridge from static to dynamic (optimal stretch) compact routing schemes, thus widening the applicability of such schemes to more realistic dynamic scenarios. We describe the *first fully dynamic* name-independent compact routing scheme with optimal-stretch in metrics of low doubling dimension. More precisely, our distributed scheme works on a dynamic set of nodes V in a metric of low doubling dimension $\alpha = O(\log\log \Delta)$, and uses routing tables, label and packet headers of size polylogarithmic in the network size and the normalized diameter. The number of messages, amortized per operation, grows as a polylogarithmic function of the network size and the normalized diameter. The supported operations are node-join, leave and move. Finally, the move operation is locality-sensitive, in that the cost of the movement of a node is proportional to the distance the node last moved. In the full version of this paper [13], we also describe an optimal $(1+\epsilon)$-stretch dynamic labeled compact routing scheme with similar bounds as our name-independent scheme.

1.1 Distributed Hash Tables and Mobile Node Location

One important application of dynamic compact routing in metrics is the design of dynamic Distributed Hash Tables (DHTs) [4, 10, 16, 17, 18, 21] in highly-scalable peer-to-peer systems. A DHT (also known as object or service location scheme) is a dictionary data structure implemented in a distributed way, thus allowing efficient object location (lookup) in the network, where an object may be some data item (e.g. file), node, or service. Object location is a major challenge for fully decentralized dynamic peer-to-peer systems, with applications that range from file sharing, to database query and indexing, to node location in mobile ad-hoc networks. A typical DHT builds a low-degree overlay network on the set of participant peers (nodes), and implements the required operations by routing messages through the resulting overlay topology. Although each peer maintains only a few physical links, the goal is to form an efficient and versatile overlay network which can be used for object location.

A full-fledged DHT should support insertion and deletion of objects (by means of *publish* and *unpublish* operations), allow nodes to join and leave the overlay network, balance the load across participants, and be resilient to failures. A DHT is said to be *locality-aware* or *locality-sensitive* if the cost of its lookup operation is proportional to the distance between the node initiating the operation and the closest copy of the object in the network (multiple copies of the same object may exist and be located at different nodes). Moreover, we say that a DHT also has *locality-sensitive (un)publish* if the cost of this operation is proportional to distance between initiating node and closest copy of the object residing at a different node, times a polylogarithmic factor. We call the maximum ratio of the lookup cost to the shortest path distance between the requesting node and the closest object copy the *stretch* of the DHT scheme. For scalability and fairness, it is important to keep the node degree and the storage overhead in a DHT low, in general polylogarithmic in the network parameters.

Designing DHTs on a network where each node publishes its own name as an object reduces to compact name-independent routing on the shortest-path metric induced by the network: the dynamic graph G maintained by the routing scheme will correspond to the DHT overlay network.

Hence, an immediate application of our dynamic compact routing scheme is that of *locating nodes in mobile ad-hoc networks*, given our locality-sensitive protocols for adapting to node movement in the network, as well as node join and leave operations. From a rigorous theoretical point-of-view, this problem had only been studied for limited classes of close-to-uniform node distributions [1, 8]. Thus, our results imply the existence of *constant stretch*, *polylogarithmic degree and storage space* node location schemes in mobile ad-hoc networks which adapt *near-optimally to node move/join/leave* operations in all networks of low doubling dimension.

Our compact routing scheme also generalizes to DHTs where nodes may hold multiple (copies of) objects and where the network may contain duplicate copies of any object. We also achieve *constant stretch*, *polylogarithmic degree and storage space*, and *locality-sensitive* node move/join/leave and publish/unpublish operations. No DHT with constant stretch has been known before for networks of low doubling dimension.

2 Our Contributions

In the following statements, $\epsilon \in (0, 1)$ is an arbitrary constant and (M, d) is a metric of doubling dimension α. We use V_t to denote the node set at time t (if clear from context, we omit the parameter t). Let Δ_t be the *normalized diameter* of V_t at time t, i.e. $\Delta_t = \max_{u,v \in V_t} d(u, v)/ \min_{u,v \in V_t} d(u, v)$. Let Δ be the *maximum normalized diameter* over all time, i.e., $\Delta = \max_t \Delta_t$. We assume $\alpha = O(\log \log \Delta)$ although our protocols still work correctly for arbitrary α (if $\alpha = \omega(\log \log \Delta)$, we no longer meet the polylogarithmic requirements for storage and packet header size). We assume that each node knows its point in the metric and the distance function $d(\cdot,\cdot)$, and that a point in a metric can be described by $O(\alpha \log \Delta)$ bits (Note that a Δ diameter area contais at most $\Delta^{O(\alpha)}$ points).

In all of our schemes, we maintain a hierarchical data structure with $O(\log \Delta)$ levels. In our analysis, we assume that we do not have concurrent updates, i.e., that at any point in time at most one update operation, due to a node joining/leaving or moving within the network, is being performed. We guarantee that (i) for each movement of node u, the move protocol at u is executed at a single level; and (ii) a level k move protocol will only be executed at u if u has moved by a distance at least 2^k from the point where the last move protocol at level no less than k was executed at u.

Theorem 1. *We maintain a graph $G = (V, E)$ over the metric (M, d) with degree $2^{O(\alpha)} \log^2 \Delta$ and achieve a $(9+\epsilon)$-stretch* name-independent *routing scheme on G with $(1/\epsilon)^{O(\alpha)} \log^4 \Delta$-bit storage per node. Each node move protocol at level k uses $(1/\epsilon)^{O(\alpha)} \log^4 \Delta \cdot k^2$ amortized number messages, each traversing a distance of $O(2^k/\epsilon)$ in G. In particular, it uses $(1/\epsilon)^{O(\alpha)} \log^6 \Delta$ amortized number of messages for a node to join, or to leave.*

Note that stretch 9 is asymptotically optimal for name-independent routing schemes in low doubling dimension (the lower bound in [11] also extends to metric routing). A unique feature of our name-independent routing scheme is that it does *not* rely on an underlying labeled scheme—as did basically all of the previous name-independent routing schemes [2, 11, 12, 14].

Another new ingredient in this scheme is an explicit scale-control procedure which adapts to the dynamic variations in network size and diameter. We perform a scale-control procedure every time the number of nodes in the network is squared or square-rooted or the normalized diameter of the network changes by at least a constant factor. The amortized number of messages exchanged at each scale-control procedure is at most a constant with the exception of the case when the normalized diameter grows by more than a constant factor, in which case the amortized number of messages is polylogarithmic on the current normalized diameter. Thus, it follows that the results in Theorems 1 and 2 still hold if we take Δ to be the *current normalized diameter* of the network, rather than the maximum diameter over time.

Location services. As discussed in the introduction, our dynamic name-independent compact routing scheme provides an efficient node location service for mobile ad-hoc networks: this location service has stretch $(9 + \epsilon)$, polylogarithmic degree and storage, and uses locality-sensitive node move/join/leave operations. It works in mobile ad-hoc networks whose shortest-path metric is of low doubling dimension. The performance guarantees match those in Theorem 1.

DHT. We use our dynamic name-independent routing scheme to provide efficient dynamic DHTs, as per the theorem below. The same notation is assumed as before, and additionally, q denotes the maximum number of objects at any node.

Theorem 2. *We maintain a graph $G = (V, E)$ with degree $2^{O(\alpha)} \log^2 \Delta$ and achieve a $(9 + \epsilon)$-stretch DHT on G with $(1/\epsilon)^{O(\alpha)} \log^2 \Delta((1/\epsilon)^{O(\alpha)} \log^2 \Delta + q)$-bit storage per node, $(1/\epsilon)^{O(\alpha)} \log^3 \Delta \cdot (\log^2 \Delta + q)$ amortized number of messages* *to publish an object, and* $(1/\epsilon)^{O(\alpha)} \log^2 \Delta$ *number of messages to unpublish an object. Each node-move protocol at level* k *uses* $((1/\epsilon)^{O(\alpha)} \log^2 \Delta(\log^2 \Delta + q) \cdot k^2$ *amortized number of messages, each message crossing distance* $O(2^k/\epsilon)$*. In particular, it takes* $((1/\epsilon)^{O(\alpha)} \log^4 \Delta(\log^2 \Delta + q)$ *amortized number of messages for a node to join, or to leave. In addition, when multiple copies of an object may exist, a node publishes an object only up to a level* k *such that another copy of the object already exists in the network within distance* $(1/\epsilon)^{O(\alpha)} 2^k$ *of the node.*

Our results rely on a data structure, which we call *skeletal trees*, that appears more suitable for the storage of dynamic information (such as routing location information) than prior search trees which resulted from standard network decompositions (such as search trees that directly mimic a hierarchical r-net decomposition of the network [2, 11, 12, 14, 19]).

2.1 Related Work

More general than our problem is the problem of designing compact routing schemes on a graph $G(V, E)$. Hence all compact routing schemes for graphs whose induced shortest-path metric is of low doubling dimension also apply to our problem. The basic idea of maintaining a hierarchy of r-nets and searching for routing information hierarchically is similar to static routing schemes [2, 7, 11, 12, 14, 19, 22]. However, our scheme not only efficiently adapts to dynamic changes in the node set, but does so while still achieving optimal stretch factors and polylogarithmic storage, label and packet header size (note however that the best-known static schemes are scale-free, i.e. independent of the normalized diameter, while ours is not). Note that our lower bound result in [11] also extends to metric routing and so excludes name-independent compact routing with stretch better than 9 in low doubling metrics.

Korman and Peleg [15] considers dynamic routing schemes in general graphs, where the topology of the underlying graph is fixed and the weights on edges may change. Their result indicates that the amortized message complexity of these dynamic updates depends on the local density of the graph. Note routing schemes in general graphs are not compact [3].

As discuseed in the introduction, the work on DHTs [4, 10, 16] is in fact closely related to name-independent routing in metrics. There is no known DHT for networks of low doubling dimension which also achieves constant stretch. PRR [16] and LAND [4], at the forefront of DHT schemes for growth-bounded metrics (a subset of low doubling dimension metrics), fail either in maintaining constant stretch (even if just in expectation), or in maintaining polylogarithmic memory overhead when applied to networks of low doubling dimension. In fact, the expected stretch of PRR can be $\Omega(n^{\log 1.5})$ and LAND may require linear space at some nodes (See the full paper [13] for counter-examples). Moreover no straightforward extension of these schemes would yield a constant-stretch DHT with the desired properties in low doubling dimension. Hildrum, Krauthgamer, and Kubiatowicz [10] provide a $(1 + \epsilon)$-stretch DHT scheme with storage depending on local network-growth rate rather than a global bounded growth rate. However, there exist doubling metrics with unbounded growth rates, such as a clique.

Awerbuch and Peleg [5], in one of the first papers on the subject, show how to keep track of mobile users in a distributed network. These results are related to the later DHT work, but differ in several important aspects. They work with general graphs, and thus cannot bound stretch below $O(\log^2 n)$. They treat mobile users in a static network. They only limit total memory usage, and do not offer per-node bounds. Finally, their results, as ours, are scale-dependent. The mobile node location schemes in [1, 8] address a more current scenario for mobile node location: They consider the problem of locating mobile nodes in a mobile ad-hoc wireless network scenario, albeit for much more restrictive classes of node distributions. The work in [8] also addresses concurrent updates.

3 Preliminaries

Given a dynamic set of nodes V in a metric space (M, d) of doubling dimension α, we define a virtual graph $G' = (V', E')$, and a host mapping $\phi : V' \to V$ that associates each virtual vertex in V' to a host node in V. This virtual graph will give us the necessary data structures to achieve the results outlined in Theorems 1 and 2. Thus it is natural to define the link set $E = \{(\phi(x), \phi(y)) \mid \forall (x, y) \in E'\}$, and therefore the dynamic graph $G = (V, E)$.

The elements of the metric space M are called points. To avoid confusion between the dynamic graph $G = (V, E)$ and the virtual graph $G' = (V', E')$, the elements of V and E are called *nodes* and *links* respectively, while the elements of V' and E' are called *vertices* and *edges* respectively. Each vertex $x \in V'$ (resp., each node $u \in V$) corresponds to a point $pnt(x)$ (resp., $pnt(u)$) in M.

For any point $x \in M$ and $r > 0$, $B_x(r)$ denotes the ball of radius r around x, i.e. $B_x(r) = \{y \in M \mid d(x, y) \leq r\}$. In the following, we give the definition of an (X, r)-net, on which our hierarchical data structures are based. Intuitively, an (X, r)-net is an r-net drawn from points in M but which is only required to cover points in a subset $X \subseteq M$. Hence an r-net is an (M, r)-net.

Definition 3 ((X, r)-net). *For any $r > 1$ and $X \subseteq M$, a set $Y \subseteq M$ is an* (X, r)-net *if (i) the distance of any pair of points $y, y' \in Y$ is at least r, i.e. $d(y, y') \geq r$; and (ii) for any point $x \in X$ there exists $y \in Y$ within distance of r, i.e. $|Y \cap B_x(r)| \geq 1$.*

We also refer to an (X, r)-net as an r-net covering X. The following is a well-known property of r-nets, which also extends to (X, r)-nets.

Lemma 1 ([9]). *Let Y be an* (X, r)-net. *For any point x in the metric space M and $r' \geq r$, we have $|Y \cap B_x(r')| \leq \left(\frac{4r'}{r}\right)^{\alpha}$.*

We use a distributed hash function to represent each node name which uses $O(\alpha \log \Delta_t)$ bits at any time t—the amortized cost of updating this hash function when necessary is considered in Section 8. (Note that at each time, we have

$\log|V_t| = O(\alpha \log \varDelta_t)$ because of the α doubling dimension.) We provide node join/leave/move protocols for nodes joining/leaving/moving within the metric. For a node to join the network, it must have some (arbitrary) bootstrap node it can connect to in the network. We assume that a node leaves the network gracefully, that is, it always performs the node-leave protocol before leaving. If sudden node departures are common (e.g., systems with high node failures), we can still achieve the same performance bounds with an extra polylogarithmic factor in the degree and storage space at each node, if each node also stores a copy of its routing table at each of its neighbors.

3.1 Virtual Graph

The virtual graph G' consists of two hierarchies of 2^i-nets: the *parent* hierarchy $X = \cup_{i=0}^{h} X_i$ and the *cluster header* hierarchy $Y = \cup_{i=0}^{h} Y_i$, where $\ell = \lceil \log \varDelta \rceil$. Let the *parent set* X_i be a 2^i-net, for each $i \in [\ell]^2$, and $X_0 = V$. Let the *cluster header set* Y_i be a 2^i-net covering X_i, for each $i \in [\ell]$. Actually we abuse the notation slightly; when we say that, for example, X_i is a 2^i-net, we mean that its point set $pnt(X_i) \subseteq M$ is a 2^i-net. Note that we treat two vertices from different levels of X_i or Y_i, or from X and Y respectively, as different vertices, even if their corresponding points in M are identical.

We define three kinds of edge relationships as follows. For each node $u \in V$ and each $i \in [\ell]$, let $p_i(u) \in X_i$ denote the *parent* of u at level i, which is selected by our dynamic protocols and initially has $d(p_i(u), u) \leq 2^i$. For each cluster header $y \in Y_i$ and $i \in [\ell]$, let $N(y) = X_i \cap B_y(2^i/\epsilon)$ be the *neighborhood set* of y. Since Y_i is a 2^i-net covering X_i, for $i \in [\ell]$, let $h_i : X_i \rightarrow Y_i$ be the *header mapping* that maps each $x \in X_i$ to a cluster header $y = h_i(x) \in Y_i$ that covers x, i.e. $d(y, x) \leq 2^i$. Thus we define edge sets: the *parent* edges $E'_p = \{(p_i(u), p_{i-1}(u)) \,|\, \forall u \in V, \forall i \in [\ell]\}$, and the *cluster* edges $E'_c = \{(y, x) \,|\, \forall y \in Y, \forall x \in N(y)\}$. Note that edges $(h_i(x), x) \in E'_c$, for all $x \in X_i$ and $i \in [\ell]$.

For each cluster header $y \in Y_i$ and $i \in [\ell]$, we organize the node set $\{u \in V \,|\, p_i(u) \in N(y)\}$ to store routing information using a virtual tree $CT(y)$, called *cluster tree*. The *cluster tree* rooted at y consists of root y, edges (y, x) and subtrees $T(x)$, called *descendant trees*, for all $x \in N(y)$. A *descendant tree* $T(x)$, for each $x \in X_i$ and $i \in [\ell]$, consists of a copy of path $\langle p_i(u) = x, p_{i-1}(u), \cdots, p_0(u) \rangle$ for each node $u \in V$ s.t. $x = p_i(u)$; for any nodes u and v with $p_i(u) = p_i(v) = x$, and for each $j \leq i$, the vertices $p_j(u)$ and $p_j(v)$ are merged into one vertex in $T(x)$ iff $p_k(u) = p_k(v)$ for all $j \leq k \leq i$. We treat any two distinct vertices u and v in the descendant tree $T(x)$ as *distinct vertices* of $T(x)$ even if they correspond to the same node in X — note that the virtual subgraph (X, E'_p), where $E'_p = \{(p_i(u), p_{i-1}(u)) \,|\, \forall u \in V, \forall i \in [\ell]\}$, may not even be a tree. The only node we directly associate with the original corresponding node in X is the root node x of $T(x)$.

Thus we have $V' {=} (X \cup Y) \bigcup \cup_{x \in X} V(T(x))$, and $E' {=} (E'_p \cup E'_c) \bigcup \cup_{x \in X} E(T(x))$.

[1] For any integer $k \geq 0$, let $[k]$ denote the set $\{0, 1, \cdots, k\}$.

3.2 General Idea

The $(9+\epsilon)$-stretch routing algorihtm works on the virtual graph G' which maintains a parent hierarchy of X_i (a 2^i-net) as similar as in [11] and a cluster header hierarchy of Y_i covering X_i. For each cluster header $y \in Y_i$ and $i \in [\ell]$, a search tree $ST(y)$ rooted at y is maintained to provide a query on the routing information of nodes in $B_y(2^i/\epsilon)$ with delay cost of $2 \cdot 2^i/\epsilon$. Given the name of the destination node v, the routing algorithm searches the routing information of the destination in search trees $ST(y)$ along the parents $p_i(u)$ of the source node u, for $i = 0, 1, \cdots$, where $y = h(p_i(u))$ is the cluster header of $p_i(u)$. Whenever it finds the routing information of the destination, the algorithm delivers the message down to the destination with the information on the search trees that contain the destination. The key idea of the dynamic protocols in Section 4, which maintain the virtual graph G' and the hosting mapping ϕ, is that we maintain a net structure for the subgraph (X, E'_p), rather than a tree structure as in [11]. In Section 5, we introduce a novel dynamic tree structure, called *skeletal tree*, as our search tree. The main idea behind the skeletal tree is that it recursively keeps *large* (in terms of number of nodes) branches of a standard search tree and omits all the small branches. This makes a skeletal tree less sensitive to changes in the network topology. In addition, we also show how a simple locality-sensitive load-balancing procedure can be used to efficiently and distributedly (re-)balance our dynamic storage structure.

4 Dynamic Protocols

In this section, we describe the protocols that dynamically maintain the virtual graph (V', E') and the host mapping ϕ when a node joins, leaves, or moves . We focus on the node join/leave/move protocols for the subgraph $(X \cup Y, E'_p \cup E'_c)$ and its host mapping, while briefly describing how to also dynamically maintain each descendant tree. All of the algorithms are described in more detail (wihout the corresponding descendant trees updates) in the boxes below.

Node Join Protocol (Algorithm 1). When a new node u joins V, for each $k \in [\ell]$, we either assign an existing vertex $x \in X_k$ as $p_k(u)$ if $\exists x \in X_k \cap B_u(2^k)$, or add a new vertex as $p_k(u)$ using the subprocedure `AddNewParent(`u, k`)`, which also associates or adds a cluster header y for $p_k(u)$.

Node Leave Protocol (Algorithm 2). When an existing node u leaves V, for each $k \in [\ell]$ with u as the host of $p_k(u)$, we either update the host of $p_k(u)$ to be v using subprocedure `UpdateHost(`u, k, v`)` if $\exists v \neq u \in V$ s.t. $p_k(v) = p_k(u)$, or delete $p_k(u)$ using the subprocedure `DeleteParent(`u, k`)`.

Node Move Protocol (Algorithm 3). When a node moves, let i be the maximal index s.t. $d(p_j(u), u) > 2^{j+1}$, $\forall j \in [i]$. Algorithm 3 presents a move operation of u at level i. For level k from i down to 0, the algorithm decouples the old parent vertex $p_k(u)$, and assigns a new parent vertex $p_k(u)$ within $B_u(2^k)$. Note that the *for* loop of Algorithm 3 is the exact combination of the *for* loops of Algorithms 1 and 2.

Thus we have the following Invariant (See the full paper [13] for the proof):

Invariant 4. *1. For any $x \in X \cup Y$, $pnt(x)$ will never change during the course of the protocol, although its host $\phi(x)$ may be updated over time.*

2. The host $\phi(x)$, $\forall x \in X_i$ and $\forall i \in [\ell]$, is a node u s.t. $p_i(u) = x$. The host $\phi(y)$, $\forall y \in Y_i$ and $\forall i \in [\ell]$, is $\phi(x)$, where $x \in X_i$ is a node that minimizes $d(x, y)$.

3. Whenever a node u updates $p_i(u)$ due to the movement of u, we have $d(p_i(u), u) > 2^{i+1}$ before the update and $d(p_i(u), u) \leq 2^i$ after the update.

4. $d(u, p_i(u)) \leq 5 \cdot 2^i$, for all $u \in V$ and $i \in [\ell]$.

We can extend Algorithms 1, 2 and 3 in order to also maintain the descendant tree $T(x)$, $\forall x \in X_i$ and $\forall i \in [\ell]$, which simply consists of a copy of path $\langle p_i(u) = x, p_{i-1}(u), \cdots, p_0(u) \rangle$ for each node $u \in V$ s.t. $x = p_i(u)$, with some of the subpaths possibly merged as explained above. In addition, we guarantee that for each vertex $z \in T(x)$ at level j, the host $\phi(z)$ is one of the nodes u such that z is a copy of $p_j(u)$.

Algorithm 1. A new node u joins V

```
for $k = \ell$ downto 0 do
    if $\exists x \in X_k \cap B_u(2^k)$ then
        Set $p_k(u) = x$
    else AddNewParent($u, k$)

Procedure AddNewParent($u, k$)
begin
    Add a new vertex $x$ with $pnt(x) = pnt(u)$ into $X_k$, and set $\phi(x) = u$
    Set $p_k(u) = x$ and add $x$ to $N(y)$ for all $y \in Y_k \cap B_x(2^k/\epsilon)$
    if $\exists y \in Y_k \cap B_x(2^k)$ then
        Set $h_k(x) = y$
        if $d(x, y) \leq d(p_k(v), y)$, where $v = \phi(y)$ then
            Update $\phi(y) = \phi(x)$
    else
        Add a new vertex $y$ with $pnt(y) = pnt(u)$ into $Y_k$, and set $\phi(y) = u$
        Set $N(y) = X_k \cap B_y(2^k/\epsilon)$, and $h_k(x) = y$
end
```

Algorithm 2. A node u leaves V

```
for $k = \ell$ downto 0 do
    if $\phi(p_k(u)) = u$ then
        if $\exists v \neq u \in V$ s.t. $p_k(v) = p_k(u)$ then
            UpdateHost($u, k, v$)
        else DeleteParent($u, k$)

Procedure UpdateHost($u, k, v$)
begin
    Set $\phi(p_k(u)) = v$
    Set $\phi(y) = v$, $\forall y \in Y_k$ with $\phi(y) = u$
end

Procedure DeleteParent($u, k$)
begin
    Remove $x = p_k(u)$ from $X_k$
    for all $y \in Y_k \cap B_x(2^k/\epsilon)$ do
        Remove $x$ from $N(y)$
        if $N(y) = \emptyset$ then
        Remove $y$ from $Y_k$
        else if $\phi(y) = u$ then
            Set $\phi(y) = \phi(z)$, where $z \in N(y)$ that minimizes $d(z, y)$
end
```

Algorithm 3. A node u moves at level i

```
Let $i$ be the maximal index s.t. $d(p_j(u), u) > 2^{j+1}$, $\forall j \in [i]$
for $k = i$ downto $0$ do
    if $\phi(p_k(u)) = u$ then
        if $\exists v \neq u \in V$ s.t. $p_k(v) = p_k(u)$ then
            UpdateHost($u, k, v$)
        else DeleteParent($u, k$)
    if $\exists x \in X_k \cap B_u(2^k)$ then Set $p_k(u) = x$
    else AddNewParent($u, k$)
```

5 Search Trees

In order to efficiently search for nodes within $CT(y)$, we maintain a *search tree* for each cluster tree $CT(y)$, $\forall y \in Y$. It might be natural to use the cluster trees themselves as search trees in a static network. However, in order to make the search relatively insensitive to frequent changes in the network, we maintain a search tree on a *subgraph* of its corresponding cluster tree. Intuitively, the subgraph, called *skeletal tree*, contains cluster tree branches of large cardinality but omits small branches.

Definition 5 (Skeletal Tree). *Given a cluster tree $CT(y)$, for $y \in Y_i$ and $i \in [\ell]$, let $CT_y(x)$ denote the subtree rooted at x of $CT(y)$, and let $s_y(x)$ denote the number of leaves in the subtree $CT_y(x)$, for each $x \in CT(y)$. The* skeletal tree *of $CT(y)$, denoted $ST(y)$, is the subgraph of $CT(y)$ including (i) the root y; and (ii) any edge (x, z) of $CT(y)$, where $x \in ST(y)$ and z is a child of x in $CT(y)$, and $s_y(z)/s_y(x) \geq \frac{1}{b\cdot(i+1)}$, for $b = \left(\frac{4}{\epsilon}\right)^{\alpha}$. We denote the subtree of $ST(y)$ rooted at x by $ST_y(x)$, for any vertex $x \in ST(y)$.*

Note that the degree of $CT(y)$ is at most b, and its height is i. Thus we have:

Lemma 2. *Given any $y \in Y_i$ and $i \in [\ell]$, the ratio of the number of leaves in the skeletal tree $ST(y)$ to the number of leaves in the cluster tree $CT(y)$ is at least $(1 - \frac{1}{i+1})^i > e^{-1} \approx 1/2.7$.*

For each cluster tree $CT(y)$, $\forall y \in Y$, we maintain a search tree on its skeletal tree $ST(y)$. For each vertex $x \in ST(y)$, let $Range_y(x)$ be the minimal interval that contains all keys stored in $ST_y(x)$. Then the interval of the root vertex, $Range_y(y)$, is the whole range of the key space, while for any vertex $x \in ST(y)$, the family $\{Range_y(z) \,|\, z \text{ is a child of } x\}$ is a partition of $Range_y(x)$. A key k together with its associated data is inserted into the leaf z of $ST(y)$ such that $k \in Range_y(z)$ along the path from the root y to the leaf z. Let each leaf z of $ST(y)$ keep its stored keys in a sorted list, denoted $List_y(z)$. Fix any $x \in ST(y)$. We can enumerate keys in $List_y(x)$ by enumerating keys in each list $List_y(z)$, for all leaves z in $ST_y(x)$ in a preorder traversal of the subtree.

Given a key k, the search procedure on a search tree $ST(y)$, $\forall y \in Y_i$ and $\forall i \in [\ell]$, searches along the path from the root to a leaf such that any vertex x on the path has $k \in Range_y(x)$, and returns to the root with the data for k or with the "not-found" message. It takes $2i$ messages and $2^{i+1}(1/\epsilon + O(1))$ delay.

As the structure of a search tree changes due to the network changes, we want to ensure that the load on nodes stays balanced. In addition, if some node's load becomes too heavy due to newly inserted keys, we also want to trigger load balancing across regions with heavy load. The following lemma gives a load balancing procedure and its performance (the proof occurs in the full paper [13]):

Lemma 3 (Load Balancing). *Given a subtree $ST_y(x)$ with height k of any search tree $ST(y)$, for $y \in Y$ and $x \in ST(y)$, there is a load balancing procedure on $ST_y(x)$ for keys in $List_y(x)$ that rearranges them in $ST_y(x)$ so that every leaf stores an equal number of keys. This takes $O(k|List_y(x)|)$ messages.*

Now we consider when to trigger the load-balancing procedure. For each cluster tree $CT(y)$, $\forall y \in Y$, we maintain a counter $t_y(x)$ for each vertex x in the search tree $ST(y)$: (i) initially $t_y(x)$ is set to zero; (ii) whenever a key inserted in the search tree $ST(y)$ is stored in the subtree $ST_y(x)$, $t_y(x)$ is increased by one; (iii) when $t_y(x)$ reaches $s(x)$, i.e. the number of leaves of $T(x)$, load-balancing is executed on the subtree $ST_y(x)$ for the keys in $List_y(x)$, and $t_y(z)$ is reset to zero for all vertices $z \in ST_y(x)$.

5.1 Dynamic Maintenance of Search Trees

Consider the dynamic maintenance of a cluster tree $CT(y)$ for a cluster header $y \in Y_i$ and $i \in [\ell]$. When a node u joins, or moves, the number of leaves of each subtree $CT_y(z)$, $\forall z \in CT(y)$, that now contains a copy of the new $p_0(u)$ is increased by 1. This might result in the addition of the branch $ST_y(z)$ to the skeletal tree $ST(y)$ at z's parent x if $s_y(z)/s_y(x) \geq \frac{1}{b\cdot(i+1)}$ and $x \in ST(y)$. In that case, we perform load balancing on the updated subtree $ST_y(x)$ for the list $List_y(x)$, which takes $k \cdot |List_y(x)|$ messages, where k is the height of $ST_y(x)$, i.e. $x \in X_k$.

When a node u leaves or moves, the number of leaves of each subtree $CT_y(z)$, $\forall z \in CT(y)$, that loses the copy of $p_0(u)$ is decreased by 1, which might result in the removal of the branch $ST_y(z)$ from the skeletal tree $ST(y)$ at z's parent x. However, we remove the branch $ST_y(z)$, only if $s_y(z)/s_y(x) < \frac{1}{2b\cdot(i+1)}$, instead of $s_y(z)/s_y(x) < \frac{1}{b\cdot(i+1)}$. This helps avoid repeated addition and removal of the same branch with only several nodes joining and leaving. We perform a load-balancing procedure on the updated subtree $ST_y(x)$ for the old list $List_y(x)$, which takes $k \cdot |List_y(x)|$ messages, where k is the height of $ST_y(x)$, i.e. $x \in X_k$.

The cost of dynamic maintenance is discribed in the following lemma, whose proof occurs in the full paper [13].

Lemma 4. *Given that each node publishes at most q pairs of $(key, data)$, for each search tree $ST(y)$, $\forall y \in Y_i$ and $i \in [\ell]$, each leaf stores at most $(1/\epsilon)^{O(\alpha)} \cdot i^2 + O(q)$ pairs. It takes $(1/\epsilon)^{O(\alpha)} i^3 \cdot (i^2 + q)$ amortized messages for a node* *joining, or leaving, and* $(1/\epsilon)^{O(\alpha)} i(i^2+q) \cdot k^2$ *amortized message for a move operation at level* k*.*

6 Name-Independent Routing

We color each vertex x in X, using a color function $c : X \rightarrow [20^{\alpha}]$ such that no two siblings in X share a color, where we say two vertices $x, x' \in X_i$, $\forall i \in [\ell - 1]$, are sibling if $\exists z \in X_{i+1}$ s.t. edges (z, x) and (z, x') are in E'_p. Note that the number of siblings of any vertex in X is at most 20^{α} by Lemma 1 and Invariant 4. Thus we can always find a valid color for a newly added vertex.

We maintain a search tree $ST(y)$ for each cluster tree $CT(y)$, $\forall y \in Y$. Let each node publishes its own name as the key, and store the ID of $p_i(u)$ and colors $\langle c(p_i(u)), c(p_{i-1}(u)), \cdots, c(p_{i-\log(1/\epsilon)}(u))\rangle$ as the data for the search tree $ST(y)$, for all $y \in Y_i$ such that $p_i(u) \in N(y)$ and for all $i \in [\ell]$. The ID of $p_i(u)$ takes $O(\alpha \log(1/\epsilon))$ bits, since $p_i(u) \in N(y)$ and $|N(y)| = (1/\epsilon)^{O(\alpha)}$.

The improved name-independent routing scheme is presented in Algorithm 4. While it shares the high-level structure of the name-independent routing algorithm in [11], it differs considerably from that algorithm since it does *not* rely on an underlying labeled routing scheme. Instead, once we get the color data of v at level i, we are only able to go down to $p_{i-\log(1/\epsilon)}(v)$. Then we recursively get the color data of v at level $i - \log(1/\epsilon)$ by querying the search tree $ST(h_{i-\log(1/\epsilon)}(p_{i-\log(1/\epsilon)}(v)))$ and go down $\log(1/\epsilon)$ levels further.

We can use a similar flavor argument as in [11] to prove the $(9+\epsilon)$ stretch of our scheme; Lemma 4 gives the storage and message complexity. Hence we have the result in Theorem 1.

Algorithm 4. A name-independent routing from u to v, given the key, i.e. v's name

```
for $i = 0$ to $\ell$ do
    $y \leftarrow h_i(p_i(u))$
    Go to $y$, and search on the search tree $ST(y)$ for the key
    if the data $\langle c(p_i(v)), c(p_{i-1}(v)), \cdots, c(p_{i-\log(1/\epsilon)}(v))\rangle$ for the key is found then
        Go to $p_i(v)$; and break
while true do
    Go down to $p_{i-\log(1/\epsilon)}(v)$ from $p_i(v)$ using the color data
    if we reach $p_0(v)$ then terminate the algorithm
    $i \leftarrow i - \log(1/\epsilon)$
    Search on the search tree $ST(h_i(p_i(v)))$ for the key
```

7 DHT

Our DHT scheme uses the same underlying virtual graph and host mapping as our name-independent routing scheme, and each node publishes an object with the object ID as the key and the same routing data as for the name-indepedent routing scheme. The lookup algorithm is also the same as the routing algorithm in Algorithm 4. In addition, when multiple copies of an object may exist, we provide the locality-sensitive publish in the following subsection. Thus we achieve the results in Theorem 2.

7.1 A Locality Sensitive Pulbish for DHTs

We consider a scenario where multiple copies of an object may exist along the network. We want to reduce the duplicated information published for each object while still guaranteeing $(9+\epsilon)$-stretch lookup.

The publish algorithm is given in Algrithm 5. For an object at a node u, we stop publishing at a level j such that there is a search tree $ST(y)$, for some $y \in Y_j$ with $p_j(u) \in N(y)$, that stores an existing entry for the object. For level $i = 0$ up to $j-1$, we still publish it at all search trees $ST(y)$, $\forall y \in Y_i$ s.t. $p_i(u) \in N(y)$.

Algorithm 5. Publish an object at node u

```
1: for $i = 0$ to $\ell$ do
2:     for each $y \in Y_i$ s.t. $p_i(u) \in N(y)$ do
3:         if there is an existing entry for the object in the search tree $ST(y)$ then
4:             Mark that the object at $u$ is duplicated; and terminate the algorithm
5:         Publish the object at the search tree $ST(y)$ with the object ID as the key,
           and the routing data to $u$ as the data
```

Note that by the above Algorithm, a search tree contains at most one entry for each object, no matter how many nodes within the search tree publish the object. When a node u unpublishes the object at a search tree, it checks whether there is another node v that stopped publishing the same object at the same search tree. If it is the case, we notice node v to resume the publish as given in Algorithm 5. Now we improve the publish algorithm to guarantee $(9+\epsilon)$-stretch lookup. If the *if* condition at Line 3 of Algorithm 5 is satisfied, we continue publishing additional $\log(c/\epsilon)$ levels as in Line 5, and making these entries as secondary, where $c > 16$ is a constant. Thus we have the following lemma (See the full paper [13] for the proof).

Lemma 5. *For any node $u \in V$, any object held at u, and any level $j \in [\ell+1]$, there exists a node $v \in B_u(2^j)$ that publishes the object at level j.*

By a similar argument of our name-independent routing scheme, we guarantee that the lookup stretch is at most $9 + O(\epsilon)$. The additional secondary entries result in at most a constant factor more on the storage of each node.

8 Scale-Control Procedure

In this section, we provide a scale-control procedure to adapt our schemes to dynamic network size and diameter. First we discuss how to adapt our schemes to dynamic changes in the number of nodes in the network. Note that the original node name might be expressed in $\log n$ bits, where n is the total number of distinct nodes over all time. However the number n_t of nodes at the current time t may be much less than n. We use a universal hash function to hash each original node name into a value represented by $c \log n_t$ bits, where $c > 2$ is a constant. Carter and Wegman [6] provide such a universal hash function, which is represented by $O(\log n_t)$ bits. Whenever the number of nodes in the network is *squared* or *square-rooted*, we update the hash function so that the number of bits for each hash value increases or decreases by c bits.

Second, we discuss how to adapt our schemes to dynamic changes in the network diameter. We update the hierarchical level ℓ according to the changes in the network diameter.

Invariant 6. *We preserve two invariants: (i)* $N(h_\ell(x)) = X_\ell, \forall x \in X_\ell$*; and (ii)* $|X_{\ell - \log \frac{1}{\epsilon}}| > 1$.

Whenever we insert a new vertex into X_ℓ, we check whether Invariant 6 (i) is preserved. The invariant is not preserved iff the diameter increases to $2^\ell/\epsilon$. We recursively define the parent set $X_{\ell+i} \subseteq X_{\ell+i-1}$ to be a $2^{\ell+i}$-net covering $X_{\ell+i-1}$, for $i = 1$ up to a value j s.t. $|X_{\ell+j}| = 1$ (Note that $j = \log \frac{1}{\epsilon} + O(1)$). Meanwhile, we add cluster header sets $Y_{\ell+i} = X_{\ell+i}$ for $i = 1, \cdots, j$, the cluster tree and the search tree for each newly added cluster header. Then we update $\ell = \ell + j$. Note that by Lemma 4, $(1/\epsilon)^{O(\alpha)}\ell^6$ amortized messages per node in the current network suffice for the operation.

Whenever we delete a vertex in $X_{\ell - \log \frac{1}{\epsilon}}$, we check whether Invariant 6 (ii) is preserved. If not, we drop all $X_{\ell-k}$ and $Y_{\ell-k}$ for $k = 0, \cdots, \log \frac{1}{\epsilon} - 1$, and update $\ell = \ell - \log \frac{1}{\epsilon}$. Note that a constant amortized number of messages per node in the current network suffice for the operation.

9 Future Work

There are a number of directions that should be investigated further. Here we just mention a few. One direction is the control of hot-spot nodes in the case where the bandwidth of each link is limited. For example, roots of search trees may suffer from congestion. Second, in this paper, we only consider one topology change at a time. Can our routing schemes be improved to deal with multiple concurrent changes of the topology? Finally, it would be interesting to generalize the compact routing problems to scenarios where not only the node set but the metric itself may be dynamic (in our scenario we assume that the metric is static, as for example if we were using an Euclidean metric), e.g. for shortest-path metrics in dynamic graphs. This seems to be a very challenging task and might open the way for finding dynamic compact routing schemes in graphs.

References

1. Abraham, I., Dolev, D., Malkhi, D.: LLS: a locality aware location service for mobile ad hoc networks. In: Proc. 2004 DIALM-POMC (2004)
2. Abraham, I., Gavoille, C., Goldberg, A.V., Malkhi, D.: Routing in networks with low doubling dimension. In: Proc. 26th ICDCS, p. 75 (2006)
3. Abraham, I., Gavoille, C., Malkhi, D.: On space-stretch trade-offs: Lower bounds. In: Proc. 18th SPAA, pp. 207–216 (2006)
4. Abraham, I., Malkhi, D., Dobzinski, O.: Land: stretch $(1 + \epsilon)$ locality-aware networks for DHTs. In: Proc. 15th SODA, pp. 550–559 (2004)
5. Awerbuch, B., Peleg, D.: Online tracking of mobile users. J. ACM 42(5), 1021–1058 (1995)
6. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. J. Comp. Sys. Sci. 18(2), 143–154 (1979)
7. Chan, H.T.-H., Gupta, A., Maggs, B.M., Zhou, S.: On hierarchical routing in doubling metrics. In: Proc. 16th SODA, pp. 762–771 (2005)
8. Flury, R., Wattenhofer, R.: MLS: an efficient location service for mobile ad hoc networks. In: Proc. 7th MobiHoc, pp. 226–237 (2006)
9. Gupta, A., Krauthgamer, R., Lee, J.R.: Bounded geometries, fractals and low-distortion embeddings. In: Proc. 44th FOCS, pp. 534–543 (2003)
10. Hildrum, K., Krauthgamer, R., Kubiatowicz, J.: Object location in realistic networks. In: Proc. 16th SPAA, pp. 25–35 (2004)
11. Konjevod, G., Richa, A.W., Xia, D.: Optimal-stretch name-independent compact routing in doubling metrics. In: Proc. 25th PODC, pp. 198–207 (2006)
12. Konjevod, G., Richa, A.W., Xia, D.: Optimal scale-free compact routing schemes in networks of low doubling dimension. In: Proc. 18th SODA, pp. 939–948 (2007)
13. Konjevod, G., Richa, A.W., Xia, D.: Dynamic routing and location services in metrics of low doubling dimension. Technical report, ASU (2008), http://thrackle.eas.asu.edu/users/goran/papers/dynamic-routing.pdf
14. Konjevod, G., Richa, A.W., Xia, D., Yu, H.: Compact routing with slack in low doubling dimension. In: Proc. 26th PODC, pp. 71–80 (2007)
15. Korman, A., Peleg, D.: Dynamic routing schemes for general graphs. In: Proc. 33rd ICALP, pp. 619–630 (2006)
16. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distributed environment. In: Proc. 9th SPAA, pp. 311–320 (1997)
17. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable Content-Addressable network. In: Proc. 2001 SIGCOMM, pp. 161–172 (2001)
18. Rowstron, A., Druschel, P.: Pastry: scalable, decentraized object location and routing for large-scale peer-to-peer systems. In: Proc. 18th Middleware (2001)
19. Slivkins, A.: Distance estimation and object location via rings of neighbors. In: Proc. 24th PODC, pp. 41–50 (2005)
20. Slivkins, A.: Towards fast decentralized construction of locality-aware overlay networks. In: Proc. 26th PODC, pp. 89–98 (2007)
21. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proc. 2001 SIGCOMM, pp. 149–160 (2001)
22. Talwar, K.: Bypassing the embedding: algorithms for low dimensional metrics. In: Proc. 36th STOC, pp. 281–290 (2004)

Leveraging Linial's Locality Limit

Christoph Lenzen and Roger Wattenhofer

Computer Engineering and Networks Laboratory (TIK)
ETH Zurich, 8092 Zurich, Switzerland
{lenzen,wattenhofer}@tik.ee.ethz.ch
www.dcg.ethz.ch

Abstract. In this paper we extend the lower bound technique by Linial for local coloring and maximal independent sets. We show that constant approximations to maximum independent sets on a ring require at least log-star time. More generally, the product of approximation quality and running time cannot be less than log-star. Using a generalized ring topology, we gain identical lower bounds for approximations to minimum dominating sets. Since our generalized ring topology is contained in a number of geometric graphs such as the unit disk graph, our bounds directly apply as lower bounds for quite a few algorithmic problems in wireless networking.

Having in mind these and other results about local approximations of maximum independent sets and minimum dominating sets, one might think that the former are always at least as difficult to obtain as the latter. Conversely, we show that graphs exist, where a maximum independent set can be determined without any communication, while finding even an approximation to a minimum dominating set is as hard as in general graphs.

1 Introduction

The recent hype about multi-hop wireless networks such as ad hoc, mesh, or sensor networks has sparked an unprecedented interest in distributed network algorithms, from inside the distributed computing community, and probably even more from outside. In the last decade reams of new distributed network algorithms have been proposed. One common theme of these algorithms is *locality*: As large networks demand fast and failure resistant algorithms, nodes should be able to make decisions solely by communicating to neighboring nodes a bounded number of times. The main challenge is to design local algorithms which can provide global guarantees. In the center of attention are classic graph optimization problems such as minimum dominating sets (MDS) and connected dominating sets, as they provide, e.g., energy-efficient backbone solutions for a variety of applications.

This abundance of distributed network algorithms is not matched by an equally rich knowledge about lower bounds and impossibility results. Indeed, on the lower bound side of locality research there are to the best of our knowledge only two results. One is a technique by Kuhn et al. [13] which proved that

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 394–407, 2008.

many classic graph optimization problems including vertex cover, matching, or MDS cannot be polylogarithmically approximated in less than $\sqrt{\log n/\log\log n}$ time. However, the proof requires quite a peculiar "fractal" graph family barely occuring in real world problems, and certainly not in the world of wireless networks. The other is a nifty lower bound by Linial [12] who proved that computing a 3-coloring or a maximal independent set (MIS) even on a ring topology cannot be done in constant time. Indeed, using an indistinguishibility argument, Linial shows that at least $\Omega(\log^* n)$ time is necessary on a ring with n nodes. Linial's lower bound is extremely weak compared to the one due to Kuhn et al., conversely the ring is a topology that may appear in virtually any network. In fact Linial's limit also holds on a simple list, hence prohibiting constant time solutions to the above problems on almost any graph families of practical use.

However, to the theorist's annoyance, Linial's bound only holds for 3-coloring and maximal independent set, leaving important problems as MDS or maximum indepent set (MaxIS) approximations aside. In fact, the MDS approximation problem on Linial's ring topology offers a trivial solution: Simply take every node and you have a 3-approximation!

The first issue we address in this paper is whether one can extend Linial's lower bound towards approximation covering and packing problems such as MDS or MaxIS, such that the lower bound still holds in natural geometric graphs existing in wireless multi-hop networks. The weakest geometric graph model is the unit disk graph (UDG). Our lower bounds hold in UDGs, and henceforth in all generalizations thereof, e.g. quasi unit disk graphs, unit ball graphs, and growth-bounded graphs.

We will prove that constant approximations of the MDS problem on a quite simple family of UDG's cannot be obtained in $o(\log^* n)$ time. To the best of our knowledge, this is the first nontrivial lower bound for this problem holding in UDG's. More generally, if we allow for an $O(f(n))$-approximation of the MDS problem in $O(g(n))$ time, the product $f(n) \cdot g(n)$ cannot be in $o(\log^* n)$. Strengthening Linial's results with respect to MIS, the same bounds are proved for the MaxIS problem on the ring.[1] Like Linial's limit our results apply to a very general computational model, where nodes can gather all information about nodes that are at most k hops away in k rounds, and may perform arbitrary local computations. In addition, all bounds still apply when allowing non-uniform algorithms, i.e., nodes to be aware of the size of the graph.

Finally we answer the question if the local complexities of both problems are in principle related to each other. In growth-bounded graphs the MIS problem is as least as hard as approximating a MDS, as a MIS is always a constant factor MDS approximation. The same order holds for all other known results. However, as is shown in the last part of the paper, this is not true in general. We will utilize a simple construction to generate a graph family for which an exact solution to the MaxIS problem is trivial, but MDS and approximations thereof stay as hard as in general graphs.

[1] Note that—as for 3-coloring—this result trivially generalizes to simple lists, since in this case most of the nodes observe the same topology as on a ring.

2 Related Work

Local algorithms have been a recurring research theme since the 80's [4,9,10,11,12]. Lately many contributions in this area have been motivated by demands of wireless ad hoc and sensor networks. The classical minimum dominating set and maximum independent set type problems are subject to quite a few basic protocols in such distributed systems. Energy consumption and communication capacities are directly affected by the quality of the given solutions. As communication ranges are limited in the systems in consideration, often the family of bounded growth graphs (or subfamilies thereof, e.g. unit disk graphs) are examined. Under this assumption the problems seem to be closely related, as a MIS becomes both a constant MDS and MaxIS approximation.

As a first highlight Luby [10] managed to compute a MIS in $\mathrm{O}(\log n)$ time, where n is the number of nodes. The algorithm works on general graphs, however, in general graphs a MIS is neither a MDS nor a MaxIS approximation. It took until the beginning of the current century for the first distributed MDS algorithm non-trivial in both time and approximation to be published [7]. It yields a $\mathrm{O}(\log \Delta)$ approximation in $\mathrm{O}(\log n \log \Delta)$ time, where n is the number of nodes and Δ is the largest node degree. Kuhn et al. followed with the first constant time algorithm providing a non-trivial approximation ratio [6]. This result has been improved [5] to the currently best result for general graphs: A MDS can be approximated up to a factor of $\mathrm{O}\left(\Delta^{1/\sqrt{k}} \log \Delta\right)$ in $\mathrm{O}(k)$ time.

For a long time the only known lower bound for local algorithms had been Linial's $\Omega(\log^* n)$ bound on 3-coloring and MIS on the ring. Later Kuhn et al. [13] opposed the positive results by showing that in general graphs local algorithms cannot compute a polylogarithmic approximation of several optimization problems, including MDS, in less than $\Omega\left(\sqrt{\log n / \log \log n}\right)$ time. Independent of and concurrent to our own results, Czygrinow et al. [15] proved the same lower bound on MaxIS approximations we show, but using a different argument. In the same work they present a randomized algorithm achieving an $(1-\varepsilon)$-approximation in $O(1/\varepsilon)$ time for any $\varepsilon > 0$, showing that in contrast to Lineal's bound randomization does help.

Since the graphs used in the lower bound proof in [13] are complex and most unlikely to occur in practice, researchers started studying geometric graph classes like unit disk graphs (UDG's) or bounded growth graphs, which are regarded as abstractions of realistic wireless network topologies. Close-to-optimum deterministic respectively randomized MIS/MDS/MaxIS algorithms were presented by [3] and [8]. Recently, Schneider et al. [1] devised an algorithm computing a MIS on bounded growth graphs within $\mathrm{O}(\log^* n)$ time. Our lower bounds show this bound to be tight also with respect to MDS or MaxIS approximations in bounded growth graphs, as a MIS yields constant approximations to both in this graph family. In other words, locally approximating a MDS or MaxIS in bounded growth graphs is not simpler than the special case of ascertaining a MIS.

Restricting the scope further, one can study UDG's with the nodes given global position information. In this setting even Linial's lower bound can be beaten (e.g., in [14] a PTAS for the MDS problem is given), as the positions can be used to partition the problem into efficiently solvable local instances. Thus, the main difficulty when approximating MDS or MaxIS in UDG's is to break the symmetry of the problem, which is reflected in Linial's Limit.

3 Model and Notation

We model a network as a simple undirected graph $G = (V, E)$, where nodes represent processors and edges represent bidirectional communication links. Basically we use Linial's classic synchronous message passing model, where in one communication round each node of the network graph can send a message to each of its direct neighbors. We allow those messages to be of arbitrary size. However, at the beginning a node $v \in V$ is only equipped with information about its communication channels and a unique identifier of $O(\log n)$ size, which for simplicity we will refer to as v as well. Thus, a node can gather knowledge about node identifiers and edges between nodes at most k hops away in k communication rounds. Each processor may perform arbitrary local computations. Thus in this model an algorithm running in at most k rounds can be expressed as a function of the topology and identifiers of the (inclusive) k-neighborhood $\mathcal{N}_k^+(v) := \{w \in V \mid w \text{ is in at most } k \text{ hops distance of } v\}$ of each node v to a result $c(v)$. For an algorithm to be correct, it is required that combining the choices $c(v)$ of all $v \in V$ yields a feasible global solution of the considered problem.

We modify this standard model by dropping the assumption of uniformity, i.e., we allow nodes to know the size $n := |V|$ of the graph. Though we need this in our proofs of the claimed lower bounds, we aquire even stronger results. We will solely consider symmetric graphs, in the sense that an embedding exists where for any to nodes $v, w \in V$ and any $k \in \mathbb{N}$, the k-neighborhood $\mathcal{N}_k^+(v)$ is identical up to translation and rotation to $\mathcal{N}_k^+(w)$. Hence an Algorithm $\mathcal{A}$ running in at most k rounds on a node $v \in V$ will be a function from $\mathcal{N}_k^+(v)$, its topology, and n to the set of possible decisions $c(v)$, independent of wether nodes can gather any *local* geometric information. This implies that our bounds also hold, e.g., when we assume an Euclidean embedding of the graph where nodes can determine the exact distance an edge bridges. As discussed in the related work section, with *global* positition information better solutions become possible.

Definition 1 (Local f-approximations of MaxIS). *Given a graph $G = (V, E)$, an independent set (IS) of G is a set $I \subseteq V$ such that for all $v, w \in I$ we have $\{v, w\} \notin E$. A maximal independent set (MIS) is an independent set M so that no set $S \supset M$ can be an IS. A maximum independent set (MaxIS) is an IS of maximum cardinality. Let f be a function from $\mathbb{N}$ to $[1, \infty) \subset \mathbb{R}$. A local f-approximation algorithm for the MaxIS problem computes for each node $v \in V$ a choice $c(v) \in \{0, 1\}$ such that $I := \{v \in V \mid c(v) = 1\}$ is an IS and for any graph G the inequality $f(n)|I| \geq |M|$ holds, where M is an arbitrary MIS of G.*

Definition 2 (Local f-approximations of MDS). *Given a graph $G = (V, E)$, a dominating set (DS) of G is a set $D \subseteq V$ such that for each $v \in V \setminus D$ a $d \in D$ exists with $\{v, d\} \in E$. A minimum dominating set (MDS) is a DS of minimum cardinality. Let f be a function from $\mathbb{N}$ to $[1, \infty) \subset \mathbb{R}$. A local f-approximation algorithm for the MDS problem computes for each node $v \in V$ a choice $c(v) \in \{0, 1\}$ such that $D := \{v \in V \mid c(v) = 1\}$ is a DS and for any graph G the inequality $|D| \leq f(n)|M|$ holds, where M is an arbitrary MDS of G.*

Definition 3 (Local 3-coloring). *A valid 3-coloring of a graph $G = (V, E)$ is a function $c : V \to \{r, g, b\}$ such that $c(v) \neq c(w)$ for all $\{v, w\} \in E$.*

Definition 4 (Unit Disk Graph (UDG)). *A Unit Disk Graph (UDG) is a graph $\mathrm{UDG}(\iota) = (V, E)$, defined by an injective function $\iota : V \to \mathbb{R}^2$, where $E = \{\{v, w\} \in V \times V \mid 0 < \|\iota(v) - \iota(w)\|_{\mathbb{R}^2} \leq 1\}$.*

Definition 5 (R_n and R_n^k). *Define the ring with n nodes as $R_n := (V_n, E_n)$, where $V_n := \{v_1, \ldots, v_n\}$ and $E_n := \{\{v_1, v_2\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. Thus R_n is simply a circle consisting of n nodes. Denote by $R_n^k := \left(V_n, E_n^k\right)$ the k-ring with n nodes, i.e., R_n extended by all edges $\{v_i, v_j\}$ with $v_j \in \mathcal{N}_k^+(v_i) \setminus \{v_i\}$ with respect to R_n (see Figure 1).*

Proposition 6. *R_n^k can be realized as UDG.*

Proof. Place all nodes equidistantly on a circle of radius $\frac{1}{2}\left(\sin\left(\frac{l\pi}{n}\right)\right)^{-1}$, as illustrated by Figure 1.

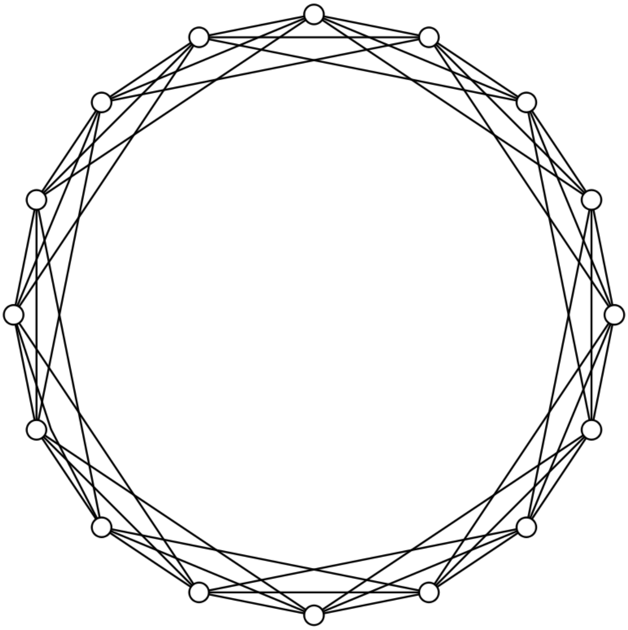


Fig. 1. R_{16}^3. Realized as UDG k is controlled by the scaling.

4 Proofs of the Bounds

For brevity, in the subsequent analysis the term *algorithm* will refer to deterministic, local algorithms in the sense of the model described in the previous section. We show the claimed lower bounds by means of a reduction of local 3-coloring of the ring. For this problem Linial [12] proved the following bound:

Theorem 7 (Lower bound for local 3-coloring of the ring). *There is no deterministic local algorithm 3-coloring the ring R_n requiring less than $\frac{1}{2}(\log^* n - 1)$ communication rounds.*

Proof. The proof in [9] applies, as it also holds when we assume the nodes to know the size of the network n.

Naor proved an analogous result for randomized algorithms [2]. We will need the following notion:

Definition 8 ($\sigma(n)$-alternating algorithm). *Suppose $\mathcal{A}$ is an algorithm operating on R_n which assigns each node $v_i \in V_n$ a value $c(v_i) \in \{0, 1\}$. We call $\mathcal{A}$ $\sigma(n)$-alternating, if the length k of any monochromatic sequence $c(v_i) = c(v_{i+1}) = \ldots = c(v_{i+k})$, indices taken modulo n, is smaller than $\sigma(n)$.*

If a $\sigma(n)$-alternating algorithm is given, one can easily obtain a 3-coloring of the ring R_n in $\mathrm{O}(\sigma(n))$ time:

Lemma 9 (3-coloring the marked ring). *Given a $\sigma(n)$-alternating algorithm $\mathcal{A}$ running in $\mathrm{O}(\sigma(n))$ rounds, a 3-coloring of the ring can be computed in $\mathrm{O}(\sigma(n))$ rounds.*

Proof. Recall that we identify nodes with their identifier, thus we can compare two nodes $v, w \in R_n$. We define the following algorithm for 3-coloring the ring R_n nodewise for each node $v \in V_n$:

1. Run $\mathcal{A}$. Let $d(v) \in \{0, 1\}$ denote the result of this run.
2. Find a pair of neighboring nodes $\{w_1, w_2\}$ with $d(w_1) \neq d(w_2)$ which is closest to v. If $v \in \{w_1, w_2\}$, set $c(v) := b$, if $d(v) = 0$, and $c(v) := r$ otherwise. Else denote by δ the distance to the closer node in $\{w_1, w_2\}$, w.l.o.g. w_1, and set $c(v) := c(w_1)$ if $\delta \in 2\mathbb{N}$ and $c(v) := c(w_2)$ else.
3. If v has a neighbor w with $c(v) = c(w)$ and $v > w$, set $c(v) := g$.
4. If v has a neighbor w with $c(v) = c(w) = g$ and $v > w$, set $c(v)$ to the color none of the neighbors of v has.
5. Return $c(v)$.

Clearly, the running time of this algorithm is in $\mathrm{O}(\sigma(n))$, as by assumption not more than $\sigma(n)$ consecutive nodes take the same decision $d(v)$ when running $\mathcal{A}$.

We now show that it yields a valid 3-coloring of R_n. In step 2 at most one of the neighbors of any node $v \in V_n$ may take the same choice, as each node chooses different from one of its neighbors. In step 3 from each pair of neighbors with the same color one chooses g. Thus only neighbors both colored with g may remain. These will be resolved in step 4, as nodes to the right and left of a pair colored by g both must have a different color than g. □

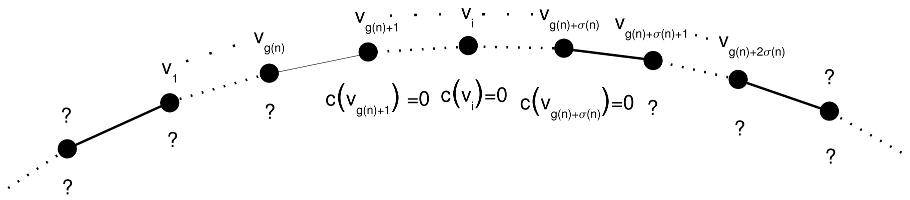


Fig. 2. An element of $\mathcal{S}_n$ displayed as part of a labeling of the ring. A "?" indicates that the identifier or output of the corresponding node is unspecified respectively unknown. Independent of the identifiers left of v_1 and right of $v_{g(n)+2\sigma(n)}$ all nodes v_i from $v_{g(n)+1}$ to $v_{g(n)+\sigma(n)}$ will compute $c(v_i) = 0$.

To establish our lower bounds, we construct $\sigma(n)$-alternating algorithms out of assumed approximation algorithms for MaxIS and MDS, respectively.

Lemma 10 (Modified MaxIS approximation). *Suppose an f-approximation algorithm $\mathcal{A}$ for the MaxIS problem on the ring R_n running in at most $g(n) \geq 1$ rounds is given, where we have $f(n)g(n) \in \mathrm{o}(\log^* n)$. Then an $\mathrm{o}(\log^* n)$-alternating algorithm $\mathcal{A}'$ requiring $\mathrm{o}(\log^* n)$ communication rounds exists.*

Proof. As stated in the last section, we identify nodes with their identifiers. Thus, the input of $\mathcal{A}$ when executed on R_n is a sequence of identifiers $(v_1, \ldots, v_n)$, where no identifier occurs twice. Recall that for a single node $v_i \in V_n$, $i \in \{1, \ldots, n\}$, we can express the output $c(v_i)$ of $\mathcal{A}$ as a function of n and the subsequence of identifiers $\big(v_{i-g(n)}, \ldots, v_i, \ldots, v_{i+g(n)}\big)$, where indices are taken modulo n. Set $\sigma(n) := 10f(n)g(n)$ and define

$$\begin{aligned}\mathcal{S}_n := \big\{&\big(v_1, \ldots, v_{\sigma(n)+2g(n)}\big)\\ &\mid \forall i \in \{g(n)+1, \ldots, \sigma(n)+g(n)\} : c(v_i) = 0\big\}\quad, \end{aligned} \tag{1}$$

i.e., exactly the set of sequences preventing that $\mathcal{A}$ is $\sigma(n)$-alternating (see also Figure 2). Note that due to the preceeding observations $\mathcal{S}_n$ is well defined, although the choices of the leading and trailing $g(n)$ many nodes may depend on further identifiers.

For n fixed we construct a sequence of identifiers for R_n. Initially we choose an arbitrary subsequence $s \in \mathcal{S}_n$ and assign the identifiers of s to $v_1, \ldots, v_{|s|}$. Now suppose we already assigned labels to the nodes $v_1, \ldots, v_j$. If there exists a sequence $s \in \mathcal{S}_n$ that can be appended to $v_1, \ldots, v_j$ without duplicating an identifier, we do so. If no further sequence fits, we add $n - j$ arbitrary identifiers not yet present in $v_1, \ldots, v_j$ to complete the labeling $(v_1, \ldots, v_n)$ of R_n. Observe that each sequence from $\mathcal{S}_n$ added implies that at least $\sigma(n)$ additional nodes will compute $c(v) = 0$ when $\mathcal{A}$ is run on the constructed labeling.

Assume for contradiction, that for arbitrarily large n it is possible to label R_n as described in the preceding paragraph, with at least $n - \frac{n}{5f(n)}$ identifiers stemming from sequences out of $\mathcal{S}_n$. By construction at least

$$\frac{\sigma(n)n}{\sigma(n)+2g(n)} - \frac{n}{5f(n)} \tag{2}$$

many nodes compute $c(v) = 0$ when $\mathcal{A}$ is passed such a labeling as input. Thus we have

$$|I| \leq n - \left(\frac{\sigma(n)n}{\sigma(n) + 2g(n)} - \frac{n}{5f(n)} \right) \leq \frac{2n}{5f(n)} \ , \tag{3}$$

where $I = \{v \in V_n \,|\, c(v) = 1\}$ denotes the resulting independent set. Since the size of a MaxIS of R_n is $\left\lfloor \frac{n}{2} \right\rfloor$, this contradicts the assumption that $\mathcal{A}$ is an f-approximation algorithm to the MaxIS problem on R_n.

Thus, an $n_0 \in \mathbb{N}$ must exist, such that for all $n \geq n_0$ we may choose a maximal set of disjoint sequences $\{s_1, \ldots, s_{j_n}\} \subset S_n$ such that

$$\left| Id(n) \setminus \left(\bigcup_{i=1}^{j_n} s_i \right) \right| \geq \frac{n}{5f(n)} \ , \tag{4}$$

where $Id(n)$ is the set of admissible identifiers for nodes on R_n. In other words, at least $\frac{n}{5f(n)}$ identifiers remain which cannot form a further sequence from $\mathcal{S}_n$. W.l.o.g. we may restrict $Id(n)$ such that $|Id(n)| = n$, as $\mathcal{A}$ must yield correct results for any admissible set of identifiers. Hence, by setting $n' := \max\{n_0, 5f(n)n\}$, we can define an injective relabeling function $r_n : Id(n) \to Id(n')$ such that no sequence $s \in \mathcal{S}_{n'}$ is completely contained in the image of r_n.

The algorithm $\mathcal{A}'$ claimed to exist now consists of redefining all identifiers by r_n and simulating a run of $\mathcal{A}$ on the modified instance, where instead of n the algorithm is given n' as the number of nodes. As $g(n) \leq g(n)f(n) \in \mathrm{o}(\log^* n)$, the running time $g(n')$ of $\mathcal{A}'$ is certainly in $\mathrm{o}(\log^* n)$ as well. Since $\mathcal{A}$ computes an IS, no two consecutive nodes are assigned $c(v) = 1$.[2] By definition no sequence from $\mathcal{S}_{n'}$ is contained completely in the image of r_n, hence at most $\sigma(n') - 1 \in \mathrm{O}(f(n')g(n')) \subset \mathrm{o}(\log^* n') = \mathrm{o}(\log^* n)$ consecutive nodes compute $c(v) = 0$. Thus $\mathcal{A}'$ is $\mathrm{o}(\log^* n)$-alternating as desired. □

We will need a similar result for the MDS approximation problem. In a ring topology choosing every node is a trivial, yet constant MDS approximation. Hence we will resort to the slightly more complex topology of R_n^k, which still is present in UDG's.

Lemma 11 (Modified MDS approximation). *Assume an f-approximation algorithm $\mathcal{A}$ for the MDS problem on UDG's running in at most $g(n) \geq 1$ rounds is given, where $f(n)g(n) \in \mathrm{o}(\log^* n)$. Then an $\mathrm{o}(\log^* n)$-alternating algorithm $\mathcal{A}'$ requiring $\mathrm{o}(\log^* n)$ communication rounds exists.*

Proof. We will extend the proof of Lemma 10. In a simple ring topology choosing all nodes is a constant MDS approximation. This is not true in R_n^k. Set $\sigma_k(n) := \max\{f(n), k\}g(n)$ and define

$$\begin{aligned} \mathcal{S}_n^k := \big\{ & \big(v_1, \ldots, v_{\sigma_k(n)+2kg(n)}\big) \\ & \,|\, \forall i \in \{kg(n)+1, \ldots, \sigma_k(n)+kg(n)\} : c(v_i) = 1 \text{ on } R_n^k \big\} \ , \end{aligned} \tag{5}$$

[2] Independence is a local property, which is only affected by the input of $\mathcal{A}$ at a node v and its neighbors, i.e., the identifiers of the $g(n)+1$-neighborhood of v, and n. Since any subsequence of identifiers $(v_{i-g(n)-1}, \ldots, v_i, \ldots, v_{i+g(n)+1}) \subset Id(n')$ may occur on $R_{n'}$, the output of $\mathcal{A}'$ must still form an IS.

i.e., the set of sequences of identifiers yielding $\sigma_k(n)$ consecutive nodes taking the decision $c(v) = 1$ when $\mathcal{A}$ is executed on R_n^k, where the choices of the leading and trailing $kg(n)$ many nodes may also depend on identifiers not in the considered sequence. As the decision of any node v depends only on identifiers of nodes in $\mathcal{N}_{kg(n)}^+(v)$, $\mathcal{S}_n^k$ is well defined.

We make a case decision. The first case is that a $k_0 \in \mathbb{N}$ exists allowing a similar relabeling procedure as in Lemma 10. More precisely, $k_0, n_0 \in \mathbb{N}$ exist, such that for $n \geq n_0$ at most $\frac{n}{2}$ identifiers can simultanuously participate in disjoint sequences from $\mathcal{S}_n^{k_0}$ in a valid labeling of $R_n^{k_0}$. Thus, by setting $n' := \max\{n_0, 2n\}$, we can define $\mathcal{A}'$ to simulate a run of $\mathcal{A}$ on $R_{n'}^{k_0}$ and return the computed result. Each simulated round of $\mathcal{A}$ will require k_0 communication rounds, thus the running time of $\mathcal{A}'$ is bounded by $k_0 g(n') \in \mathrm{o}(\log^* n)$. At most $2k_0$ consecutive nodes will compute $c(v) = 0$, as $\mathcal{A}$ determines a DS, and by definition of $\mathcal{S}_{n'}^{k_0}$ at most $\sigma_{k_0}(n') - 1 \in \mathrm{O}(f(n')g(n')) \subset \mathrm{o}(\log^* n') = \mathrm{o}(\log^* n)$ consecutive nodes take the decision $c(v) = 1$.

The second case is that no pair $k_0, n_0 \in \mathbb{N}$ as assumed in the first case exists. Similar to the proof of Lemma 10, we construct a labeling of R_n^k with at least $\frac{n}{2}$ many identifiers stemming from sequences in $\mathcal{S}_n^k$. Running $\mathcal{A}$ on this instance will yield at least

$$\frac{\sigma_k(n)n}{2(\sigma_k(n) + 2kg(n))} \geq \frac{n}{6} \in \Omega(n) \tag{6}$$

many nodes choosing $c(v) = 1$. On the other hand, varying k, we get minimum dominating sets with $\mathrm{O}\left(\frac{n}{k}\right)$ many nodes. Define n_k to be the minimum n, such that it is possible to construct labelings of R_n^k with $\frac{n}{2}$ identifiers from sequences in $\mathcal{S}_n^k$. Since $\mathcal{A}$ is an f-approximation algorithm to the MDS problem on R_n^k, we conclude

$$f(n_k) \in \Omega(k) \ . \tag{7}$$

We choose $k(n)$ minimum such that $n' := 2n < n_{k(n)}$, allowing to define a injective relabeling function $r_n : Id(n) \to Id^{k(n)}(n')$, such that no element of $\mathcal{S}_{n'}^{k(n)}$ lies completely in the image of r_n. Here $Id(n)$ and $Id^k(n)$ denote the sets of admissible identifiers of R_n and R_n^k, respectively, where w.l.o.g. we assume $|Id(n)| = |Id^k(n)| = n$. We define $\mathcal{A}'$ to be the algorithm operating on R_n, but returning the result of a simulated run of $\mathcal{A}$ on $R_{n'}^{k(n)}$, where we relabel all nodes $v \in R_n$ by $r_n(v)$. By definition of $k(n)$ we have $n_{k(n)-1} \leq n'$. Together with (7) this yields

$$k(n) = (k(n) - 1) + 1 \in O(f(n_{k(n)-1}) + 1) = O(f(n')) = O(f(n)) \ , \tag{8}$$

since f grows asymptotically sublinear. Hence we can estimate the running time of $\mathcal{A}'$ by $k(n)g(n') \in O(f(n)g(n))$, using that g grows asymptotically sublinear as well.

Since the simulated run of $\mathcal{A}$ yields a dominating set, at worst $2k(n) \in O(f(n)) \subseteq O(f(n)g(n))$ many consecutive nodes may compute $c(v) = 0$. By the definitions of $\mathcal{S}_n^k$ and r_n at most $s_{k(n)}(n') - 1 < \max\{f(n'), k(n)\}g(n') \in O(f(n)g(n))$ consecutive nodes may take the decision $c(v) = 1$. Thus $\mathcal{A}'$ is $\mathrm{o}(\log^* n)$-alternating, as claimed. □

The two main theorems follow immediately from the preceding statements.

Theorem 12 (Lower bound on MaxIS approximations). *No f-approximation algorithm to the MaxIS problem on the ring R_n running in at most $g(n) \geq 1$ communication rounds with $f(n)g(n) \in \mathrm{o}(\log^* n)$ exists.*

Proof. Assuming the contrary, we can combine Lemma 10 and Lemma 9 to construct an algorithm contradicting Theorem 7.

Theorem 13 (Lower bound on MDS approximations on UDG's). *No f-approximation algorithm to the MDS problem on UDG's running in at most $g(n) \geq 1$ time with $f(n)g(n) \in \mathrm{o}(\log^* n)$ exists.*

Proof. Assuming the contrary, we can combine Lemma 11 and Lemma 9 to construct an algorithm contradicting Theorem 7.

Note that $g(n) \geq 1$ is just a formal restriction. If $g(n) = 0$ for infinitely many n, the approximation ratio f must be trivial, i.e., $f(n) \notin \mathrm{o}(n)$.

5 MaxIS Graphs

In this section we address the question wether the difficulties of MaxIS and MDS approximations are related in general. As shown in Theorem 12, on a ring topology one cannot compute a constant MaxIS approximation in constant time. Conversely, for MDS this is trivially possible by taking all nodes to be in the DS. On UDG's[3] a MaxIS, or even any maximal independent set, is a constant MDS approximation, since the number of independent neighbors of a node is bounded by a constant. Schneider et al. [1] showed how a maximal independent set can be computed on UDG's in $\mathrm{O}(\log^* n)$ time. As direct consequence of Theorem 7 this bound is tight, as a maximal independent set on a ring allows for a 3-coloring in a single round. Moreover, Theorem 13 shows this bound to be tight also with respect to MDS approximations on UDG's.

In the light of these results one might expect that approximating MaxIS is always at least as difficult as approximating MDS. On the contrary, we will now present an example showing that finding a MaxIS can be trivial (see Lemma 15), while computing a MDS remains as hard as in general graphs (see Lemma 16). Towards this end, we construct a family of graphs for which an exact solution for the MaxIS problem can be given without any communication and without knowing the problem size n. Conversely, MDS approximation on general graphs reduces to MDS approximation on this graph family. Kuhn et al. [13] showed that this problem cannot be approximated well in less than $\Omega\left(\sqrt{\log n/\log\log n}\right)$ time. The graph class examined is constructed for this special purpose, being of no practical relevance. Indeed, the solution to the MaxIS problem will already be encoded in the local topology of the graph. The main impact is that the (local) complexities of MDS and MaxIS type problems depend strongly on the considered types of graphs.

[3] More generally, on bounded growth graphs.

Definition 14 (MaxIS Graphs). *Given an arbitrary graph $G = (V, E)$, we construct a new graph $H = (V_H, E_H)$ from G. We define $V_H := V_1 \cup V_2 \cup V_3 \cup V_4$, where the V_i, $i \in \{1, 2, 3, 4\}$, are four disjoint copies of V. Denote by $v_i \in V_i$ the nodes that are copies of a node $v \in V$. The edge set connects all nodes $v_1 \in V_1$ and $v_2 \in V_2$ to all w_i that are copies of some $w \in \mathcal{N}_1^+(v)$, i.e.,*

$$E_H := \left\{\{v_i, w_j\} \,|\, w \in \mathcal{N}_1^+(v),\ i \in \{1, 2\},\ j \in \{1, 2, 3, 4\},\ v_i \neq w_j\right\} \ . \qquad (9)$$

Thus, the subgraphs induced by V_1 and V_2 are copies of G, and each $v_3 \in V_3$ (and $v_4 \in V_4$, respectively) is connected exactly to both copies of $\mathcal{N}_1^+(v)$ in V_1

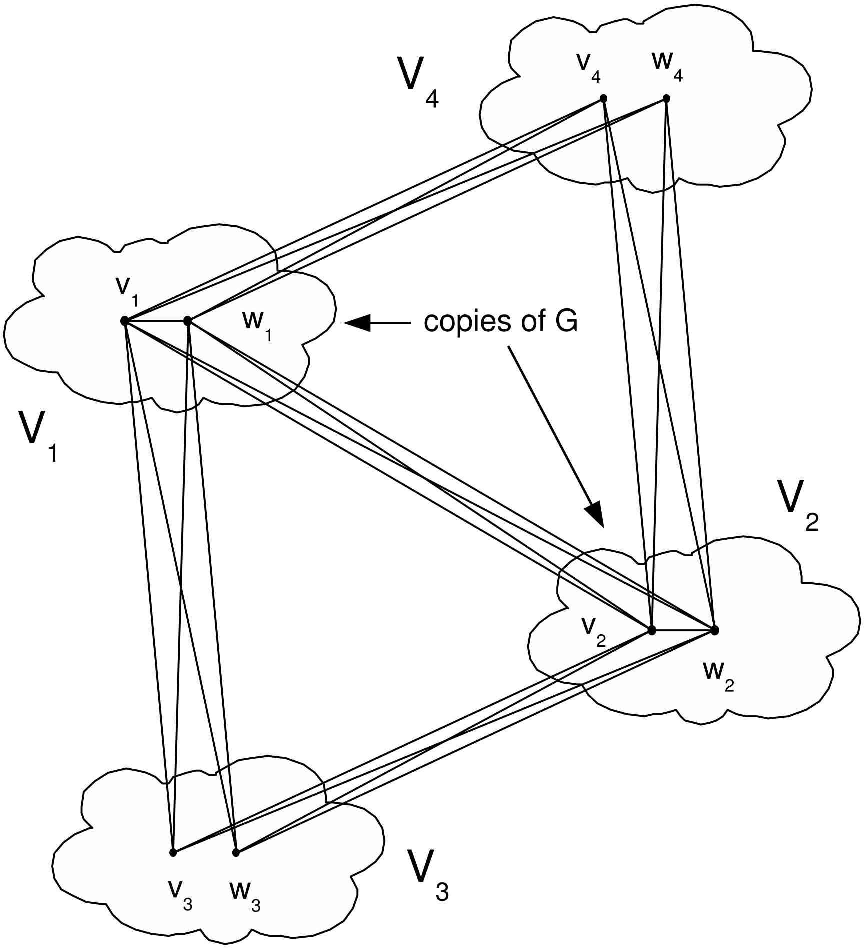


Fig. 3. Overview of the structure of a MaxIS Graph H constructed out of some graph $G = (V, E)$. The displayed vertices and egdes form the subgraph S of H induced by the copies of two nodes $v, w \in V$, where $\{v, w\} \in E$. This subgraph is the complete graph without any edges between two nodes both in $V_3 \cup V_4$. Thus the degree of a node in $V_1 \cup V_2$ with respect to S equals the one of a node in $V_1 \cup V_2$ plus three.

and V_2. An illustration is given in Figure 3. Any graph that can be constructed in this way is a MaxIS Graph.

We will proof our statements by giving a simple criterion allowing to compute a MaxIS on this class of graphs in zero rounds, and a local reduction of the general MDS approximation problem to MaxIS Graphs.

Lemma 15 (Local computation of a MaxIS on MaxIS Graphs). *The set $\{v \in V \mid |\mathcal{N}_1^+(v)| \bmod 2 = 1\}$ is a MaxIS for any MaxIS Graph. It can be determined locally, without communication.*

Proof. We use the notation of Definition 14. The set of nodes $I := V_3 \cup V_4$ is independent by construction. A node $v_i \in V_i$, $i \in \{3,4\}$, has $2|\mathcal{N}_1^+(v)|$ many neighbors in $V_1 \cup V_2$, hence $|\mathcal{N}_1^+(v_i)|$ is odd. On the other hand, for a node $v_i \in V_i$, $i \in \{1,2\}$, we have $|\mathcal{N}_1^+(v_i)| = 4|\mathcal{N}_1^+(v)|$, which is even. Thus $I = \{v \in V \mid |\mathcal{N}_1^+(v)| \bmod 2 = 1\}$ holds. Moreover, since the sequence of nodes (v_1, v_3, v_2, v_4) forms a cycle, at most two of them may participate in an IS, thus I is a MaxIS.

As nodes know the number of their neighbors, each node can determine wether it is in I or not without any communication. □

Lemma 16 (Reduction of MDS to MDS on MaxIS Graphs). *We use the notation of Definition 14. Given an f-approximation algorithm $\mathcal{A}$ to the MDS problem on MaxIS Graphs running in $g(n)$ time, we can define the following algorithm $\mathcal{A}'$ operating on an arbitrary graph G:*

1. *Simulate a run of $\mathcal{A}$ on the MaxIS Graph H constructed from G.*
2. *Return for each node $v \in V$ $c(v) = 1$ if $\mathcal{A}$ computed $c(v_i) = 1$ for some $i \in \{1,2,3,4\}$, and $c(v) = 0$ else.*

Algorithm $\mathcal{A}'$ is an $f(4n)$-approximation algorithm to the MDS problem running in $g(4n)$ rounds. Up to constants it is as efficient as the original one, i.e., it is an $\mathrm{O}(f(n))$ approximation running in $\mathrm{O}(g(n))$ time.

Proof. Since $\mathcal{A}$ computes a MDS of H, Algorithm $\mathcal{A}'$ will return a MDS of G: If $v_1 \in V_1$ is covered by $w_i \in V_i$ for some $i \in \{1,2,3,4\}$, w will cover v in G. Thus $\mathcal{A}'$ works correctly.

If M is a MDS of G, the copy $M_1 := \{m_1 \mid m \in M\}$ is a MDS of H. Conversely, any node $v_1 \in V_1$ can only be covered by copies of nodes $w \in \mathcal{N}_1^+(v)$, hence the sizes of minimum dominating sets of G and H coincede. Thus, if $\mathcal{A}$ is an $f(n)$-approximation algorithm to the MDS problem on MaxIS Graphs, $\mathcal{A}'$ will be an $f(4n)$-approximation algorithm to the MDS problem on general graphs. Since any MDS approximation algorithm will trivially reach at least an approximation ratio of n, $\mathcal{A}'$ is an $O(f(n))$-approximation.

Having the nodes of G simulate a run of $\mathcal{A}$ on H does not require any additional communication rounds. As we have no restrictions to message sizes, we can simply append the information which edge in H is used to all communications, while each node $v \in V$ simulates v_i, $i \in \{1,2,3,4\}$. This is a simple task, as the neighbourhood of each v_i is determined solely by $\mathcal{N}_1^+(v)$, and we have no restrictions to local computations. Hence, if $\mathcal{A}$ runs in $g(n)$ time, $\mathcal{A}'$ will require at most $g(4n)$ time to complete. As the diameter of any graph is bounded by the number of nodes n, any algorithm in our computational model can be realized using at most n communication rounds. Thus $\mathcal{A}'$ needs at worst $O(g(n))$ rounds to complete. □

Finally we conclude that the local complexities of the MaxIS and MDS approximation problems are incomparable.

Theorem 17. *Assume an f_1-approximation algorithm to the MaxIS problem and an f_2-approximation algorithm to the MDS problem, both on a family of graphs $\mathcal{F}$, are given. Denote by $g_1(n) \geq 1$ and $g_2(n) \geq 1$ bounds for their running times. Furthermore assume the products $p_1(n) := f_1(n) \cdot g_1(n)$ and $p_2(n) := f_2(n) \cdot g_2(n)$ are minimum, i.e., the algorithms are optimum in this sense. Then neither $p_1 \in \mathrm{O}(p_2)$ nor $p_2 \in \mathrm{O}(p_1)$ holds independent of $\mathcal{F}$.*

Proof. The lower bound of Kuhn et al. [13] and Lemma 16 show that on MaxIS Graphs we have $p_2 \in \Omega\left(\sqrt{\log n / \log\log n}\right)$, while Lemma 15 yields $p_1 \in \mathrm{O}(1)$. Conversely, for a ring topology we trivially have $p_2 \in \mathrm{O}(1)$, while Theorem 12 gives $p_1 \notin \mathrm{o}(\log^* n)$, implying $p_1 \notin \mathrm{O}(1) = \mathrm{O}(p_2)$. □

6 Conclusion

In this paper we extended Linial's lower bound to the well-known MDS and MaxIS problems on UDG's. The product between running time and approximation quality of any deterministic local algorithm for these problems cannot be in $\mathrm{o}(\log^* n)$.

In a couple of graph classes, especially geometric graphs, a MIS is a special case of the MDS approximation problem. Consequently one might believe that coming up with a distributed algorithm for MIS is harder, or at least not simpler, than for MDS. In the second part of the paper we showed that this is not always true. The constructed MaxIS graphs demonstrate that the two problems are generally incomparable: In this graph class a MaxIS can be "computed" locally without any communication, while any MDS approximation algorithm on MaxIS graphs could be used to solve the problem on general graphs. Adding one more tessera to the picture, the extension of Linial's lower bound to the MaxIS approximation problem holds on virtually any graph class.

We hope that our findings will help to get a better understanding of distributed algorithms, eventually permitting a classification of local problems reflecting their complexity.

References

1. Schneider, J., Wattenhofer, R.: A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In: Proc. Twenty-Seventh Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (2008)
2. Naor, M.: A Lower Bound on Probabilistic Algorithms for Distributive Ring Coloring. SIAM Journal on Discrete Mathematics 4(3), 409–412 (1991)

3. Kuhn, F., Moscibroda, T., Wattenhofer, R.: On the Locality of Bounded Growth. In: Proc. 24th ACM Symposium on the Principles of Distributed Computing (2005)
4. Cole, R., Vishkin, U.: Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. Information and Control 70(1), 32–53 (1986)
5. Kuhn, F., Moscibroda, T., Wattenhofer, R.: The Price of Being Near-Sighted. In: Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (2006)
6. Kuhn, F., Wattenhofer, R.: Constant-Time Distributed Dominating Set Approximation. In: Proc. 22nd ACM Symposium on the Principles of Distributed Computing (2003)
7. Jia, L., Rajaraman, R., Suel, T.: An Efficient Distributed Algorithm for Constructing Small Dominating Sets. Distributed Computing 15(4), 193–205 (2002)
8. Gfeller, B., Vicari, E.: A Randomized Distributed Algorithm for the Maximal Independent Set Problem in Growth-Bounded Graphs. In: Proc. 26th annual ACM symposium on Principles of distributed computing (2007)
9. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2000)
10. Luby, M.: A Simple Parallel Algorithm for the Maximal Independent Set Problem. SIAM J. Comput. 15(4), 1036–1055 (1986)
11. Naor, M., Stockmeyer, L.: What Can Be Computed Locally? SIAM J. Comput. 24(6), 1259–1277 (1995)
12. Linial, N.: Locality in Distributed Graph Algorithms. SIAM J. Comput. 21(1), 193–201 (1992)
13. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What Cannot Be Computed Locally. In: Proc. 23rd annual ACM symposium on Principles of distributed computing (2004)
14. Wiese, A., Kranakis, E.: Local PTAS for Independent Set and Vertex Cover in Location Aware Unit Disk Graphs. In: Proc. 4th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (2008)
15. Czygrinow, A., Hańćkowiak, M., Wawrzyniak, W.: Fast distributed approximations in planar graphs. In: Proc. 22nd International Symposium on Distributed Computing (2008)

Continuous Consensus with Failures and Recoveries

Tal Mizrahi and Yoram Moses

Department of Electrical Engineering, Technion, Haifa 32000 Israel
deweastern@yahoo.com, moses@ee.technion.ac.il

Abstract. A *continuous consensus* (CC) protocol maintains for each process i at each time k an up-to-date core $M_i[k]$ of information about the past, so that the cores at all processes are guaranteed to be identical. This is a generalization of simultaneous consensus that provides processes with the ability to perform simultaneously coordinated actions, and saves the need to compute multiple instances of simultaneous consensus at any given time. For an indefinite ongoing service of this type, it is somewhat unreasonable to assume a bound on the number of processes that ever fail. Moreover, over time, we can expect failed processes to be corrected. A failure assumption called (m,t) interval-bounded failures, closely related to the *window of vulnerability* model of Castro and Liskov, is considered for this type of service. The assumption is that in any given interval of m rounds, at most t processes can display faulty behavior.

This paper presents an efficient CC protocol for the (m,t) bound in the crash and sending omissions failure models. A matching lower bound proof shows that the protocol is optimal in all runs (and not just in the worst case): For each and every behavior of the adversary, and at each time instant m, the core that our protocol maintains at time m is a superset of the core maintained by any other correct CC protocol under the same adversary. The lower bound is a significant generalization of previous proofs for common knowledge, and it applies to continuous consensus in a wide class of benign failure models, including the general omissions model, for which no similar proof existed.

Keywords: Agreement problem, Consensus, Continuous Consensus, Distributed algorithm, Early decision, Common Knowledge, Lower bound, Modularity, Process crash failure, Omission failures, Process recovery, Round-based computation model, Simultaneity, Synchronous message-passing system.

1 Introduction

Fault-tolerant systems often require a means by which independent processes or processors can arrive at an exact mutual agreement of some kind. As a result, reaching consensus is one of the most fundamental problems in fault-tolerant distributed computing, dating back to the seminal work of Pease, Shostak, and Lamport [17]. In the first consensus algorithms, decisions were reached in the same round of communication by all correct processes. It was soon discovered, however, that allowing decisions to be made in different rounds ("eventual agreement") at different sites gives rise to simpler protocols in which the processes can often decide much faster than they would if we insist that decisions be simultaneous [5]. There are cases in which eventual agreement often suffices: In recording the outcomes of transactions, for example. In other

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 408–422, 2008.

cases, however, a simultaneous decision or action is often required or beneficial: E.g., when one distributed algorithm ends and another one begins, and the two may interfere with each other if executed concurrently. Similarly, many synchronous algorithms assume that the starting round is the same at all sites. Finally, there are cases in which the responses in a given round to external requests at different sites for, say resource allocation, must be consistent. A familiar application that assists in many of these is the Firing Squad problem [1, 4].

Motivated by [5, 9], the problem of reaching simultaneous consensus was shown in [7, 13] to require the correct processes to attain *common knowledge* about the existence of particular initial values. By computing common knowledge efficiently in the crash failure model, they designed protocols for simultaneous consensus that are optimal *in all runs*, and not just in the worst case: In every execution, they decided as fast as any correct protocol could, given the same behavior of the adversary.[1] Finally, they also observed that computing facts that are common knowledge, in essentially the same manner, can solve other simultaneous coordination problems such as the firing squad problem. While [7] considered crash failures, [13] extended the analysis of common knowledge to omission failure models. They designed an efficient protocol for computing all facts that are common knowledge at a given point, and used this to derive optimal protocols for a wide class of *simultaneous choice* problems. More recently, this work was further generalized in [11], where a general service called *Continuous Consensus* (CC) that serves to support simultaneous coordination was defined.[2] It is described as follows.

Suppose that we are interested in maintaining a simultaneously consistent view regarding a set of events $\mathcal{E}$ in the system. These are application-dependent, but will typically record inputs that processes receive at various times, values that certain variables have at a given time, and faulty behavior in the form of failed or inconsistent message deliveries. A *(uniform)*[3] *continuous consensus* protocol maintains at all times $k \geq 0$ a *core* $M_i[k]$ of events of $\mathcal{E}$ at every site i. In every run of this protocol the following properties are required to hold, for all processes i and j.

Accuracy: All events in $M_i[k]$ occurred in the run.
Consistency: $M_i[k] = M_j[k]$ at all times k.
Completeness: If e occurs at a process j at a point at which j is nonfaulty, then $e \in M_i[k]$ must hold at some time k.

The continuous consensus problem generalizes many problems having to do with simultaneous coordination. Using the core of a CC primitive, processes can independently choose compatible actions in the same round. Thus, the *firing squad* problem [4]

[1] We think of the adversary as determining the pattern of initial votes and the pattern in which failures occur, in each given execution of the protocol. The performance of a protocol P can be compared to that of P' by looking at respective runs in which the adversary behaves in the same way.

[2] A different problem, by the same name, was independently defined by Dolev and Rajsbaum. Sometimes called *Long-lived consensus*, it concerns maintaining consensus on a single bit in a self-stabilizing fashion [6].

[3] In [11] we defined both non-uniform and a uniform variants of continuous consensus. In the (m,t) model processes can fail and recover repeatedly, so we focus on the uniform variant, in which *all* processes maintain the same core at all times.

can immediately be implemented, but so can, say, consistent resource allocation, mutual exclusion, etc. In a world in which distributed systems increasingly interact with an outside world, a CC protocol facilitates the system's ability to present a consistent view to the world.

The consensus problem has rightfully attracted considerable attention in the last thirty years, since it is a basic primitive without which many other tasks are unattainable. Continuous consensus offers a strict generalization of (simultaneous) consensus: While consensus is concerned with agreeing on a single bit, continuous consensus allows decisions to be taken based on a broader picture involving a number of events of interest. In particular, it provides the processes with Byzantine Agreement regarding the values of all relevant external inputs that do or do not arrive at the various processes (where relevance is determined by $\mathcal{E}$). It thus eliminates the need for initiating a separate instance of a consensus protocols for each of the facts of interest, and provides the same benefits at a lower cost.

Popular failure assumptions bound the overall number of failures that may occur during the execution of a protocol [17]. Clearly, if the adversary can cause all of the processes to fail (by crashing, say) then the best protocols cannot be expected to achieve much. Typically, a process is considered *faulty* in a given run if it ever displays incorrect behavior during the course of the run. Such assumptions are reasonable for applications that are short-lived. For applications such as continuous consensus, however, which is an ongoing service that should operate indefinitely, expecting (or assuming) that certain processes remain correct throughout the lifetime of the system may be overly optimistic. Conversely, it would also be natural to expect various failed processes to be repaired over time, and thus resume correct participation in the protocol. In such a setting, it is more reasonable to consider bounds on the number and/or types of failures that can occur over limited intervals of time. Indeed, Castro and Liskov designed a protocol for state-machine replication in an Byzantine environment that is correct provided that no more than $n/3$ processes fail during an execution-dependent period that they call a *window of vulnerability* [2]. We follow a similar path in this paper, and consider continuous consensus in omissive settings under the (m,t)-*interval bound* assumption (called the (m,t) *model* for short), which states that there is no m-round interval in which more than t processes display faulty behavior. The contributions of this paper are:

- The (m,t)-interval bounded omission failure model is introduced.
- A continuous consensus protocol mt-Cc for the (m,t) model whenever $t < m$ is presented.
- mt-Cc is shown to be optimal *in all runs* for this model: For any CC protocol P and any given behavior of the adversary, the core maintained by mt-Cc at each time k is a superset of the core maintained by P under the same conditions.
- The lower bound used in proving the optimality of mt-Cc is the first one tackling the possibility of process recoveries in a simultaneous agreement problem. All previous lower bounds for simultaneous consensus and continuous consensus in the presence of crash or omission failures make essential use of the fact that failures accumulate monotonically over time [7, 11, 12, 13] The new lower bound proof in this paper overcomes this hurdle.
- The main lemmas in the lower bound apply to a large class of benign failure models, including the general omissions model studied in [13].

2 Model and Preliminary Definitions

Our treatment of the continuous consensus problem will be driven by a knowledge-based analysis. A general approach to modeling knowledge in distributed systems was initiated in [9] and given a detailed foundation in [8] (most relevant to the current paper are Chapters 4 and 6). For ease of exposition, our definitions will be tailored to the proofs for continuous consensus in our setting.

The Communication Network. We consider a synchronous network with $n \geq 2$ possibly unreliable processes, denoted by $\mathbb{P} = \{1,2,\ldots,n\}$. Each pair of processes is connected by a two-way communication link. Processes correctly identify the sender of every message they receive. They share a discrete global clock that starts out at time 0 and advances by increments of one. Communication in the system proceeds in a sequence of *rounds*, with round $k+1$ taking place between time k and time $k+1$. Each process starts in some *initial state* at time 0. Then, in every following round, the process first sends a set of messages to other processes, and then receives messages sent to it by other processes during the same round. In addition, a process may also receive requests for service from clients external to the system (think, for example, of deposits and withdrawals at branches of a bank), or input from sensors with information about the world outside of the system (e.g., smoke detectors). Finally, the process may perform local computations based on the messages it has received. The history of an infinite execution of such a network will be called a *run*.

Modeling the Environment: Inputs and Failures. A protocol is designed to satisfy a specification when executed within a given setting, which determines the aspects of the execution that are not controlled by the protocol. In our framework the setting can be described in terms of a set of adversaries that control the two central aspects of any given run: inputs and failures.

Inputs. Every process starts out in an initial local state from some set Σ_i, and can receive an external input in any given round k (this input is considered as *arriving* at time k). The initial local state of each process can be thought of as its external input at time 0. We represent the external inputs in an infinite execution as follows. Define a set $\mathbb{V} = \mathbb{P} \times \mathbb{N}$ of *process-time nodes* (or *nodes*, for short). We shall use a node $(i,k) \in \mathbb{V}$ to refer to process i at time k. We denote by $\mathbb{V}(k)$ the set $\mathbb{P} \times \{k\} \subset \mathbb{V}$ of all nodes (i,k), and if $k \leq \ell$ then we define $\mathbb{V}[k,\ell] = \mathbb{P} \times \{k,k+1,\ldots,\ell\} = \mathbb{V}(k) \cup \cdots \cup \mathbb{V}(\ell)$.

An *(external) input assignment* is a function ζ associating with every node $(i,0)$ at time 0 an initial state from Σ_i and with each node (i,k) with $k > 0$ an input from a set of possible inputs, which is denoted by $\mathbb{I}$. (The set $\mathbb{I}$ typically contains a special symbol $\perp$, corresponding to a "null" external input.) An *input model* consists of a set Ξ of input assignments. For the purpose of the analysis in this paper, we will focus on input models in which the inputs at different nodes of $\mathbb{V}$ are not correlated. An input model Ξ is said to be *independent* if for every $\zeta, \zeta' \in \Xi$ and every set $T \subseteq \mathbb{V}$, we are guaranteed that $\zeta_T \in \Xi$, where ζ_T is the input assignment that coincides with ζ on T and with ζ' on $\mathbb{V} \setminus T$. In the classical consensus problem, for example, $\Sigma_i = \{0,1\}$ and $I = \{\perp\}$. The input model consists of all possible input assignments based on these Σ_i and on I, and is clearly independent.

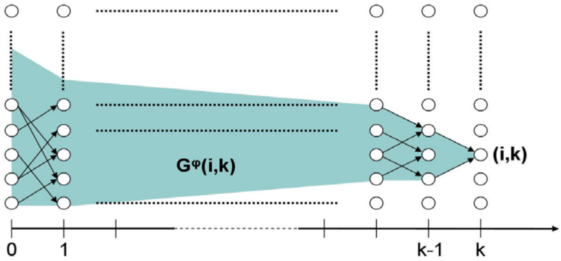


Fig. 1. A communication graph G^{φ} and i's subgraph $\mathsf{G}^{\varphi}(i,k)$

Benign failures and recoveries. The second aspect of a run that is determined by the adversary has to do with the type of failures possible, and the patterns in which they can occur. When focusing on benign failures, as in this paper, any message that is delivered is sent according to the protocol.

A *failure pattern* for benign failures is a function $\varphi : \mathbb{V} \to 2^{\mathbb{P}}$. Intuitively, a failure pattern determines which channels are blocked (due to failure) in any given round. More concretely, $\varphi(i,k)$ is the set of processes to which messages sent by i in round $k+1$ will not be delivered.[4] Equivalently, a message sent by i to j in round $k+1$ will be delivered iff $j \notin \varphi(i,k)$. We denote by ☺ the failure-free pattern satisfying that ☺$(v) = \emptyset$ for all $v \in \mathbb{V}$.

We identify a failure pattern φ with a *communication graph*,[5] denoted by G^{φ}, detailing the active channels in an execution. We define the graph $\mathsf{G}^{\varphi} = (\mathbb{V}, E^{\varphi})$, where $E^{\varphi} = \{\langle (i,k),(j,k+1)\rangle : \ j \notin \varphi(i,k)\}$. Notice that φ uniquely determines G^{φ} and vice-versa. For a node $v = (i,k) \in \mathbb{V}$, we denote by $\mathsf{G}^{\varphi}(i,k)$ (or $\mathsf{G}^{\varphi}(v)$) the subgraph of G^{φ} generated by v and all nodes $w \in \mathbb{V}$ such that there is a directed path from w to v in G^{φ}. This subgraph captures the potential "causal past" of v under φ: all nodes by which v can be affected via communication, either directly or indirectly. An illustration of a graph $\mathsf{G}^{\varphi}(i,k)$ is depicted in Figure 1. Given a set of nodes $S \subseteq \mathbb{V}$, we denote by $\mathsf{G}^{\varphi}(S)$ the subgraph of G^{φ} obtained by taking the union of the graphs $\mathsf{G}^{\varphi}(v)$, for $v \in S$.

A *(benign) failure model* for synchronous systems is a set Φ of failure patterns φ. Intuitively we can view a benign failure model as one in which failures affect message deliveries. Any message that *is* delivered is one that was sent according to the protocol. The standard benign models, including t-bounded crash failures, sending omissions, receiving omissions, or general omissions [13] are easily modeled in this fashion. The same applies to models that bound the number of undelivered messages per round (e.g., [3, 18]).

[4] Note that a failure pattern can model process failures as well as communication failures. If $i \in \varphi(i,k)$ then i "does not receive its own message" in round $k+1$. This corresponds to a loss of i's local memory, and captures the *recovery from a process crash.* It allows a distinction between recovery from a local process crash and reconnection following disconnected operation.

[5] Communication graphs were first used, informally, in the anasysis of consensus by Merritt [10]. They were formalized in [8, 13]. Our modelling is taken from the latter. Similar modelling was more recently used in the Heard-Of model of Charron-Bost and Schipper [3].

The (m,t) *interval-bounded failure model.* For the purpose of this paper, it is convenient to consider a process $i \in \mathbb{P}$ as being *faulty at* (i,k) *according to* φ if $\varphi(i,k) \neq \emptyset$ (i.e., if one or more of i's outgoing channels is blocked in round $k+1$). We say that $i \in \mathbb{P}$ is *faulty according to* φ in the interval $\mathbb{V}[k,k+m-1]$ if i is faulty at (i,k') for some k' satisfying $k \leq k' \leq k+m-1$. The (m,t) interval-bounded failure model is captured by the set $\mathrm{OM}(m,t)$ of all failure patterns φ such that, for every $k \geq 0$, no more than t processes are faulty according to φ in $\mathbb{V}[k,k+m-1]$. The standard (sending) omissions failure model is then captured by $\mathrm{OM}(\infty,t) = \bigcap_{m\geq 1} \mathrm{OM}(m,t)$.

Environments and Protocols. We shall design protocols for continuous consensus with respect to an environment that is specified by a pair $\Delta = \Phi \times \Xi$, where $\Phi = \mathrm{OM}(m,t)$ and Ξ is an independent input model. Each element (φ,ζ) of the environment Δ we call an *adversary*. A protocol for process i is a deterministic function P_i from the local state of a process to local actions and an assignment of at most one message to each of its outgoing links. Its local state at time $k+1$ consists of its local variables after the local actions at time k are performed, its external input at $(i,k+1)$, and the messages delivered to it in round $k+1$ on its incoming channels.[6]

A *joint protocol* (or just protocol for short) is a tuple $P = \{P_i\}_{i\in\mathbb{P}}$. For a given adversary $(\varphi,\zeta) \in \Phi \times \Xi$, a protocol P determines a unique *run* $r = P(\varphi,\zeta) : \mathbb{V} \rightarrow L \times M \times \mathbb{P}$, which assigns a local state to each node (i,k) in the unique manner consistent with P, φ and ζ: initial states and external inputs are determined by ζ, protocol P determines local states and outgoing messages, and φ determines which of the sent messages on incoming channels produces a delivered message.

3 Continuous Consensus and Common Knowledge

Knowledge theory, and specifically the notion of common knowledge, are central to the study of simultaneously coordinated actions. This connection has been developed and described in [7, 8, 11, 13, 15, 16]. In particular, [11] showed that in a continuous consensus protocol, the contents of the core at a given time k are guaranteed to be common knowledge. In this section we will review this connection, in a manner that will be light on details and focus on the elements needed for our analysis of CC protocols. In particular, we will make use of a very lean logical language in which the only modal construct is common knowledge. For a more complete exposition of knowledge theory see [8].

We are interested in studying CC in an environment $\Delta = (\Phi,\Xi)$ in which Ξ is an independent input model. We say that Ξ is *non-degenerate* at a node $(i,k) \in \mathbb{V}$ if there are $\zeta,\zeta' \in \Xi$ such that $\zeta(i,k) \neq \zeta'(i,k)$. A CC application needs to collect information only about non-degenerate nodes, since on the remaining nodes the external input is fixed a priori. We will consider a set of *primitive events* $\mathcal{E}$ w.r.t. Ξ each of the form $e = \langle \alpha,i,k \rangle$ where $\alpha \in \Xi$ and Ξ non-degenerate at (i,k). The event $e = \langle \alpha,i,k \rangle$ is said to occur in run $r = P(\varphi,\zeta)$ exactly if $\zeta(i,k) = \alpha$. The core will be assumed to consist of a set of such primitive events.

[6] To model local crashes, we would replace the local variables in i's local state at time k by λ if $i \in \varphi(i,k+1)$.

Knowledge is analyzed within the context of a *system* $\mathcal{R} = \mathcal{R}(P,\Delta)$ consisting of the set of runs $r(P,\varphi,\zeta)$ of a protocol P in Δ. A pair (r,k) where r is a run and k is a time is called a *point*. Process i's local state at (i,k) in r is denoted by $r_i(k)$. We say that i *cannot distinguish* (r,k) from (r',k) if $r_i(k) = r'_i(k)$. We consider knowledge formulas to be true or false at a given point (r,k) in the context of a system $\mathcal{R}$. In the standard definition [8], process i knows a fact A at (r,k) if A holds at all points that i cannot distinguish from (r,k).

For every $e \in \mathcal{E}$, we denote by $C(e)$ the fact that e is common knowledge, which intuitively means that everybody knows e, everybody knows that everybody knows e, and so on ad infinitum. We define common knowledge formally in the following way, which can be shown to capture this intuition in a precise sense. We define a *reachability relation* $\sim$ among points of $\mathcal{R}$ to be the least relation satisfying:

1. if $r_i(k) = r'_i(k)$ then $(r,k) \sim (r',k)$, and
2. if, for some $r'' \in \mathcal{R}$, both $(r,k) \sim (r'',k)$ and $(r'',k) \sim (r',k)$, then $(r,k) \sim (r',k)$.

In other words, define the *similarity graph* over $\mathcal{R}$ to be an undirected graph whose nodes are the points of $\mathcal{R}$, and where two points are connected by an edge if there is a process that cannot distinguish between them. Then $(r,k) \sim (r',k)$ if both points are in the same *connected component* of the similarity graph over $\mathcal{R}$. Notice that $\sim$ is an equivalence relation, since connected components define a partition on the nodes of an undirected graph. We denote by $(\mathcal{R},r,k) \models C(e)$ the fact that e is common knowledge to the processes at time k in r. We formally define:

$$(\mathcal{R},r,k) \models C(e) \quad \text{if event } e \text{ occurs in every } r' \in \mathcal{R} \text{ satisfying } \ (r,k) \sim (r',k).$$

We can formally prove that the events in the core of a CC protocol must be common knowledge:

Theorem 1 ([7, 11]). *Let P be a CC protocol for Δ, and let $\mathcal{R} = \mathcal{R}(P,\Delta)$. For all runs $r \in \mathcal{R}$, times $k \geq 0$ and events $e \in \mathcal{E}$, we have:*

$$\textit{If} \quad e \in M^r_i[k] \quad \textit{then} \quad (\mathcal{R},r,k) \models C(e).$$

4 Lower Bounds for CC in Benign Failure Models

We can show that an event at a node (i,k) cannot be in the the core of a CC protocol at time ℓ if we prove that the event is not common knowledge by time ℓ. It turns out that the failure models and failure patterns play a central role in forcing events *not* to be common knowledge. By working with these directly, we avoid some unnecessary notational clutter.

Given a failure model Φ, we define the *similarity* relation $\approx$ on $\Phi \times \mathbb{N}$ to be the least relation satisfying:

1. if $\mathsf{G}^{\varphi}(i,k) = \mathsf{G}^{\varphi'}(i,k)$ holds for some process $i \in \mathbb{P}$, then $(\varphi,k) \approx (\varphi',k)$, and
2. if, for some $\varphi'' \in \Phi$, both $(\varphi,k) \approx (\varphi'',k)$ and $(\varphi'',k) \approx (\varphi',k)$, then $(\varphi,k) \approx (\varphi',k)$.

As in the case of $\sim$, the $\approx$ relation is an equivalence relation, because condition (1) is reflexive and symmetric, while condition (2) is symmetric and guarantees transitivity.

We say that process i is *shut out* in round $k+1$ by φ (equivalently, (i,k) is shut out by φ), if $\varphi(i,k) \supseteq \mathbb{P} \setminus \{i\}$. Intuitively, this means that no process receives a message from i in round $k+1$. We say that a node (i,k) is *hidden by* φ *at time* ℓ if $(\varphi,\ell) \approx (\psi,\ell)$ for some pattern ψ in which i is shut out rounds $k+1,\ldots,\ell$. If a node is hidden by the failure pattern, then its contents cannot be common knowledge, no matter what protocol is being used, as formalized by the following theorem:

Theorem 2. *Let $\mathcal{R} = \mathcal{R}(P,\Delta)$, where $\Delta = \Phi \times \Xi$ and Ξ is independent, let $e = (\alpha,i,k)$ be a primitive event and let $r = P(\varphi,\zeta) \in \mathcal{R}$. If (i,k) is hidden by φ at ℓ then $(\mathcal{R},r,\ell) \not\models C(e)$.*

A major tool in our impossibility results is the notion of a covered set of nodes. Intuitively, a covered set S at a given point satisfies three main properties. (i) it is not common knowledge that even on node of s is faulty, as there is a reachable point in which all nodes if S are nonfaulty. (ii) for every node in s, its contents are not common knowledge, in the sense that there is a reachable point in which this node is silent. (iii) Finally, the reachable points in (i) and (ii) all agree with the current point on the nodes not in S. Formally, we proceed as follows.

Definition 1 (Covered Nodes). *Let $S \subset \mathbb{V}$, and denote $\bar{S}_\ell = \mathbb{V}[0,\ell] \setminus S$. We say that S is* covered *by φ at time ℓ (w.r.t. Φ) if*

- *for every node $(i,k){=}v \in S$ there exist φ_v such that (a) $(\varphi,\ell) \approx (\varphi_v,\ell)$, (b) $\mathsf{G}^{\varphi}(\bar{S}_\ell) = \mathsf{G}^{\varphi_v}(\bar{S}_\ell)$, and (c) i is shut out in rounds $k+1,\ldots,\ell$ of φ_v. Moreover,*
- *there exists $\varphi' \in \Phi$ such that (d) $(\varphi,\ell) \approx (\varphi',\ell)$, (e) $\mathsf{G}^{\varphi}(\bar{S}_\ell) = \mathsf{G}^{\varphi'}(\bar{S}_\ell)$, and (f) $\varphi'(v) = \emptyset$ for all nodes $v \in S$.*

The fact that $\approx$ is an equivalence relation immediately yields

Lemma 1. *Fix Φ and let $S \subseteq \mathbb{V}$. If $(\varphi,\ell) \approx (\varphi',\ell)$ and $\mathsf{G}^{\varphi}(\bar{S}_\ell) = \mathsf{G}^{\varphi'}(\bar{S}_\ell)$ then S is covered by φ at ℓ iff S is covered by φ' at ℓ.*

One set that is guaranteed to be covered at time ℓ is $\mathbb{V}(\ell)$:

Lemma 2. *$\mathbb{V}(\ell)$ is covered by φ at ℓ, for every failure pattern φ and $\ell \geq 0$.*

Proof. Denote $\mathbb{V}(\ell)$ by S. Choose $\varphi = \varphi_v$ for each $v = (j,\ell) \in S$ to satisfy clauses (a), (b) and (c) of the definition for $v \in S$. The clauses are immediate for φ_v. To complete the claim we need to show that S is covered by φ at ℓ: Define φ' for clauses (d), (e) and (f) to be the pattern obtained from φ by setting $\varphi'(j,\ell) = \emptyset$ for every $j \in \mathbb{P}$. ■

Monotonicity. Our "lower bound" proofs take the form of proving impossibility of common knowledge under various circumstances. To make the results in this section widely applicable, we state and prove results with respect to classes of failure models rather than just to the (m,t) model. We shall focus on models with the property that reducing the set of blocked edges in a failure pattern yields a legal failure pattern. Formally, we say that ψ *improves on* φ, and write $\psi \sqsubseteq \varphi$, if $\psi(v) \subseteq \varphi(v)$ for every $v \in \mathbb{V}$. (Notice that the fault-free failure pattern ☺ satisfies ☺ $\sqsubseteq \varphi$ for all patterns φ.) It is easy to check that $\psi \sqsubseteq \varphi$ iff G^{φ} is a subgraph of G^{ψ}. A failure model Φ is *monotonically closed* (or *monotone*, for short) if for every pattern ψ that improves on a pattern in Φ is itself in φ. Formally, $\varphi \in \Phi$ and $\psi \sqsubseteq \varphi$ implies $\psi \in \Phi$. Clearly, if Φ is monotonically closed, then ☺ $\in \Phi$. Observe that $\textsc{Om}(m,t)$ is monotonically closed.

Single Stalling. In many failure models, a crucial obstacle to consensus is caused by executions in which a single process is shut out in every round. This will also underly the following development, where we show essentially that for a broad class of failure models, the adversary's ability to fail an arbitrary process per round will keep any events from entering the CC core. We call this *single stalling*. To capture this formally for general failure models, we make the following definitions.

We say that two patterns φ and φ' *agree* on a set of nodes $T \subseteq \mathbb{V}$ if $\mathsf{G}^{\varphi}(T) = \mathsf{G}^{\varphi'}(T)$. Notice that this neither implies or is implied by having $\varphi(v) = \varphi'(v)$ for all $v \in T$. This notion of agreement depends on the nodes with directed paths into nodes $v \in T$, while $\varphi(v)$ specifies the outgoing edges from v. In a precise sense, though, the execution of a given protocol P on failure pattern φ can be shown to be indistinguishable at the nodes of T from its execution on φ'. This is closely related to the the structure underlying the proof of Theorem 2.

Definition 2 (Single stalling). *Let Φ be a monotone failure model. Fix Φ, let $\varphi \in \Phi$, and let $W \subset \mathbb{V}$. We say that* $\mathsf{G}^{\varphi}(W)$ admits single stalling in $[k,\ell]$ *(w.r.t. Φ) if, for every sequence $\sigma = p_{k+1},\ldots,p_{\ell}$ of processes (possibly with repetitions), there exists a pattern $\varphi_{\sigma} \in \Phi$ agreeing with φ on W such that both (a) $(\varphi,\ell) \approx (\varphi_{\sigma},\ell)$, and (b) each process p_j in σ is shut out in round j of in φ_{σ}, for all $j = k+1,\ldots,\ell$.*

The heart of our lower bound proof comes in the following lemma. Roughly speaking, it states that if a set of nodes S containing the block $\mathbb{V}[k+1,\ell]$ of nodes from time $k+1$ to ℓ are all covered, and it is consistent with the information stored in the complement set $\bar{S}_{\ell}$ that the node (j,k) could be shut out, then (j,k) can be added to the set of covered nodes. to the set of covered nodes. This allows to prove incrementally that the nodes at and after the critical time $c = c(\varphi)$ that are not in the critical set are all covered. Formally, we now show the following:

Lemma 3. *Fix a monotone failure model Φ. Let $k < \ell$ and assume that the set $S \supseteq \mathbb{V}[k+1,\ell]$ is covered by φ at ℓ. Let $T = S \cup \{(j,k)\}$, and let ψ be the pattern obtained from φ by shutting out (j,k). If $\psi \in \Phi$ and $\mathsf{G}^{\psi}(\bar{T}_{\ell})$ admits single stalling in $[k+1,\ell]$, then T is hidden by φ at ℓ.*

The proof of Lemma 3 appears in the Appendix. Based on Lemma 3, we obtain:

Lemma 4. *Let $k < \ell$. If $S = \mathbb{V}[k+1,\ell]$ is covered by φ at ℓ and $\mathsf{G}^{\varphi}(\bar{S}_{\ell})$ admits single stalling in $[k,\ell]$, then $\mathbb{V}[k,\ell]$ is covered by φ at ℓ.*

A new lower-bound construction. Previous lower bounds on common knowledge in the presence of failures, including the ones for CC protocols in [11], are all based on the fixed-point construction of Moses and Tuttle [13] and, for uniform consensus, on its extension by Neiger and Tuttle [16]. Their construction applies to crash and sending omissions, and it (as well as earlier proofs in [7]) depends in an essential way on the fact that faultiness in these models is forever. Somewhat in the spirit of the "Heard-of" model of [3], our generic lower bound results in Lemmas 2–4 are oblivious of the notion of process failures per-se. Lemmas 3 and 4 are the basis of our new a fixed-point construction for general monotone failure models that subsumes the construction in [13].

5 A Continuous Consensus Protocol for the (m,t) Model

The purpose of this section is to present mt-CC, a CC protocol for $\textsc{Om}(m,t)$ that is efficient and optimal in runs. The protocol makes use of instances of the UniConCon protocol (UCC) presented in [11]. This is an optimal (uniform) CC protocol for $\textsc{Om}(\infty,t)$ with the following properties: Beyond storing the relevant events in $\mathcal{E}$—which are application dependent—UCC uses $O(\log k)$ new bits of storage and performs $O(n)$ computation in each round k. In addition, in every round $k+1$ each process i sends everyone a message specifying the set $f_i[k]$ of processes it knows are faulty at time k, as well as any new information about events $e \in \mathcal{E}$ that it has obtained in the latest round. The failure information in every message requires $\min\{O(n), O(|f_i[k]|\log n)\}$ bits. Our analysis will not need further details about UCC beyond these facts.

The intuition behind our protocol is based on the following observation. The adversary's ability to cause failures in the $\textsc{Om}(m,t)$ model in a given round depends only on the failures that the culprit caused in the m-round interval ending with this round. In a precise sense, the (crucial) impact of the adversary's actions on the core at time ℓ depends only on the previous cores and the failures that occur in the last m rounds. Consequently, our strategy is to invoke a new instance of UCC in every round, and keep it running for m rounds. Each instance takes into account only failures that occur after it is invoked. For times $\ell \leq m$, the core $\hat{M}[\ell]$ coincides with the one computed by the UCC initiated at time 0. For later times $\ell > m$, the core $\hat{M}[\ell]$ is obtained by taking the union of $\hat{M}[\ell-1]$ and the core $M[\ell]_{\ell-m}$ obtained by the UCC instance invoked at time $\ell-m$.[7]

The mt-CC protocol shown in Figure 2 presents the mt-CC protocol for CC in $\textsc{Om}(m, t)$. It accepts m and $t < n$ as parameters. We denote by $M[k]_s$ the core computed by $\textsc{Ucc}_s(t)$ at time k. The message sending operations in the instances $\textsc{Ucc}_s$ are suppressed, replaced by the message μ sent by mt-CC. More formally, $\textsc{Ucc}_s$ is an instance of UCC that is invoked at time s, in which only failures that occur from round $s+1$ on are counted. It is initiated with no failure information. Incoming failure information for $\textsc{Ucc}_s$ is simulated based on the incoming messages μ, where only failures that occur in rounds $s+1\ldots,s+m$ are taken into account. Similarly, maintaining the events in the core is handled by mt-CC. Based on the structure of UCC it is straightforward to show:

Lemma 5. *Let $\varphi,\varphi' \in \textsc{Om}(m,t)$ be failure patterns, such that no process fails in the first s rounds of φ', and the same channels are blocked in both patterns from round $s+1$ on. Let r be a run of* $\textsc{Ucc}_0$ *with adversary (φ',ζ), and let r' be a run with adversary (φ,ζ) in which* $\textsc{Ucc}_s$ *is invoked at time s. Then $M_x[s+m]_s = M'_x[s+m]$ for all $x \in \mathbb{P}$.*

[7] In fact, the UCC instance computes the subset of nodes of $\mathbb{V}[0,\ell]$ that determines its core, and the core $M[\ell]_{\ell-m}$ consists of the events that occur at the nodes of this subset.

```
mt-Cc_x        % Executed by $x$, on parameters $m$ and $t$
0  $\hat{M}_x[0] \leftarrow \emptyset$
1  **invoke** an instance of $\text{Ucc}_0$
   **for** every round $k \geq 1$ **do**
2      **send** $\mu = \langle$failure-info, new-events$\rangle$ to all
3      $s \leftarrow max(0, k-m)$
4      **receive** incoming messages;
5      **simulate** a round of $\text{Ucc}_s, \ldots, \text{Ucc}_{k-1}$;
6      $\hat{M}_x[k] \leftarrow (\hat{M}_x[k-1] \cup M_x[k]_s)$
7      **invoke** an instance of $\text{Ucc}_k$
   **endfor**
```

Fig. 2. Process x's computation in mt-Cc

The failure-info component of the message μ on line 2 consists of a list of the processes j that x knows have displayed faulty behavior in the last m rounds, and for each such j the latest round number $(\text{mod } m)$ in which x knows j to have been faulty. In mt-Cc at most m instances of Ucc_k need to be active at any given time. As we show in the full paper, simple accounting mod m suffices for x to be able to construct the incoming (simulated) messages for each active instance, based on the failure-info components of incoming messages. The failure component in mt-Cc message is thus of size $\min\{O(n \log m), O(|f_i[k]| \cdot \log nm)\}$ bits, where now $f_i[k]$ is the number of processes known by i to have failed after round $k-m$. The new-events component in mt-Cc messages is the same as in a single instance of Ucc in the standard sending omissions model $\text{Om}(\infty, t)$.

While mt-Cc appears to be rather straightforward, the use of a *uniform* CC protocol to construct it, and the way in which the results of the different instances are combined is rather subtle. The real technical challenge, which brought about the new lower bound techniques in Section 4, is proving that it is optimal in all runs. We now consider the properties of mt-Cc. Our first result is an upper bound, which is proved in Section B of the appendix:

Lemma 6. *The mt-Cc(m,t) is a CC protocol for $m > t$. When $m \leq t$ it satisfies Accuracy and Consistency.*

Based on the lower bound in Section 4 we then prove that mt-Cc is optimal for $m > t$. Moreover, the results of Santoro and Widmayer can be used to show that no CC protocol exists for $m \leq t$ (this also easily follows from our lower bounds). Nevertheless, mt-Cc is optimal among the protocols that satisfy accuracy and consistency for $m \leq t$. The optimality property implies that, essentially, mt-Cc is as complete as possible for $m \leq t$.

Being based on the cores computed by instances of Ucc, the core at time ℓ in a run of mt-Cc is the set of events that take place at a particular set $T \subset \mathbb{V}[0,\ell]$. Moreover, since Ucc_s in $\text{Om}(m,t)$ is equivalent to Ucc in the standard sending omissions model (by Lemma 5) the set A has the same structure as derived in the Moses and Tuttle construction. Let c denote the maximal time in nodes of T. To prove that mt-Cc is optimal, we show that all nodes that are at and after the critical time $c = c(\varphi, \ell)$ and are not in A are covered. Formally, we apply Lemma 4 to mt-Cc in the $\text{Om}(m,t)$ model to obtain:

Lemma 7. *Let r be a run of* mt-$\textsc{Cc}(m,t)$ *with failure pattern φ, and assume that $\hat{M}[\ell]$ is generated by the set of nodes $A \subseteq \mathbb{V}[0,\ell]$. If c is the maximal time of any node in A, then $S = (\mathbb{V}[c,\ell] \setminus A)$ is covered by φ at ℓ.*

In the $\textsc{Om}(m,t)$ model, if a process fails to deliver a message in round k, it is faulty. Hence, all nodes of $\mathbb{V}[0,\ell-1]$ that do not appear in the causal past of A belong to faulty processes. We can use Lemma 7 to silence them at a reachable point using the covered nodes in the same way as in the proof on Lemma 3. Once we do this, all nodes from before time ℓ that do not appear in the view $\mathsf{G}^{\varphi}(A)$ are eliminated from the graph. Formally, we can show:

Lemma 8. *Let r be a run of* mt-$\textsc{Cc}(m,t)$ *and let ℓ and A be as in Lemma 7. Then all nodes in $\mathbb{V}[0,\ell] \setminus A$ are hidden by φ at ℓ in* $\textsc{Om}(m,t)$*.*

Lemma 8 provides the final ingredient for the optimality proof for mt-$\textsc{Cc}$: Lemma 6 guarantees that mt-$\textsc{Cc}$ solves continuous consensus. Lemma 8 states that all nodes not in the core view $\mathsf{G}^{\varphi}(A)$ of mt-$\textsc{Cc}$ are hidden given the current failure pattern, for all protocols. Theorem 2 states that an event at a hidden node cannot be common knowledge, which by Theorem 1 implies that it cannot be contained in the common core under *any* protocol whatsoever. It follows that, for each and every adversary all times ℓ, the core provided by mt-$\textsc{Cc}$ is a superset of the core that any other correct CC protocol can construct. We obtain:

Theorem 3. *Let $m > t \geq 0$, and let $\Delta = \textsc{Om}(m,t) \times \Xi$ where Ξ is independent. Then* mt-$\textsc{Cc}(m,t)$ *is optimal in all runs in Δ.*

We can also show that mt-$\textsc{Cc}$ provides optimal behavior for $m \leq t$; it is accurate, consistent and maximal. Indeed, for $t \geq m > 1$, there are runs of mt-$\textsc{Cc}$ in which nontrivial events enter the core. However, the completeness property is not achievable in some of the runs. Using Lemmas 4 and 3 we can show:

Lemma 9. *Let $0 < m \leq t$, and let $\Delta = \textsc{Om}(m,t) \times \Xi$ where Ξ is independent. Let r be a run of a CC protocol with failure pattern $\varphi \in \textsc{Om}(m,t)$ in which no more than one process fails per round in rounds $1,\ldots,\ell$. Then the core at time ℓ is necessarily empty.*

By Lemma 9, as well as from the results of Santoro and Widmayer in [18], it follows that Continuous consensus is not solvable in $\textsc{Om}(m,t)$ for $1 \leq m \leq t$. Lemma 9 implies that if $m = t = 1$ then the core is necessarily empty at all times. However, whenever $m > 1$ there are runs in which the core is *not* empty. In these cases, Lemma 6 guarantees that mt-$\textsc{Cc}$ is Accurate and Consistent. In fact, it is as close to a CC protocol as one could hope for:

Theorem 4. *For all $m \leq t$, the* mt-$\textsc{Cc}$ *protocol is optimal in all runs among the protocols that satisfy the Accuracy and Consistency properties of CC.*

6 Conclusions

Continuous consensus (CC) is a powerful primitive in synchronous distributed systems. Being an ongoing activity, the classical t-bounded failure model in which failures can only accumulate and recoveries are not considered is not satisfactory. This paper considers the CC problem in an (m,t) omission failure model. mt-Cc, an efficient protocol for CC in such models is presented, and is shown to be a optimal in all runs: It maintains the largest core at all times, in each and every run (i.e., against each behavior of the adversary). We remark that while this paper focused on optimally fast CC protocols, it is an interesting open problem how to trade speed against message complexity in CC protocols.

The lower bound proof makes essential use of the theory of knowledge in distributed systems. Using a new technique the lower bound manages to sidestep previous dependence on the stability of faultiness in order to apply to models with failures and recoveries, such as the (m,t) omission model. It gives rise to a lower-bound construction generalizing that of Moses and Tuttle [13] in the standard sending omissions model. The fact that mt-Cc is optimal in all runs proves that, in fact, the construction completely characterizes what is common knowledge (and what can appear in a CC core) in Om(m,t). The new construction applies to monotone failure models in general. It thus strictly generalizes the MT construction and applies to many other failure models, including the elusive general omissions failure model [13] in which a faulty process can fail to send *and* to receive messages. Models that bound the number of messages lost are also monotone, as are ones in which there are different reliability guarantees for different subsets of the processes (say central servers vs. plain workstations). We believe that a slight variation on this construction can be used to solve a twenty-year old open problem regarding optimum simultaneous consensus in the general omissions model. Finally, in future work we plan to show that, with slight modifications, the new lower bound proof is also applicable to topologies in which the communication network is not a complete graph.

References

1. Burns, J.E., Lynch, N.A.: The byzantine firing squad problem. Technical Report MIT/LCS/TM-275 (1985)
2. Castro, M., Liskov, B.: Proactive recovery in a Byzantine-fault-tolerant system. In: Proc. 4th OSDI: Symp. Op. Sys. Design and Implementation, pp. 273–288 (2000)
3. Charron-Bost, B., Schiper, A.: The Heard-Of Model: Unifying all Benign Failures. EPFL LSR-REPORT-2006-004 (2006)
4. Coan, B.A., Dolev, D., Dwork, C., Stockmeyer, L.J.: The distributed firing squad problem. SIAM J. Comput. 18(5), 990–1012 (1989)
5. Dolev, D., Reischuk, R., Strong, H.R.: Eventual is earlier than immediate. In: Proc. 23rd IEEE Symp. on Foundations of Computer Science, pp. 196–203 (1982)
6. Dolev, S., Rajsbaum, S.: Stability of long-lived consensus. J. Comput. Syst. Sci. 67(1), 26–45 (2003)
7. Dwork, C., Moses, Y.: Knowledge and common knowledge in a Byzantine environment: crash failures. Information and Computation 88(2), 156–186 (1990)
8. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (1995) (revised 2003)
9. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. Journal of the ACM 37(3), 549–587 (1990)

10. Merritt, M.J.: Unpublished notes on the Dolev-Strong lower bound for Byzantine Agreement (1984)
11. Mizrahi, T., Moses, Y.: Continuous consensus via common knowledge. Distributed Computing 20(5), 305–321 (2008)
12. Moses, Y., Raynal, M.: Revisiting Simultaneous Consensus with Crash Failures. Tech Report 1885, 17 pages, IRISA, Université de Rennes 1, France (2008), http://hal.inria.fr/inria-00260643/en/
13. Moses, Y., Tuttle, M.R.: Programming simultaneous actions using common knowledge. Algorithmica 3, 121–169 (1988)
14. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Synchronous condition-based consensus. Distributed Computing 18(5), 325–343 (2006)
15. Neiger, G., Bazzi, R.A.: Using knowledge to optimally achieve coordination in distributed systems. Theor. Comput. Sci. 220(1), 31–65 (1999)
16. Neiger, G., Tuttle, M.R.: Common knowledge and consistent simultaneous coordination. Distributed Computing 6(3), 181–192 (1993)
17. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM 27(2), 228–234 (1980)
18. Santoro, N., Widmayer, P.: Time is not a healer. In: Proc. 6th Symp. Theo. Asp. Comp. Sci (STACS), pp. 304–313 (1989)

A Lower Bound Proof

This section contains the proof of our central lower bound claim.

Proof of Lemma 3: Let k, ℓ, φ, ψ, S and $T = S \cup \{(j,k)\}$ satisfy the conditions of the lemma. Since S is covered by φ at ℓ, we have that $(\varphi,\ell) \approx (\varphi',\ell)$ where φ' agrees with φ on $\bar{S}_\ell$, and in which all nodes of S are correct in φ'. For ease of exposition we assume w.l.o.g. that all nodes of S are correct in φ.

Denote by φ_m the pattern that agrees with φ on $V \setminus \{(j,k)\}$, where $\varphi_m(j,k)$=$\varphi(j,k) \cup \{1,\ldots,m\}$. In particular, we vacuously have $\varphi_0 = \varphi$. Observe that $\varphi \sqsubseteq \varphi_m \sqsubseteq \psi$ holds for all m. We prove by induction on m that $(\varphi,\ell) \approx (\varphi_m,\ell)$. The claim trivially holds for $m = 0$. Let $m = h+1 > 0$ and assume inductively that the claim holds for h. Thus, $(\varphi,\ell) \approx (\varphi_h,\ell)$. By Lemma 1 it follows that $S = V[k+1,\ell]$ is covered at φ_h. Denote $w = (h+1,k+1)$. Clearly, $w \in S$. Let φ_w be a failure pattern such that $(\varphi_h,\ell) \approx (\varphi_w,\ell)$, $\mathsf{G}^{\varphi_h}(\bar{S}_\ell) = \mathsf{G}^{\varphi_w}(\bar{S}_\ell)$, and $h+1$ is shut out in rounds $k+1,\ldots,\ell$ of φ_w. Let φ'_w be a failure pattern that is obtained from φ_w by dropping the edge $\langle (j,k),(h+1,k+1)\rangle$. We first claim that $\varphi'_w \in \Phi$. Denote by $\hat{\varphi}$ pattern that agrees with φ on $\bar{T}_\ell$, in which w is shut out in round $k+1$, and $h+1$ is shut out in rounds $k+2$ through ℓ. The lemma's statement ensures that $\hat{\varphi} \in \Phi$ for every pattern $\hat{\varphi} \in \Phi$ that agrees with φ on $\bar{T}_\ell$, in which (j,k) is shut out (in round $k+1$) and exactly one node is shut out in rounds $k+2$ through ℓ. Since $\varphi'_w \sqsubseteq \hat{\varphi}$, we have by monotonicity of Φ that $\varphi'_w \in \Phi$, as claimed.

For every process $i \neq h+1$ we have that $\mathsf{G}^{\varphi_w}(i,\ell) = \mathsf{G}^{\varphi'_w}(i,\ell)$, since G^{φ_w} and $\mathsf{G}^{\varphi'_w}$ differ only on the incoming edges of $w = (h+1,k+1)$. Since $h+1$ is shut out from time $k+1$ to ℓ, the node w appears neither in $\mathsf{G}^{\varphi_w}(i,\ell)$ nor in $\mathsf{G}^{\varphi'_w}(i,\ell)$. It follows that $(\varphi_w,\ell) \approx (\varphi'_w,\ell)$. By transitivity of $\approx$ we have that $(\varphi,\ell) \approx (\varphi'_w,\ell)$, which by Lemma 1 implies that S is covered by φ'_w. The pattern that agrees with φ'_w on S in which the nodes of S are correct is φ_{h+1}. The fact that S is covered by φ'_w at ℓ implies that $(\varphi'_w,\ell) \approx$

(φ_{h+1},ℓ), and again by transitivity of $\approx$ we have that $(\varphi,\ell) \approx (\varphi_{h+1},\ell)$, completing the inductive proof. We thus obtain that $(\varphi,\ell) \approx (\varphi_n,\ell)$. Moreover, since S is covered at ℓ by φ_n, the pattern φ_u that is guaranteed with respect to φ_n by clause (a) of the definition of hidden for $u = (j,k+1)$ satisfies $\mathsf{G}^{\varphi_u}(\bar{S}_\ell) = \mathsf{G}^{\varphi}(\bar{S}_\ell)$ and has j shut out in rounds $k+1,\ldots,n$. It follows that φ_u satisfies all three conditions (a), (b) and (c) necessary to establish the first part of the definition of $T = S \cup \{(j,k)\}$ being covered by φ at time ℓ, as required.

The second part of the proof that T is covered by φ at ℓ follows the same steps. In this case, however, we define patterns $\hat{\varphi}_m(j,k) = \varphi(j,k) \setminus \{1,\ldots,m\}$ and rather than blocking edges from (j,k) in round $k+1$, we incrementally add such edges. A completely analogous inductive proof produces a pattern ψ' such that $(\varphi,\ell) \approx (\psi',\ell)$, $\mathsf{G}^{\varphi}(\bar{T}_\ell) = \mathsf{G}^{\psi'}(\bar{T}_\ell)$, and $\varphi'(j,k) = \emptyset$. By Lemma 1 we obtain that ψ' covers S at ℓ. The pattern φ' guaranteed with respect to ψ' and S satisfies that $(\varphi',\ell) \approx (\psi',\ell) \approx (\varphi,\ell)$, that $\mathsf{G}^{\varphi'}(\bar{T}_\ell) = \mathsf{G}^{\psi'}(\bar{T}_\ell) = \mathsf{G}^{\varphi}(\bar{T}_\ell)$, and that $\varphi'(j,k) = \psi'(j,k) = \emptyset$ and $\varphi'(v) = \emptyset$ for every $v \in S$. We thus have that φ' satisfies conditions (d), (e) and (f) completing the proof that $T = S \cup \{(j,k)\}$ is covered by φ at time ℓ. ■

B Upper Bound Proof

Proof of Lemma 6: In order to prove that mt-Cc is a CC protocol, we have to show that Accuracy, Consistency and Completeness hold. We note that Ucc satisfies these properties, as shown in [11].

Initially, $\hat{M}_x[0] = \emptyset$ and is thus vacuously accurate. By line 4 of mt-Cc, the core $\hat{M}_x[k+1]$ is constructed by extending $\hat{M}_x[k]$ with events from $M_x[k]_s$, computed by Ucc$_s$. Since Ucc$_s$ satisfies Accuracy, the resulting core $\hat{M}_x[k+1]$ is accurate, as required.

For Consistency, let r be the run of mt-Cc(m,t) with adversary (φ,ζ), let $k \geq 1$, and let $\hat{M}_x[k]$ denote the core computed by mt-Cc at time k in r. We proceed by induction on k. For $k=0$, by line 0 of mt-Cc we have $\hat{M}_x[k] = \hat{M}_z[k] = \emptyset$. Now assume the inductive hypothesis holds for $k-1$, i.e., that $\hat{M}_x[k-1] = \hat{M}_z[k-1]$. By line 4 of mt-Cc we have $\hat{M}_x[k] = \hat{M}_x[k-1] \cup M_x[k]_s$. By Lemma 5, we have that $M_x[k]_s = M_z[k]_s$. Since $\hat{M}_x[k-1] = \hat{M}_z[k-1]$ by the inductive hypothesis, we have that $\hat{M}_x[k-1] \cup M_x[k]_s = \hat{M}_z[k-1] \cup M_z[k]_s$, and thus $\hat{M}_x[k] = \hat{M}_z[k]$. Thus, Consistency holds for mt-Cc.

Finally, for Completeness, recall that Ucc(t) guarantees that $e \in M[s+t+1]$ for every event $e \in \mathcal{E}$ that is known to a nonfaulty process j at time s. By Lemma 5, if $m > t$ then $M_x[s+m]_s = M'_x[s+m]$ where M'_x is obtained by Ucc in a run r' with adversary (φ'_s,ζ) as defined in that lemma. Since all messages delivered in r are delivered in r', it follows that if j knows e at time s in r it knows e at j in r' as well. Since $e \in M'_x[s+t+1]$ and $M'_x[s+t+1] \subseteq M'_x[s+m]$, it follows by Lemma 5 that $e \in M_x[s+m]_s$. By line 4 of mt-Cc we have that $e \in \hat{M}_x[s+m]$, as desired. ■

No Double Discount: Condition-Based Simultaneity Yields Limited Gain

Yoram Moses[1] and Michel Raynal[2]

[1] Department of Electrical Engineering, Technion, Haifa, 32000 Israel
[2] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
moses@ee.technion.ac.il, raynal@irisa.fr

Abstract. Assuming that each process proposes a value, the *consensus* problem requires the non-faulty processes to agree on the same value that, moreover, must be one of the proposed values. Solutions to the consensus problem in synchronous systems are based on the round-based model, namely, the processes progress in synchronous rounds. The well-known worst-case lower bound for consensus in the synchronous setting is $t{+}1$ rounds, where t is an *a priori* bound on the number of failures. *Simultaneous consensus* is a variant of consensus in which the non-faulty processes are required to decide in the exact same round, in addition to the deciding on the same value. Dwork and Moses showed that, in a synchronous system prone to t process crashes, the earliest round at which a common decision can be simultaneously obtained is $(t+1)-W$ where t is a bound on the number of faulty processes and W is a non-negative integer determined by the actual failure pattern F. In the *condition-based* approach consensus the consensus requirement is relaxed by assuming that the input vectors (consisting of the proposed initial values) are restricted to belong to a predetermined set C. Initially considered as a means to achieve solvability for consensus in the asynchronous setting, condition-based consensus was shown by Mostéfaoui, Rajsbaum and Raynal to allow solutions with better worst-case behavior. They defined a hierarchy of sets of conditions $\mathcal{C}_t^t \subset \cdots \subset \mathcal{C}_t^d \subset \cdots \subset \mathcal{C}_t^0$ (where the set $\mathcal{C}_t^0$ contains the condition made up of all possible input vectors). It has been shown that $t+1-d$ is a tight lower bound on the minimal number of rounds for synchronous condition-based consensus with a condition in $\mathcal{C}_t^d$.

This paper considers condition-based simultaneous consensus in the synchronous model. It first presents a simple algorithm in which processes decide simultaneously at the end of the round $RS_{t,d,F} = (t+1) - \max(W, d)$. The paper then shows that $RS_{t,d,F}$ is a lower bound for simultaneous condition-based consensus. A consequence of the analysis is that the algorithm presented is optimal *in each and every run*, and not just in the worst case: For every choice of failure pattern by the adversary (and every input configuration), the algorithm reaches simultaneous agreement as fast as any correct algorithm could do under the same conditions. This shows that, contrary to what could be hoped, when considering condition-based consensus with simultaneous decision, we can benefit from the best of both actual worlds (either the failure world when $RS_{t,d,F} = (t{+}1){-}W$, or the condition world when $RS_{t,d,F} = d + 1$), but we cannot benefit from the sum of savings offered by both. Only the best discount applies.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 423–437, 2008.

1 Introduction

The consensus problem. Fault-tolerant systems often require a means by which processes or processors can arrive at an exact mutual agreement of some kind [14]. If the processes defining a computation have never to agree, that computation is actually made up of a set of independent computations, and consequently is not an inherently distributed computation. The agreement requirement is captured by the *consensus* problem that is one of the most important problems of fault-tolerant distributed computing. It actually occurs every time entities (usually called agents, processes -the word we use in the following-, nodes, sensors, etc.) have to agree. The consensus problem is surprisingly simple to state: Each process is assumed to propose a value, and all the processes that are not faulty have to agree/decide (termination), on the same value (agreement), that has to be one of the proposed values (validity). The failure model considered in this paper is the process crash model.

While consensus is impossible to solve in pure asynchronous systems despite even a single process crash [4] ("pure asynchronous systems" means systems in which there is no upper bound on process speed and message transfer delay), it can be solved in synchronous systems (i.e., systems where there are such upper bounds) whatever the number n of processes and the number t of process crashes ($t < n$).

An important measure for a consensus algorithm is the time it takes for the non-faulty processes to decide. As a computation in a synchronous system can be abstracted as a *sequence of rounds*, the time complexity of a synchronous consensus algorithm is measured in terms of the minimal number of rounds (R_t) that a process participates in before deciding, in the worst case scenario. It has been shown (see, e.g., in [8]) that $R_t = t + 1$. Moreover, this bound is tight: There exist algorithms where no process ever executes more than R_t rounds (e.g., [15]); these algorithms are thus optimal with respect to the worst-case bound.

Early decision. While $t + 1$ rounds are needed in the worst case scenario, the major part of the executions have few failures or are even failure-free. So, an important issue is to be able to design *early deciding* algorithms, i.e., algorithms in which the processes decide "as early as possible" in good scenarios. Let f, $0 \leq f \leq t$, be the actual number of process crashes in an execution. It has been shown that the lower bound on the number of rounds is then $R_{t,f} = \min(f+2, t+1)$ (e.g., [1,8]). As before, this bound is tight: Algorithms in which no process ever executes more than $R_{t,f}$ rounds exist (e.g., see [1,15]).

Condition-based approach and the hierarchy of synchronous conditions. The condition-based approach was originally introduced to circumvent the impossibility of solving consensus in an asynchronous system prone to crash failures [12]. It consists in restricting the set of input vectors of values that can be proposed (such a set is called a *condition*). A main result associated with this approach is the following [12]: A condition C allows to solve consensus in an asynchronous system prone to up to x process crashes iff it is x-legal (roughly speaking, a condition is x-legal if each of its input vectors contains a particular value more than x times, and the Hamming distance between two vectors from which different values can be decided is greater than x). There is a strong connection between the condition-based approach and error-correcting codes [5].

While the condition-based approach is useful to extend computability of consensus in asynchronous systems, it was also shown in [13] to allow for faster agreement in synchronous systems. More precisely, let us consider a synchronous system where up to t processes can crash, and let $\mathcal{C}_t^d$ be the set of all d-legal conditions. $\mathcal{C}_t^t \subset \cdots \subset \mathcal{C}_t^d \subset \cdots \subset \mathcal{C}_t^0$. For every condition $C \in \mathcal{C}_t^d$, it is shown in [13] that synchronous consensus can be solved uniformly for all input vectors $I \in C$ in one round when $d = t$, and in $t + 1 - d$ rounds when $0 \leq d \leq t$. It is also shown that $t + 1 - d$ is a tight worst-case lower bound.

Simultaneous decision. Consensus is a data agreement property, namely, that the processes must agree on the same value. Depending on the actual failure pattern, and the way this pattern is perceived by the processes, it is possible for several processes to decide in distinct rounds. The *simultaneous (decision)* consensus problem aims at eliminating this uncertainty. It requires that the processes decide on the same value (*data agreement*), during the very same round (*time agreement*).

Many "classical" consensus algorithms where all the processes that do not crash decide systematically at the end of the round $R_t = t + 1$ ensure simultaneous decision. Simultaneous consensus was considered in [2], where it was shown that simultaneous decisions can be performed in anywhere between 2 and $t + 1$ rounds, depending on the failure pattern. Indeed, for every failure pattern F there is a measure called the *waste*, denoted by $W(F)$, such that decisions can be performed in $RS_{t,F} = t+1-W$ rounds, and no earlier. $W(F)$ represents the number of rounds "wasted" by the adversary relative to the worst case bound of $t + 1$ rounds. Interestingly, the greatest values of W are obtained when many processes crash early. If at most one process crashes in each round, then $W = 0$ and simultaneous decision cannot be performed before the end of round $t + 1$. At first glance, this may appear counter-intuitive. Actually it is not; roughly speaking, it is a consequence of the fact that once many processes crash, the adversary's options for the remaining rounds become more restricted. Simultaneous decision requires common knowledge about proposed input values, which becomes easier to attain as the uncertainty about past and future failures is reduced. Algorithms that solve simultaneous consensus optimally—requiring precisely $t + 1 - W(F)$ for every run and each failure pattern F—are described in [2,7,9,11].

Content of the paper. This paper investigates the simultaneous decision requirement in the context of the condition-based approach. Let $C \in \mathcal{C}_t^d$ (thus, C is d-legal). On the one hand, we have from [13] that simultaneous consensus can be guaranteed in round $t + 1 - d$. On the other hand, since a solution for unrestricted simultaneous consensus is still correct under any condition C, the existing solution from [2,9,11] can be used to guarantee simultaneous agreement in $t + 1 - W$ rounds. This paper investigates the interaction between conditioning and simultaneity. For a positive result, it shows that the simultaneity requirement allows a seamless combination of the two solutions. Essentially, it is possible to simulate both types of protocols and decide (without violating agreement!) in the round in which the first of the two algorithms decides. This solves condition-based simultaneous consensus in $t+1-\max\{W, d\}$ rounds. On the negative side, it shows that, in a precise sense, this is the best possible. For natural d-legal conditions, the algorithm presented here is optimal in a very strong sense, inherited from the optimal solutions for simultaneous consensus: For each and every input vector and failure pattern, the algorithm decides as soon as any other simultaneous consensus algorithm for C would decide. So the algorithm is optimal *in every case* and not just in the *worst* case run. The main technical challenge is in the lower-bound proof, which is based on a knowledge-based analysis, using the connection between simultaneous agreement and common knowledge [2,3,6].

2 Computation Model, Conditions and Problem Specification

2.1 Computation Model

Round-based synchronous system. The system model consists of a finite set of processes, namely, $\Pi = \{p_1, \ldots, p_n\}$, that communicate and synchronize by sending and receiving messages through channels. (Sometimes we also use p and q to denote processes.) Every pair of processes is connected by a bi-directional reliable channel (which means that there is no creation, alteration, loss or duplication of messages).

The system is *round-based synchronous*. This means that each execution consists of a sequence of *rounds*. Those are identified by the successive integers $1, 2$, etc. For the processes, the current round number appears as a global variable r that they can read, and whose progress is managed by the underlying system. A round is made up of three consecutive phases:

- A send phase in which each process sends the same message to all the processes (including itself).
- A receive phase in which each process receives messages. The fundamental property of the synchronous model lies in the fact that a message sent by a process p_i to a process p_j at round r, is received by p_j at the very same round r.
- A computation phase during which each process processes the messages it received during that round and executes local computation.

Process failure model. A process is *faulty* during an execution if its behavior deviates from that prescribed by its algorithm, otherwise it is *correct*. We consider here the crash failure model, namely, a faulty process stops its execution prematurely. After it has crashed, a process does nothing. If a process crashes in the middle of a sending phase, only a subset of the messages it was supposed to send might actually be received.

As already indicated, the model parameter t ($1 \leq t < n$) denotes an upper bound on the number of processes that can crash in a run. A *failure pattern* F is a list of at most t triples $\langle q, k_q, B_q \rangle$. A triple $\langle q, k_q, B_q \rangle$ states that the process q crashes while executing the round k_q (hence, it sends no messages after round k_q), while the set B_q denotes the set of processes that do not receive the message sent by q during the round k_q.

2.2 The Condition-Based Approach

Notation. Let $\mathcal{V}$ be the set of values that can be proposed, with $|\mathcal{V}| \geq 2$. An *input configuration* is an assignment $\mathcal{I} : \Pi \to \mathcal{V}$ of an initial value $v_i \in \mathcal{V}$ to each process p_i. An *input vector* is a size n vector corresponding to an input configuration. A condition is a set of input vectors.

Let $\perp$ denote a default value such that $\perp \notin \mathcal{V}$ and $\forall a \in \mathcal{V}$, $\perp < a$. We usually denote by I an input vector (all its entries are in $\mathcal{V}$), and by J a vector that may have some entries equal to $\perp$. Such a vector J is called a *view*. The number of occurrences of a value $a \in \mathcal{V} \cup \{\perp\}$ in the vector J is denoted $\#_a(J)$. Given two vectors $I1$ and $I2$, *dist*$(I1, I2)$ denotes their Hamming distance.

Definition 1 (x-legal). [5,12] *A condition C is x-*legal *if there exists a function $\mathcal{H} : C \mapsto \mathcal{V}$ with the following properties:* (1) $\forall I \in C$*:* $\#_{\mathcal{H}(I)}(I) > x$*, and* (2) $\forall I1, I2 \in C$*:* $\mathcal{H}(I1) \neq \mathcal{H}(I2) \Rightarrow dist(I1, I2) > x$*.*

This means that the value extracted from I by $\mathcal{H}()$ appears "often enough" in I (more than x times), and two vectors from which different values are extracted by $\mathcal{H}()$ are "far enough apart" in terms of Hamming distance.

Example 1. Intuitively, a condition selects a proposed value to become the decided value, namely, the value selected from an input vector I is $\mathcal{H}(I)$. The *max-value* condition M_x is shown in in [12] to be x-legal. Using $\max(I)$ to refer to the greatest value in I, the condition M_x is defined by:

$$M_x \quad = \quad \{I : \#_{\max(I)}(I) > x\}.$$

A straightforward consequence of the definition of x-legal conditions is that the sets $\mathcal{C}_t^d$ form a strict hierarchy, as captured by:

Theorem 1. [12] *The set $\mathcal{C}_t^{x+1}$ of $(x+1)$-legal conditions strictly contains the set $\mathcal{C}_t^x$ of (x)-legal conditions. Thus, $\mathcal{C}_t^t \subset \cdots \subset \mathcal{C}_t^d \subset \cdots \subset \mathcal{C}_t^0$.*

A main result of [12] is the characterization of the largest set of conditions that allow to solve consensus in an asynchronous system:

Theorem 2. [12] *If C is x-legal then consensus can be solved under C in an asynchronous system prone to up to x process crashes. Conversely, there exists an $(x-1)$-legal condition C' for which consensus is unsolvable in the presence of x process crashes.*

2.3 The Condition-Based Simultaneous (cb-s) Consensus Problem

The problem has been informally stated in the introduction: Every process p_i *proposes* a value v_i (called its *initial value*). Assuming that the vector of proposed values belongs to a predetermined condition C, the processes have to *decide*, during the very same round, on the same value, is required to be one of the proposed values. This can be stated as a set of four properties that any algorithm solving the problem has to satisfy.

- Decision. Every correct process decides.
- Validity. A decided value is a proposed value.
- Data agreement. No two processes decide different values.
- Simultaneous decision. No two processes decide during distinct rounds.

3 An Optimal Condition-Based Simultaneous Consensus Algorithm

This section presents a simple condition-based simultaneous consensus algorithm in which the processes decide at the end of the round $RS_{t,d,F} = (t+1) - \max(W, d)$. It will be shown in Section 4 that $RS_{t,d,F}$ is the smallest number of rounds for condition-based simultaneous consensus.

The proposed algorithm is built modularly. It combines two base algorithms. One is a condition-based algorithm that, when instantiated with a d-legal condition C (i.e., $C \in \mathcal{C}_t^d$, with $0 \leq d \leq t$, and the input vector I belongs to C) directs the processes to decide simultaneously at the end of round $t+1-d$. The second is a simultaneous (non-condition-based) consensus algorithm that directs the processes to decide at the end of the round $t+1-W$. These base algorithms are first presented.

In general, obtaining a consensus algorithm based on running two consensus algorithms is a subtle matter. In our case, the fact that both guarantee simultaneous decisions simplifies this task. As a consequence, the combination of the two algorithms yields a cb-s consensus algorithm in which the processes simultaneously decide at the end of round $RS_{t,d,F} = (t+1) - \max(W, d)$.

3.1 A Simple Condition-Based Simultaneous Consensus Algorithm

It is convenient to consider partial input vectors, which specify some of the initial values, and contain $\perp$ instead of the missing values. We write $J \leq I$ if I is "more informative" than J. Namely, if (i) every location containing $\perp$ in I contains $\perp$ in J as well, and (ii) if I and J agree on all locations that are not $\perp$ in J. One property of x-legal conditions that is very useful for solving condition-based consensus is captured by the following lemma:

Lemma 1. [13] *Let C be an x-legal condition. If $I1, I2 \in C$, $\#_\perp(J) \leq x$, $J \leq I1$ and $J \leq I2$, then $\mathcal{H}(I1) = \mathcal{H}(I2)$.*

Given Lemma 1, any subset of $n-x$ values in an input vector I determine the value of $\mathcal{H}(I)$. It is thus possible to extend $\mathcal{H}$ to partial input vectors that contain any number $k \geq n-x$ values. Since fewer values suffice, it turns out that condition-based consensus admits faster algorithms in the synchronous round model than (unconditional) consensus. Indeed, Mostéfaoui et al. in [13] present a condition-based consensus algorithm for the synchronous round model that, when applied with respect to a condition in $\mathcal{C}_t^d$, guarantees that all non-faulty processes decide in the first $t+1-d$ rounds. Their algorithm, however, does not guarantee simultaneous decision.[1] The algorithm of [13] can

[1] The algorithm described in [13] works whether the input vector is in the condition or not. When the input vector I belongs to the condition C, a process decides in two rounds if $f \leq d$, and in at most $t+1-d$ rounds when $f > d$. When I is not in the condition a process decides in at most $t+1$ rounds. The condition-based protocols developed in the current paper are guaranteed to be correct only if I belongs to a condition of the proper class. They do not degrade gracefully when the input vector is arbitrary.

```
operation COND_PROPOSE($v_i, t, d, \mathcal{H}$): % code for $p_i$ %
(101)  $V_i \leftarrow [\bot, \ldots, \bot]$; $v\_cond_i \leftarrow \bot$; $v\_nocond_i \leftarrow \bot$;
(102)  when $r = 1$
(103)  begin round
(104)     send $(v_i)$ to all;
(105)     for each $v_j$ received do $V_i[j] \leftarrow v_j$ end for;
(106)     if $(\#_\bot(V_i) \leq d)$ then $v\_cond_i \leftarrow \mathcal{H}(V_i)$ end if;
(107)     $v\_nocond_i \leftarrow \max($ all the $v_j$ received during $r)$
(108)  end round;
(109)  when $r = 2, 3, \ldots$ do
(110)  begin round
(111)     send $(v\_cond_i, v\_nocond_i)$ to all;
(112)     $v\_cond_i \leftarrow \max($ all the $v\_cond_j$ received during $r)$;
(113)     $v\_nocond_i \leftarrow \max($ all the $v\_tmf_j$ received during $r)$;
(114)     if $\big(r = (t+1) - d\big)$ then
(115)        if $(v\_cond_i \neq \bot)$ then return $(v\_cond_i)$ else return $(v\_nocond_i)$ end if
(116)     end if
(117)  end round.
```

Fig. 1. COND_PROPOSE: A synchronous condition-based simultaneous consensus algorithm

be adapted so that decisions are performed simultaneously in round $t+1-d$, essentially by delaying all decisions until this round. The adapted algorithm `Cond` is presented in Figure 1 and it satisfies these properties. By the analysis in [13], we have:

Theorem 3. *The* COND_PROPOSE *algorithm presented in Figure 1 solves the condition-based consensus problem. Moreover, the processes decide in* $(t+1)-d$ *rounds.*

For completeness, COND_PROPOSE is presented in the next section, which can be skipped in a first reading.

The COND_PROPOSE algorithm. Each process p_i uses three local variables.

- V_i is an array whose aim is to contain p_i's local view of the input vector. Initialized to $[\bot, \ldots, \bot]$, it contains at most t entries equal to $\bot$ at the end of the first round (line 105).
- v_cond_i (initialized to $\bot$) is destined to contain the value $\mathcal{H}(I)$ that the condition C associates with the input vector I (line 106). As the condition C is d-legal, $\mathcal{H}(I)$ can be computed from $\mathcal{H}(V_i)$ only when the local view V_i of p_i has at most d entries equal to $\bot$. Thus, the value of $\mathcal{H}$ can be computed at this point.
- v_nocond_i (initialized to $\bot$) is destined to contain the value to be decided when no value can be decided from the condition because there are too many failures during the first round (more than d processes crash). When this occurs, a process will decide the greatest proposed value it has ever seen.

The behavior of a process p_i is simple. During the first round (lines 102–108), p_i determines its local view V_i of the input vector I, computes v_cond_i if it sees not too many failures (i.e., at most d crashes), and computes v_nocond_i in case the condition is useless because there are more than d crashes. Then, from the second round until round $t+1-d$, the processes use the flood set strategy to exchange their current values v_cond_i and v_nocond_i, and keep the greatest ones of each. At the end of the round $t+1-d$, a process p_i decides on the value in v_cond_i if that value is not $\bot$; otherwise, it decides on the value in v_nocond_i (which is then different from $\bot$).

3.2 An Optimal Algorithm for (Unconditional) Simultaneous Consensus

The second basic algorithm that we shall use is taken from our paper [9], which is in turn based on [2,7]. It optimally solves the simultaneous consensus problem: For every failure pattern F and input vector I, it decides as soon as any other algorithm can decide on (F, I). Before describing the algorithm, we need a few definitions.

Preliminary definitions. For simplicity we assume that each process sends a message to all the processes in each round. As a consenquence, process failures can be easily detected, and this detection is done as soon as possible. Moreover, for every algorithm P, a run ρ determines to a unique pair (I, F) consisting of its input vector and failure pattern; we write $\rho = P(I, F)$ in this case.

Definition 2 (Failure discovery). *With respect to a failure pattern F, the failure of a process q is discovered in round r if r is the first round in which there is a process p that (1) does not receive a round r message from q, and (2) p survives (i.e., completes without crashing) round r. Process q has detectably crashed in F by round r if its failure is discovered no later than in round r.*

Definition 3 (Waste). *Fix a failure pattern F. Define by $\boldsymbol{C}(r)$ the set of processes whose failure is discovered in F by round r. In particular, $\boldsymbol{C}(0) = 0$. We define the* Waste *inherent in F to be $W(F) = \max_{r \geq 0}(\boldsymbol{C}(r) - r)$.*

Since $\mathbf{C}(0) = 0$ and $\mathbf{C}(r) \leq t - r$, it immediately follows that $0 \leq W(F) \leq t-1$ holds for all failure patterns F.

The SIM_PROPOSE algorithm described below is a simultaneous consensus algorithm that is optimal in all runs:

Theorem 4. *[2] Let $t < n-1$. The* SIM_PROPOSE *algorithm of Figure 2 solves simultaneous consensus. In every run, it reaches is reached in round $RS_{t,F} = t+1-W$. Moreover, no simultaneous consensus algorithm can ever decide in fewer than $t+1-W(F)$ in a run with failure pattern F.*

As in the condition-based case, we now present for the sake of completeness the algorithm called SIM_PROPOSE, taken from [7,9], which will be our concrete optimal unconditional simultaneous consensus algorithm. For more details and a proof of optimality, see [9].

The SIM_PROPOSE algorithm

The Horizon notion. The algorithm is based on the notion of a *horizon* of the previous round that each process computes in every round. Very roughly speaking, the horizon is a current best estimate for when initial values will become common knowledge, so that decisions can be based on them. The horizon is updated to be the minimal value of $t+1-|f'_i(r-1)|$, where the latter term $f'_i(r-1)$ refers roughly to the number of processes that, according to i's knowledge at the end of round r, were definitely discovered by the end of round $r-1$. (For this and the finer points of SIM_PROPOSE, See [9].)

Local variables. Each process p_i manages the following local variables. Some variables are presented as belonging to an array. This is only for notational convenience, as such array variables can be implemented as simple variables.

- est_i contains, at the end of r, p_i's current estimate of the decision value. Its initial value is v_i, the value proposed by p_i.
- $f_i[r]$ denotes the set of processes from which p_i has not received a message during the round r. (So, this variable is the best current estimate that p_i can have of the processes that have crashed.)
  Let $\overline{f_i[r]} = \Pi \setminus f_i[r]$ (i.e., the set of processes from which p_i has received a round r message).
- $f'_i[r-1]$ is a value computed by p_i during the round r, but that refers to crashes that occurred up to the round $r-1$ (included), hence the notation. It is the value $\bigcup_{p_j \in \overline{f_i[r]}} f_j[r-1]$, which means that $f'_i[r-1]$ is the set of processes that were known as crashed at the end of the round $r-1$ by at least one of the processes from which p_i has received a round r message. This value is computed by p_i during the round r. As a process p_i receives its own messages, we have $f_i[r-1] \subseteq f'_i[r-1]$.
- $bh_i[r]$ represents the best (smallest) horizon value known by p_i at round r. It is p_i's best estimate of the smallest round for a simultaneous decision. Initially, $bh_i[0] = h_i(0) = t+1$.

Process behavior. Each process p_i not crashed at the beginning of r sends to all the processes a message containing its current estimate of the decision value (est_i), and the set $f_i[r-1]$ of processes it currently knows as faulty. After it has received the round r messages, p_i computes the new value of est_i and the value of $bh_i[r]$. The new value of est_i is the smallest of the estimates values it has seen so far. As far as the value of $bh_i[r]$ is concerned, we have the following.

- The computation of $bh_i[r]$ has to take into account $h_i(r)$. This is required to benefit from the fact that there is a clean round y such that $r \leq y \leq h_i(r)$. A clean round is one in which all processes hear from the exact set of processes. When this clean round will be executed, any two processes p_i and p_j that execute it will have $est_i = est_j$, and (as they will receive messages from the same set of processes) will be such that $f'_i[r-1] = f'_j[r-1]$. It follows that, we will have $h_i(y) = h_j(y)$, thereby creating correct "seeds" for determining the smallest round for a simultaneous decision. This allows the processes to determine rounds at which they can simultaneously decide.

```
algorithm SIM_PROPOSE($v_i, t$): % code for $p_i$ %
(201)  $est_i \leftarrow v_i$; $bh_i[0] \leftarrow t+1$; $f_i[0] \leftarrow \emptyset$; $decided_i \leftarrow false$;    % initialization %
(202)  when $r = 1,\ 2,\ \ldots$ do    % $r$: round number %
(203)  begin round
(204)      send $(est_i, f_i[r-1])$ to all;    % including $p_i$ itself %
(205)      let $f'_i[r-1]$ = the union of the $f_j[r-1]$ sets received during $r$;
(206)      let $f_i[r]$ = the set of proc. from which $p_i$ has not received a message during $r$;
(207)      $est_i \leftarrow \min($ all the $est_j$ received during $r)$;
(208)      let $h_i(r) = (r-1) + (t+1-|f'_i[r-1]|)$;
(209)      $bh_i[r] \leftarrow \min\big(bh_i[r-1], h_i(r)\big)$;
(210)      if $r = bh_i[r]$ then $decided_i \leftarrow true$; return $(est_i)$ end if
(211)  end round.
```

Fig. 2. SIM_PROPOSE: Optimal simultaneous consensus despite up to t crash failures

- As we are looking for the first round where a simultaneous decision is possible, $bh_i[r]$ has to be set to $\min\big(h_i(0), h_i(1), \ldots, h_i(r)\big)$, i.e., $bh_i[r] = \min\big(bh_i[r-1], h_i(r)\big)$.

Finally, according to the previous discussion, the algorithm directs a process p_i to decide at the end of the first round r that is equal to the best horizon currently known by p_i, i.e., when $r = bh_i[r]$.

The resulting algorithm is presented in Figure 2, where $h_i(r)$ (see line 208) is expressed as a function of $r-1$ to emphasize the fact that it could be computed at the end of the round $r-1$ by an external omniscient observer. The local boolean variable $decided_i$ is used only to prove the optimality of the combined algorithm (see Section 4). Its suppression does not alter the algorithm.

3.3 An Optimal Condition-Based Simultaneous Consensus Algorithm

We now show how the two simultaneous consensus algorithms presented in Figures 1 and 2 can be combined to give a correct simultaneous consensus algorithm that decides as soon as either one does. In fact, a similar combination can be done in general for simultaneous consensus algorithms. The issue of parallel execution of consensus algorithms without the simultaneity property is much more subtle is not always possible.

Running simultaneous consensus algorithms in parallel. Suppose that A and B are two algorithms for simultaneous consensus, whose decision round is determined by conditions ψ_A and ψ_B, respectively. Define $A\&B$ to be the algorithm that runs both A and B in parallel, but changes the decision rule in the following way:

- When algorithm A's decision rule (ψ_A) is satisfied, then decision is performed according to A, provided that a decision has not been performed in previous rounds.
- When algorithm B's decision rule (ψ_B) is satisfied, if A's rule is *not* satisfied, and no decision has been performed in previous rounds, then decision is performed according to B.

Thus, $A\&B$ decides as soon as the first of A and B decides. Ties are broken by accepting A's decision in case both algorithms happen to decide in the same round. As result, the agreement property, as well as all other properties of simultaneous consensus, is properly maintained.

For completeness, we describe the changes to the algorithms in Figures 1 and 2 that yield a properly combined algorithm:

1. The r-th round, $1 \leq r \leq t+1-d$, of the combined algorithm is a simple merge of the r-th round of both algorithms. This means that the message sent by p_i at round r now piggybacks v_cond_i, v_nocond_i, est_i and $f_i[r-1]$ (a closer look at the base algorithms shows that their variables v_nocond_i and est_i play the same role. Consequently, only of them has to be kept in the combined algorithm).
2. The lines 114–116 of the algorithm in Figure 1, and the line 210 of the algorithm in Figure 2 are replaced by the following lines:

```
if $(r = bh_i[r]) \ \vee \ (r = t+1-d)$ then
   if $(r = bh_i[r])$ then $decided_i \leftarrow true$; return $(est_i)$
                      else  if $(v\_cond_i \neq \bot)$ then return $(v\_cond_i)$
                                                  else  return $(v\_nocond_i)$ end if
end if.
```

We immediately obtain:

Theorem 5. *Let $t < n-1$. The algorithm obtained by the combined execution (as described in the previous items) of the algorithms described in Figures 1 and 2 solves the condition-based simultaneous consensus problem. In a run with failure pattern F, decision is reached in round $t+1-\max(W(F),d)$.*

4 Optimality: $t+1-\max(W,d)$ Is a Lower Bound

This section proves that the algorithm described in Section 3.3 (COND_SIM_PROPOSE in the following) is optimal, namely, in a synchronous system prone to up to t process crashes (with $t < n-1$), there is no deterministic algorithm that can ever solve the condition-based simultaneous consensus problem in fewer than $(t+1)-\max(W,d)$ rounds. The proof relies on a knowledge-based analysis using notions introduced in [2,7,9]. Due to space limitations, only the main lemmas and theorems are stated. The reader can consult [10] for the full details of the proof. We start with some intuition and background.

4.1 Similarity Graphs and Common Knowledge

Our lower bound is based on the well-known connection between simultaneous agreement and common knowledge [2,11]. This connection implies that it is possible to simultaneously decide on a value $v \in \mathcal{V}$ only once it becomes common knowledge that one of the initial values of the current input vector is v:

Proposition 1 ([2]). *If P solves condition-based simultaneous consensus for condition C, then whenever the nonfaulty processes decide v, it is common knowledge that v appears in the input vector.*

Given Proposition 1, if we are able to show that no value is commonly known to be an initial value in round r of a run ρ, then decision is not possible in this round. We will use this in order to prove lower bounds on when decisions can be performed in a cb-s consensus protocol. Rather than develop the logic of knowledge in detail here, we will use a simple graph-theoretic interpretation of common knowledge in our setting.

The *local state* of process p_i at the end of round r of an algorithm P is identified with the state of p_i's memory - its variables and their values.

Definition 4 (Similarity Graph). *Fix a round r. We say that runs ρ and ρ' are* indistinguishable to p after round r, *denoted by $\rho \sim_p \rho'$ if p has survived round r in both runs, and p's local state at the end of round r is the same in both runs. For a given algorithm P, we define $\mathsf{sG}(r) = (V, E)$, where V are the runs of P, and $\{\rho, \rho'\} \in E$ iff $\rho \sim_p \rho'$ holds for some process p.*

The similarity graph $\mathsf{sG}(r)$ has the property that the existence of an initial value of v is common knowledge in round r of a run ρ if and only if every run in ρ's connected component in $\mathsf{sG}(r)$ contains v at least as one of its initial values. We write $\rho \approx \rho'$ if ρ and ρ' are in the same connected component of $\mathsf{sG}(r)$. In other words, $\rho, r \approx \rho'$ holds if there is a sequence of runs and processes such that $\rho = \rho_0 \sim_{p_0} \rho_1 \sim_{p_1} \cdots \sim_{k-1} \rho_k = \rho'$.

A well-known corollary of Proposition 1 is captured by:

Lemma 2 ([2]). *Fix r and let $\rho = P(I, F)$ and $\rho' = P(I_{\bar{v}}, F')$ be runs of a deterministic algorithm P that satisfies the requirements of simultaneous consensus. If v does not appear in $I_{\bar{v}}$ and $\rho \overset{r}{\approx} \rho'$ then If some non-faulty process decides on value v in round r of ρ and $\rho \approx \rho'$, then the nonfaulty processes cannot decide v in round r of ρ.*

We say that a round r is *premature* in a run $\rho = P(I, F)$ if $r < t + 1 - W(F)$. It was shown in [2] showed that before round $t + 1 - W$, the only fact about failures that is common knowledge is that $r < t + 1 - W$. It is not common knowledge that even one failure has occurred. More formally,

Lemma 3 ([2,9]). *Let P be a deterministic protocol, and denote $\rho = P(I, F)$ and $\rho' = P(I, F')$. If round r is premature in both F and F', then $\rho \overset{r}{\approx} \rho'$.*

The proof of this lemma in [9] depends only on the fact that the set of runs in $\mathsf{sG}(r)$ contains all runs $\hat{\rho} = P(I, \hat{F})$ with failure patterns $\hat{F}$ containing no more than t crashes. It thus applies to the condition-based case, and indeed will serve us in the proof of optimality. Before we complete the proof, we need to revisit conditions.

4.2 Properties of Conditions

Consider a condition $C = \{0^n, 1^n\} \in \mathcal{C}_t^{n-1}$ containing only two extreme initial configurations: All zeros and all ones. Clearly, consensus can be solved for C with no rounds of communication. Observe that C is x-legal for every $0 \leq x \leq n$. It follows that we have no hope of basing a nontrivial lower-bound on the notion of x-legality; the property of being x-legal is useful mainly for proving upper bounds as demonstrated in Theorem 3. The property of being x-legal imposes a requirement that input vectors need to be sufficiently distant from each other. We now define a property of conditions called k-coverable, which goes in the opposite direction and stipulates that the input vectors in a condition be sufficiently Hamming close.

Definition 5 **(k-graph of C).** *Given a condition C and a natural number k, the* k-graph *over C is $\mathcal{G}_k[C] = (C, E_k)$ where $E_k = \{\{I, I'\} : \mathit{dist}(I, I') \leq k\}$.*

Thus, two vectors of C are neighbors in $\mathcal{G}_k[C]$ exactly if they disagree on the values of at most k processes. We now use the k-graph to define a closeness property on conditions:

Definition 6 **(y-coverability).** *The condition C is* k-coverable *if for every value $v \in \mathcal{V}$ and input vector $I \in C$ there is an input vector $I_{\bar{v}} \in C$ such that (i) v does not appear in $I_{\bar{v}}$, and (ii) I and $I_{\bar{v}}$ are in the same connected component of $\mathcal{G}_k[C]$.*

A natural question is how the properties of being x-legal and of being k-coverable are related. It is straightforward to show:

Lemma 4. *If C is x-legal then it is not x-coverable.*

Definition 7 **(x-tight).** *A condition C is* x-tight *if it is both x-legal and $(x+1)$-coverable.*

It has been shown by [13] that the max-value condition $M_x = \{I : \#_{\max(I)}(I) > x\}$ is x-legal. In fact, we can extend this result to show:

Lemma 5. *The* max-value *condition M_x is x-tight.*

By Lemma 5, M_x is x-tight, for every value of x. It follows that every class in the $\mathcal{C}_t^d$ hierarchy contains a d-tight condition. In the sequel, we shall prove a lower bound using the y-coverability property, and obtain results that are optimal in all runs for conditions that are x-tight.

4.3 Proving the Bounds

The crux of our lower bound proof is based on the following lemma, whose proof will be outlined and which can be found in [10].

Lemma 6. *Fix a protocol P and let $\rho = P(I, F)$ and $\rho' = P(I', F')$ be two runs with input configurations I, I', respectively. Let $r \geq 0$ and assume that r is premature in both ρ and ρ'. Denote the Hamming distance $\mathit{dist}(I, I') = k$. If $k \leq t + 1 - r$, then $\rho \overset{r}{\approx} \rho'$.*

Proof. The fact that $\mathit{dist}(I, I') = k$ implies that I and I' differ on a set of values $v_{i_1}, \ldots, v_{i_k}$. Consider the failure pattern $\hat{F}$ in which $p_{i_1}, \ldots, p_{i_k}$ are crashed and silent from the outset, while no other processes fail in $\hat{F}$. Observe that $W(\hat{F}) = k - 1$, and $t + 1 - Waste(\hat{F}) = t + 2 - k$. The assumption that $k \leq t + 1 - r$ implies that

$r \leq t+1-k < t+2-k = t+1-W(\hat{F})$. Thus, r is premature in both $\rho_1 = P(I, \hat{F})$ and in $\rho_2 = P(I', \hat{F})$. Observe that every process has the same local state at the end of all rounds (and in particular at the end of round r) of both ρ_1 and ρ_2. Since $t < n-1$ at least one of the non-crashed processes is non-faulty, and so $\rho_1 \overset{r}{\approx} \rho_2$. Moreover, By Lemma 3 we have that $\rho \overset{r}{\approx} \rho_1$ and that $\rho_2 \overset{r}{\approx} \rho'$. By transitivity of $\overset{r}{\approx}$ we obtain that $\rho \overset{r}{\approx} \rho'$, as desired. $\Box_{Lemma\ 6}$

Given Lemma 6, we can now conclude that decision is impossible any faster than is obtained in our "optimal" algorithm:

Lemma 7. *Fix a protocol P and a y-coverable condition C, let $\rho = P(I, F)$ be a run of P and let $r < \min\{t+1-W(F), y+1\}$. Then no value v can be decided on at the end of round r in ρ.*

Proof. Let P, C, y, $\rho = P(I, F)$ and r satisfy the conditions of the lemma. Assume by way of contradiction that v is common knowledge at the end of round r. Since C is y-coverable, there is a vector $I_{\bar{v}} \in C$ such that v does not appear in $I_{\bar{v}}$, while I and $I_{\bar{v}}$ are in the same connected component of $\mathcal{G}_y[C]$. Thus, there is a finite path $I = I^1, I^2, \ldots I^k = I_{\bar{v}}$ in $\mathcal{G}_y[C]$ connecting I with $I_{\bar{v}}$. Define $\rho^j = P(I^j, F)$ for $j = 1, \ldots, k$. We prove by induction on $j \leq k$ that ρ and ρ^j are in the same connected component of $\mathsf{sG}(r)$. The case $j = 1$ is trivial since $\rho^j = \rho$. The inductive step follows immediately by Lemma 6, since all runs in the sequence have the same failure pattern F, which has a fixed waste. Since $\rho^k = P(I^k, F)$ does not have an initial value of v, it follows by Lemma 2 that v cannot be decided on at the end of round r of ρ. The claim follows. $\Box_{Lemma\ 7}$

We now immediately conclude:

Theorem 6. *For every x-tight condition C, the* COND_SIM_PROPOSE *algorithm is optimal for cb-s consensus in every run. I. e., for every cb-s consensus protocol P for C, every $I \in C$ and every failure pattern F, if $\rho = P(I, F)$ decides in round r, then* COND_SIM_PROPOSE *decides on (I, F) either in round r or in an earlier round.*

5 Conclusion

This paper focused on simultaneous decision in the condition-based consensus setting. It has presented two results. The first is a condition-based consensus algorithm in which processes decide simultaneously at the end of the round $RS_{t,d,F} = (t+1) - \max(W, d)$ where $W \geq 0$ is a value that depends on the actual failure pattern, and d depends on the position of the condition C (the algorithm is instantiated with) in the hierarchy of legal conditions. The second result is a proof that $RS_{t,d,F}$ is a lower bound on the number of rounds of the simultaneous condition-based consensus problem. This bound shows that we can benefit from the best world provided by the actual run (failure world when $RS_{t,d,F} = (t+1) - W$ or condition world when $RS_{t,d,F} = (t+1) - d$), but that the two benefits do not compound. There is no double discount.

Acknowledgements. We thank the anonymous referees of the PC for comments that improved the presentation of this paper.

References

1. Dolev, D., Reischuk, R., Strong, R.: Early Stopping in Byzantine Agreement. Journal of the ACM 37(4), 720–741 (1990)
2. Dwork, C., Moses, Y.: Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures. Information and Computation 88(2), 156–186 (1990)
3. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (2003)
4. Fischer, M.J., Lynch, N., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
5. Friedman, R., Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Asynchronous Agreement and its Relation with Error-Correcting Codes. IEEE Trans. on Computers 56(7), 865–876 (2007)
6. Halpern, J.Y., Moses, Y.: Knowledge and Common Knowledge in a Distributed Environment. Journal of the ACM 37(3), 549–587 (1990)
7. Mizrahi, T., Moses, Y.: Continuous Consensus via Common Knowledge. Distributed Computing 20(5), 305–321 (2008)
8. Moses, Y., Rajsbaum, S.: A Layered Analysis of Consensus. SIAM Journal of Computing 31(4), 989–1021 (2002)
9. Moses, Y., Raynal, M.: Revisiting Simultaneous Consensus with Crash Failures. Tech Report 1885, IRISA, Université de Rennes 1, France, 17 pages (2008)
10. Moses, Y., Raynal, M.: No Double Discount: Condition-based Simultaneity Yields Limited Gain. Tech Report 1898, IRISA, Université de Rennes 1, France, 20 pages (2008)
11. Moses, Y., Tuttle, M.R.: Programming Simultaneous Actions Using Common Knowledge. Algorithmica 3, 121–169 (1988)
12. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. Journal of the ACM 50(6), 922–954 (2003)
13. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Synchronous Condition-Based Consensus. Dist. Computing 18(5), 325–343 (2006)
14. Pease, L., Shostak, R., Lamport, L.: Reaching Agreement in Presence of Faults. Journal of the ACM 27(2), 228–234 (1980)
15. Raynal, M.: Consensus in Synchronous Systems: a Concise Guided Tour. In: roc. 9th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC 2002), pp. 221–228. IEEE Computer Press, Los Alamitos (2002)

Bosco: One-Step Byzantine Asynchronous Consensus*

Yee Jiun Song and Robbert van Renesse

Cornell University, Ithaca, NY 14850, USA

Abstract. Asynchronous Byzantine consensus algorithms are an important primitive for building Byzantine fault-tolerant systems. Algorithms for Byzantine consensus typically require at least two communication steps for decision; in many systems, this imposes a significant performance overhead. In this paper, we show that it is possible to design Byzantine fault-tolerant consensus algorithms that decide in one message latency under contention-free scenarios and still provide strong consistency guarantees when contention occurs. We define two variants of one-step asynchronous Byzantine consensus and show a lower bound on the number of processors needed for each. We present a Byzantine consensus algorithm, Bosco, for asynchronous networks that meets these bounds, even in the face of a strong network adversary.

1 Introduction

Informally, the consensus problem is the task of getting a set of processors to agree on a common value. This simple primitive can be used to implement atomic broadcast, replicated state machines, and view synchrony, thus making consensus an important building block in distributed systems.

Many variants of the consensus problem have been proposed. The differences between them lie mainly in the failure assumptions and the synchronicity assumptions. In this paper, we are concerned with Byzantine consensus in an asynchronous environment, i.e., faulty processors can behave in an arbitrary manner and there are no assumptions about the relative speed of processors nor about the timely delivery of messages.

Consensus algorithms allow processors to converge on a value by exchanging messages. Previous results have shown that algorithms that solve asynchronous Byzantine consensus must have correct executions that require at least two communication steps even in the absence of faults [1], where a single communication step is defined as a period of time where each processor can i) send messages; ii) receive messages; and iii) do local computations, in that order. However, this

* The authors were supported by AFRL award FA8750-06-2-0060 (CASTOR), NSF award 0424422 (TRUST), AFOSR award FA9550-06-1-0244 (AF-TRUST), DHS award 2006-CS-001-000001 (I3P), as well as by ISF, ISOC, CCR, and Intel Corporation. The views and conclusions herein are those of the authors.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 438–450, 2008.

does not mean that such algorithms must *always* take two or more communication steps. We show that when there is no contention, it is possible for processors to decide a value in one communication step.

One-step decisions can improve performance for applications where contention is rare. Consider a replicated state machine: if a client broadcasts its operation to all machines, and there is no contention with other clients, then all correct machines propose the same operation and can respond to the client immediately. Thus an operation completes in just two message latencies, the same as for a Remote Procedure Call to an unreplicated service.

Previously such *one-step* asynchronous consensus algorithms have been proposed for crash failure assumptions [2,3,4,5,6,7]; Friedman et al. proposed a common coin-based one-step consensus algorithm that tolerates Byzantine failures and terminates with probability 1 but requires that the network scheduler has no knowledge of the common coin oracle [8]. In this paper, we consider one-step algorithms for Byzantine asynchronous consensus in the presence of a strong network adversary. We define two different notions of one-step Byzantine asynchronous algorithms and prove a lower bound for the number of processors that are required for each. Next we show that the lower bounds are tight by extending the work presented in [2] to handle Byzantine failures, resulting in Bosco, an algorithm that meets these bounds.

The rest of the paper is organized as follows: Section 2 defines the model and the Byzantine consensus problem; Section 3 proves lower bounds for the two versions of one-step Byzantine consensus; Section 4 describes Bosco, a one-step consensus algorithm; Section 5 discusses some properties of Bosco; Section 6 presents a brief survey of some related work; finally, Section 7 concludes.

2 The Byzantine Consensus Problem

The Byzantine consensus problem was first posed in [9], albeit for a synchronous environment. In this paper we focus on an asynchronous environment.

In this problem, there is a set of n processors $P = \{p, q, ...\}$ each of which have an initial value, 0 or 1. An unknown subset T of P contains *faulty* processors. These faulty processors may exhibit arbitrary (*aka* Byzantine) behavior, and may collude maliciously. Processors in $P - T$ are *correct* and behave according to some protocol. Processors communicate with each other by sending messages via a network. The network is assumed to be fully asynchronous but reliable, that is, messages may be arbitrarily delayed but between two correct processors, will be eventually be delivered. Links between processors are private so a Byzantine processor cannot forge a message from a correct processor.

In addition, we assume a *strong network adversary*. By this, we mean that the network is controlled by an adversary that, with full knowledge of the contents of messages, may choose to arbitrarily delay messages as long as between any two correct processes, messages are eventually delivered.

The goal of a Byzantine consensus protocol is to allow all correct processors to eventually *decide* some value. Specifically, a protocol that solves Byzantine consensus must satisfy:

Definition 1. *Agreement. If two correct processors decide, then they decide the same value. Also, if a correct processor decides more than once, it decides the same value each time.*

Definition 2. *Unanimity. If all correct processors have the same initial value v, then a correct processor that decides must decide v.*

Definition 3. *Validity. If a correct processor decides v, then v was the initial value of some processor.*

Definition 4. *Termination. All correct processors must eventually decide.*

Note that algorithms that satisfy all of the above requirements are not possible in asynchronous environments when even a single crash failure must be tolerated [10]. In practice, algorithms circumvent this limitation by assuming some limitation in the extent of asynchrony in the system, or by relaxing the Termination property to a probabilistic one where all correct processors terminate with probability 1.

Unanimity requires that the outcome be predetermined when the initial values of all correct processors are unanimous. A one-step algorithm takes advantage of such favorable initial conditions to allow correct processors to decide in one communication step.

We define two notions of one-step protocols:

Definition 5. *Strongly one-step. If all correct processors have the same initial value v, a* strongly one-step *Byzantine consensus algorithm allows all correct processors to decide v in one communication step.*

Definition 6. *Weakly one-step. If there are no faulty processors in the system and all processors have the same initial value v, a* weakly one-step *Byzantine consensus algorithm allows all correct processors to decide v in one communication step.*

While both can decide in one step, strongly one-step algorithms make fewer assumptions about the required conditions and in particular cannot be slowed down by Byzantine failures when all correct processors have the same initial value. Strongly one-step algorithms optimize for the case where some processors may be faulty, but there is no contention among correct processors, and weakly one-step algorithms optimize for cases that are both contention-free *and* failure-free.

3 Lower Bounds

We show that a Byzantine consensus algorithm that tolerates t Byzantine failures among n processors requires $n > 7t$ to be strongly one-step and $n > 5t$ to be weakly one-step.[1] These results are for the best case scenario in which each correct processor broadcasts its initial value to all other processors in the first communication step and thus they hold for any algorithm.

3.1 Lower Bound for Strongly One-Step Byzantine Consensus

Lemma 1. *A strongly one-step Byzantine consensus algorithm must allow a correct processor to decide v after receiving the same initial value v from $n - 2t$ processors.*

Proof. Assume otherwise, that there exists a run in which a strongly one-step Byzantine algorithm $\mathcal{A}$ does not allow a correct processor p to decide v after receiving the same initial value v from $n - 2t$ processors. Since $\mathcal{A}$ is a strongly one-step algorithm, the fact that processor p does not decide after the first round implies that some correct processor q has an initial value v', $v' \neq v$. Now consider a second run, in which all correct processors *do* have the same initial value v. Without blocking, p can wait for messages from at most $n - t$ processors. Among these, t can be Byzantine and send arbitrary initial values. This means that processor p is only guaranteed to receive $n - 2t$ messages indicating that $n - 2t$ processors have the initial value v. Given that $\mathcal{A}$ is a strongly one-step algorithm, p must decide v at this point. However, from the point of view of p, this second run is indistinguishable from the first run. This is a contradiction. □

Theorem 1. *Any strongly one-step Byzantine consensus protocol that tolerates t failures requires at least $7t + 1$ processors.*

Proof. Assume that there exists a strongly one-step Byzantine consensus algorithm $\mathcal{A}$ that tolerates up to t Byzantine faults and requires only $7t$ processors. We divide the processors into three groups: G_0 and G_1 each contain $3t$ processors, of which the correct processors have initial values 0 and 1 respectively; G_* contain the remaining t processors.

Now consider the following configurations $\mathcal{C}_0$ and $\mathcal{C}_1$. In $\mathcal{C}_0$, t of the processors in G_1 are Byzantine, and processors in G_* have the initial value 0. Assume that Byzantine processors act as if they are correct processors with initial value 0 when communicating with processors in G_*, and initial value 1 when communicating with processors not in G_*. Now consider that a correct processor $p_0 \in G_*$ collects messages from $n - t$ processors in the first communication step. Given that the network adversary controls the order of message delivery, p_0 can be made to receive messages from all processors in G_0 and G_*, and the t Byzantine processors in G_1. p_0 thus receives $n - 2t$ messages indicating that the $n - 2t$ senders have initial value 0. By Lemma 1, p_0 must decide 0 after that first communication step. In order to satisfy Agreement, $\mathcal{A}$ must ensure that any correct processor that ever decides in $\mathcal{C}_0$ decides 0. We say that $\mathcal{C}_0$ is 0-*valent.*

In $\mathcal{C}_1$, t of the processors in G_0 are Byzantine, and processors in G_* have the initial value 1. In addition, Byzantine processors act as if they are correct

[1] These results are for threshold quorum systems, but may be generalized to use arbitrary quorum systems.

processors with initial value 1 when communicating with processors from G_* and initial value 0 when communicating with processors not in G_*. A correct processor $p_1 \in G_*$ collects messages from $n - t$ in the first communication step. Suppose that the network adversary chooses to deliver messages from G_1 and G_*, as well as from the t Byzantine processors. Now p_1 collects $n - 2t$ messages indicating that $n - 2t$ senders have initial value 1. By Lemma 1, p_1 must decide 1 after the first communication step. In order to satisfy Agreement, $\mathcal{A}$ must ensure that any correct processor that ever decides in $\mathcal{C}_1$ decides 1. We say that $\mathcal{C}_1$ is 1-*valent*.

Further assume that for both configurations, messages from any processor in G_* to any processor not in G_* are arbitrarily delayed such that in any asynchronous round, when a processor that is not in G_* awaits $n - t$ messages, it receives messages from every processor that is not in G_*. Now, any correct process $q_0 \notin G_*$ executing $\mathcal{A}$ in $\mathcal{C}_0$ will be communicating with $3t$ processors that behave as if they are correct processors with initial value 0 and $3t$ processors that behave as if they are correct processors with initial value 1. As we have shown above, $\mathcal{C}_0$ is a 0-valent configuration, so $\mathcal{A}$ must ensure that q_0 decides 0, if it ever decides. Similarly, a correct processor $q_1 \notin G_*$ executing $\mathcal{A}$ in $\mathcal{C}_1$ will also be communicating with $3t$ processors that behave as if they are correct processors with initial value 0 and $3t$ processors that behave as if they are correct processors with initial value 1. However, since we have shown that $\mathcal{C}_1$ is a 1-valent configuration, $\mathcal{A}$ must ensure that q_1 decides 1, even though it sees exactly the same inputs as q_0. This is a contradiction. □

3.2 Lower Bound for Weakly One-Step Byzantine Consensus

We now show the corresponding lower bound for weakly one-step algorithms. The lower bound for weakly one-step algorithms happens to be identical to that for two-step algorithms. The bound for two-step algorithms was shown in [11]. We show a corresponding bound for weakly one-step algorithm for completeness, but note that this is not a new result.

We weaken the requirement on Lemma 1 as follows:

Lemma 2. *A weakly one-step Byzantine consensus algorithm must allow a processor to decide v after learning that $n - t$ processors have the same initial value v.*

Proof. A processor can only wait for messages from $n - t$ processors without risking having to wait indefinitely. Since a weakly one-step Byzantine consensus algorithm must decide in one communication step if all correct processors have the same initial value and there are no Byzantine processors, it must decide if all of the $n - t$ messages claim the same initial value. □

Theorem 2. *A weakly one-step Byzantine consensus protocol that tolerates t failures requires at least $5t + 1$ processors.*

Proof. We provide only a sketch of the proof since it is similar to that of Theorem 1. Proof by contradiction. Assume that a Byzantine consensus algorithm

$\mathcal{A}$ is weakly one-step and requires only $5t$ processors. We divide the $5t$ processors into three groups, G_0, G_1, and G_*, containing $2t$, $2t$, and t processors respectively. All correct processors in G_0 have the initial value 0 and all correct processors in G_1 have the initial value 1.

As in the proof of Theorem 1, we construct two configurations $\mathcal{C}_0$ and $\mathcal{C}_1$. In $\mathcal{C}_0$, processors in G_* have the initial value 0 and t processors in G_1 are Byzantine. Correspondingly, in $\mathcal{C}_1$, processors in G_* have the initial value 1 and t processors in G_0 are Byzantine. These Byzantine processors behave as they do in the proof of Theorem 1. It is thus possible for processors in G_* to decide 0 and 1 in $\mathcal{C}_0$ and $\mathcal{C}_1$ respectively. Therefore, correct processors in G_0 and G_1 must not decide any value other than 0 and 1 respectively. However, if all messages from any processor in G_* to any processor not in G_* are delayed, then correct processors in $\mathcal{C}_0$ and $\mathcal{C}_1$ see exactly the same inputs. This is a contradiction. □

4 Bosco

We now present Bosco (**B**yzantine **O**ne-**S**tep **CO**nsensus), an algorithm that meets the bounds presented in the previous section. To the best of our knowledge, Bosco is the first strongly one-step algorithm that solves asynchronous Byzantine consensus with optimal resilience. The idea behind Bosco is simple, and resembles the one presented in [2]. We simply extend the results of [2] to handle Byzantine failures. The Bosco algorithm is shown in Algorithm 1.

Algorithm 1. Bosco: a one-step asynchronous Byzantine consensus algorithm

```
Input: $v_p$
broadcast $\langle$VOTE, $v_p\rangle$ to all processors
wait until $n - t$ VOTE messages have been received
if more than $\frac{n+3t}{2}$ VOTE messages contain the same value $v$ then
    DECIDE($v$)
if more than $\frac{n-t}{2}$ VOTE messages contain the same value $v$,
  and there is only one such value $v$ then
    $v_p \leftarrow v$
Underlying-Consensus($v_p$)
```

Bosco is an asynchronous Byzantine consensus algorithm that satisfies Agreement, Unanimity, Validity, and Termination. Bosco requires $n > 3t$, where n is the number of processors in the system, and t is the maximum number of Byzantine failures that can be tolerated, in order to provide these correctness properties. In addition, Bosco is weakly one-step when $n > 5t$ and strongly one-step when $n > 7t$.

The main idea behind Bosco is that if all processors have the same initial value, then given enough processors in the system, a correct processor is able to observe sufficient information to safely decide in the first communication round. Additional mechanisms ensure that if such an early decision ever happens, all correct processors *must* either i) early decide the same value; or ii) set their local estimates to the value that has been decided.

When the algorithm starts, each processor p receives an input value v_p, that is the value that the processor is trying to get decided and the value that it will use for its local estimate. Each processor broadcasts this initial value in a `VOTE` message, and then waits for `VOTE` messages from $n - t$ processors (likely including itself). Since at most t processors can fail, votes from $n - t$ processors will eventually be delivered to each correct processor.

Among the votes that are collected, each processor checks two thresholds: if more than $\frac{n+3t}{2}$ of the votes are for some value v, then a processor decides v; if more than $\frac{n-t}{2}$ of the votes are for some value v, then a processor sets its local estimate to v. Each processor then invokes `Underlying-Consensus`, a protocol that solves asynchronous Byzantine consensus (satisfies Agreement, Unanimity, Validity, and Termination), but is not necessarily one-step.

We first prove that Bosco satisfies Agreement, Unanimity, Validity, and Termination, when $n > 3t$.

Lemma 3. *If two correct processors p and q decide values v and v' in line 4, then $v = v'$.*

Proof. Assume otherwise, that two correct processors p and q decide values v and v' in line 4 such that $v \neq v'$. p and q must have collected more than $\frac{n+3t}{2}$ votes for v and v' each. Since there are only n processors in the system, these two sets of votes share more than $\frac{3t}{2}$ common senders. Given that only t of these senders can be Byzantine, more of $\frac{t}{2}$ of these senders are correct processors. Since a correct processor must send the same vote to all processors (in line 1), $v = v'$. This is a contradiction. □

Lemma 4. *If a correct processor p decides a value v in line 4, then any correct processor q must set its local estimate to v in line 6.*

Proof. Assume otherwise, that a correct processor p decides a value v in line 4, and a correct processor q does not set its local estimate to v in line 6. Since processor p decides in line 4, it must have collected more than $\frac{n+3t}{2}$ votes for v in line 2. Since processor q does not set its local estimate to v in line 6, it must have collected no more than $\frac{n-t}{2}$ votes for v, or collected more than $\frac{n-t}{2}$ votes for some value v', $v' \neq v$. For the first case, consider that since there are only n processors in the system, processor q must have collected votes from at least $n - 2t$ of the senders that processor p collected from. Among these, more than $\frac{n+t}{2}$ sent a vote for v to q. Since at most t of these processors can be Byzantine, processor q must have received more than $\frac{n-t}{2}$ votes for v. This is a contradiction. For the second case, if q collects more than $\frac{n-t}{2}$ votes for some value v', $v' \neq v$, then more than t of these senders must be among those that sent a vote for v to processor q. This is a contradiction, since, no more than t of the processors in the system can be Byzantine. □

Theorem 3. *Bosco satisfies Agreement.*

Proof. There are two cases to consider. In the first case, no processor collects sufficient votes containing the same value to decide in line 4. This means that all decisions occur in `Underlying-Consensus`. Since `Underlying-Consensus` satisfies Agreement, Bosco satisfies Agreement. In the second case, some correct processor p decides some value v in line 4. By Lemma 3, any other processor that decides in line 4 must decide the same value. By Lemma 4, all correct processors must change their local estimates to v in line 6. Therefore, all correct processors will invoke `Underlying-Consensus` with the value v. Since `Underlying-Consensus` satisfies Unanimity, all correct processors that decide in `Underlying-Consensus` must also decide v. □

Theorem 4. *Bosco satisfies Unanimity.*

Proof. Proof by contradiction. Suppose a processor p decides v', but all correct processors have the same initial value v, $v' \neq v$. Since only t Byzantine processors can broadcast vote messages that contain $v \neq v'$, no correct processor can collect sufficient votes to either decide in line 4 or to set its local estimate in line 6. Therefore, in order for a processor to decide v, `Underlying-Consensus` must allow correct processors to decide v even though all correct processors start `Underlying-Consensus` with the initial value v'. This is a contradiction since `Underlying-Consensus` satisfies Unanimity. □

Theorem 5. *Bosco satisfies Validity.*

Proof. If a processor decides v in line 4, more than $\frac{n+3t}{2}$ processors voted v and more than $\frac{n+t}{2}$ of these processors are correct and had initial value v. Similarly, if a processor sets its local estimate to v in line 6, more than $\frac{n-t}{2}$ processors voted v and more than $\frac{n-3t}{2}$ of these processors are correct and had initial value v. Combined with the fact that `Underlying-Consensus` satisfies Validity, Bosco satisfies Validity. □

Note that satisfying Validity in general in a consensus protocol is non-trivial, particularly if the range of initial values is large. A thorough examination of the hardness of satisfying Validity is beyond the scope of this paper; we simply assume that `Underlying-Consensus` satisfies Validity for the range of initial values that it allows.

Theorem 6. *Bosco satisfies Termination.*

Proof. Since each processor awaits messages from $n - t$ processors in line 2, and there can only be t failures, line 2 is guaranteed not to block forever. Each processor will therefore invoke the underlying consensus protocol at some point. Therefore, Bosco inherits the Termination property of `Underlying-Consensus`. □

Next, we show that Bosco offers strongly and weakly one-step properties when $n > 7t$ and $n > 5t$ respectively.

Theorem 7. *Bosco is Strongly One-Step if $n > 7t$.*

Proof. Assume that all correct processors have the same initial value v. Now consider any correct processor that collects $n - t$ votes in line 2. At most t of these votes can be from Byzantine processors and contain values other than v. Therefore, all correct processors must obtain at least $n - 2t$ votes for v. Since $n > 7t$, $2n - 4t > n + 3t$. This means that $n - 2t > \frac{n+3t}{2}$. Therefore, all correct processors will collect sufficient votes and decide in line 4. □

Theorem 8. *Bosco is Weakly One-Step if $n > 5t$.*

Proof. Assume that there are no failures in the system and that all processors have the same initial value v. Then any correct processor must collect $n - t$ votes that contain v in line 2. Given that $n > 5t$, $2n - 2t > n + 3t$. This means that $n - t > \frac{n+3t}{2}$. Therefore, all correct processors will collect sufficient votes and decide in line 4. □

5 Discussion

One important feature of Bosco, from which it draws its simplicity, is its dependence on an underlying consensus protocol that it invokes as a subroutine. This allows the specification of Bosco to be free of complicated mechanisms typically found in consensus protocols to ensure correctness. While it is clear that any Byzantine fault-tolerant consensus protocol that provides Agreement, Unanimity, Validity, and Termination can be used for the subroutine in Bosco, the FLP impossibility result [10] states that such a protocol cannot actually exist! Two common approaches have been used to sidestep the FLP result: assuming partial synchrony or relaxing the termination property to a probabilistic termination property. Thankfully, such algorithms can be used as subroutines to Bosco, resulting in one-step algorithms that either require partial synchrony assumptions, or provide probabilistic termination properties (or both). An example of an algorithm that can be used as a subroutine in Bosco is the Ben-Or algorithm [12]. Algorithms that do not provide validity, such as PBFT [13], cannot be used by Bosco.

While abstracting away the underlying consensus protocol simplifies the specification and correctness proof of Bosco, for practical purposes it may be advantageous to unroll the subroutine. This potentially allows piggybacking of messages and improves the efficiency of implementations. As an example, Algorithm 2 shows RS-Bosco, a randomized strongly one-step version of Bosco which does not depend on any underlying consensus protocol. RS-Bosco is strongly one-step and requires that $n > 7t$. It does not satisfy Termination as defined in section 2, but instead provides Probabilistic Termination:

Definition 7. *Probabilistic Termination. All correct processors decide with probability 1.*

Algorithm 2. RS-Bosco: a randomized strongly one-step asynchronous Byzantine consensus algorithm

```
Initialization
    $x_p \leftarrow v_p$
    $r_p \leftarrow 0$
Round $r_p$
    Broadcast $\langle \texttt{VOTE}, r_p, x_p \rangle$ to all processors
    Collect $n-t$ $\langle \texttt{VOTE}, r_p, * \rangle$ messages
    if more than $\frac{n+3t}{2}$ VOTE msgs contain $v$ then
        DECIDE($v$)
    if more than $\frac{n-t}{2}$ VOTE msgs contain $v$ then
        Broadcast $\langle \texttt{CANDIDATE}, r_p, v \rangle$
    else
        Broadcast $\langle \texttt{CANDIDATE}, r_p, \perp \rangle$
    end
    Collect $n-t$ $\langle \texttt{CANDIDATE}, r_p, * \rangle$ messages
    if at least $t+1$ msgs are NOT of the form $\langle \texttt{CANDIDATE}, r_p, x_p \rangle$ then
        $x_p \leftarrow$RANDOM() // pick randomly from {0,1}
    $r_p \leftarrow r_p + 1$
```

For brevity, the proof of correctness of RS-Bosco is omitted. We note that RS-Bosco suffers from two limitations as currently constructed. First, RS-Bosco solves only binary consensus. Second, RS-Bosco uses a local coin to randomly update local estimates when a threshold of identical votes cannot be obtained. This mechanism is similar to that in the Ben-Or algorithm and causes the algorithm to require an exponential number of rounds for decision when contention is present. We believe that these limitations can be overcome in practical implementations, but a thorough discussion is beyond the scope of this paper.

6 Related Work

One-step consensus algorithms for crash failures have previously been studied. Brasileiro et al. [2] proposed a general technique for converting any crash-tolerant consensus algorithm into a crash-tolerant consensus algorithm that terminates in one communication step if all correct processors have the same initial value. Bosco is an extension of the ideas presented in that work to handle Byzantine failures. The key difference between handling crashed failures and Byzantine failures is that when Byzantine failures need to be tolerated, equivocation must be handled correctly.

A simple and elegant crash-tolerant consensus algorithm of the same flavor, One-Third-Rule, appears in [4]. This work has been extended to handle Byzantine faults by considering transmission faults where messages can be corrupted in addition to being dropped [14]. The algorithms in [4,14] differ from the algorithms we have presented because they use a different failure model, where failures are attributed to message transmissions, rather than to processors.

Friedman et al. [8] proposed a weakly one-step algorithm that tolerates Byzantine faults and terminates with probability 1 but does not tolerate a strong network adversary. In particular, their protocol is dependent on a common coin oracle and assumes that the network adversary has no access to this common coin; a strong network adversary with access to the common coin can prevent termination. In comparison, Bosco does not explicitly depend on any oracles, although the subroutine invoked by Bosco may have such dependencies. With a judicious choice of the consensus subroutine, Bosco can tolerate a strong network adversary that can arbitrarily re-order messages and collude with Byzantine processors. In particular, RS-Bosco does not require any oracles and tolerates strong network adversaries.

Zielinski [15] presents a framework for expressing various consensus protocols using an abstraction called *Optimistically Terminating Consensus* (OTC). Among the algorithms constructed by Zielinski are two Byzantine consensus algorithms with one-step characteristics that require $n > 5t$ and $n > 3t$. The first of these algorithms is a weakly one-step algorithm that requires partial synchrony; the second algorithm, while appearing to violate the lower bounds we have shown in this paper, is neither weakly nor strongly one-step because processors can only decide in the first communication step when, in addition to the system being failure-free and contention-free, all processors are fast enough that the timeout mechanism in the algorithm is not triggered.

Many techniques have been proposed to improve the performance and reduce the overhead of providing Byzantine fault tolerance. Abd-El-Malek et al. [16] proposed the optimistic use of quorums rather than agreement protocols to obtain higher throughput. However, in the face of contention, optimistic quorum systems perform poorly. HQ combines the use of quorums and consensus techniques to provide high performance during normal operation and minimize overhead during periods of contention [17]. Probabilistic techniques have also been proposed to reduce the overhead of using quorum systems to provide Byzantine fault-tolerance [18,19]. Hendricks et al. [20] proposed the use of erasure coding to minimize the overhead of a Byzantine fault-tolerant storage system. Zyzzyva, another recently proposed Byzantine fault-tolerant system, uses optimistic speculation to decrease the latency observed by clients [21]. In comparison, the one-step Byzantine consensus algorithms presented in this paper aims to improve performance by exploiting contention-free and failure-free situations to provide decisions in one communication step.

Lamport [5] presents lower bounds for the number of message delays and the number of processors needed for several kinds of asynchronous non-Byzantine consensus algorithm in; in particular, *Fast Learning* algorithms are one-step algorithms for non-Byzantine settings. A one-step version of Paxos [22], Fast Paxos, is presented in [3,6]. Fast Paxos tolerates only crash failures, although [6] alludes to the possibility of a Byzantine fault-tolerant version of Fast Paxos.

7 Conclusion

Byzantine fault tolerance has drawn significant interest from both academia and the industry recently. While Byzantine fault tolerance aims to provide resilience against arbitrary failures, in many applications, failures and contention are not the norm. This paper explores optimization opportunities in contention-free and failure-free situations.

Overall, this paper makes three contributions: 1) we provide two definitions of one-step asynchronous Byzantine consensus algorithms that provide low latency performance in favorable conditions while guaranteeing strong consistency when failures and contention occur; 2) we prove lower bounds in the number of processors required for such algorithms; and 3) we present Bosco, a one-step algorithm for Byzantine asynchronous consensus that meets these bounds.

References

1. Keidar, I., Rajsbaum, S.: On the cost of fault-tolerant consensus when there are no faults. SIGACT News 32(2), 45–63 (2001)
2. Brasileiro, F.V., Greve, F., Mostéfaoui, A., Raynal, M.: Consensus in one communication step. In: Proc. of the 6th International Conference on Parallel Computing Technologies, pp. 42–50. Springer, London (2001)
3. Boichat, R., Dutta, P., Frolund, S., Guerraoui, R.: Reconstructing Paxos. ACM SIGACT News 34 (2003)
4. Charron-Bost, B., Schiper, A.: The Heard-Of model: Unifying all benign failures. Technical Report LSR-REPORT-2006-004, EPFL (2006)
5. Lamport, L.: Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-72, Microsoft Research (2004)
6. Lamport, L.: Fast Paxos. Distributed Computing 19(2), 79–103 (2006)
7. Dobre, D., Suri, N.: One-step consensus with zero-degradation. In: DSN 2006: Proceedings of the International Conference on Dependable Systems and Networks, pp. 137–146. IEEE Computer Society, Washington (2006)
8. Friedman, R., Mostefaoui, A., Raynal, M.: Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. IEEE Transactions on Dependable and Secure Computing 2(1), 46–56 (2005)
9. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
10. Fischer, M., Lynch, N., Patterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
11. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. In: Proceedings of the International Conference on Dependable Systems and Networks, pp. 402–411 (June 2005)
12. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols. In: Proc. of the 2nd ACM Symp. on Principles of Distributed Computing, Montreal, Quebec, ACM SIGOPS-SIGACT, pp. 27–30 (August 1983)
13. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA (February 1999)

14. Biely, M., Widder, J., Charron-Bost, B., Gaillard, A., Hutle, M., Schiper, A.: Tolerating corrupted communication. In: PODC 2007: Proceedings of the twenty-sixth annual ACM symposium on Principles of Distributed Computing, pp. 244–253. ACM, New York (2007)
15. Zielinski, P.: Optimistically terminating consensus: All asynchronous consensus protocols in one framework. In: ISPDC '06: Proceedings of the Proceedings of The Fifth International Symposium on Parallel and Distributed Computing, Washington, DC, pp. 24–33. IEEE Computer Society Press, Los Alamitos (2006)
16. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. SIGOPS Operating Systems Review 39(5), 59–74 (2005)
17. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In: OSDI 2006: Proceedings of the 7th symposium on Operating Systems Design and Implementation, pp. 177–190. USENIX Association, Berkeley (2006)
18. Merideth, M.G., Reiter, M.K.: Probabilistic opaque quorum systems. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 403–419. Springer, Heidelberg (2007)
19. Malkhi, D., Reiter, M.K., Wool, A., Wright, R.N.: Probabilistic quorum systems. Information and Computation 170(2), 184–206 (2001)
20. Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead Byzantine fault-tolerant storage. In: Proc. of twenty-first ACM SIGOPS Symposium on Operating Systems Principles, pp. 73–86. ACM, New York (2007)
21. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative Byzantine fault tolerance. In: Proc. of twenty-first ACM SIGOPS symposium on Operating Systems Principles, pp. 45–58. ACM, New York (2007)
22. Lamport, L.: The part-time parliament. Trans. on Computer Systems 16(2), 133–169 (1998)

A Limit to the Power of Multiple Nucleation in Self-assembly

Aaron D. Sterling[*]

Department of Computer Science, Iowa State University, Ames, IA 50014, USA
sterling@cs.iastate.edu

Abstract. Majumder, Reif and Sahu presented in [7] a model of reversible, error-permitting tile self-assembly, and showed that restricted classes of tile assembly systems achieved equilibrium in (expected) polynomial time. One open question they asked was how the model would change if it permitted multiple nucleation, *i.e.,* independent groups of tiles growing before attaching to the original seed assembly. This paper provides a partial answer, by proving that no tile assembly model can use multiple nucleation to achieve speedup from polynomial time to constant time without sacrificing computational power: if a tile assembly system $\mathcal{T}$ uses multiple nucleation to tile a surface in constant time (independent of the size of the surface), then $\mathcal{T}$ is unable to solve computational problems that have low complexity in the (single-seeded) Winfree-Rothemund Tile Assembly Model. The proof technique defines a new model of distributed computing that simulates tile assembly, so a tile assembly model can be described as a distributed computing model.

Keywords: self-assembly, multiple nucleation, locally checkable labeling.

1 Introduction

1.1 Overview

Nature is replete with examples of the self-assembly of individual parts into a more complex whole, such as the development from zygote to fetus, or, more simply, the replication of DNA itself. In his Ph.D. thesis in 1998, Winfree proposed a formal mathematical model to reason algorithmically about processes of self-assembly [15]. Winfree connected the experimental work of Seeman [12] (who had built "DNA tiles," molecules with unmatched DNA base pairs protruding in four directions, so they could be approximated by squares with different "glues" on each side) to a notion of tiling the integer plane developed by Wang in the 1960s [14]. Rothemund, in his own Ph.D. thesis, extended Winfree's original Tile Assembly Model [10].

Informally speaking, Winfree effectivized Wang tiling, by requiring a tiling of the plane to start with an individual *seed tile* or a connected, finite *seed assembly.*

[*] This research was supported in part by National Science Foundation Grants 0652569 and 0728806.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 451–465, 2008.

Tiles would then accrete one at a time to the seed assembly, growing a *seed supertile*. A *tile assembly system* is a finite set of differently defined *tile types*. Tile types are characterized by the names of the "glues" they carry on each of their four sides, and the binding strength each glue can exert. We assume that when the tiles interact "in solution," there are infinitely many tiles of each tile type. Tile assembly proceeds in discrete stages. At each stage s, from all possibilities of tile attachment at all possible locations (as determined by the glues of the tile types and the binding requirements of the system overall), one tile will bind. If more than one tile type can bind at stage s, a tile type and location will be chosen uniformly at random. Winfree proved that his Tile Assembly Model is Turing universal, so it is a robust model of computation.

The standard Winfree-Rothemund tile assembly model is *error-free* and *irreversible*—tiles always bind correctly, and, once a tile binds, it can never unbind. Adleman *et al.* were the first to define a notion of time complexity for tile assembly, using a one-dimensional error-permitting, reversible model, where tiles would assemble in a line with some error probability, then be scrambled, and fall back to the line [1]. Adleman *et al.* proved bounds on how long it would take such models to achieve equilibrium. Majumder, Reif and Sahu have recently presented a two-dimensional stochastic model for self-assembly [7], and have shown that some tiling problems in their model correspond to *rapidly mixing Markov chains*—Markov chains that reach stationary distribution in time polynomial in the state space. The tile assembly model in [7], like the standard model, allows only for a single seed assembly, and one of the open problems in [7] was how the model might change if it allowed multiple nucleation, *i.e.*, if multiple supertiles could build independently before attaching to a growing seed supertile.

The main result of this paper provides a time complexity lower bound for tile assembly models that permit multiple nucleation: there is no way to use multiple nucleation to achieve a speedup to tiling a surface in constant time (time independent of the size of the surface) without sacrificing computational power. This result holds for tile assembly models that are reversible, irreversible, error-permitting or error-free. In fact, a speedup to constant time is impossible, even if we relax the model to allow that, at each step s, there is a positive probability for every available location that a tile will bind there (instead of requiring that exactly one tile bind per stage).

Our method of proof appears novel: given a tile assembly model and a tile assembly system $\mathcal{T}$ in that model, we construct a distributed network of processors that can simulate the behavior of $\mathcal{T}$ as it assembles on a surface. Our result then follows from the theorem by Naor and Stockmeyer that locally checkable labeling (LCL) problems have no local solution in constant time [8]. This is true for both deterministic and randomized algorithms, so no constant-time tile assembly system exists that solves an LCL problem with a positive probability of success. We consider one LCL problem in specific, the weak c-coloring problem, and demonstrate a tile set of only seven tile types that solves the weak c-coloring problem in the Winfree-Rothemund Tile Assembly Model, even though weak c-coloring is impossible to achieve in constant time by multiple nucleation, regardless of the rate of convergence to equilibrium.

1.2 Background

In the standard Tile Assembly Model, one tile is added per stage, so the primary complexity measure is not one of time, but of how much information a tile set needs in order to solve a particular problem. Several researchers [1] [3] [4] [11] [13] have investigated the *tile complexity* (the minimum number of distinct tile types required for assembly) of finite shapes, and sets of "scale-equivalent" shapes (essentially a $\mathbb{Z} \times \mathbb{Z}$ analogue of the Euclidean notion of similar figures). For example, it is now known that the number of tile types required to assemble a square of size $n \times n$ (for n any natural number) is $\Omega(\log n / \log \log n)$ [11]. Or, if T is the set of all discrete equilateral triangles, the asymptotically optimal relationship between triangle size and number of tiles required to assemble that triangle, is closely related to the Kolmogorov Complexity of a program that outputs the triangle as a list of coordinates [13].

Despite these advances in understanding of the complexity of assembling finite, bounded shapes, the self-assembly of infinite structures is not as well understood. In particular, there are few lower bounds or impossibility results on what infinite structures can be self-assembled in the Tile Assembly Model. The first such impossibility result appeared in [6], when Lathrop, Lutz and Summers showed that no finite tile set can assemble the discrete Sierpinski Triangle by placing a tile only on the coordinates of the shape itself. (By contrast, Winfree had shown that just seven tile types are required to tile the first quadrant of the integer plane with tiles of one color on the coordinates of the discrete Sierpinski Triangle, and tiles of another color on the coordinates of the complement [15].) Recently, Patitz and Summers have extended this initial impossibility result to other discrete fractals [9], and Lathrop *et al.* [5] have demonstrated sets in $\mathbb{Z} \times \mathbb{Z}$ that are Turing decidable but cannot be self-assembled in Winfree's sense.

To date, there has been no work comparing the strengths of different tile assembly models with respect to infinite (nor to finite but arbitrarily large) structures. Since self-assembly is an asynchronous process in which each point has only local knowledge, it is natural to consider whether the techniques of distributed computing might be useful for comparing models and proving impossibility results in nanoscale self-assembly. This paper is an initial attempt in that direction.

Aggarwal *et al.* in [3] proposed a generalization of the standard Tile Assembly Model, which they called the q-Tile Assembly Model. This model permitted multiple nucleation: tiles did not need to bind immediately to the seed supertile. Instead, they could form independent supertiles of size up to some constant q before then attaching to the seed supertile. While the main question considered in [3] was *tile* complexity, we can also ask whether multiple nucleation would allow an improvement in *time* complexity. Intuitively, Does starting from multiple points allow us to build things strictly faster than starting from a single point?

As mentioned above, Majumder, Reif and Sahu recently presented a stochastic, error-permitting tile assembly model, and calculated the rate of convergence to equilibrium for several tile assembly systems [7]. The model in [7] permitted only a single seed assembly, and addition of one tile to the seed supertile at each stage. Majumder, Reif and Sahu left as an open question how the model might be extended to permit the presence and binding of multiple supertiles.

Therefore, we can rephrase the "intuitive" question above as follows: Can we tile a surface of size $n \times n$ in a constant number of stages, by randomly selecting nucleation points on the surface, building supertiles of size q or smaller from those points in $\leq q$ stages, and then allowing $\leq r$ additional stages for tiles to fall off and be replaced if the edges of the supertiles contain tiles that bind incorrectly? (The assembly achieves equilibrium in constant time because q and r do not depend on n.)

The main result of this paper is that the answer is: Not without losing significant computational power.

Section 2 of this paper describes the "standard" Winfree-Rothemund Tile Assembly Model, and then considers generalizations of the standard model that permit multiple nucleation. Section 3 reviews the distributed computing results of Naor and Stockmeyer needed to prove the impossibility result. In Section 4 we present our main result. Section 5 concludes the paper and suggests directions for future research.

2 Description of Tile Assembly Models

2.1 The Winfree-Rothemund Tile Assembly Model

Winfree's objective in defining the Tile Assembly Model was to provide a useful mathematical abstraction of DNA tiles combining in solution in a random, nondeterministic, asynchronous manner [15]. Rothemund [10], and Rothemund and Winfree [11], extended the original definition of the model. For a comprehensive introduction to tile assembly, we refer the reader to [10]. In our presentation here, we follow [6], which gives equal status to finite and infinite tile assemblies. Throughout this paper, we will consider only two-dimensional tile assemblies.

Intuitively, a tile of type t is a unit square that can be placed with its center on a point in the integer lattice. A tile has a unique orientation; it can be translated, but not rotated. We identify the side of a tile with the direction (or unit vector) one must travel from the center to cross that side. The literature often refers to west, north, east and south sides, starting at the leftmost side and proceeding clockwise. Each side $\overrightarrow{u} \in U_2$ (where U_2 is the set of unit vectors in two coordinates) of a tile is covered with a "glue" that has *color* $\text{col}_t(\overrightarrow{u})$ and *strength* $\text{str}_t(\overrightarrow{u})$. Figure 1 shows how a tile is represented graphically.

If tiles of types t and t' are placed adjacent to each other (*i.e.*, with their centers at $\overrightarrow{m}$ and $\overrightarrow{m} + \overrightarrow{u}$, where $\overrightarrow{m} \in \mathbb{Z}^2$ and $\overrightarrow{u} \in U_2$) then they will *bind* with strength $\text{str}_t(\overrightarrow{u}) \cdot [\![t(\overrightarrow{u}) = t'(-\overrightarrow{u})]\!]$, where $[\![\phi]\!]$ is the Boolean value of the statement ϕ. Note that this definition of binding implies that if the glues of the adjacent sides do not have the same color or strength, then their binding strength is 0. Later, we will permit pairs of glues to have negative binding strength, to model error occurrence and correction.

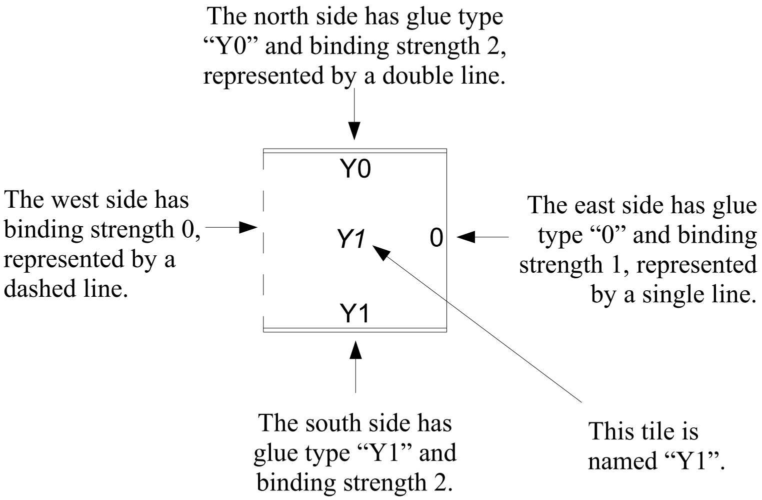

The north side has glue type "Y0" and binding strength 2, represented by a double line.
Y0
The west side has binding strength 0, represented by a dashed line.
Y1
0
The east side has glue type "0" and binding strength 1, represented by a single line.
Y1
The south side has glue type "Y1" and binding strength 2.
This tile is named "Y1".

Fig. 1. An example tile with explanation

One parameter in a tile assembly model is the minimum binding strength required for tiles to bind "stably." This parameter is usually termed *temperature* and denoted by τ, where $\tau \in \mathbb{N}$.

As we consider only two-dimensional tile assemblies, we limit ourselves to working in $\mathbb{Z}^2 = \mathbb{Z} \times \mathbb{Z}$. U_2 is the set of all unit vectors in $\mathbb{Z}^2$.

A *binding function* on an (undirected) graph $G = (V, E)$ is a function $\beta : E \longrightarrow \mathbb{N}$. If β is a binding function on a graph $G = (V, E)$ and $C = (C_0, C_1)$ is a cut of G, then the *binding strength* of β on C is

$$\beta_C = \{\beta(e) \mid e \in E, \{e\} \cap C_0 \neq \emptyset, \text{ and } \{e\} \cap C_1 \neq \emptyset\} \ .$$

The *binding strength* of β on G is then $\beta(G) = \min\{\beta_C \mid C \text{ is a cut of } G\}$. Intuitively, the binding function captures the strength with which any two neighbors are bound together, and the binding strength of the graph is the minimum strength of bonds that would have to be severed in order to separate the graph into two pieces.

A *binding graph* is an ordered triple $G = (V, E, \beta)$ where (V, E) is a graph and β is a binding function on (V, E). If $\tau \in \mathbb{N}$, a binding graph $G = (V, E, \beta)$ is *τ-stable* if $\beta(V, E) \geq \tau$.

Recall that a *grid graph* is a graph $G = (V, E)$ where $V \subseteq \mathbb{Z} \times \mathbb{Z}$ and every edge $\{\overrightarrow{m}, \overrightarrow{n}\} \in E$ has the property that $\overrightarrow{m} - \overrightarrow{n} \in U_2$.

Definition 1. *A* tile type *over a (finite) alphabet Σ is a function $t : U_2 \longrightarrow \Sigma^* \times \mathbb{N}$. We write $t = (\mathrm{col}_t, \mathrm{str}_t)$, where $\mathrm{col}_t : U_2 \longrightarrow \Sigma^*$, and $\mathrm{str}_t : U_2 \longrightarrow \mathbb{N}$ are defined by $t(\overrightarrow{u}) = (\mathrm{col}_t(\overrightarrow{u}), \mathrm{str}_t(\overrightarrow{u}))$ for all $\overrightarrow{u} \in U_2$.*

Definition 2. *If T is a set of tile types, a T-*configuration *is a partial function $\alpha : \mathbb{Z}^2 \dashrightarrow T$.*

Definition 3. *The* binding graph *of a T-configuration $\alpha : \mathbb{Z}^2 \dashrightarrow T$ is the binding graph $G_\alpha = (V, E, \beta)$, where (V, E) is the grid graph given by*

$$\begin{aligned} &V = \mathrm{dom}(\alpha),\\ &E = \big\{\{\overrightarrow{m}, \overrightarrow{n}\} \in [V]^2 \mid \overrightarrow{m} - \overrightarrow{n} \in U_2, \mathrm{col}_{\alpha(\overrightarrow{m})}(\overrightarrow{n} - \overrightarrow{m}) = \mathrm{col}_{\alpha(\overrightarrow{n})}(\overrightarrow{m} - \overrightarrow{n}), \textit{ and}\\ &\quad \mathrm{str}_{\alpha(\overrightarrow{m})}(\overrightarrow{n} - \overrightarrow{m}) > 0\big\}, \end{aligned}$$

and the binding function $\beta : E \longrightarrow \mathbb{Z}^+$ is given by $\beta(\{\overrightarrow{m}, \overrightarrow{n}\}) = \mathrm{str}_{\alpha(\overrightarrow{m})}(\overrightarrow{n} - \overrightarrow{m})$ for all $\{\overrightarrow{m}, \overrightarrow{n}\} \in E$.

Definition 4. *For T a set of tile types, a T-configuration α is* stable *if its binding graph G_α is τ-stable. A* τ-T-assembly *is a T-configuration that is τ-stable. We write $\mathcal{A}^\tau_T$ for the set of all τ-T-assemblies.*

Definition 5. *Let α and α' be T-configurations.*

1. *α is a* subconfiguration *of α', and we write $\alpha \sqsubseteq \alpha'$, if $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\alpha')$ and, for all $\overrightarrow{m} \in \mathrm{dom}(\alpha)$, $\alpha(\overrightarrow{m}) = \alpha'(\overrightarrow{m})$.*
2. *α' is a* single-tile extension *of α if $\alpha \sqsubseteq \alpha'$ and $\mathrm{dom}(\alpha') \smallsetminus \mathrm{dom}(\alpha)$ is a singleton set. In this case, we write $\alpha' = \alpha + (\overrightarrow{m} \mapsto t)$, where $\{\overrightarrow{m}\} = \mathrm{dom}(\alpha') \smallsetminus \mathrm{dom}(\alpha)$ and $t = \alpha'(\overrightarrow{m})$.*
3. *The notation $\alpha \xrightarrow[\tau,T]{1} \alpha'$ means that $\alpha, \alpha' \in \mathcal{A}^\tau_T$ and α' is a single-tile extension of α.*

Definition 6. *Let $\alpha \in \mathcal{A}^\tau_T$.*

1. *For each $t \in T$, the* τ-t-frontier *of α is the set*
$$\partial^\tau_T \alpha = \left\{ \overrightarrow{m} \in \mathbb{Z}^2 \smallsetminus \mathrm{dom}(\alpha) \,\middle|\, \sum_{\overrightarrow{u} \in U_2} \mathrm{str}_t(\overrightarrow{u}) \cdot \llbracket \alpha(\overrightarrow{m} + \overrightarrow{u})(-\overrightarrow{u}) = t(\overrightarrow{u}) \rrbracket \geq \tau \right\} .$$
2. *The* τ-frontier *of α is the set*
$$\partial^\tau \alpha = \bigcup_{t \in T} \partial^\tau_t \alpha \ .$$

Definition 7. *A* τ-T-assembly sequence *is a sequence $\overrightarrow{\alpha} = (\alpha_i \mid 0 \leq i < k)$ in $\mathcal{A}^\tau_T$, where $k \in \mathbb{Z}^+ \cup \{\infty\}$ and, for each i with $1 \leq i + 1 < k$, $\alpha_i \xrightarrow[\tau,T]{1} \alpha_{i+1}$.*

Definition 8. *The* result *of a τ-T-assembly sequence $\overrightarrow{\alpha} = (\alpha_i \mid 0 \leq i < k)$ is the unique T-configuration $\alpha = \mathrm{res}(\overrightarrow{\alpha})$ satisfying: $\mathrm{dom}(\alpha) = \cup_{0 \leq i < k} \mathrm{dom}(\alpha_i)$ and $\alpha_i \sqsubseteq \alpha$ for each $0 \leq i < k$.*

Definition 9. *Let $\alpha, \alpha' \in \mathcal{A}^\tau_T$. A* τ-T-assembly sequence from α to α' *is a τ-T-assembly sequence $\overrightarrow{\alpha} = (\alpha_i \mid 0 \leq i < k)$ such that $\alpha_0 = \alpha$ and $\mathrm{res}(\overrightarrow{\alpha}) = \alpha'$. We write $\alpha \xrightarrow[\tau,T]{} \alpha'$ to indicate that there exists a τ-T-assembly from α to α'.*

Definition 10. *An assembly $\alpha \in \mathcal{A}^\tau_T$ is* terminal *if $\partial^\tau \alpha = \emptyset$.*

Intuitively, a configuration is a set of tiles that have been placed in the plane, and the configuration is stable if the binding strength at every possible cut is at least as high as the temperature of the system. Informally, an assembly sequence is a sequence of single-tile additions to the frontier of the assembly constructed at the previous stage. Assembly sequences can be finite or infinite in length. We are now ready to present a definition of a tile assembly system.

Definition 11. *Write $\mathcal{A}^{\tau}_{T}$ for the set of configurations, stable at temperature τ, of tiles whose tile types are in T. A* tile assembly system *is an ordered triple $\mathcal{T} = (T, \sigma, \tau)$ where T is a finite set of tile types, $\sigma \in \mathcal{A}^{\tau}_{T}$ is the* seed assembly*, and $\tau \in \mathbb{N}$ is the* temperature*. We require* dom(σ) *to be finite.*

Definition 12. *Let $\mathcal{T} = (T, \sigma, \tau)$ be a tile assembly system.*

1. *Then the* set of assemblies produced by $\mathcal{T}$ *is*

$$\mathcal{A}[\mathcal{T}] = \left\{\alpha \in \mathcal{A}^{\tau}_{T} \middle| \sigma \underset{\tau,T}{\longrightarrow} \alpha\right\} \ ,$$

where "$\sigma \underset{\tau,T}{\longrightarrow} \alpha$" means that tile configuration α can be obtained from seed assembly σ by a legal addition of tiles (as formalized in Appendix A).

2. *The* set of terminal assemblies produced by $\mathcal{T}$ *is*

$$\mathcal{A}_{\Box}[\mathcal{T}] = \{\alpha \in \mathcal{A}[\mathcal{T}] \mid \alpha \textit{ is terminal}\} \ ,$$

where "terminal" describes a configuration to which no tiles can be legally added.

If we view tile assembly as the programming of matter, the following analogy is useful: the seed assembly is the input to the computation; the tile types are the legal (nondeterministic) steps the computation can take; the temperature is the primary inference rule of the system; and the terminal assemblies are the possible outputs.

We are, of course, interested in being able to *prove* that a certain tile assembly system always achieves a certain output. In [13], Soloveichik and Winfree presented a strong technique for this: local determinism.

Informally, an assembly sequence $\overrightarrow{\alpha}$ is locally deterministic if (1) each tile added in $\overrightarrow{\alpha}$ binds with the minimum strength required for binding; (2) if there is a tile of type t_0 at location $\overrightarrow{m}$ in the result of α, and t_0 and the immediate "OUT-neighbors" of t_0 are deleted from the result of α, then no other tile type in $\mathcal{T}$ can legally bind at $\overrightarrow{m}$; the result of α is terminal.

Definition 13 (Soloveichik and Winfree [13]). *A τ-$\mathcal{T}$-assembly sequence $\overrightarrow{\alpha} = (\alpha_i \mid 0 \leq i \leq k)$ with result α is* locally deterministic *if it has the following three properties.*

1. *For all $\overrightarrow{m} \in dom(\alpha) - dom(\alpha_0)$,*

$$\sum_{\overrightarrow{u} \in \mathrm{IN}^{\overrightarrow{\alpha}}(\overrightarrow{m})} \mathrm{str}_{\alpha_{i_\alpha(\overrightarrow{m})}}(\overrightarrow{m}, \overrightarrow{u}) = \tau \ .$$

2. *For all* $\overrightarrow{m} \in dom(\alpha) - dom(\alpha_0)$ *and all* $t \in \mathcal{T} - \{\alpha(\overrightarrow{m})\}$, $\overrightarrow{m} \notin \partial_t^\tau(\overrightarrow{\alpha} \backslash \overrightarrow{m})$.
3. $\partial^\tau \alpha = \emptyset$.

Definition 14 (Soloveichik and Winfree [13]). *A tile assembly system* $\mathcal{T}$ *is* locally deterministic *if there exists a locally deterministic* τ-$\mathcal{T}$*-assembly sequence* $\alpha = (\alpha_i \mid 0 \leq i < k)$ *with* $\alpha_0 = \sigma$.

Local determinism is important because of the following result.

Theorem 1 (Soloveichik and Winfree [13]). *If* $\mathcal{T}$ *is locally deterministic, then* $\mathcal{T}$ *has a unique terminal assembly.*

2.2 Generalizations of the Winfree-Rothemund Tile Assembly Model

We will consider three generalizations of the standard tile assembly model: (1) multiple nucleation; (2) assembly in which glues bind incorrectly according to some error probability; and (3) negative glue strengths, allowing incorrectly bound tiles to be released from the assembly so it is possible for a correctly-binding tile to attach in that space. We move from an *irreversible* tiling model, in which tiles are placed in an error-free manner and can never be removed, to a *reversible* tiling model, in which a terminal assembly is defined by equilibrium, not by the disappearance of a frontier to which tiles can be legally added.

Aggarwal *et al.* in [3] formulated and studied a model that permitted multiple nucleation, which they called the *q-tile* or *multiple tile* model. Essentially, they allowed supertiles to form, independent of the seed, up to size bounded by a constant q. Then the independent supertile would have to bind to the growing seeed supertile. Legal supertiles were defined recursively: each tile type was a legal supertile, and any two supertiles whose combined size was $\leq q$ could form a legal supertile if the binding strength at their adjacent frontiers was at least the temperature of the system.

Models of reversible tiling have been considered in [15] and [1], and more recently in [7], which contains a summary of previous work in the area. Majumder, Reif and Sahu in [7] introduced the concept of *bond pair equilibrium*, as follows.

Definition 15 (Majumder, Reif and Sahu [7]). *Suppose* α *is a finite configuration that contains* m *different tile types* $t_1, \ldots, t_m$, *with* γ_i *the relative fraction of tiles of type* t_i *(so* $\sum \gamma_i = 1$*).*

1. *Define* a_{ij} *to be the fraction of* t_i *tiles bonded to the east to a* t_j *tile.*
2. *Define* b_{ik} *to be the fraction of* t_i *tiles bonded to the north to a* t_k *tile.*
3. *Define* p_{ij} *to be the fraction of* t_i *tiles bonded to the west to a* t_j *tile.*
4. *Define* q_{ik} *to be the fraction of* t_i *tiles bonded to the south to a* t_k *tile.*
5. $A_{ij} = \gamma_i a_{ij}$. $B_{ik} = \gamma_i b_{ik}$.

Definition 16 (Majumder, Reif and Sahu [7]). *A configuration* α *in an error-permitting, reversible tile assembly system has achieved* bond pair equilibrium *when, for every tile type* t_i *in* α, *the (expected value of the) number of pairs* (A_{ij}, B_{kj}) *is invariant over time steps.*

Informally, bond pair equilibrium is achieved when, if the configuration is considered as a whole, the quantity of each distinct bond interaction does not change over time. If we assume the system has a property of *bond independence*—the bond on one side of a tile does not affect the binding on the other three sides—then bond pair equilibrium is a sufficient condition for thermodynamic equilibrium.

Theorem 2 (Majumder, Reif and Sahu [7]). *Bond pair equilibrium and bond independence implies strong (thermodynamic) equilibrium.*

This theorem provides justification for us to replace the notion of terminal assembly with the notion of assembly that has achieved bond pair equilibrium, if we relax the Winfree-Rothemund Tile Assembly Model to include the possiblity of error in binding, and the reversibility of tile assembly.

Majumder, Reif and Sahu studied the rate of convergence of several tile assembly systems in a model that only permitted addition of one tile at a given time step. They defined the notion of a Markov Chain that corresponds to an assembly system, and demonstrated several tile assembly systems whose Markov chains were rapidly mixing, *i.e.,* they reached stationary distribution in time polynomial in the state space.

In what follows, we will see that a speedup to constant time is impossible without losing computational power, even if we add multiple nucleation to a model of reversible tile assembly. First, though, we review the distributed computing impossibility results that imply this.

3 Distributed Computing Results of Naor and Stockmeyer

In a well known distributed computing paper, Naor and Stockmeyer investigated whether "locally checkable labeling" problems could be solved over a network of processors in an entirely local manner, where a local solution means a solution arrived at "within time (or distance) independent of the size of the network" [8]. One locally checkable labeling problem Naor and Stockmeyer considered was the *weak c-coloring problem.*

Definition 17 (Naor and Stockmeyer [8]). *For $c \in \mathbb{N}$, a* weak c-coloring *of a graph is an assignment of numbers from $\{1, \ldots, c\}$ (the possible "colors") to the vertices of the graph such that for every non-isolated vertex v there is at least one neighbor w such that v and w receive different colors. Given a graph G, the* weak c-coloring problem for G *is to weak c-color the nodes of G.*

In the context of tiling, to solve the weak c-coloring problem for an $n \times n$ surface means tiling the surface so each tile has at least one neighbor (to the north, south, east or west) of a different color. In the next section, we will present a simple solution to the weak c-coloring problem in the Winfree-Rothemund Tile Assembly Model. By contrast, Naor and Stockmeyer showed that no local, constant-time algorithm can solve the weak c-coloring problem for grid graphs.

Theorem 3 (Naor and Stockmeyer [8]). *For any c and t, there is no local algorithm with time bound t that solves the weak c-coloring problem for the class of finite square grid graphs over the integer lattice.*

This theorem is a consequence of Theorem 6.3 in [8]. The original result is a stronger statement.

A second theorem from the same paper says that randomization does not help. As before, the original result is stronger than the formulation I provide here.

Theorem 4 (Naor and Stockmeyer [8]). *Fix a class $\mathcal{G}$ of graphs closed under disjoint union. If there is a randomized local algorithm P with time bound t that solves the weak c-coloring problem for $\mathcal{G}$ with error probability ϵ for some $\epsilon < 1$, then there is a deterministic local algorithm A with time bound t that solves the weak c-coloring problem for $\mathcal{G}$.*

4 Proof of Main Result

In order to apply the theorems of Naor and Stockmeyer to the realm of tile assembly, we build a distributed network of processors that simulates assembly of tile assembly system $\mathcal{T}$ in tile assembly model $\mathcal{M}$. We accomplish this by defining a class of tile assembly models that generalize the standard model and permit multiple nucleation; and we show that for any tileset defined in that class of models, there is a system of distributed processors that simulates the assembly behavior of that tileset.

Theorem 5. *For any (reversible or irreversible) tile assembly model $\mathcal{M}$ that permits multiple nucleation, and any tile set $\mathcal{T}$ in $\mathcal{M}$, there is a model of distributed computing $\mathcal{N}$ that simulates the assembly of $\mathcal{T}$ on a surface of size n^2, using n^2 processors laid out in a grid graph, and constant-size message complexity.*

Proof. Fix a tile assembly model $\mathcal{M}$ with the following properties:

1. The binding function β of $\mathcal{M}$ assigns a real number to each pair of glue types. This assignment can be positive, zero or negative.
2. The definition of the binding function β and the definition of each tile type t_i induces a function

$$\hat{\beta} : T \times (\{\text{glue colors of } T, \text{glue strengths of } T\} \cup \{\emptyset\})^4 \longrightarrow [0, 1] \ ,$$

   such that for any T-configuration α and any location $\overrightarrow{m}$ at stage s,

$$\hat{\beta}\big[\alpha(\overrightarrow{m}), \alpha(\overrightarrow{m} + (1, 0)), \alpha(\overrightarrow{m} + (-1, 0)), \alpha(\overrightarrow{m} + (0, 1)), \alpha(\overrightarrow{m} + (0, -1))\big]$$

   is the probability that the tile at location $\overrightarrow{m}$ will remain in that location at the end of stage s. (In words, $\hat{\beta}$ is a function from a tile type and each possible set of glues—including no glue—adjacent to that tile type, to a probability that the tile will remain in that location at the end of the stage.) Note that in a model of irreversible tiling, if there is a tile in location $\overrightarrow{m}$ that is part of configuration α, then we can drop the part of $\hat{\beta}$ that depends on the tile's neighbors, and $\hat{\beta}[\alpha(\overrightarrow{m})]$ always takes the value 1.

3. $\mathcal{M}$ can allow multiple nucleation. In addition to the placement of the seed assembly at the first stage of assembly, there is some probability π such that (at the first stage of assembly only) a tile is placed on each location of the surface in question with probability π, determined uniformly at random. (Note that if $\pi = 0$, then $\mathcal{M}$ does not allow multiple nucleation.)
4. At each stage s of assembly, there is a probability $\pi_{s,\overrightarrow{m}}$ for each location $\overrightarrow{m}$ in the frontier of each supertile that a tile will be placed there. In particular, it is possible to place more than one tile per stage. Tiles that are placed in stage s do not interact with one another (with either positive or negative binding strength) until stage $s + 1$.

For example, if we want $\mathcal{M}$ to be the standard Winfree-Rothemund Tile Assembly Model, we set all values of β to 0 or a positive integer, all values of $\hat{\beta}$ to 1, $\pi = 0$, and the values of $\pi_{s,\overrightarrow{m}}$ sufficiently small for all stages s and locations $\overrightarrow{m}$ that, with high probability, at most one tile appears per stage. Then we count time steps only when a tile is added to the existing configuration.

We simulate assembly sequences of $\mathcal{T}$ on an $n \times n$ surface by a network of processors $\mathcal{N}$ whose network graph is an $n \times n$ grid graph. Each processor will simulate the presence or absence of a tile in the same location on the $n \times n$ tiling surface. Processors do not have unique ID's, and do not know their own coordinates. Each processor $p_i \in \mathcal{N}$ is of the following form.

Processor p_i
Four input message buffers: $\text{inbuf}_{i,n}$, $\text{inbuf}_{i,s}$, $\text{inbuf}_{i,e}$ and $\text{inbuf}_{i,w}$.
Four output message buffers: $\text{outbuf}_{i,n}$, $\text{outbuf}_{i,s}$, $\text{outbuf}_{i,e}$ and $\text{outbuf}_{i,w}$.
A color variable: COLOR_i, a variable that can take a value from $\{1, \ldots, c\}$, where c is a global constant.
A local state: Each processor is in one of $|T| + 1$ different local states q during a given execution stage s. There is one stage q_k to simulate each tile type $t_k \in T$, and an additional stage QUIET, to simulate the absence of a tile from the surface location that p_i is simulating.
A state transition function: This function takes the current processor state and the messages received in the current round, and (deterministically or probabilistically, depending on $\mathcal{M}$) directs what state the processor will adopt in the next round.

The messages processors send on the network are of form ⟨glue type, glue strength⟩. The input message buffers of processor p_i simulate the glue types of the edges the tile at p_i's location is adjacent to. The output message buffers of p_i simulate the glues on the edges of the tile p_i is simulating. The purpose of COLOR_i is to simulate the color of the tile placed at the location simulated by p_i.

All processors in $\mathcal{N}$ are hardcoded with the same state transition function, which is determined from the definition of $\hat{\beta}$ in $\mathcal{M}$, in the natural way: if, in round r of the algorithm execution, p_i is in state q_k, a simulation of $t_k \in T$, and hears messages that simulate glue types $g_1, \ldots, g_4$, then at the end of round r, if $\hat{\beta}(t_k, g_1, g_2, g_3, g_4) = \gamma$, then with probability γ the transition function directs p_i to remain in state q_k, and with probability $1 - \gamma$ to enter state QUIET.

To simulate the process of tile assembly, we run the following distributed algorithm on $\mathcal{N}$.

Algorithm execution proceeds in synchronized rounds. Before execution begins, all processors start in state QUIET. In round $r = 0$, (through the intervention of an omniscient operator) each processor in the locations corresponding to the seed assembly enters the stage to simulate the tile type at that location in the seed assembly.

Also in round $r = 0$, each processor not simulating part of the seed assembly "wakes up" (enters a state other than QUIET) with probability π. If a processor wakes up, it enters state $q \neq$ QUIET, chosen uniformly at random. For any round $r > 0$, each processor runs either Algorithm 1 or Algorithm 2, depending on whether it is in state QUIET.

The interaction between tiles in $\mathcal{M}$ is completely defined by the glues of a tile's immediate neighbors, as specified in the function $\hat{\beta}$, and the processors of $\mathcal{N}$ simulate that behavior with Algorithm 2. Since the processors of $\mathcal{N}$ simulate

Algorithm 1. For p_i in state QUIET at round r

```
if $r = 0$ then
  wake up with probability $\pi$, and cease execution for this round.
end if
if $r > 0$ then
  Read the four input buffers.
  if no messages were received then
    cease execution for the round
  else
    let $q_0$ be the state change (probabilistically) indicated by the value of $\hat{\beta}$ for
    a location that has adjacent glue types that are simulated by the messages
    received this round.
    Send the messages indicated by state $q_0$.
    Set the value of COLOR$_i$ according to $q_0$.
    Enter state $q_0$ and cease execution for this round.
  end if
end if
```

Algorithm 2. For p_i in state $q \neq$ QUIET (at any round)

```
Read the four input buffers.
if no messages were received then
  Send the messages indicated by state $q$ and cease execution for this round.
else
  Let $q_0$ be the state change directed by the function $\hat{\beta}$ applied to the glue types
  simulated by the messages received this round. {Note that $q_0$ will either equal $q$
  or QUIET, and $q_0$ might be chosen probabilistically.}
  Send the messages indicated by state $q_0$.
  Set the value of COLOR$_i$ according to $q_0$.
  Enter state $q_0$ and cease execution for this round.
end if
```

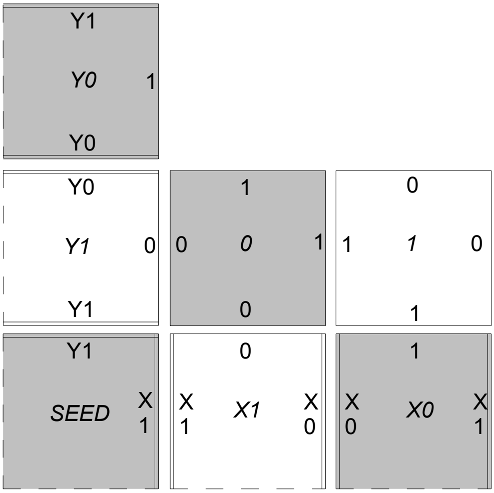


Fig. 2. The tileset $\mathcal{T}^*$ used in the proof of Lemma 2

empty spaces with Algorithm 1, by a straightforward induction argument, $\mathcal{N}$ can simulate all possible $\mathcal{T}$-assembly sequences, and the theorem is proved.

Combining Theorem 5 and the impossibility results of Naor and Stockmeyer, we obtain our main result, as follows.

Theorem 6 (Main Result). *Any (multiply nucleating) tileset that tiles a surface in constant time is unable to solve the weak c-coloring problem, even though the weak c-coloring problem has a low-complexity solution in the Winfree-Rothemund Tile Assembly Model.*

We break down the proof of this theorem into the following two lemmas.

Lemma 1. *Let $\mathcal{T}$ and $\mathcal{M}$ be such that, for all n sufficiently large, the expected time $\mathcal{T}$ takes to assemble on an $n \times n$ is some constant k, independent of n. Then $\mathcal{T}$ does not weak c-color the surface.*

Proof. Suppose $\mathcal{M}$ is an irreversible tiling model. If $\mathcal{T}$ can weak c-color surfaces in constant time, then there is a deterministic algorithm for the distributed network $\mathcal{N}$ that weak c-colors $\mathcal{N}$ locally, and in constant time. By Theorem 3 that is impossible.

So assume $\mathcal{M}$ is a reversible tiling model, and when $\mathcal{T}$ assembles, it weak c-colors the tiling surface, and achieves bond pair equilibrium in constant time. Then there is a local probabilistic algorithm for $\mathcal{N}$ that weak c-colors $\mathcal{N}$ in constant time, with positive probability of success. By Theorem 4 that is impossible as well. Therefore, no $\mathcal{T}$ exists that weak c-colors surfaces in constant time.

Lemma 2. *There is a tileset in the Winfree-Rothemund model that weak c-colors the first quadrant.*

Proof. Figure 2 exhibits a tileset $\mathcal{T}^*$ that assembles into a weak c-coloring of the first quadrant, starting from an individual seed tile placed at the origin. One can verify by inspection that $\mathcal{T}^*$ is locally deterministic, so it will always produce the same terminal assembly. All assembly sequences generated by $\mathcal{T}^*$ produce a checkerboard pattern in which a monochromatic "+" configuration never appears. Hence, it solves the weak c-coloring problem for the entire first quadrant, and also for all $n \times n$ squares, for any n.

The main result of the paper follows immediately from Lemmas 1 and 2.

5 Conclusion

In this paper, we showed that no tile assembly model can use multiple nucleation to solve locally checkable labeling problems in constant time, even though the Winfree-Rothemund Tile Assembly Model can solve a locally checkable labeling problem using just seven tile types. This was the first application of a distributed computing impossibility result to the field of nanoscale self-assembly.

There are still many open questions regarding multiple nucleation. Aggarwal *et al.* asked in [3] whether multiple nucleation might reduce the tile complexity of finite shapes. The answer is not known. Furthermore, we can ask for what class of computational problems does there exist some function f such that we could tile an $n \times n$ square in time $\mathcal{O}(1) < \mathcal{O}(f) < \mathcal{O}(n^2)$, and "solve" the problem with "acceptable" probability of error, in a tile assembly model that permits multiple nucleation. Finally, we hope that this is just the start of a conversation between researchers in distributed computing and biomolecular computation.

Acknowledgements

I am grateful to Soma Chaudhuri, Dave Doty, Jim Lathrop and Jack Lutz for helpful discussions on earlier versions of this paper.

References

1. Adleman, L., Cheng, Q., Goel, A., Huang, M.D.: Running time and program-size for self-assembled squares. In: Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing, pp. 740–748 (2001)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. Series on Parallel and Distributed Computing. Wiley, Chichester (2004)
3. Aggarwal, G., Goldwasser, M., Kao, M.-Y., Schweller, R.: Complexities for Generalized Models of Self-Assembly. In: Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms, pp. 880–889 (2004)

4. Cheng, Q., de Espanes, P.M.: Resolving two open problems in the self-assembly of squares. Technical Report 793, University of Southern California (2003)
5. Lathrop, J., Lutz, J., Patitz, M., Summers, S.: Computability and complexity in self-assembly. In: Logic and Theory of Algorithms: Proceedings of the Fourth Conference on Computability in Europe (to appear, 2008)
6. Lathrop, J., Lutz, J., Summers, S.: Strict self-assembly of discrete Sierpinski triangles. In: Computation and Logic in the Real World: Proceedings of the Third Conference on Computability in Europe, pp. 455–464. Springer, Heidelberg (2007)
7. Majumder, U., Reif, J., Sahu, S.: Stochastic Analysis of Reversible Self-Assembly. In: Computational and Theoretical Nanoscience (to appear, 2008)
8. Naor, M., Stockmeyer, L.: What can be computed locally? SIAM Journal of Computing 24(6), 1259–1277 (1995)
9. Patitz, M., Summers, S.: Self-assembly of discrete self-similar fractals. In: Proceedings of the Seventh International Conference on Unconventional Computation, Springer, Heidelberg (to appear, 2008)
10. Rothemund, P.W.K.: Theory and Experiments in Algorithmic Self-Assembly. Ph.D. thesis, University of Southern California, Los Angeles (2001)
11. Rothemund, P., Winfree, E.: The program-size complexity of self-assembled squares. In: Proceedings of the 32nd Annual ACM Symposium on Theory of Computing, pp. 459–468 (2000)
12. Seeman, N.: Denovo design of sequences for nucleic-acid structural-engineering. Journal of Biomolecular Structure and Dynamics 8(3), 573–581 (1990)
13. Soloveichik, D., Winfree, E.: Complexity of self-assembled shapes. SIAM Journal of Computing 36(6), 1544–1569 (2007)
14. Wang, H.: Proving theorems by pattern recognition II. Bell Systems Technical Journal 40, 1–41 (1961)
15. Winfree, E.: Algorithmic Self-Assembly of DNA. Ph.D. thesis, California Institute of Technology, Pasadena (1998)

Using Bounded Model Checking to Verify Consensus Algorithms

Tatsuhiro Tsuchiya[1,⋆] and André Schiper[2,⋆⋆]

[1] Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
t-tutiya@ist.osaka-u.ac.jp
[2] École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland
Andre.Schiper@epfl.ch

Abstract. This paper presents an approach to automatic verification of asynchronous round-based consensus algorithms. We use model checking, a widely practiced verification method; but its application to asynchronous distributed algorithms is difficult because the state space of these algorithms is often infinite. The proposed approach addresses this difficulty by reducing the verification problem to small model checking problems that involve only single phases of algorithm execution. Because a phase consists of a finite number of rounds, bounded model checking, a technique using satisfiability solving, can be effectively used to solve these problems. The proposed approach allows us to model check some consensus algorithms up to around 10 processes.

1 Introduction

Model checking, a method for formally verifying state transition systems, has now become popular, because it allows the fully automatic analysis of designs. This paper presents an approach to model checking of *asynchronous consensus algorithms*. *Consensus* is central to the construction of fault-tolerant distributed systems. For example, atomic broadcast, which is at the core of state machine replication, can be implemented as a sequence of consensus instances [1]. Other services, such as view synchrony and membership, can also be constructed using consensus [2,3]. Because of the importance, much research has been being devoted to developing new algorithms for this problem.

Model checking of asynchronous consensus algorithms is difficult, because these algorithms usually induce an infinite state space, making model checking infeasible. Sources of infinite state spaces include unbounded round numbers and unbounded message channels. In our previous work [4], we succeeded in model checking several asynchronous consensus algorithms by adopting a round-based computation model, called the *Heard-Of (HO) model* [5], and by devising a finite abstraction of unbounded round numbers. The scalability, however, still needs

⋆ Supported in part by the Grant-in-Aid from MEXT of Japan (no. 20700026).
⋆⋆ Research funded by the Swiss National Science Foundation under grant number 200021-111701 and Hasler Foundation under grant number 2070.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 466–480, 2008.

to be addressed, because the system size that can be model checked is rather small, typically three or four processes.

This paper presents a different approach, which divides the verification problem into several problems that can be solved by model checking. Importantly, these model checking problems can be solved by only analyzing single phases of the execution of the consensus algorithm. Because of this, the time and memory space required for verification can be significantly reduced, resulting in increasing the size of systems that can be verified.

Related Work: The TLA specifications of some *Paxos* algorithms [6,7] were debugged with the aid of TLC, the TLA model checker. The models that were model checked consisted of two or three processes and a small number of rounds [8]. In [9], automatic discovery of consensus algorithms was performed using a procedure that determines if a given decision rule satisfies the safety properties of a single phase. This procedure cannot be used for liveness verification or to verify an entire consensus algorithm. In [10], a synchronous consensus algorithm was model checked for three processes. Studies on model checking of shared memory-based randomized consensus algorithms can be found in [11,12].

Roadmap: Sect. 2 describes the HO model and the consensus problem. Sect. 3 and Sect. 4 describe our proposed model checking techniques for verification of safety and liveness, respectively. Sect. 5 sketches automatic procedures for validating two important assumptions used in the safety and liveness verification. A detailed description can be found in our technical report [13]. Sect. 6 shows the results of case studies. Sect. 7 concludes the paper.

2 Consensus in the Heard-of Model

The HO model [5] is a communication-closed round model that generalizes the asynchronous round model by Dwork et al. [14] with some features of [15] and [16]. Let $\Pi = \{1, 2, \cdots, n\}$ be the set of processes. An algorithm proceeds in phases, each of which consists of $k\,(\geq 1)$ rounds.[1] An algorithm comprises, for each process p and each round r, a sending function S_p^r and a transition function T_p^r. In each round r, every process p sends messages according to $S_p^r(s_p)$, where s_p is the state of p. Then, p makes a state transition according to $T_p^r(Msg, s_p)$, where Msg is the collection of all messages that p has received in round r.

In the HO model both synchrony degree and faults are represented in the form of *transmission faults*. We denote by $HO(p, r)$ $(\subseteq \Pi)$ the set of processes from which p receives a message in round r: $HO(p, r)$ is the "heard of" set of p in round r. A transmission fault is a situation where $q \notin HO(p, r)$ while q sent (or was supposed to send) a message to p in round r. Transmission faults can occur if messages missed a round due to the asynchrony of communication and processing, or if a process or a link is faulty.

[1] In [1] and [17], a round is decomposed in phases. "Round" and "phase" are swapped here to use the classical terminology [14].

Algorithm 1. The *LastVoting* (*Paxos*) algorithm [5]

```
Initialization:
    $x_p \in Val$, initially the proposed value of $p$
    $vote_p \in Val \cup \{?\}$, initially ?
    $commit_p$ a Boolean, initially false
    $ready_p$ a Boolean, initially false
    $ts_p \in \mathbb{N}$, initially 0

Round $r = 4\phi - 3$:
  $S_p^r$ :
    send $\langle x_p , ts_p \rangle$ to $Coord(p, \phi)$
  $T_p^r$ :
    if $p = Coord(p, \phi)$ and
    number of $\langle \nu, \theta \rangle$ received $> n/2$ then
        let $\overline{\theta}$ be the largest $\theta$ from $\langle \nu, \theta \rangle$ received
        $vote_p$ := one $\nu$ such that $\langle \nu, \overline{\theta} \rangle$ is received
        $commit_p$ := true

Round $r = 4\phi - 2$:
  $S_p^r$ :
    if $p = Coord(p, \phi)$ and $commit_p$ then
        send $\langle vote_p \rangle$ to all processes
  $T_p^r$ :
    if received $\langle v \rangle$ from $Coord(p, \phi)$ then
        $x_p := v$ ; $ts_p := \phi$

Round $r = 4\phi - 1$:
  $S_p^r$ :
    if $ts_p = \phi$ then
        send $\langle ack \rangle$ to $Coord(p, \phi)$
  $T_p^r$ :
    if $p = Coord(p, \phi)$ and
    number of $\langle ack \rangle$ received $> n/2$ then
        $ready_p$ := true

Round $r = 4\phi$:
  $S_p^r$ :
    if $p = Coord(p, \phi)$ and $ready_p$ then
        send $\langle vote_p \rangle$ to all processes
  $T_p^r$ :
    if received $\langle v \rangle$ from $Coord(p, \phi)$ then
        DECIDE($v$)
    if $p = Coord(p, \phi)$ then
        $ready_p$ := false
        $commit_p$ := false
```

Consensus is the problem of getting all processes to agree on the same decision. Each process is assumed to have a proposed value at the beginning and is required to eventually decide on a value proposed by some process. In the HO model, consensus is specified by the following three conditions:

Integrity. Any decision value is the proposed value of some process.
Agreement. No two processes decide differently.
Termination. All processes eventually decide.

Note that in the HO model the termination property requires all processes to decide. Discussion of the reason for this specification can be found in [5,18]. For most consensus algorithms, integrity is trivially satisfied; thus we limit our discussion to the verification of agreement and termination.

This computation model can naturally be extended to represent coordinator-based algorithms. Let $Coord(p, \phi)$ denote the coordinator process of process p in phase ϕ. The sending function and the state transition function are now represented as $S_p^r(s_p, Coord(p, \phi))$ and $T_p^r(Msg, s_p, Coord(p, \phi))$, where ϕ is the phase that round r belongs to. *LastVoting* (Algorithm 1) is an example of a coordinator-based consensus algorithm [5]. This algorithm can be viewed as an HO model-version of *Paxos* [17]. It is also close to the $\Diamond\mathcal{S}$ consensus algorithm [1].

Since there is no deterministic consensus algorithm in a pure asynchronous system, some synchrony condition must be assumed to solve the problem. In the HO model such a condition is represented as a predicate over the collections of HO sets $(HO(p, r))_{p \in \Pi, r>0}$ and of coordinators $(Coord(p, \phi))_{p \in \Pi, \phi > 0}$. For example, the following predicate specifies a sufficient condition for the *LastVoting* algorithm to solve consensus:

$$\begin{aligned}&\exists\phi_0 > 0, \exists co \in \Pi, \forall p \in \Pi :\\ &\quad co = \mathit{Coord}(p, \phi_0) \wedge |HO(co, 4\phi_0 - 3)| > n/2 \wedge |HO(co, 4\phi_0 - 1)| > n/2\\ &\quad \wedge\ co \in HO(p, 4\phi_0 - 2) \wedge\ co \in HO(p, 4\phi_0)\end{aligned} \tag{1}$$

In words, phase ϕ_0 is a synchronous phase where: all processes agree on the same coordinator *co*; *co* can hear from a majority of processes in the first and third rounds of that phase; and every process can hear from *co* in the second and fourth rounds. If such a phase ϕ_0 occurs, then all processes will make a decision at the end of this phase. This condition is required only for termination. Agreement can never be violated no matter how bad the HO set is. For simplicity, in the paper we limit our discussion to verification of such algorithms — algorithms that are always safe, even in completely asynchronous runs.

3 Verification of Agreement

Our reasoning consists of two levels. Sect. 3.1 presents the phase-level reasoning, which shows that agreement verification can be accomplished by examining only single phases of algorithm execution. Sect. 3.2 then describes how model checking can be used to analyze the single phases at the round level.

3.1 Phase Level Analysis

At the upper-level of our reasoning, we perform a phase-wise analysis, rather than round-wise. We define a *configuration* as a $(n+1)$-tuple consisting of the states of the n processes and the phase number. Let $\mathcal{C}$ be the set of all possible configurations; that is, $\mathcal{C} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n \times \mathbb{N}^+$ where $\mathcal{S}_p$ is a set of states of a process p and $\mathbb{N}^+$ is a set of positive integers. Given a configuration $c = (s_1, \cdots, s_n, \phi) \in \mathcal{C}$, we denote by $\phi(c)$ the phase number ϕ of c. It should be noted that the state of a process is a value assignment to the variables of the process. Hence any set of configurations can be represented by a predicate over the process variables of all processes and ϕ; that is, the predicate represents a set of configurations for which it evaluates to true. We therefore use the notions of a set of configurations and of such a predicate interchangeably.

Let Val be the set of values that may be proposed. We define a ternary relation $R \subseteq \mathcal{C} \times 2^{Val} \times \mathcal{C}$ as follows: $(c, d, c') \in R$ iff the system can transit from the configuration c at the beginning of phase $\phi(c)$ to the next configuration c' at the beginning of the next phase $\phi(c')$ while deciding the values in d during phase $\phi(c)$. By definition $\phi(c) + 1 = \phi(c')$ if $(c, d, c') \in R$.

Let *Init* be the set of the configurations that can occur at the beginning of phase 1. We define a *run* as an infinite sequence$c_1 d_1 c_2 d_2 \cdots$ $(c_i \in \mathcal{C}, d_i \subseteq Val)$ such that $c_1 \in \mathit{Init}$ and $(c_i, d_i, c_{i+1}) \in R$ for all $i \geq 1$. We let *Run* denote the set of all runs. Let *Reachable* be a set of all configurations that can occur in a run; that is, $\mathit{Reachable} = \{c \mid \exists c_1 d_1 c_2 d_2 \cdots \in \mathit{Run}, \exists i \geq 1 : c = c_i\}$. We say that a configuration c is *reachable* iff $c \in \mathit{Reachable}$. Agreement holds iff:

$$\forall c_1 d_1 c_2 d_2 \cdots \in \mathit{Run} : \left| \bigcup_{i>0} d_i \right| \leq 1 \tag{2}$$

The key feature of our verification approach is that it can determine whether (2) holds or not without exploring all runs. In doing this, the notion of *univalence* plays a crucial role. A configuration is said to be *univalent* if there is only one value that can be decided from this configuration [19]. If the configuration is univalent and v is the only value that can be decided, then the configuration is said to be *v-valent*. Formally, a configuration c_i is *v-valent* iff $\bigcup_{j\geq i} d_j = \emptyset$ or $\bigcup_{j\geq i} d_j = \{v\}$ holds for every sequence $c_i d_i c_{i+1} d_{i+1} \cdots$ such that $\forall j \geq i : (c_j, d_j, c_{j+1}) \in R$.

In the proposed verification approach, we assume that some property, represented by $U(v)$ and Inv, holds on the algorithm under verification. $U(v)$ is shown below. The assumption for Inv will be described later. Indeed, we will have to validate $U(v)$ and Inv.

Assumption 1 ($U(v)$). For any $v \in Val$, $U(v)$ is a set of configurations such that if $c \in U(v) \cap Reachable$, then c is v-valent. In other words, any configuration in $U(v)$ is either (i) reachable and v-valent or (ii) unreachable.

Example 1. Like many other consensus algorithms, *LastVoting* uses a majority quorum of processes to "lock" a decision value. A reachable configuration is v-valent if a majority of processes have the same estimate v for the decision value and have greater timestamps than the other processes. Thus we have:

$$U(v) := \exists Q \subseteq \Pi : (|Q| > n/2 \wedge \forall p \in Q : (x_p = v \wedge \forall q \in \Pi \backslash Q : ts_p > ts_q))$$

Theorem 1. Agreement holds if:

$$\forall c \in Reachable : \forall (c, d, c') \in R : (d = \emptyset \vee \exists v \in Val : (d = \{v\} \wedge c' \in U(v))) \quad (3)$$

Proof. We show that for any $c_1 d_1 c_2 d_2 \cdots \in Run$, (3) implies that for any $l \geq 1$, either (i) $\bigcup_{0<i\leq l} d_i = \emptyset$ or (ii) for some $v \in Val$, $\bigcup_{0<i\leq l} d_i = \{v\}$ and $c_l \in U(v)$, meaning that (2) holds. The proof is by induction on l. Base case: (3) implies that either $d_1 = \emptyset$ or $d_1 = \{v\} \wedge c_2 \in U(v)$ for some $v \in Val$. Inductive step: Suppose that the above (i) or (ii) holds for some $l \geq 1$. If (i) holds for l, then $d_{l+1} = \emptyset$ or $d_{l+1} = \{v\} \wedge c_{l+1} \in U(v)$ for some $v \in Val$. Hence (i) or (ii) holds for $l + 1$. If (ii) holds for l, then $d_{l+1} = \emptyset$ or $d_{l+1} = \{v\}$ since $c_l \in U(v)$. Also $c_{l+1} \in U(v)$ because c_l is v-valent. Thus (ii) also holds for $l + 1$. □

It should be noted that Formula (3) only refers to individual phase-level transitions from *Reachable*, rather than to runs. This property is critical for reducing the verification problem to a model checking problem of single phases. However, directly checking this formula would do little good, because roughly speaking, obtaining *Reachable* is as hard as examining all runs.

The key here is that *Reachable* can be substituted by its *over-approximation*. An over-approximation of the set of reachable states is usually referred to as an *invariant*. That is, a set of configurations is an *invariant* iff it contains all reachable configurations. We assume that an invariant Inv is available; that is, $Inv \subseteq Reachable$ (**Assumption 2**).

Example 2. It is easy to see that for *LastVoting*, the following predicate is always true at the beginning of every phase ϕ:

$$Inv := \forall p \in \mathit{\Pi} : (commit_p = \texttt{false} \wedge \ ready_p = \texttt{false} \wedge \ ts_p < \phi)$$

Theorem 2. Agreement holds if:

$$\forall c \in Inv : \forall (c, d, c') \in R : (d = \emptyset \vee \exists v \in Val : (d = \{v\} \wedge c' \in U(v))) \qquad (4)$$

Proof. Because *Reachable* $\subseteq Inv$, (4) implies (3). By Theorem 1, (3) implies agreement. □

This theorem leads directly to the following verification steps:

Step A1: Check if (4) holds or not.
Step A2: If (4) holds, then agreement holds. Otherwise, further analysis is needed because in this case (i) the consensus algorithm is incorrect or (ii) $U(v)$ or Inv are too small or too large, respectively.

3.2 Model Checking of Single Phases

This section shows how model checking can be used to determine if (4) holds or not. Since our problem involves only single phases, we only need to consider k consecutive state transitions of the consensus algorithm, where k is the number of rounds per phase (see Sect. 2).

The behavior of the system in a single phase, say phase $\mathit{\Phi}$, can be represented as a tuple $(c^1 ho^1 dv^1 c^2 ho^2 dv^2 \cdots c^k ho^k dv^k c^{k+1}, Coord)$, where

- c^i $(1 \leq i \leq k)$ is the configuration at the beginning of the i-th round of the phase, i.e., round $k * (\mathit{\Phi} - 1) + i$, while c^{k+1} corresponds to the first round of the next phase. Hence $\mathit{\Phi} = \phi(c^1) = \phi(c^2) = \cdots = \phi(c^k)$ and $\mathit{\Phi} + 1 = \phi(c^{k+1})$.
- $ho^i = (HO(1, k(\mathit{\Phi} - 1) + i)), \cdots, HO(n, k(\mathit{\Phi} - 1) + i)))$ is a collection of n HO sets – one per process – in the i-th round.
- $dv^i = (dv^i_1, \cdots, dv^i_n)$, where $dv^i_p \in Val \cup \{?\}$ for any $p \in \mathit{\Pi}$, is the collection of values decided by each process in the i-th round. If a process p does not decide in the round, then $dv^i_p =?$.
- $Coord = (Coord(1, \mathit{\Phi}), \cdots, Coord(n, \mathit{\Phi}))$ is a collection of n coordinators – one per process – in phase $\mathit{\Phi}$.

We call such a tuple a *one-phase execution* iff it is consistent with the given consensus algorithm.[2] By definition, $(c, d, c') \in R$ iff there is a one-phase execution $(c^1 ho^1 dv^1 \cdots c^k ho^k dv^k c^{k+1}, Coord)$ such that $c = c^1$, $c' = c^{k+1}$, and $d = \left(\bigcup_{p \in \mathit{\Pi}, 1 \leq i \leq k} \{dv^i_p\}\right) \backslash \{?\}$. Let X denote the set of all one-phase executions that start with a configuration in Inv; that is, a one-phase execution $(c^1 ho^1 dv^1 \cdots c^k \ ho^k dv^k c^{k+1}, Coord)$ is in X iff $c^1 \in Inv$.

[2] Here whether c^1 is reachable or not is irrelevant. In other words, a one-phase execution specifies how the system would behave in a phase, provided that the phase begins with c^1.

Our model checking problem is described as follows: Given (i) an algorithm to be verified, (ii) $U(v)$, and (iii) Inv, determine if the following condition holds for all $(c^1 ho^1 dv^1 \cdots c^k ho^k dv^k c^{k+1}, Coord) \in X$:

$$d = \emptyset \vee \exists v \in Val : (d = \{v\} \wedge c^{k+1} \in U(v)) \tag{5}$$

where $d = \left(\bigcup_{p \in \Pi, 1 \leq i \leq k} \{dv_p^i\}\right) \backslash \{?\}$.

Clearly (4) holds iff (5) holds for all one-phase executions in X.

This model checking problem is unique in that it only concerns exactly k consecutive transitions. Because of this, *bounded model checking* [20] can be most effectively used to solve this problem. As the name suggests, bounded model checking searches state transitions of bounded length. This restriction allows the model checking problem to be reduced to the *satisfiability problem* for a set of constraints in some logic. In our case the constraints are boolean combinations of linear (in)equalities and boolean expressions with integer and boolean variables. The variables involved in the constraints are:

- Variables that represent the values of the process variables at each configuration c^i $(1 \leq i \leq k+1)$. If the process variable is *boolean*, so is the corresponding representing variable; otherwise the representing variable is *integer*.
- An integer variable Φ, which represents the phase number; that is, $\Phi = \phi(c^1) = \phi(c^2) = \cdots = \phi(c^k)$.
- Boolean variables $ho_{p,q}^i (p, q \in \Pi, 1 \leq i \leq k)$, which represent whether p hears of q in the i-th round. That is, $q \in HO(p, k(\Phi - 1) + i)$ iff $ho_{p,q}^i = \mathtt{true}$.
- Integer variables $dv_p^i (p, q \in \Pi, 1 \leq i \leq k)$, which represent the value that is decided by p in the i-th round.
- Integer variables $Coord_p (p \in \Pi)$, which represent the coordinator of p in phase Φ.

Example 3. For the *LastVoting* algorithm, the variables involved in model checking are: $x_p^i, vote_p^i, ts_p^i$ (integer) and $commit_p^i$, $ready_p^i$ (boolean) for $p \in \Pi, 1 \leq i \leq 5$; Φ (integer); $ho_{p,q}^i$ (boolean) for $p, q \in \Pi, 1 \leq i \leq 4$; dv_p^i (integer) for $p \in \Pi, 1 \leq i \leq 4$; $Coord_p$ (integer) for $p \in \Pi$.

It should be noted that any one-phase execution is uniquely represented as a value assignment to these variables. In order to check (5), we proceed as follows. We consider two set of constraints on the above variables:

- $\mathcal{X}$, which represents all one-phase executions in X. That is, $\mathcal{X}$ represents exactly the value assignments to variables corresponding to a one-phase execution in X.
- $\overline{\mathcal{U}}$, which represents the value assignments to variables that correspond to a one-phase execution in X that does *not* meet (5).

$\mathcal{X}$ is derived from the consensus algorithm and Inv, while $\overline{\mathcal{U}}$ is derived from $U(v)$. Note that $\mathcal{X}$ and $\overline{\mathcal{U}}$ can only be simultaneously satisfied by a value assignment corresponding to a one-phase execution that (i) belongs to X *and* (ii) for which (5) does not hold. Therefore every one-phase execution in X meets (5) iff $\mathcal{X} \cup \overline{\mathcal{U}}$ is unsatisfiable.

Step A1 and Step A2 can now be replaced with Step B1 and Step B2, respectively.

Step B1: Check the satisfiability of $\mathcal{X} \cup \overline{\mathcal{U}}$. This check can be done by an off-the-shelf *Satisfiability Modulo Theories* (SMT) solver, such as Yices [21].

Step B2: If no satisfying value assignment exists, then every one-phase execution in X satisfies (5), meaning that (4) holds. As a result, it is guaranteed that agreement holds. On the other hand if there is a satisfying assignment, further analysis is needed to obtain a conclusive answer (see Step A2).

We now show how to construct $\mathcal{X}$ and $\overline{\mathcal{U}}$.

Constraints $\mathcal{X}$: $\mathcal{X}$ is composed as $\mathcal{X} := Dom \cup \mathcal{T}^1 \cup \mathcal{T}^2 \cup \cdots \cup \mathcal{T}^k \cup \mathcal{I}$, where Dom, $\mathcal{T}^i$ and $\mathcal{I}$ are as follows:

Dom restricts the domains of the variables. We represent the set Val of possible proposed values by the set of all positive integers and ? by zero. For the *LastVoting* algorithm, Dom consists of the following constraints:

- $\forall p \in \Pi, \forall i, 1 \leq i \leq 5$: $x_p^i > 0 \wedge vote_p^i \geq 0 \wedge ts_p^i \geq 0$
- $\Phi > 0$
- $\forall p \in \Pi, \forall i, 1 \leq i \leq 4$: $dv_p^i \geq 0$
- $\forall p \in \Pi : 1 \leq Coord_p \leq n$

$\mathcal{T}^i$ represents the i-th round of the algorithm; $\mathcal{T}^i$ is satisfiable iff the system's states – represented by the variable values at the beginning of the ith and i+1-th rounds – are consistent with the algorithm under verification. For example:

$$\mathcal{T}^3 := \forall p \in \Pi : \\ \begin{cases} \quad x_p^3 = x_p^4 \wedge vote_p^3 = vote_p^4 \wedge ts_p^3 = ts_p^4 \wedge commit_p^3 = commit_p^4 \\ \wedge\, ite(Coord_p = p \wedge \bigvee_{Q \in Maj} \bigwedge_{q \in Q} (Coord_q = p \wedge ts_q^3 = \Phi \wedge ho_{p,q}^3 = \texttt{true}), \\ \quad ready_p^4 = \texttt{true},\ ready_p^3 = ready_p^4) \\ \wedge\, dv_p^3 = 0 \end{cases}$$

where $Maj \equiv \{Q \mid Q \subseteq \Pi, |Q| > n/2\}$. The first four terms express that process variables x_p, $vote_p$, ts_p and $commit_p$ do not change in the third round. The ite term[3] represents how variable $ready_p$ is updated: It specifies that $ready_p$ will be updated to `true` if p considers itself the coordinator and receives messages from a majority of processes who consider the coordinator to be p (see lines 24–27 of Algorithm 1). The last term expresses that no decision is made in this round.

$\mathcal{I}$ enforces that $c^1 \in Inv$. For example, consider Inv shown in Example 2 that refers to phase Φ. In this case we have:

$$\mathcal{I} :=\ \forall p \in \Pi : (commit_p^1 = \texttt{false} \wedge\ ready_p^1 = \texttt{false} \wedge\ ts_p^1 < \Phi)$$

[3] $ite(a, b, c) = b$ if $a = \texttt{true}$; $ite(a, b, c) = c$, otherwise.

Constraints $\overline{\mathcal{U}}$*:* The negation of (5) is: $d \neq \emptyset \land \forall v \in Val : \neg(d = \{v\} \land c^{k+1} \in U(v))$ where $d = \left(\bigcup_{p \in \Pi, 1 \leq i \leq k} \{dv_p^i\}\right) \backslash \{?\}$. If $v \notin d$, then $d \neq \{v\}$, which allows us to replace Val with d:

$$d \neq \emptyset \land \forall v \in d : \neg(d = \{v\} \land c^{k+1} \in U(v)) \tag{6}$$

$\overline{\mathcal{U}}$ consists of constraints that represent (6). For example, consider $U(v)$ given in Example 1. Then $\overline{\mathcal{U}}$ consists of the conjunction of the following constraints:

- $\displaystyle\bigvee_{p \in \Pi, 1 \leq i \leq k} dv_p^i \neq 0$
- $\forall p \in \Pi, \forall i, 1 \leq i \leq k :$
  $$dv_p^i \neq 0 \longrightarrow \neg \left(\bigwedge_{q \in \Pi, 1 \leq j \leq k} \begin{pmatrix} dv_q^j = dv_p^i \\ \lor\ dv_q^j = 0 \end{pmatrix} \land \bigvee_{Q \in Maj} \bigwedge_{q \in Q} \begin{pmatrix} x_q^{k+1} = dv_p^i \ \land \\ \bigwedge_{q' \in \Pi \backslash Q} ts_q^{k+1} > ts_{q'}^{k+1} \end{pmatrix} \right)$$

Basically, the second constraint means that some process has decided ($dv_p^i \neq 0$) while the configuration is not univalent ($\neg(...)$).

4 Verification of Termination

As stated in Sect. 2, in the context of the HO model, the condition for termination is specified by a predicate over the collections of HO sets and coordinators. In this section we consider a condition of the form $\exists \phi > 0 : P^{sync}(\phi)$, where $P^{sync}(\phi)$ is a predicate over the HO sets and the coordinators in phase ϕ such that:

- $P^{sync}(\phi)$ is invariant under phase changes; that is, for any $\phi, \phi' > 0$, we have $P^{sync}(\phi) = P^{sync}(\phi')$ if the HO sets and the coordinators in phases ϕ and ϕ' are identical. We henceforth denote $P^{sync}(\phi)$ as P^{sync}.
- P^{sync} is not the constant `false`.

For example, Condition (1) in Sect. 2 is of this form. The verification method described here determines whether the given algorithm always terminates in a phase where P^{sync} holds.

Let $R^{sync} \subseteq \mathcal{C} \times 2^{\Pi} \times \mathcal{C}$ be a ternary relation such that $(c, \pi, c') \in R^{sync}$ iff whenever phase $\phi(c)$ meets P^{sync}, a one-phase execution from c to c' is possible in which π is the set of processes that decide. Hence termination is satisfied if:[4]

$$\forall c \in Reachable : (\forall (c, \pi, c') \in R^{sync} : \pi = \Pi) \tag{7}$$

Theorem 3. Termination holds if:

$$\forall c \in Inv : (\forall (c, \pi, c') \in R^{sync} : \pi = \Pi) \tag{8}$$

Proof. Because $Reachable \subseteq Inv$ (Assumption 2), (8) implies (7). □

[4] Note that we exclude the exceptional case where no $(c, \pi, c') \in R^{sync}$ exists for some c, because P^{sync} is not the constant `false`.

Whether (8) holds or not can be determined using bounded model checking, as was done for agreement verification: Let X^{sync} be the set of all one-phase executions that start with a configuration in Inv, and can occur if P^{sync} holds for the phase. That is, $(c^1 ho^1 dv^1 \cdots c^k ho^k dv^k c^{k+1}, Coord) \in X^{sync}$ iff:

- $c^1 \in Inv$ (hence $X^{sync} \subseteq X$); and
- P^{sync} holds for the HO sets, $ho^1, ho^2, \cdots, ho^k$, and the coordinators, $Coord$.

Hence $(c, \pi, c') \in R^{sync}$ iff there is a one-phase execution $(c^1 ho^1 dv^1 \cdots c^k ho^k dv^k c^{k+1}, Coord) \in X^{sync}$ such that $c = c^1$, $c' = c^{k+1}$, and $\pi = \{p \mid \exists i, 1 \leq i \leq k : dv^i_p \neq ?\}$.

The problem we want to solve is to determine if the following condition (9) holds for all $(c^1 ho^1 dv^1 \cdots c^k ho^k dv^k c^{k+1}, Coord) \in X^{sync}$:

$$\forall p \in \Pi, \exists i, 1 \leq i \leq k : dv^i_p \neq ? \tag{9}$$

In words, (9) states that every process decides in some round of a phase where P^{sync} holds. By definition, (8) holds iff (9) holds for all executions in X^{sync}.

The problem of deciding whether all one-phase executions in X^{sync} satisfy (9) is reduced to the satisfiability problem, as was done in Sect. 3.2. The constraints to be checked are $\mathcal{X} \cup Sync \cup \overline{Term}$, where:

- $Sync$ represents P^{sync}. $Sync$ consists of constraints over ho^i_p and $Coord_p$ ($p \in \Pi, 1 \leq i \leq k$) and is satisfied iff P^{sync} holds for the HO sets and the coordinators represented by ho^i_p and $Coord_p$. As a result, $\mathcal{X} \cup Sync$ represents X^{sync}.
- $\overline{Term}$ is satisfied by a value assignment corresponding to a one-phase execution iff it does *not* meet (9); that is, some process exists that does not decide in the execution. $\overline{Term}$ is composed of only a single constraint $\bigvee_{p \in \Pi} \bigwedge_{1 \leq i \leq k} dv^i_p = 0$.

The constraints $\mathcal{X} \cup Sync \cup \overline{Term}$ can be simultaneously satisfied by, and only by, a value assignment corresponding to a one-phase execution that (i) is in X^{sync} *and* (ii) for which (9) does not hold. Therefore every one-phase execution in X^{sync} satisfies (9) iff no satisfying assignment exists. As a result, termination can be verified as follows:

Step C1: Check the satisfiability of $\mathcal{X} \cup Sync \cup \overline{Term}$.

Step C2: If no satisfying value assignment exists, then every one-phase execution in X^{sync} satisfies (9), meaning that (4) holds. It is therefore guaranteed that termination holds. On the other hand, if the constraints are satisfiable, then it means that (i) the algorithm is incorrect or (ii) Inv is too large. In this case further analysis is required to obtain a conclusive answer.

Example 4. Consider condition (1) for termination of *LastVoting*. We have:

$$Sync := \bigvee_{p \in \Pi} \left(\begin{array}{l} p = Coord_1 = \cdots = Coord_n \\ \wedge \bigvee_{Q \in Maj} \bigwedge_{q \in Q} ho^1_{p,q} = \mathtt{true} \wedge \bigvee_{Q \in Maj} \bigwedge_{q \in Q} ho^3_{p,q} = \mathtt{true} \\ \wedge \bigwedge_{q \in \Pi} ho^2_{q,p} = \mathtt{true} \wedge \bigwedge_{q \in \Pi} ho^4_{q,p} = \mathtt{true} \end{array} \right)$$

5 Validating *Inv* and $U(v)$

So far we have assumed that $U(v)$ (see Example 1) and Inv (see Example 2) satisfy Assumption 1, respectively Assumption 2 (see Sect. 3.1). Here we present automatic procedures for checking that $U(v)$ and Inv indeed satisfy the corresponding assumptions. Because of space limitations, we refer the reader to our technical report [13] for details.

5.1 Validating *Inv*

Here we remove Assumption 2; that is, it is not known whether or not Inv is an invariant.

Theorem 4. Suppose that Inv is a set of configurations. Inv is an invariant if the following two conditions hold:

$$Init \subseteq Inv \tag{10}$$

$$\forall c \in Inv : (\forall (c, d, c') \in R : c' \in Inv) \tag{11}$$

Testing (10) can be straightforwardly conducted using satisfiability solving, because it suffices to simply check if there is a configuration that is in $Init$ but not in Inv.

Testing whether (11) holds or not can be done by determining if $c^{k+1} \in Inv$ for all $(c^1 ho^1 dv^1 \cdots c^k ho^k dv^k c^{k+1}, Coord) \in X$. This problem can be reduced to the satisfiability problem of $\mathcal{X} \cup \overline{\mathcal{I}}'$, where $\overline{\mathcal{I}}'$ is a constraint set that is satisfied iff $c^{k+1} \notin Inv$. Thus if there is no satisfying solution, then every one-round execution in X ends with a configuration in Inv, i.e., (11) holds.

5.2 Validating $U(v)$ with Assumption 2

Here we remove Assumption 1; that is, it is not known that $U(v)$ represents v-valent configurations. However, we assume that Inv correctly represents an invariant; that is, $Reachable \subseteq Inv$ (Assumption 2).

Theorem 5. Suppose that $v \in Val$ and $U(v)$ is a set of configurations. Any $c \in U(v) \cap Reachable$ is v-valent if:

$$\forall c \in Inv : \forall (c, d, c') \in R : c \in U(v) \longrightarrow d \in \{\emptyset, \{v\}\} \wedge c' \in U(v) \tag{12}$$

Again, bounded model checking can be used to check if (12) holds for all $v \in Val$. The problem to be solved is to determine whether or not for any one-phase execution $(c^1 ho^1 dv^1 \cdots c^k ho^k dv^k c^{k+1}, Coord) \in X$, the following condition holds:

$$\forall v \in Val : c^1 \in U(v) \longrightarrow (d = \emptyset \vee d = \{v\}) \wedge c^{k+1} \in U(v) \tag{13}$$

where $d = \left(\bigcup_{p \in \Pi, 1 \leq i \leq k} \{dv_p^i\}\right) \backslash \{?\}$.

The problem is reduced to the satisfiability problem of $\mathcal{X} \cup \overline{\mathcal{V}}$, where $\overline{\mathcal{V}}$ is a constraint set that is satisfied by a one-phase execution in X iff it does *not* meet (13). Therefore if $\mathcal{X} \cup \overline{\mathcal{V}}$ is unsatisfiable, then (13) holds for any execution in X, ensuring that any $c \in U(v) \cap Reachable$ is v-valent.

Algorithm 2. The *Hybrid-1(α)* algorithm ($\alpha \leq \lfloor n/4 \rfloor$)

```
Initialization:
    $x_p \in Val$, initially the proposed value of $p$
    $vote_p \in Val \cup \{?\}$, initially ?
    $voteToSend_p$ a Boolean, initially false
    $ts_p \in \mathbb{N}$, initially 0

Round $r = 3\phi - 2$:
  $S_p^r$ :
    send $\langle x_p\ , ts_p, Coord(p,\phi)\rangle$ to all processes
  $T_p^r$ :
    if $(\phi = 1)$ and $\#\langle -,-,-\rangle \geq n - \alpha$ then
        if $n - \alpha$ messages received are equal to $\langle \overline{x}, -, -\rangle$ then
            DECIDE($\overline{x}$)
    if $p = Coord(p,\phi)$ and $\#\langle -,-,p\rangle$ received $> \max(n/2, 2\alpha)$ then
        if the messages received are all equal to $\langle -, 0, p\rangle$ and, except at most $\alpha$, are equal to $\langle \overline{x}, 0, p\rangle$ then
            $vote_p := \overline{x}$
        else
            let $\overline{\theta}$ be the largest $\theta$ from $\langle -, \theta, p\rangle$ received
            $vote_p :=$ one $\overline{x}$ such that $\langle \overline{x}, \overline{\theta}, p\rangle$ is received
        $voteToSend_p :=$ true

Round $r = 3\phi - 1$:
  $S_p^r$ :
    if $p = Coord(p,\phi)$ and $voteToSend_p$ then
        send $\langle vote_p \rangle$ to all processes
  $T_p^r$ :
    if received $\langle v \rangle$ from $Coord(p,\phi)$ then
        $x_p := v$ ; $ts_p := \phi$

Round $r = 3\phi$:
  $S_p^r$ :
    if $ts_p = \phi$ then
        send $\langle ack, x_p\rangle$ to all processes
  $T_p^r$ :
    if $\exists v$ s.t. $\#\ \langle ack, v\rangle$ received $> n/2$ then
        DECIDE($v$)
    $voteToSend_p :=$ false
```

6 Case Studies

In this section we present the results of applying the proposed approach to two consensus algorithms:

- *LastVoting* (*Paxos*) [5], see Algorithm 1.
- *Hybrid-1(α)*, an improved version of the *Fast Paxos* algorithm [7], presented in [22], see Algorithm 2.

For *LastVoting*, the condition for termination is specified by (1) in Sect. 2, while $U(v)$ and *Inv* are given in Examples 1 and 2.

Similarly to *LastVoting*, *Hybrid-1(α)* is always safe if $\alpha \leq \lfloor n/4 \rfloor$. The condition for termination is specified in Table 1(a); $U(v)$ and *Inv* are given in Table 1(b). *Hybrid-1(α)* combines a fast phase and an ordinary phase of *Fast Paxos* into the same phase. In the first round of Phase 1, if a process has received the

Table 1. Properties for *Hybrid-1(α)*

(a) Condition for Liveness

$\exists \phi > 0, \exists co \in \Pi, \forall p \in \Pi : \begin{cases} \lvert HO(p,r)\rvert > \max(n/2, 2\alpha) \wedge co = Coord(p,\phi) \\ \wedge\ \forall 0 \leq i \leq 2 : co \in HO(p, 3\phi - i) \end{cases}$

(b) $U(v)$ and *Inv*

$U(v) := \begin{cases} \exists Q \subseteq \Pi : \lvert Q\rvert \geq n - \alpha \ \wedge \forall p \in Q : (x_p = v) \wedge \forall p \in \Pi \backslash Q : (ts_p = 0) \\ \vee\ \exists Q \subseteq \Pi : \lvert Q\rvert > n/2 \ \wedge \forall p \in Q : (x_p = v \wedge \forall q \in \Pi \backslash Q : ts_p > ts_q) \end{cases}$
$Inv := \forall p \in \Pi : voteToSend_p = \texttt{false} \wedge ts_p < \phi$

Table 2. Execution time (in seconds)

	LastVoting								*Hybrid-1($\lfloor n/4 \rfloor$)*							
n	4	5	6	7	8	9	10	11	4	5	6	7	8	9	10	11
agreement	0.1	0.3	0.6	5.0	10	139	312	11066	0.1	0.8	2.2	57	539	11848	t.o.	t.o.
termination	0.0	0.1	0.1	0.5	1.0	3.6	7.6	32	0.0	0.1	0.3	1.1	3.2	13	28	121
$U(v)$	0.1	0.6	1.7	104	2356	t.o.	t.o.	t.o.	0.3	2.3	8.2	142	16650	t.o.	t.o.	t.o.
Inv	0.0	0.1	0.1	0.4	0.9	2.7	5.2	29	0.0	0.1	0.3	0.8	2.7	10	26	92

Table 3. Traditional approach on *LastVoting* with different model checkers

Model Checker	n	Time (sec)
NuSMV [23]	4	167
SPIN [24]	3	2922
ALV [25]	3	1921

same estimate from $n - \alpha$ processes, then the process can immediately decide (line 12). To prevent processes from deciding different values in later rounds (line 33), if $n - \alpha$ or more processes have the same proposed value, then it must be guaranteed that no process having a different estimate will be allowed to update its timestamp. That is, when at least $n - \alpha$ processes have the same estimate v and the remaining processes have a timestamp equal to zero, the system must be v-valent. The first term of the $U(v)$ in Table 1(b) states this formally.

Experiments: The experiments were performed on a Linux workstation with an Intel Xeon processor 2.2GHz and 4Gbyte memory. We used the Yices [21] satisfiability solver. For both algorithms, we conducted the four kinds of checks, namely agreement, termination, $U(v)$, and Inv, up to $n = 11$. Table 2 shows the execution time required for these checks. Notation "t.o." (timeout) indicates that the check was not completed within 5 hours.

No satisfiable solution was found in any of the checks. Therefore Table 2 shows that for the two algorithms (i) $U(v)$ and Inv meet Assumption 1 up to $n = 8$, Assumption 2 up to $n = 11$, and (ii) agreement and termination are guaranteed up to $n = 8$ and $n = 11$, respectively.

Traditional approach: For comparison, we evaluated a different approach in which the whole state space of an algorithm is explored with an existing model checker. Specifically we verified *LastVoting* against the agreement property with three model checkers: NuSMV [23], SPIN [24], and ALV [25].

SMV and SPIN are the two best known model checkers. NuSMV is one of the latest implementations of SMV. Although SMV and SPIN can only deal with finite state systems, the abstraction technique proposed in [4] allows one to model check the whole state space with these model checkers. In model checking with SPIN we made an extensive optimization to reduce the state space. For example, we completely removed the information on HO sets from the state space, by specifying all possible behaviors as non-deterministic ones. This optimization does not work for SMV because HO sets are already compactly represented by the data structure used in SMV. ALV is a well known model checker that can analyze infinite state systems with unbounded integer variables. It does not require any finite state abstraction, while the same optimization as with SPIN was performed to avoid explicit representation of HO sets.

Table 3 presents the maximum number of processes that each of the model checkers was able to handle without running out of memory or time (5 hours) together with the time needed to model check the largest model. Comparing Tables 2 and 3 one can clearly see that the proposed approach scales much better than the approach using existing model checkers. This improvement can be explained by the fact that our approach can avoid explosive growth of the search space by limiting it to single phases.

7 Conclusion

We proposed an approach to automatic verification of asynchronous consensus algorithms. Using the notions of univalence and invariant, we reduced agreement and termination verification to the problem of model checking only single phases of the algorithm. This unique property of the model checking problem allowed us to effectively use bounded model checking. As case studies we applied the proposed approach to two consensus algorithms and mechanically verified that they satisfy agreement up to 8 processes and termination up to 11 processes. Comparing the performance of the traditional model checking approach showed that the performance improvement was significant.

References

1. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of ACM 43(2), 225–267 (1996)
2. Guerraoui, R., Schiper, A.: The generic consensus service. IEEE Trans. on Software Engineering 27(1), 29–41 (2001)
3. Schiper, A., Toueg, S.: From set membership to group membership: A separation of concerns. IEEE Transactions on Dependable and Secure Computing (TDSC) 3(1), 2–12 (2006)
4. Tsuchiya, T., Schiper, A.: Model checking of consensus algorithms. In: Proc. 26th Symp. on Reliable Distributed Systems (SRDS), Beijing, China, pp. 137–148 (October 2007)
5. Charron-Bost, B., Schiper, A.: The Heard-Of model: Computing in distributed systems with benign failures. Technical Report LSR-REPORT-2007-001, EPFL (2007)
6. Gafni, E., Lamport, L.: Disk Paxos. Distributed Computing 16(1), 1–20 (2003)
7. Lamport, L.: Fast Paxos. Distributed Computing 19(2), 79–103 (2006)
8. Lamport, L.: Personal Communication (2006)
9. Zieliński, P.: Automatic verification and discovery of Byzantine consensus protocols. In: Proc. Int'l Conf. on Dependable Systems and Network (DSN 2007), Edinburgh, UK, pp. 72–81. IEEE CS Press, Los Alamitos (June 2007)

10. Hendriks, M.: Model checking the time to reach agreement. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 98–111. Springer, Heidelberg (2005)
11. Cheung, L.: Randomized Wait-Free Consensus Using an Atomicity Assumption. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 47–60. Springer, Heidelberg (2005)
12. Kwiatkowska, M.Z., Norman, G., Segala, R.: Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 194–206. Springer, Heidelberg (2001)
13. Tsuchiya, T., Schiper, A.: Using bounded model checking to verify consensus algorithms. Technical Report LSR-REPORT-2008-005, EPFL (July 2008)
14. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of ACM 35(2), 288–323 (1988)
15. Gafni, E.: Round-by-round fault detectors: Unifying synchrony and asynchrony. In: Proc. 17th ACM Symp. on Principles of Distributed Computing (PODC-17), pp. 143–152. ACM Press, New York (1998)
16. Santoro, N., Widmayer, P.: Time is not a healer. In: Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1989), Paderborn, Germany. LNCS, vol. 346, pp. 304–313. Springer, Heidelberg (1989)
17. Lamport, L.: The part-time parliament. ACM Trans. on Computer Systems 16(2), 133–169 (1998)
18. Charron-Bost, B., Schiper, A.: Harmful dogmas in fault tolerant distributed computing. SIGACT News 38(1), 53–61 (2007)
19. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of ACM 32(2), 374–382 (1985)
20. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
21. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
22. Charron-Bost, B., Schiper, A.: Improving Fast Paxos: Being optimistic with no overhead. In: Proc. of 12th Pacific Rim International Symposium on Dependable Computing (PRDC 2006), Riverside, CA, USA, pp. 287–295. IEEE CS Press, Los Alamitos (2006)
23. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404. Springer, Heidelberg (2002)
24. Holzmann, G.J.: The model checker SPIN. IEEE Trans. on Software Engineering 23(5), 279–295 (1997)
25. Bultan, T., Yavuz-Kahveci, T.: Action language verifier. In: Proc. 16th IEEE Int'l Conf. on Automated Software Engineering (ASE 2001), San Diego, CA, USA, pp. 382–386 (2001)

Theoretical Bound and Practical Analysis of Connected Dominating Set in Ad Hoc and Sensor Networks

Alireza Vahdatpour, Foad Dabiri, Maryam Moazeni, and Majid Sarrafzadeh

Computer Science Department
University of California Los Angeles
{alireza,dabiri,mmoazeni,majid}@cs.ucla.edu

Abstract. *Connected dominating set* is widely used in wireless ad-hoc and sensor networks as a routing and topology control backbone to improve the performance and increase the lifetime of the network. Most of the distributed algorithms for approximating connected dominating set are based on constructing *maximal independent set*. The performance of such algorithms highly depends on the relation between the size of the maximum independent set ($mis(G)$) and the size of the minimum connected dominating set ($cds(G)$) in the graph G. In this paper, after observing the properties of such subgraphs, we decrease the previous ratio of 3.453 to 3.0 by showing that $mis(G) \leq 3 \cdot mcds(G) + 3$. Additionally, we prove that this bound is tight and cannot be improved. Finally, we present practical analysis of constructing connected dominating set based on maximal independent set in wireless networks. It is shown that the theoretical bound for *unit disk graph* is still practically applicable for general wireless networks.

1 Introduction

Wireless ad-hoc and sensor networks are gaining more interest in a variety of applications. In addition to applications such as habitat and environment monitoring, these networks are useful in emergency operations such as search and rescue, crowd control and commando operations [15]. In such wireless networks, each node communicates with other nodes either directly or through other intermediate nodes. These networks can not have fixed or centralized infrastructure, since network nodes are not assumed to be fixed in the environment. Although a wireless ad-hoc network has no physical infrastructure, connected dominating set can be used as a virtual backbone in the network [19]. A connected dominating set (CDS) of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that each node in $V - S$ is adjacent to some nodes in S and the communication graph induced by S is connected. Centralized and distributed construction of dominating set in sensor networks has been studied widely before, since it has great impact on the efficiency of routing, power management, and topology control protocols.

The problem of finding a minimum CDS (MCDS) in a graph has been shown to be NP-hard [2]. Many efforts [1], [6], [7], [18], [19], [20] have been made to design

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 481–495, 2008.

approximations and heuristics for the MCDS in a network. Among distributed approximation algorithms, which are the most suitable algorithms for ad-hoc and sensor networks, using the idea of constructing a maximal independent set and then converting it to a connected graph is a common trend. This approach has several advantages comparing to centralized and other distributed MCDS approximation algorithm. Sensor and ad-hoc networks have severe limitation in processing and communication capabilities, therefore, localized distributed algorithms are the most suitable solutions for issues arising from inefficient energy consumption in the network. Since every maximal independent set is a dominating set and can be constructed locally, most of the well-known algorithms usually construct a maximal independent set at the first step and then convert it to a CDS with a local algorithm. Consequently, the approximation performance ratio would be determined by two factors: 1) How large a maximal independent set can be compared to a minimum connected dominating set. 2) How many vertices are required to connect a maximal independent set. Simple algorithms are available that construct a connected graph from maximal independent set by the ratio of 2 [13], [19].

Several studies have been done to find the ratio of number of nodes in MIS to the size of MCDS in a graph. All the studies assume the graph to be unit disk graph. A *unit disk graph (UDG)* is the intersection graph of a family of unit circles in the Euclidean plane. In such graph, a vertex v is connected to vertex u, if and only if $|uv| \leq 1$, where $|uv|$ represents the Euclidean distance between u and v. Even assuming the graph to be UDG, finding MCDS has been proven to be NP-Hard [2]. Using the geometric properties of unit disk graph, [13], [19], [10], [21], [14] have shown different constant ratios for the relation between number of nodes in MCDS and MIS subgraphs. The study in [21] showed that in every unit disk graph: $mis(G) \leq 3.8cds(G) + 1.2$. where $mis(G)$ is the size of a maximum independent set and $cds(G)$ is the size of a minimum connected dominating set in G. More recently, Funke et. al. [10] decreased the bound to $3.453 \cdot cds(G) + 8.291$. In this paper, we decrease the bound and show that $mis(G) \leq 3 \cdot cds(G) + 3$. Additionally, we analyze the ratio in practical networks, where the network does not necessarily obey the characteristics of a unit disk graph. Through running large set of simulation experiments, we observed that although theoretically the bound is not applicable to a general network where nodes have indeterministic wireless coverage. Furthermore, we analyze the effect of network density and connectivity on the construction of connected dominating set and its practical relation to maximal independent set in wireless networks.

The rest of this paper is organized as follows. Section 2, discusses the applications and usage of MCDS in sensor and ad-hoc networks. In section 3, related work and observations that lead to the main results of this paper are presented. Section 4 presents the main results which includes the proof of the new bound for the relation between the number of nodes in MCDS and MIS. Section 5 includes experimental results for practical analysis and observations. Finally, the last section contains some concluding remarks.

2 Applications of Connected Dominating Set in Sensor and Ad Hoc Networks

Several approaches have been proposed in the literature to maximize the network lifetime, as solutions to great challenges that energy constraints impose on sensor and ad-hoc networks. Using connected dominating set as a network backbone for routing and topology control is one of the most mature approaches that can lead to efficient energy consumption in the network. In this approach, messages can be routed from the source to one of its neighbors in the dominating set, along the CDS to the dominating set member closest to the destination node, and finally to the destination [12]. *Dominating set based routing* [7], [20], *spine based routing* [5] or *backbone based routing* [3] are the terms used in the literatures particularly addressing this routing method.

Furthermore, CDS has been widely used to increase the efficiency of multicast and broadcast routing. According to [12], [16] *broadcast storm problem* occurs when a large number of intermediate nodes unnecessarily forwards a message. Therefore, causing several nodes to hear the same message several times, which causes unwanted energy consumption in the network. In another approach CDS is applied to increase the number of nodes that can be in the sleep mode, while the connectivity of the network is maintained and messages can effectively be forwarded throughout the network [17], [20]. In [8] the nodes in dominating set coordinate with one another and use orthogonal spreading codes in their neighborhoods to improve spatial reuse with code division spread spectrum techniques.

Recently, Banik and Radhakrishnan [4] discussed that while using MCDS as backbone for broadcasting minimizes the the total number of transmissions, it does not guarantee minimum broadcast latency. Hence, they proposed a collision free broadcast scheduling algorithm among nodes in MCDS to minimize the broadcast latency.

Different approaches are applied to construct connected dominating set over a network. Centralized algorithms are not suitable for sensor and ad-hoc networks since they require one node to perform most of the computation and communication, which leads to unbalanced energy consumption in the network. In contrast, a common distributed approach that has been addressed by [13] for the first time, is to use maximal independent set to form the connected dominating set. This approach is popular and efficient among distributed CDS algorithms since distributed construction of maximal independent set in large networks needs few number of local communications among the nodes. Currently, the best known approximation algorithm for connected dominating set uses the above approach [14]. [21], [19], [13] investigated the relation of minimum connected dominating set and maximal independent set. The common assumption in most of the work proposing algorithms for constructing MCDS over the network is that a wireless network can be modeled as a Unit Disk Graph (UDG). Exploiting the geometric properties of unit disk graphs, introducing effective approximation algorithms for MCDS is possible. [13] has addressed few simple heuristics for unit disk graphs that leads to acceptable approximation bounds.

3 Motivation, Observations and Related Work

Table 1 summarizes performance comparison of well known distributed CDS construction algorithms [12]. The best performance of distributed algorithms is achieved by [14] where it first finds a maximal independent set and then converts it to a connected dominating set. Finding maximal independent set and converting it to a dominating set is relatively an easy task in a graph, therefore, most of distributed algorithms try to efficiently convert the dominating set constructed by maximal independent set to a connected tree which has the minimum cardinality from the optimal MCDS. In most of these conversion algorithms the size of the connected tree is at most twice of the input dominating set. Hence, the final approximation ratio of the connected dominating set is highly depended on the approximation ratio of the size of maximal independent set to the size of the MCDS.

In [13] authors assumed Unit Disk Graphs (UDG) and showed that in this type of geometric graphs, the ratio of number of nodes in MIS to the number of nodes in MCDS is at most 5. In a UDG there is an edge between two vertices if the distance between them is less than one unit. Since wireless sensor/ad-hoc networks are usually considered to be consisted of homogeneous nodes with equal wireless ranges, using UDG for the theoretical model is reasonable. Environment effects and slight geometric conditions are ignored in UDG, however. Modeling a wireless network as a UDG, each vertex of MCDS cannot dominate more than five independent nodes, since no more than five independent nodes can lie in a unit disk area (Figure 1.a). This immediately results in that the maximum ratio of nodes in every MIS is not more than five times of the size of the MCDS. [19] decreased the ratio to 4, by showing that the intersection between the dominating areas (communication coverage) of each two adjacent nodes does not let every vertex in MCDS to dominate five independent nodes. Figure 1.b shows part of a minimum connected dominating set. v and u are two adjacent vertices in MCDS. It can be seen that U_v lie in a sector of at most 240° within the coverage range of vertex v. It can be easily shown that at most 4 independent nodes can lie in this area.

Table 1. Performance comparison of previous CDS construction distributed algorithms [12] (Here, n and m are the number of vertices and edges in the graph. Δ and H denote the maximum degree of the vertices in the graph and the harmonic function, respectively.)

	Graph Model	Approximation Ratio	Time Complexity	Message Complexity
[7]-I	general	$2H(\Delta)+1$	$O((n+\|C\|)\Delta)$	$O(n\|C\|+m+nlog(n))$
[7]-II	general	$2H(\Delta)$	$O(\|C\|(\Delta+\|C\|))$	$O(n\|C\|)$
[19]	UDG	$8opt+1$	$O(n)$	$O(nlog(n))$
[1]	UDG	$192opt+48$	$O(n)$	$O(n)$
[6]	UDG	$8opt$	$O(n)$	$O(nlog(n))$
[20]	general	$O(n)$	$O(\Delta^3)$	$\theta(m)$
[14]	UDG	$6.8opt$	-	-

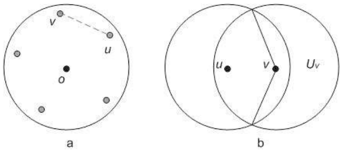


Fig. 1. a) A neighboring area with five independent nodes **b)** U_v lies in a sector of at most 240°

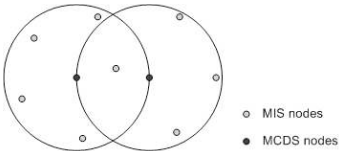


Fig. 2. An example where 8 nodes lie in the neighboring area of two vertices

[21] has extended the observation and proved that the number of nodes in the MIS is at most $3.8cds(G)+1.2$. Using this upper-bound, the approximation ratio of the algorithms [1], [6], [19] that use MIS to find an approximate connected dominating set on a network would be 7.6. The key observation in [21] is that the neighboring area around two adjacent nodes in MCDS cannot have more than eight independent nodes. Figure 2 depicts an example where eight nodes lie in the neighboring area of two adjacent nodes in MCDS.

Recently, [10] et al. presented a better bound using Voronoi diagram of the vertices in the MCDS. Up to now, $3.453 \cdot cds(G) + 8.291$ achieved by this paper is the best bound. In the next section, we will extend UDG properties discussed before and show that the ratio of number of nodes in MIS to the number of nodes in MCDS is $3 \cdot cds(G) + 3$. The idea behind the proof in the next section is to apply the previous observation to a number of nodes connected together. It will be shown that the number of independent nodes in one vertex's neighboring area depends on the position and the number of independent nodes in the adjacent MCDS vertices' neighboring areas and by generalizing the relation, the minimum upper bound can be driven.

4 Main Results

In this section, we will present our contribution which is a tight bound on the approximation ratio between MIS and $MCDS$.

Definition 1. *For any node $u \in G(V, E)$, the neighboring set of u, $N(u)$ is defined to be the set of nodes adjacent to u in G, in other words:*

$$N(u) = \{v | e_{uv} \in E\}$$

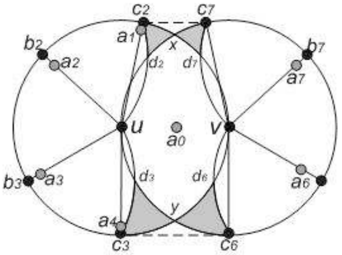


Fig. 3. Four small dark areas

Lemma 1. *For any $u \in G$:*

$$|N(u) \cap mis(G)| \leq 5 \tag{1}$$

where $mis(G)$ is a maximal independent set of graph G.

Proof. If u and v are independent ($|uv| > 1$), it is depicted in figure 1.a, that the angle made by connecting vertices v and u to vertex o is greater than 60°. Therefore, at most five mutually independent nodes can exist in a unit disk. □

Lemma 2. *Each two adjacent nodes in MCDS cannot have more than 8 independent nodes in their neighboring area:*

$$\forall u, v \in mcds(G), |uv| \leq 1; |(N(u) \cup N(v)) \cap mis(G)| \leq 8 \tag{2}$$

Proof. The complete proof can be found in [21]. However, since the next lemma is a generalization of this lemma, we will briefly overview the proof scheme here. Figure 3 depicts two adjacent vertices u and v, and their neighboring area. Without loss of generality, we assume that $|uv| = 1$, therefore, $N(u) \cup N(v)$ is maximum. First, if two independent nodes lie in the $N(u) \cap N(v)$, according to lemma 1, $N(u) - N(v)$ and $N(v) - N(u)$ can include at most three independent nodes, resulting in the total of at most 8 independent nodes in the $N(u) \cup N(v)$.

In the second case, assume that only node $a_0 \in mis(G)$ lies in $N(u) \cap N(v)$. In order to have more than 8 independent nodes in $N(u) \cup N(v)$, four nodes must lie in both $N(v) - N(u)$ and $N(u) - N(v)$. Lets assume that four independent nodes (a_1, a_2, a_3, a_4) lie in $N(u) - N(v)$. Considering that a_6 and a_7 are lying in $N(v) - N(u)$ and the fact that the sector between each two independent nodes in a circle is at least 60° (see lemma 1), a_5 and a_8 must lie in the dark areas yd_6c_6 and xd_7c_7.

According to the proof available in [21], the maximum euclidian distance between any point of c_3d_3y and any point of d_6c_6y is less than unit. Similarly, points in c_2d_2x and c_7d_7x has the same property accordingly. Therefore the dark areas in $N(v)$ cannot include independent nodes and hence four nodes cannot lie in $N(v) - N(u)$ This completes the proof. □

Definition 2. ***Semi-exclusive neighboring set:*** *let U be any set of independent nodes in $G = (V, E)$, and T be any spanning tree on $mcds(G)$. Assume*

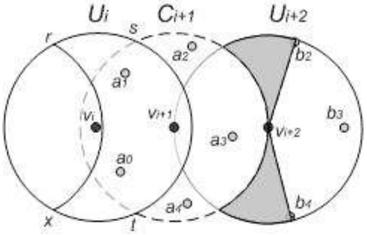


Fig. 4. Three adjacent vertices in MCDS and their neighboring areas

$v_1, v_2, ..., v_T$ is an arbitrary preorder traversal of T. For any i, $2 \leq i \leq |T|$, consider $U_i = N_i \cap U - \bigcup_{j=1}^{i-1} N(j)$ be the subset of nodes in U that are adjacent to v_i but not to any of $v_1, v_2, ..., v_{i-1}$. We will call U_i the semi-exclusive neighboring set of node v_i.

Lemma 3. *For all $i, j \in mcds(G), |U_i| = |U_j| = 4$ there exists a node $k \in mcds(G)$ on the path between i and j, where $|U_k| < 3$.*

Proof. Lets $v_0, v_1, v_2, ..., v_m$ be an arbitrary path in the $mcds(G)$ and $N_1, N_2, ..., N_m$ and $U_1, U_2, ..., U_m$ be the neighboring and semi-exclusive sets associated with vertices $v_1, v_2, ..., v_m$. Suppose that U_i and U_j contain four independent nodes. We will show that there exists v_k, $i < k < j$ such that U_k contains less than three independent nodes.

Figure 4 shows vertex v_i and its three immediate neighbor vertices in $mcds(G)$, v_{i-1}, v_{i+1} and v_{i+2}. As seen in Figure 4, since $\widetilde{st} \geq 120°$ and $\widetilde{rx} \leq 240°$, $N_{i+1} \cap U_i$ will include either three or two independent nodes. In the first case, according to lemma 1 $|U_{i+1}| \leq 2$ and this satisfies the lemma. In the second case, assume $a_0, a_1 \in N_{i+1} \cap Ui$. According to lemma 2, U_{i+1} can contain at most three independent nodes a_2, a_3 and a_4 (otherwise, by putting one independent node in $\widetilde{rx}$, $|N_{i-1} \cup N_i|$ would be greater than six). It is obvious that $a_4\widehat{v_{i+1}}a_2 > 180°$ (note that $a_4\widehat{v_{i+1}}a_2$ is the angle obtained by **counterclockwise** rotation of $v_{i+1}a_2$ toward $v_{i+1}a_4$). Assuming $|U_{i+1}| = 3$, either one or two of the independent nodes in it will also be in N_{i+2}. In the case that $|N_{i+2} \cap U_{i+1}| = 2$, N_{i+2} has the same property as of N_{i+1} and we can continue the reasoning with the relation between N_{i+3} and U_{i+2}.

If only one independent node lies in the intersection area (as it is assumed in figure 4,) since **clockwise** angle between a_2v_{i+1} and a_4v_{i+1} is less then 180°, according to the proof available in [21], independent nodes in U_{i+2} have to be inside the $\widetilde{b_2b_4}$ sector area. Since $b_1\widehat{v_{i+1}}b_2 \leq 180°$, at most three independent nodes can lie in it. Therefore, when $|U_{i+2}| = 3$, the same angular property of these nodes enforces the same condition on the nodes in U_{i+3}.

The above reasoning can be extended to all U_h, $h > i$, while $\forall s, i \leq s < h$, U_s contains exactly three independent nodes. This immediately results in the fact that for all $v_j, j > i$, $|U_j|$ is at most three unless the semi-exclusive neighboring set of a vertex on the path from v_i to v_j contains less than three independent nodes. □

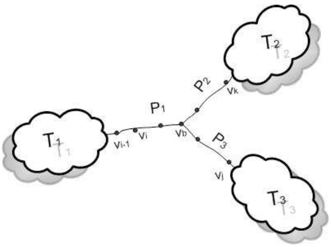


Fig. 5. Fork subgraph

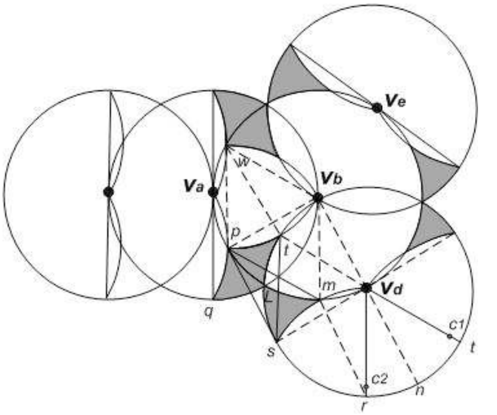


Fig. 6. Five vertices in the branch location and their neighboring areas

Lemma 4. *Assume a preorder traversal of the vertices in $mcds(G)$. The fork subgraph and dominated independent nodes with the following properties doesn't exist:(figure 5, the root for traversal is assumed to be in T_1)*

- *v_b is the vertex where paths from v_i to v_j and v_k get separated.*
- $|U_i| = |U_j| = |U_k| = 4$, $|U_b| = 2$
- *For every vertex v_h on the path from v_b to v_i, v_j or v_k,* $|U_h| = 3$

Proof. Figure 6 depicts v_b and its immediate neighbors in $mcds(G)$. Without loss of generality, assume $\widehat{v_d v_b v_e} = \widehat{v_d v_b v_a} = \widehat{v_a v_b v_e} = 120°$ (we have shown in the appendix why this assumption does not cause loss of generality.) Three cases are possible for two independent nodes in U_b:

First, assume $|U_b \cap N_e| = 2$ and $|U_b \cap N_d| = 0$. As an immediate consequence, the vertices in the path followed by vertex v_e has the specification of the vertices in lemma 3 and for every v_h on the path followed from v_b, the semi-exclusive neighboring set U_h contain at most three independent nodes while for all v_s on the path between v_e and v_h, $|U_s| = 3$. This completes the proof of the lemma, assuming the first case.

In the second case, assume $|U_b \cap N_e| = 2$ and $|U_b \cap N_d| = 1$. Since the path followed by vertex v_e has the same condition of the previous case, the lemma is also true in this case.

In the third case, assume $|U_b \cap N_e| = 1$ and $|U_b \cap N_d| = 1$. We will show that the location of dark areas in figure 6 is such that either v_d or v_e has the properties of the nodes in lemma 3 (the nodes in semi-exclusive set are limited to a 180° sector), and consequently, the path continued from it cannot lead to a U_k with four dominating independent nodes unless there is a vertex v_h on the path between v_b and v_k, where U_h contains less than three independent nodes.

As figure 6 depicts, since the total neighboring area is maximum, $\widehat{v_e v_b v_d} = 120°$. Consequently, because of the following geometric properties, the distance between the points in dark areas pql and msl is less than or equal to a unit. Hence, only one of them can include an independent node. Since $rm = mv_b = v_b v_d = v_d r = 1$, therefore $rm \parallel mv_b \parallel v_b v_d \parallel v_d r$. This follows that $rm \parallel v_d n$ and since $\widehat{nv_d r} = 30°$, therefore, $\widehat{mrv_d} = 30°$. Hence, we can conclude that rm is dividing $\widetilde{smv_d}$ to two equal sections and $\widetilde{sm} = \widetilde{mv_d}$. Similarly $\widetilde{lm} = \widetilde{v_d m} = \widetilde{pl} = 30°$ and hence $\widetilde{pm} = 60°$. This results in $pm = v_b p = v_b m = 1$.

To show that all point in two dark area are closer than a unit to each other, it remains to prove that $ps \leq 1$. According to the assumption, $\widehat{pv_b v_d} = \widehat{sv_d v_b} = 90°$ and since $pv_b = v_b v_d = sv_d = 1$, consequently, $pv_b v_d s$ is a square and $ps = 1$, which completes the claim.

Note that c_1 and c_2 are two independent nodes in U_d and the $c_1 v_d c_2 = 60°$. $v_d r$ and $v_d t$ are achieved by connecting v_d to c_1 and c_2 respectively. Sector $smv_d r$ and similar sectors are obtained by drawing the *unit arc* [21] of independent nodes in U_d.

By the same reasoning, two other pairs of dark areas have the same property, leading to the fact that either U_e or U_d is limited to a 180° sector. This follows the property of lemma 3 and as it was shown, the path continuing from such vertex cannot lead to a U_k containing four independent nodes while all the U_h betweens w and k has three independent nodes in their semi-exclusive neighboring areas and this completes the proof for lemma 4. □

Theorem 1. *For any unit disk graph G, the size of a maximal independent set is at most* $3cds(G) + 3$ *where* $cds(G)$ *is the size of a minimum connected dominating set.*

Proof. Let T be a subgraph induced by a minimum connected dominating set in the given unit disk graph. Let $|T|$ be the number of vertices in G. We will show by induction on $|T|$ that there exists at most $3|T| + 3$ independent nodes in the neighboring area of T. According to lemmas 1 and 2, the claim is true for $|T| = 1$ and 2. Now, we assume that $|T| \geq 3$. Consider the semi-exclusive neighboring area of a leaf vertex v_l in an arbitrary tree that spans all the vertices of T. By removing v_l from the MCDS tree and removing all the independent nodes in its semi-exclusive area from graph G, by induction hypothesis, the neighboring area of $T' = T - v_l$ contains at most $3cds(T') + 3$ independent vertices. Next, we consider v_l and its semi-exclusive neighboring area U_l. If U_l includes less

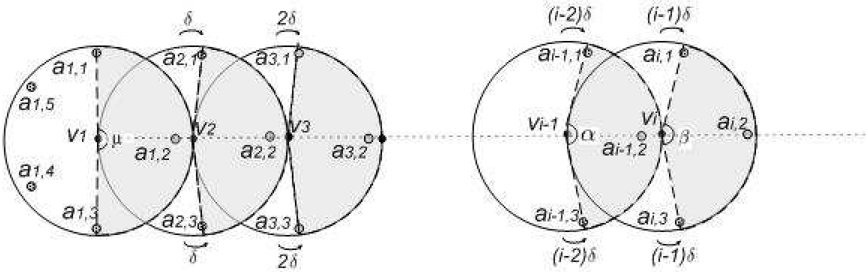


Fig. 7. Graph G, with $mis(G) = 3 \cdot cds(G) + 2$, Note that $\mu = 180° - \epsilon, \epsilon \rightarrow 0$

than three independent vertices, then the proof is complete. In the case that U_l contains four independent nodes, consider a preorder traversal of T.

According to lemma 3 on the path between v_l and the closest predecessor vertex v_j of T that U_j contains four independent nodes, there exists at least a vertex v_k, where U_k includes less than three independent nodes. Assume $T'' = T - \{v_k, v_{k+1}, ..., v_l\}$, according to induction hypothesis, the neighboring area of T'' contains at most $3cds(T'') + 3$. Now we consider the subtree P rooted by v_j which contains v_m, $m > j$. The number of independent nodes lie in the exclusive neighboring area of nodes in P is less than $3|P|$. This is because node e is the only vertex in P that its semi-exclusive neighboring set includes four nodes (more than three nodes), which cannot make counter example since U_j includes only two independent nodes (v_j in P).

Furthermore, according to lemma 4 another branch cannot exist in the subtree which would lead to a vertex v_h containing more than three nodes in its semi-exclusive neighboring area unless there is a vertex in the branch and on the path to v_h with less than three independent nodes in its semi-exclusive neighboring area. Consequently, the neighboring area of $T = T'' + P$ consists less than $3|T| + 3$ independent nodes and this completes the proof of the theorem. □

Corollary 1. *For approximation algorithms in [19], [6] for the minimum connected dominating set, the performance ratio can be reduced from 7.8 to 6.*

4.1 Tightness

Theorem 2. *The upper bound $3 \cdot cds(G) + C$ for the size of maximal independent set in unit disk graph G, where $cds(G)$ is the size of the minimum connected dominating set and C is a constant, is tight.*

Proof. To prove the theorem, we construct an extendable graph topology for vertices in a MCDS and MIS subgraphs such that for each $cds(G) = n, n \in N$, we have:

$$mis(G) = 3 \cdot cds(G) + C$$

Consider figure 7, where black vertices represent the nodes in the minimum connected dominating set and gray nodes are the set of independent vertices in a maximal independent set. Most of the edges that are connecting gray vertices are not showed for clarity of the figure, however, since the graph is a UDG, if $|uv| \leq 1$ there is an edge between u and v. Vertices in MCDS subgraph are all on a straight line and $\forall i, |v_i v_{i+1}| = 1$. As it is depicted in figure 7, the placement of the independent vertices in the MIS subgraph is such that $N(v_1)$ includes five independent nodes. For each vertex $v_i, i > 1$, U_i includes three independent nodes, which results in the factor of 3 for the relation between the size MIS and the size of MCDS. Lets consider the relation of the semi-exclusive areas U_{i-1}, U_i, and U_{i+1}. As shown in figure 7, assume the sector containing three independent nodes in U_{i-1} is α degree wide (The sector obtained by clockwise rotation of $v_{i-1}a_{i-1,1}$ to $v_{i-1}a_{i-1,3}$). According to lemma 3, the sector in U_i that can contain independent nodes is β degrees wide , where β is smaller than α ($\beta = \alpha - 2\delta$). In addition, according to lemma 1, β should be larger than 120° to be capable of containing three independent nodes. To construct a graph that meets the above relation with $cds(G) = n$, we define:

$$\delta = \frac{30}{n-1}$$

$$\widehat{a_{i,1}v_i a_{i,2}} = \widehat{a_{i,2}v_i a_{i,3}} = 90 - (i-1)\delta - \epsilon$$

Therefore,

$$|a_{i,j}a_{i-1,k}| > 1, \begin{cases} 2 \leq i \leq n \\ 1 \leq j \leq 3 \\ 1 \leq k \leq 3 \end{cases}$$

which are the immediate results of lemmas 3 and 2. Hence, we place two nodes in each semi-exclusive area on the edges of the mentioned sector and one on the line connecting MCDS vertices. Consequently, the number of nodes in MIS subgraph is $3(n-1) + C$, where C is the number of independent nodes in $N(v_1)$. Up to five independent nodes can lie in this unit disk, and consequently $mis(G) = 3 \cdot cds(G) + 2$ which completes the proof. □

5 Experimental Analysis

In this section, a practical analysis of the relation between connected dominating set and maximal independent set in wireless networks is presented. Since finding MCDS is NP-Hard, finding the size of optimal CDS in reasonable amount of time is not possible. Here, we will use the linear approximation algorithm introduced by Guha et. al. [11], which finds a CDS in an arbitrary graph with maximum cardinality of $2(1 + H(\Delta)).|OPT_{mcds}|$, where Δ and H denote the maximum degree and the harmonic function respectively. The algorithm for constructing MIS is such that in each iteration a random node is selected to be a member of MIS, which results in the elimination of its neighbors from the list of candidate MIS members. This process is continued until all nodes in the graph are either selected or eliminated from being a member of MIS.

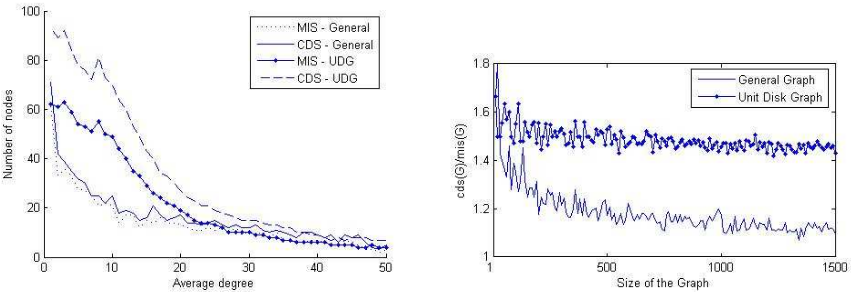


Fig. 8. cds(G) and mis(G) vs. the average degree in UDG (Left) and $\frac{cds(G)}{mis(G)}$ vs. size of the graph (Right)

To run the experiments, 400 nodes were distributed randomly in a space of size 100×100. We controlled the random graph generation phase by setting the average degree of the nodes in the graphs. It can be seen in figure 8 that as the average degree of the graph is increasing, the connectivity of the nodes increases resulting in significant decrease in the size of both $mis(G)$ and $cds(G)$. However, both sizes maintain the same slope as the connectivity increases.

Figure 8 also compares $\frac{cds(G)}{mis(G)}$ resulted from a general graph and one resulted from a unit disk graph. While maintaining the size of the space fixed, the size of the connected graph is increased from 10 to 1500 (resulting in a denser graph). Here, a *general graph* is a graph that is structured by vertices with arbitrary coverage radii. We observe that in the general graph the size of MIS is still comparable to the size of CDS. While the ratio is approximately constant in the UDG, the size of MIS and CDS subgraphs of the general graph are getting closer. Hence, the probability that the ratio does not meet the theoretical bound of UDG is higher in the denser graph. However, since in both cases $\frac{cds(G)}{mis(G)} \geq 1$, it can practically be assumed that the theoretical bound for unit disk graph is still practically applicable for wireless networks.

6 Conclusion

Applying protocols and techniques based on minimum connected dominating set plays an important role in improving the performance and energy consumption of wireless sensor and ad-hoc networks. A common approach to distributively approximate MCDS is to construct maximal independent set and then convert it to a connected tree. In this paper, we decreased the known bound between the size of maximal independent set and the size of minimum connected dominating set. We proved that $mis(G) \leq 3 \cdot cds(G) + 3$, where $mis(G)$ and $cds(G)$ denote the number of nodes in MIS and MCDS subgraphs respectively. Also, by proving that this bound is tight, we show that it cannot improved anymore. The new relation results in the new approximation ratio of 6 for distributed algorithm proposed by [6], [19]. In addition, we analyzed the behavior of general wireless networks in terms of size of MIS and MCDS subgraphs and concluded that the theoretical bound for unit disk graph is still practically applicable for wireless networks.

References

1. Alzoubi, K.M., Wan, P.-J., Frieder, O.: Message-optimal connected dominating sets in mobile ad hoc networks. In: MobiHoc 2002: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing, pp. 157–164. ACM, New York (2002)
2. Clark, C.C.B., Johnson, D.: Unit disk graphs. Descrete Mathmatics 86, 165–177 (1990)
3. Das, R.S.B., Bharghavan, V.: Routing in ad-hoc networks using a virtual backbone. In: ICCCN 1997, pp. 1–20 (1997)
4. Banik, S.M., Radhakrishnan, S.: Minimizing broadcast latency in ad hoc wireless networks. In: ACM-SE 45: Proceedings of the 45th annual southeast regional conference, pp. 533–534. ACM, New York (2007)
5. Bevan Das, R.S., Bharghavan, V.: Routing in ad-hoc networks using a spine. In: Internatioal Conference on Computers and Communaation Networks, pp. 376–380 (1997)
6. Cardei, M., Cheng, X., Cheng, X., Du., D.-Z.: Connected domination in multihop ad hoc wireless networks. In: Proceedings of Sixth International Conference in Computer Science and Informatics, CSI
7. Das, B., Bharghavan, V.: Routing in ad-hoc networks using minimum connected dominating sets. In: ICC, pp. 376–380 (1997)
8. De Gaudenzi, T.G.F.L.M., Garde, R.: Ds-CDMA techniques for mobile and personal satellite communications: An overview. In: IEEE Second Symposium Conommunications and Vehicular Technology (1994)
9. Deb, B., Nath, B.: On the node-scheduling approach to topology control in ad hoc networks. In: MobiHoc 2005: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing, pp. 14–26. ACM, New York (2005)
10. Funke, S., Kesselman, A., Meyer, U., Segal, M.: A simple improved distributed algorithm for minimum CDS in unit disk graphs. ACM Transaction on Sensor Networks 2(3), 444–453 (2006)
11. Guha, S., Khuller, S.: Approximation algorithms for connected dominating sets. Algorithmica 20(4), 374–387 (1998)
12. Blum, A.T.J., Ding, M., Cheng, X.: Connected dominating set in sensor networs and manets. In: Du, D.-Z., Pardalos, P. (eds.) Handbook of Combinatorial Optimization, pp. 329–369. Kluwer Academic Publishers, Dordrecht (2004)
13. Marathe, M.V., Breu, H., Hunt III, H.B., Ravi, S.S., Rosenkrantz, D.J.: Simple heuristics for unit disk graphs. Networks 25, 59–68 (1995)
14. Min, M., Du, H., Jia, X., Huang, C.X., Huang, S.C.-H., Wu, W.: Improving construction for connected dominating set with steiner tree in wireless sensor networks. J. of Global Optimization 35(1), 111–119 (2006)
15. Mnif, K., Rong, B., Kadoch, M.: Virtual backbone based on mcds for topology control in wireless ad hoc networks. In: PE-WASUN 2005: Proceedings of the 2nd ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks, pp. 230–233. ACM, New York (2005)
16. Ni, S.-Y., Tseng, Y.-C., Chen, Y.-S., Sheu, J.-P.: The broadcast storm problem in a mobile ad hoc network. In: MobiCom 1999: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking, pp. 151–162. ACM, New York (1999)

17. Shaikh, J.A., Solano, J., Stojmenovic, I., Wu, J.: New metrics for dominating set based energy efficient activity scheduling in ad hoc networks. In: LCN 2003: Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks, Washington, DC, USA, p. 726. IEEE Computer Society Press, Los Alamitos (2003)
18. Stojmenovic, I., Seddigh, M., Zunic, J.: Dominating sets and neighbor elimination-based broadcasting algorithms in wireless networks. IEEE Transactions on Parallel and Distributed Systems 13(1), 14–25 (2002)
19. Wan, P., Alzoubi, K., Frieder, O.: Distributed construction of connected dominating set in wireless ad hoc networks (2002)
20. Wu, J., Li, H.: On calculating connected dominating set for efficient routing in ad hoc wireless networks. In: DIALM 1999: Proceedings of the 3rd international workshop on Discrete algorithms and methods for mobile computing and communications, pp. 7–14. ACM, New York (1999)
21. Wu, W., Du, H., Jia, X., Li, Y., Huang, S.C.-H.: Minimum connected dominating sets and maximal independent sets in unit disk graphs. Theoretical Computer Science 352(1), 1–7 (2006)

Appendix: Proof of Lemma 4 for a Branch Location with Unknown Angles

In this appendix we provide a proof to show why lemma 4 is still correct in general cases. To generalize the case, we consider rotating v_e and v_d in figure 6 for d_1 and d_2 degrees. Therefore, $\widehat{v_a v_b v_e} = 120 + d_1{}^\circ$ and $\widehat{v_a v_b v_d} = 120 + d_2{}^\circ$ (See figure 9). d_1 and d_2 are positive if the rotation is clockwise and are negative if the rotation is counterclockwise.

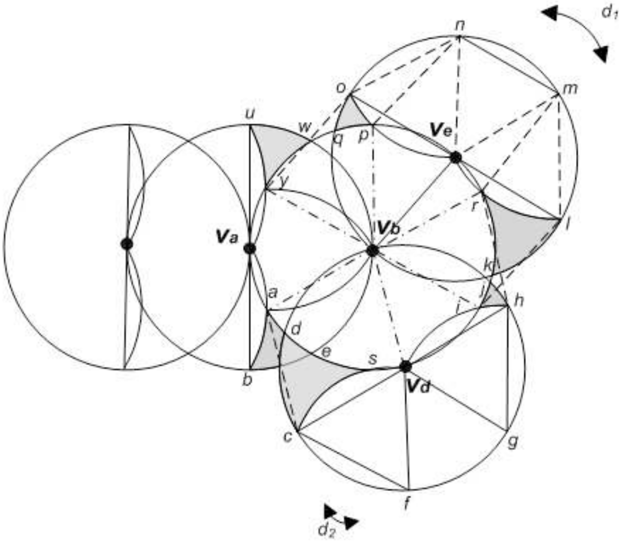


Fig. 9. The general case for the branch

Note that in figure 9, diameters ol and ch are such that $|ac| = |yo| = 1$. Also, f, g, m and n are location of independent nodes lying in exclusive neighboring sets of v_e and v_d. Arcs $\widehat{cv_d}$, $\widehat{v_dh}$, $\widehat{lv_e}$ and $\widehat{v_eo}$ are from the perimeters of unit arcs containing v_d and one of f, g, m or n, respectively. We first prove that the distance between all the points in rlk and hij is less than a unit by showing that $|hr| = |lj| = 1$. First, note that $|ac| = |cv_d| = |v_dv_b| = |v_ba| = 1$ and hence, acv_dv_b is a parallelogram. Consequently, $\widehat{av_bv_d} = \widehat{v_bv_dh}$. Now by showing that v_br and v_ba are on a straight line, we show that ra is a diameter and as a result, $v_dh \parallel v_br$ (since $v_dh \parallel v_ba$). Now, assume parallelograms v_bv_emr and v_bv_enp. Since $\widehat{nm} = 60°$ and $\widehat{mv_ev_b} + \widehat{v_bv_en} = 300°$ (onv_e and lmv_e are two unit arcs and $\widehat{ol} = 180°$, therefore $\widehat{mn} = 60°$), hence $\widehat{rv_bp} = \widehat{rv_bv_e} + \widehat{pv_bv_e} = 360° - (\widehat{mv_ev_b} + \widehat{v_bv_en}) = 60°$. Considering that $\widehat{rv_e}$ and $\widehat{v_ep}$ are both 30° before rotating v_e, $\widehat{rv_e} = 30 - {d_1}^\circ$ and $\widehat{pv_e} = 30 + {d_1}^\circ$. In addition, according to the fact that $\widehat{wq} = d_1^\circ$ (since it is the result of rotating v_d), we can conclude that $\widehat{ar} = 180°$. Therefore, v_dh is parallel to v_br, and since $|v_dh| = |v_br| = |v_dv_b| = 1$, v_bv_drh is parallelogram, and hence $|rh| = 1$.

With similar reasoning (considering ilv_ev_b and showing that it is a parallelogram), we can see that $|il| = 1$ and hence distance between all the points in regions krl and ijh are less than one. To show the same condition for other pairs of dark areas, similar set of proof will apply, resulting in the fact that rotating v_e and v_d around v_b doesn't have any effect on the generality of lemma 4.

Brief Announcement: On the Solvability of Anonymous Partial Grids Exploration by Mobile Robots

Roberto Baldoni[1], François Bonnet[2], Alessia Milani[3], and Michel Raynal[2]

[1] Università di Roma "La Sapienza", Italy
[2] IRISA, Université de Rennes, France
[3] LADyR, GSyC, Universidad Rey Juan Carlos, Spain

Graph exploration by robots. The graph exploration problem consists in making one or several mobile entities visit each vertex of a connected graph. The mobile entities are sometimes called agents or robots (in the following we use the word "robot"). The exploration is perpetual if the robots have to revisit forever each vertex of the graph. Perpetual exploration is required when robots have to move to gather continuously evolving information or to look for dynamic resources (resources whose location changes with time). If nodes and edges have unique labels, the exploration is relatively easy to achieve.

The graph exploration problem becomes more challenging when the graph is anonymous (i.e., the vertices, the edges, or both have no label). In such a context, several bounds have been stated. They concern the total duration needed to complete a visit of the nodes, or the size of the robot memory necessary to explore a graph (e.g., it has been shown that a robot needs $O(D \log d)$ bits of local memory in order to explore any graph of diameter D and maximum degree d). Impossibility results for one or more robots with bounded memory (computationally speaking, a robot is then a finite state automaton) to explore all graphs have also been stated. The major part of the results on graph exploration consider that the exploration is made by a single robot. Only very recently, the exploration of a graph by several robots has received attention also from a practical side. This is motivated by research for more efficient graph explorations, fault-tolerance, or the need to overcome impossibilities due to the limited capabilities of a single robot.

The constrained exploration problem. Considering the case where the graph is an anonymous partial grid (connected grid with missing vertices/edges), and where the robots can move synchronously but cannot communicate with each other, the paper considers the following instance of the graph exploration problem, denoted the *Constrained Perpetual Graph Exploration* problem (*CPGE*). This problem consists in designing an algorithm executed by each robot that (1) allows as many robots as possible (let k be this maximal number), (2) to visit infinitely often all the vertices of the grid, in such a way that no vertex hosts more than one robot at a time, and each edge is traversed by at most one robot at a time. These mutual exclusion constraints are intended to abstract the problem of collision that robots may incur when moving within a short distance from each other

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 496–497, 2008.

or the necessity for the robots to access resources in mutual exclusion. The designed algorithm has to avoid collisions despite the number of robots located on the grid and their initial position. On the other hand, complete exploration may not be ensured if robots are too much.

Results exposed in [2] rest on three parameters, denoted p, q and ρ. The first parameter p is related to the size of the grid, namely, it is the number of vertices of the partial connected grid. The second parameter q is related to the structure of the partial grid. This parameter is defined from a *mobility tree* (a new notion introduced in [1]) that can be associate with each partial grid. So, each pair (p, q) represents a subset of all possible partial grids with p vertices. Finally, the third parameter ρ is not related to the grid, but captures the power of the robots as far as the subgrid they can currently see is concerned. More precisely, a robot sees the part of the grid centered at its current position and covered by a radius ρ. From an operational point of view, the radius notion allows the robots that are at most ρ apart one from the other to synchronize their moves without violating the vertex and edge mutual exclusion constraints.

Bounds on the number of robots. The technical reports [1,2] analyze the solvability of the $CPGE$ problem with respect to the number of robots k that can be placed in a partial grid characterized by the pair p and q when considering a given ρ for robots. They presents the following results.

- Case $\rho = 0$ [2]. In that case, the $CPGE$ problem cannot be solved (i.e., we have $k = 0$) for any grid such that $p > 1$ (a grid with more than one vertex). This means that, whatever the grid, if the robots cannot benefit from some view of the grid, there is no algorithm run by robots that can solve the $CPGE$ problem.
- Case $\rho = +\infty$ [1]. In that case, $k \leq p - q$ is a necessary and sufficient requirement for solving the $CPGE$ problem. Let us observe that $\rho = +\infty$ means that the robots knows the structure of the grid and the current position of the robots on that grid. (The initial anonymity assumption of the vertices and the robots can then easily be overcome.)
- Case $\rho = 1$ [2]. In that case, assuming a grid with more than one vertex, $k \leq p - 1$ when $q = 0$, and $k \leq p - q$ otherwise, are necessary and sufficient requirements for solving the $CPGE$ problem.

References

1. Baldoni, R., Bonnet, F., Milani, A., Raynal, M.: Anonymous Graph Exploration without Collision by Mobile Robots. Tech Report #1886, 10 pages. IRISA, Université de Rennes 1, France (2008), ftp.irisa.fr/techreports/2008/PI-1886.pdf
2. Baldoni, R., Bonnet, F., Milani, A., Raynal, M.: On the Solvability of Anonymous Partial Grids Eploration by Mobile Robots. Tech Report #1892, 21 pages. IRISA, Université de Rennes 1, France (2008), ftp.irisa.fr/techreports/2008/PI-1892.pdf

The Dynamics of Probabilistic Population Protocols*

Brief Announcement

Ioannis Chatzigiannakis[1] and Paul G. Spirakis[1]

Research Academic Computer Technology Institute (CTI), and Department of Computer
Engineering and Informatics (CEID), University of Patras, 26500, Patras, Greece
{ichatz,spirakis}@cti.gr

Abstract. Angluin et al. [1] introduced the notion of "Probabilistic Population Protocols" (PPP) where extremely limited agents are represented as finite state machines that interact in pairs under the control of an adversary scheduler. We provide a very general model that allows to examine the continuous dynamics of population protocols and we show that it includes the model of [1], under certain conditions, with respect to the continuous dynamics of the two models.

1 Switching Probabilistic Protocols

The network is modeled as a complete graph $G = (V, E)$ where $n = |V|$, the set of vertices that represent nodes and E, the set of edges that represent communication links between nodes. Each node executes an "agent" (or process) which consists of a finite set of states K, with $k = |K|$. Let n_i the number of agents that are on state $i \in \{1, 2, \ldots, k\}$, so $n = \sum_{i=1}^{k} n_i$. The configuration of the population at time t is described as a vector $\boldsymbol{x}(t) = (x_1(t), \ldots, x_k(t))$ where $x_i(t) = \frac{n_i}{n}$.

In the sequel we assume that $n \to \infty$. We are interested, thus, in the evolution of $\boldsymbol{x}(t)$ as time goes on. We imagine that all agents in the population are infinitely lived and that they interact forever. Each agent sticks to some state in K for some time interval, and now and then *reviews* her state. This depends on $\boldsymbol{x}(t)$ and may result to a change of state of the agent. Based on this concept, a *Switching Population Protocol* (SPP) is specified by the (i) *time rate* at which agents review their state and (ii) the *switching probabilities* of a reviewing agent. The time rate depends on the current, "local", performance of the agent's state and also on the configuration $\boldsymbol{x}(t)$. The switching probabilities defines the probability that an agent, currently in state q_i at a review time, will *switch* to state q_j is in general a function $p_{ij}(\boldsymbol{x}(t))$, where $p_i(\boldsymbol{x}) = (p_{i1}(\boldsymbol{x}), \ldots, p_{ik}(\boldsymbol{x}))$ is the resulting distribution over the set K of states in the protocol.

In a large, finite, population n, we assume that the review times of an agent are the "birth times" of a Poisson process of rate $\lambda_i(\boldsymbol{x})$. At each such time, the agent i selects a new state according to $p_i(\boldsymbol{x})$. We assume that all such Poisson processes are independent. Then, the aggregate of review times in the sub-population of agents in state q_i is itself a Poisson process of birth rate $x_i \lambda_i(\boldsymbol{x})$. As in the probabilistic model

* This work has been partially supported by the ICT Programme of the European Union under contract number ICT-2008-215270 (FRONTS).

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 498–499, 2008.

of [1] we assume that state switches are independent random variables accross agents. Then, the rate of the (aggregate) Poisson process of switches from state q_i to state q_j in the whole population is just $x_i(t)\lambda_i\left(\boldsymbol{x}(t)\right)p_{ij}\left(\boldsymbol{x}(t)\right)$.

When $n \to \infty$, we can model the aggregate stochastic processes as deterministic flows. The outflow from state q_i is $\sum_{j\neq i} x_j\lambda_j\left(\boldsymbol{x}\right)p_{ij}\left(\boldsymbol{x}\right)$. Then, the rate of change of $x_i(t)$ (i.e. $\frac{dx_i(t)}{dt}$ or $\dot{x_i}(t)$) is just

$$\dot{x_i} = \sum_{j\in K} x_j p_{ji}\left(\boldsymbol{x}\right)\lambda_j\left(\boldsymbol{x}\right) - \lambda_i\left(\boldsymbol{x}\right)x_i \tag{1}$$

for $i = 1,\ldots,k$. We assume here that both $\lambda_i\left(\boldsymbol{x}\right)$ and $p_{ij}\left(\boldsymbol{x}\right)$ are Lipschitz continuous functions in an open domain Σ containing the simplex Δ where $\Delta = \Big\{(x_i,\ldots,x_k) : \sum_{i=1}^{K} x_i = 1,\, x_i \geq 0,\, \forall i\Big\}$. By the theorem of Picard - Linderlöf (see, e.g., [2] for a proof), Eq. 1 has a *unique* solution for any initial state $\boldsymbol{x}(0)$ in Δ and such a solution trajectory $\boldsymbol{x}(t)$ is *continuous* and never leaves Δ.

Theorem 1. Assume that a set of differential equations represent the continuous time dynamics of PPP as a limit of the discrete model. Then, the continuous time dynamics of SPP include those of PPP as a special case.

A full version of this paper is available at `http://arxiv.org/abs/0807.0140`

References

1. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 290–299. ACM, New York (2004)
2. Hirsch, M., Smale, S.: Differential Equations, Dynamical Systems and Linear Algebra. Academic Press, London (1974)

A Distributed Algorithm for Computing and Updating the Process Number of a Forest*

David Coudert, Florian Huc, and Dorian Mazauric

Mascotte, INRIA, I3S, CNRS UMR6070, University of Nice Sophia Antipolis
Sophia Antipolis, France

Treewidth and pathwidth have been introduced by Robertson and Seymour as part of the graph minor project. Those parameters are very important since many problems can be solved in polynomial time for graphs with bounded treewidth or pathwidth. By definition, the treewidth of a tree is one, but its pathwidth might be up to $\log n$. A linear time centralized algorithms to compute the pathwidth of a tree has been proposed in [1], but so far no dynamic algorithm exists.

The algorithmic counter part of the notion of pathwidth is the cops and robber game or node graph searching problem [2]. It consists in finding an invisible and fast fugitive in a graph using the smallest set of agents. A search strategy in a graph G can be defined as a serie of the following actions: (i) put an agent on a node, and (ii) remove an agent from a node if all its neighbors are either cleared or occupied by an agent. The node is now cleared. The fugitive is caught when all nodes are cleared. The minimum number of agents needed to clear the graph is the node search number (ns), and gives the pathwidth (pw) [3]. More precisely, it has been proved that $\text{ns}(G) = \text{pw}(G) + 1$ [4].

Other graph invariants closely related to the notion of pathwidth have been proposed (see [5] for a recent survey) such as the process number (pn) [6] and the edge search number (es), but so far it is not known if those parameters are strictly equivalent to pathwidth. However, we know that $\text{pw}(G) \leq \text{pn}(G) \leq \text{pw}(G)+1$ [6] and $\text{pw}(G) \leq \text{es}(G) \leq \text{pw}(G) + 2$ [2]. A process strategy can be defined similary to a search strategy with the extra rule that the fugitive is forced to move at each round. Therefore, a node is also cleared when all its neighbors are occupied by an agent (the node is surrounded). For examples, $\text{pn}(K_n) = n-1 = \text{ns}(K_n)-1$, where K_n is a n-clique, and $\text{pn}(C_k) = \text{ns}(C_k) = 3$, where C_k is a cycle of length $k \geq 5$. The process number is the minimum number of agents needed.

Here we propose a distributed algorithm to compute those parameters on trees and to update them in a forest after the addition or deletion of any edge [7]. Only initial conditions differ from one parameter to another. It is fully distributed, can be executed in an asynchronous environment and needs the transmission of only a small amount of information. It uses Theorem 1 which is also true for other parameters [6] and enforces each parameter to grow by 1, thus implying that for any tree $\text{ns}(T)$, $\text{es}(T)$, $\text{pw}(T)$, and $\text{pn}(T)$ are less than $\log_3(n)$.

* This work has been supported by the European projects IST FET Aeolus and COST 293 Graal, ARC CARMA, ANR JC OSERA, CRC CORSO, and Région PACA.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 500–501, 2008.

Theorem 1 ([8]). *Let G_i, $i = 1, 2, 3$ be such that* $\mathrm{ns}(G_i) = k > 1$. *The graph G obtained by connecting each of the G_i's to a new node v is such that* $\mathrm{ns}(G) = k+1$.

The principle of our algorithm, `algoHD`, is to perform a hierarchical decomposition of the tree. Each node collects a compact view of the subtree rooted at each of its soons, computes a compact view of the subtree it forms and sends it to its father, thus constructing a the hierarchical decomposition. A similar idea was used in [4] to design an algorithm computing the node search number in linear time. However their algorithm is centralized and its distributed version will transmit $\log \log n$ times more bits than ours. So we obtained,

Lemma 1. *Given a n nodes tree T, `algoHD` computes* $\mathrm{pn}(T)$, $\mathrm{ns}(T)$ *or* $\mathrm{es}(T)$, *in n steps, overall $O(n \log n)$ operations, and $n-1$ messages of* $\log_3 n + 2$ *bits.*

We have extended our algorithm to a fully dynamic algorithm, `IncHD`, allowing to add or remove any edge. Each update can be performed in $O(D)$ steps, each of time complexity $O(\log n)$, and using $O(D)$ messages of $\log_3 n + 3$ bits, where D is the diameter of the tree. We have also extended our algorithms to trees and forests of unknown sizes by using messages of size $2L(t) + 4 + \varepsilon$, where $\varepsilon = 1$ for `IncHD`, and $L(t) \leq \mathrm{pn}(T) \leq \log_3 n$ is the minimum number of bits required to encode the local view of a subtree.

Finally, we have characterized the trees for which the process number (resp. edge search number) equals the node search number and so the pathwidth.

Lemma 2. *Given a tree T,* $\mathrm{pn}(T) = \mathrm{pw}(T) + 1 = p + 1$ *(resp.* $\mathrm{pn}(T) = \mathrm{es}(T) + 1 = p+1$*) iff there is a node v such that any components of $T - \{v\}$ has pathwidth at most p and there is at least three components with process number (resp. edge search number) p of which at most two have pathwidth p.*

A challenging task is now to give such characterisations for more general classes of graphs, as well as distributed and dynamic algorithms.

References

1. Skodinis, K.: Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. Journal of Algorithms 47(1), 40–59 (2003)
2. Kirousis, M., Papadimitriou, C.: Searching and pebbling. Theoretical Computer Science 47(2), 205–218 (1986)
3. Kinnersley, N.G.: The vertex separation number of a graph equals its pathwidth. Information Processing Letters 42(6), 345–350 (1992)
4. Ellis, J., Sudborough, I., Turner, J.: The vertex separation and search number of a graph. Information and Computation 113(1), 50–79 (1994)
5. Fomin, F.V., Thilikos, D.: An annotated bibliography on guaranteed graph searching. Theoretical Computer Science, Special Issue on Graph Searching (to appear)
6. Coudert, D., Perennes, S., Pham, Q.C., Sereni, J.S.: Rerouting requests in wdm networks. In: AlgoTel 2005, Presqu'île de Giens, France, pp. 17–20 (2005)
7. Coudert, D., Huc, F., Mazauric, D.: A distributed algorithm for computing and updating the process number of a forest. Research Report 6560, INRIA (2008)
8. Parsons, T.D.: Pursuit-evasion in a graph. In: Theory and applications of graphs. Lecture Notes in Mathematics, vol. 642, pp. 426–441. Springer, Berlin (1978)

BRIEF ANNOUNCMENT: Corruption Resilient Fountain Codes*

Shlomi Dolev and Nir Tzachar

Department of Computer Science
Ben Gurion University of the Negev, Israel

A new aspect for erasure coding is considered, namely, the possibility that some portion of the arriving packets are corrupted in an undetectable fashion. In practice, the corrupted packets may be attributed to a portion of the communication paths that are leading to the receiver and are controlled by an adversary. Alternatively, in case packets are collected from several sources, the corruption may be attributed to a portion of the sources that are malicious.

Corruption resistant fountain codes are presented; the codes resemble and extend the LT and Raptor codes. To overcome the corrupted packets received, our codes use information theoretics techniques, rather than cryptographic primitives such as homomorphic one-way-(hash) functions. Our schemes overcome adversaries by means of using slightly more packets than the minimal number required for revealing the encoded message, and using a majority over the possible decoded values. We also present a more efficient randomized decoding scheme. Beyond the obvious use, as a rateless erasure code, our code has several important applications in the realm of efficient distributed robust data storage and maintenance.

Fountain codes such as the LT codes can be described by the following succinct encoding algorithm: given a message of length n bits, divide the message into k input symbols, $s_1, s_2, \ldots, s_k$, each of length $m = \frac{n}{k}$. To generate a packet, choose a random subset of input symbols and XOR them. The packet is then composed of a bit field, denoting which symbols were XORed and the result of the XOR, which results in a packet of length $k+m$. The choice of the random set of symbols forms the critical part of the encoding algorithm, defines the number of packets needed to correctly decode the input symbols and enables linear time decoding using the Belief Propagation decoder (BP-decoder). Unfortunately, the BP-decoder is very susceptible to attacks; an adversary may easily hamper the decoding process by corrupting a small fraction of the received packets.

A different approach to realize rateless codes that is resilient to corruption was recently suggested in [1]. The approach is based on embedding vector subspaces to encode and decode messages. Unfortunately, the approach taken in [1] requires packets to be of non-constant size. In particular, to recover from a corruption of a third of the received vectors, packet lengths must be in $\Omega(n)$, where n is the length of the message,

* Partially supported by Deutsche Telekom, the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Rita Altura Trust Chair in Computer Sciences, and the Lynne and William Frankel Center for Computer Sciences. Emails: {dolev,tzachar}@cs.bgu.ac.il. An extended version appears as TR#08-12 of the Dept. of Computer Science, BGU.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 502–503, 2008.

while our is the first to support shorter messages as well. We use the same encoding scheme as in [2,3]. Our contribution lies in the decoding schemes. We present several possible ways to decode a value, where the trade off between the number of packets which need to be collected and the decoding time is investigated. We assume that the adversary may corrupt a predetermined fraction of the received messages. For brevity, we assume the number of corrupted packets is bounded by f, and limit the discussion to decoding a given symbol, s_l, where all symbols may be decoded in parallel, using the same technique.

• **Majority voting.** To reconstruct a given message piece, s_l, given that at most f faults occurred, we need to collect $2f + 1$ pairwise disjoint sets of packets, such that each set can successfully decode a possible message (for example, by using the BP-decoder). From each independent set, S_j, we can reconstruct an s'_j as a candidate message piece. It then follows that the majority of the s'_j values is the correct message piece.
• **Exhaustive search algorithm.** By viewing the set of collected packets as an equation system and by collecting approximately $2f + k$ packets, the original symbols are the only solution to the equation system satisfying at least $f + k$ equations. However, finding such a maximal solution – a solution which satisfies the maximal number of equations – is NP-hard. Thus, we suggest to perform an exponential search, testing all possible values for a given symbol and, for each solution, checking if the solution solves at least $k + f$ equations. Once the correct solutions is found, the decoding is complete. Overall, the decoding time is exponential in k.
• **Randomized decoding algorithm.** We use randomization in order to reduce the decoding complexity, given that the adversary may corrupt at most f packets. Assume that we have collected N packets, such that $|N| = g(k) + k + f \geq k + 2f$ where $g(x) \geq f$. Each subset of $k + \epsilon$ uncorrupted packets has a very high probability (where $\epsilon << k$) of containing a subset of k independent packets. Let this probability be p_ϵ. The algorithm will work as follows: choose a random subset, $S \subset N$, such that $|S| = k + \epsilon$. Check if there exists a subset $S' \subset S$ which contains exactly k independent packets. If such an S' exists, solve the associated equation system. If the obtained solution satisfies more than $k + f + \epsilon$ equations out of N then, as before, we are done. Now, let p_k be the probability of choosing a subset of N with no corrupted packets. It follows that the expected runtime of the algorithm (iteration-wise) is $\frac{1}{p_k \cdot p_\epsilon}$. Setting $\epsilon = c \log k$, we get that $p_\epsilon > 1 - 1/k^c$ (see [3]). We can then bound p_k by $\exp(-f \frac{k+\epsilon}{g(k)})$. Choosing $g(x) > \frac{f \cdot (k+\epsilon)}{\log b}$, for a constant b, results in $p_k > 1/b$. The expected runtime of the decoding algorithm is then b. Such a choice of $g(x)$ is only relevant when the adversary can corrupt at most a fraction of $\frac{1}{k}$ of the collected packets and, furthermore, minimizes the run time at the expense of having to collect many messages ($g(x) \approx kf$).

References

1. Koetter, R., Kschischang, F.: Coding for Errors and Erasures in Random Network Coding, http://aps.arxiv.org/pdf/cs.IT/0703061
2. Luby, M.: LT Codes. FOCS, 271–280 (2002)
3. Shokrollahi, A.: Raptor Codes. IEEE Trans. on Info. Theory 52, 2551–2567 (2006)

Brief Announcement: An Early-Stopping Protocol for Computing Aggregate Functions in Sensor Networks *

Antonio Fernández Anta[1], Miguel A. Mosteiro[1,2], and Christopher Thraves[3]

[1] LADyR, GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Madrid, Spain
[2] Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
[3] Universite Bordeaux I, LaBRI, domaine Universitaire, 33405 Talence, France

Introduction. Nodes in a Sensor Network can collaborate to process the sensed data but, due to unreliability, a monitoring strategy can not rely on individual sensors values. Instead, the network should use aggregated information from groups of sensor nodes [2,3,7]. The topic of this work is the efficient computation of aggregate functions in the highly constrained Sensor Network setting, where node restrictions are modeled as in [4], the random node-deployment is modeled as a geometric graph, and the resulting topology, node identifiers assignment and the assignment of input-values to be aggregated is adversarial.

While algebraic aggregate functions are well defined, the implementation of such computations in Sensor Networks has to deal with various issues that make even the definition of the problem difficult. First, the input-values at each node might change over time. Therefore, it is necessary to fix to which time step correspond those input-values. Second, arbitrary node failures make the design of protocols challenging. It has been shown [1] that the problem of computing an aggregate function among all nodes in a network where some nodes join and leave the network arbitrarily in time is intractable.

Results. The protocol proposed interleaves two algorithms, one following a tree-based approach and one following a mass-distribution approach. The tree-based algorithm will provide the correct result with low time and energy complexity in a failure-free setting. If the presence of failures prevents the tree-based computation from finishing, the mass-distribution algorithm will compute and disseminate an approximation of the result. The time taken by this algorithm is larger, but it is only incurred in presence of failures. Hence, the combined algorithm is *early stopping*.

* A full version of this work is available at [5]. This research was supported in part by Comunidad de Madrid grant S-0505/TIC/0285; Spanish MEC grant TIN2005-09198-C02-01; EU Marie Curie European Reintegration Grant IRG 210021; NSF grants CCF0621425, CCF 05414009, CCF 0632838; and French ANR Masse de Donnèes project ALPAGE.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 504–506, 2008.

The efficiency is measured in two dimensions: time and number of transmissions. These metrics are strongly influenced by collisions, especially because no collision detection mechanisms are available in this setting. In order to reduce collisions and energy consumption, a two-level hierarchy of nodes is used. The actual computation is done by a small set of nodes, called *delegate nodes,* that collect the sensed input-values from the non-computing nodes, called *slug nodes.* This structure has several advantages. First, collisions are reduced since they can only occur while the delegate nodes collect the sensed input-values from the slug nodes. After that, delegate nodes are able to communicate in a collision-free fashion. Second, energy is saved because the slug nodes can idle during the computation. Third, the subnetwork of delegate nodes has constant degree, which allows to easily build a constant-degree spanning tree. Finally, since the set of delegate nodes is small, there is a smaller probability that the tree-based algorithm will fail (since only failures of delegate nodes impact on it). Notice that, in presence of failures, the two-level structure may have to be reconstructed; fortunately, this can be done fast and locally.

The main contribution of this work is the presentation of a time-optimal early-stopping protocol that computes the average function in Sensor Networks under a harsh model of sensor restrictions [1]. More precisely, it is shown that, in a failure-free setting, with high probability, this protocol returns the exact value and terminates in $O(D + \Delta)$ steps, which is also shown to be optimal, and the overall number of transmissions is in $O(n(\log n + \Delta/\log n + \log \Delta))$ in expectation. On the other hand, in presence of failures, the protocol computes the average of the input-values of a subset of nodes that depends on the failure model. More precisely, it is shown that, after the last node fails and w.h.p., the protocol takes an extra additive factor of $O(\log(n/\varepsilon)/\Phi^2)$ in time and an extra additive factor of $O(n \log(1/\varepsilon)/\Phi^2)$ in the expected number of transmissions, where $\varepsilon > 0$ is the maximum relative error, and Φ is the conductance [6] of the network of delegates.

References

1. Bawa, M., Garcia-Molina, H., Gionis, A., Motwani, R.: Estimating aggregates on a peer-to-peer network. Technical report, Stanford Univ., Database group (2003)
2. Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE/ACM Transactions on Networking 14(SI), 2508–2530 (2006)
3. Chen, J.-Y., Pandurangan, G., Xu, D.: Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis. In: Proc. of the 4th Intl. Symp. on Information Processing in Sensor Networks, p. 46 (2005)
4. Farach-Colton, M., Fernandes, R.J., Mosteiro, M.A.: Bootstrapping a hop-optimal network in the WSM. ACM Trans. on Algorithms (in press, 2008)
5. Fernández Anta, A., Mosteiro, M.A., Thraves, C.: An early-stopping protocol for computing aggregate functions in sensor networks. Technical Report 3, LADyR, GSyC, Universidad Rey Juan Carlos (May 2008)

[1] Other aggregate functions can be computed using a protocol for average without extra cost as described in [3,7].

6. Jerrum, M., Sinclair, A.: Conductance and the rapid mixing property for markov chains: the approximation of permanent resolved. In: Proc. of the 20th Ann. ACM Symp. on Theory of Computing, pp. 235–244 (1988)
7. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proc. of the 44th IEEE Ann. Symp. on Foundations of Computer Science, pp. 482–491 (2003)

Easy Consensus Algorithms for the Crash-Recovery Model

Felix C. Freiling, Christian Lambertz, and Mila Majster-Cederbaum

Department of Computer Science, University of Mannheim, Germany

Problem Setting. One of the most popular failure models for asynchronous fault-tolerant distributed systems is called *crash-stop*, which allows that a certain number of processes stops executing steps during the computation. Despite its theoretical interest, crash-stop is not expressive enough to model many realistic scenarios. In practice, processes crash but their processors reboot and the crashed process is restarted from a recovery point and rejoins the computation. This behavior is formalized as a failure model called *crash-recovery*, in which the processes can crash and recover multiple times.

Crash-recovery is a strict generalization of crash-stop and thus, any impossibility result for crash-stop also holds in crash-recovery. However, algorithms designed for crash-stop will not necessarily work in crash-recovery due to the additional behavior. While in crash-stop processes are usually classified into two distinct classes (those which eventually crash and those which do not), in crash-recovery we have four distinct classes of processes: (1) *always up* (processes that never crash), (2) *eventually up* (processes that crash at least once but eventually recover and do not crash anymore), (3) *eventually down* (processes that crash at least once and eventually do not recover anymore), and (4) *unstable* (processes that crash and recover infinitely often). (1) and (2) are called *correct* processes, and (3) and (4) *incorrect*. Since processes usually lose all state information when they crash, the notion of *stable storage* was invented to model a type of expensive persistent storage to access pre-crash data.

We choose the problem of *consensus* as benchmark problem to study the differences between crash-stop and crash-recovery. Roughly speaking, consensus requires that processes agree on a common value from a set of input values. Consensus is fundamental to many fault-tolerant problems but has mainly been studied in crash-stop. We use the *failure detector* abstraction (FD) to help solve consensus. A FD gives information about the failures of processes. We look at two classes of FDs: the class of *perfect FDs* ($\mathcal{P}$) that tell exactly who has failed, and the class of *eventually perfect FDs* ($\Diamond\mathcal{P}$) that can make finitely many mistakes about who has failed.

Contributions. Similarly to Aguilera, Chen and Toueg [1], we study consensus in the crash-recovery model under varying assumptions. However, our approach is to re-use existing algorithms from crash-stop in a modular way and to improve the comprehensibility. One main task of our algorithms is to partly *emulate* a crash-stop system on top of a crash-recovery system to be able to run a crash-stop consensus algorithm.

Table 1 summarizes the cases we study along three dimensions: (1) the availability of stable storage (large columns), (2) a process state assumption (rows of the table), and (3) the availability of FDs (sub-columns within large columns). Impossibility results are denoted by "×" and solvability by "✓". Impossibility results with stronger

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 507–508, 2008.

Table 1. Overview of our results [2]. An arrow depicts a logical implication

assumptions	**no stable storage**		**stable storage**	
	$\Diamond\mathcal{P}$	$\mathcal{P}$	$\Diamond\mathcal{P}$	$\mathcal{P}$
one correct	×	×	×	× Lemma 2
correct majority	×	× Sect. 4.1	✓ Theorem 2	✓
one always up	×	✓ Theorem 1	× Lemma 1	✓ Theorem 1
correct majority & one always up	× [1]	✓	✓	✓
more always up than incorrect	✓ [1]	✓	✓ [1]	✓
always up majority	✓ Sect. 4.3	✓	✓ Sect. 4.3	✓

assumptions imply impossibility for cases with weaker ones, while solvability with weak assumptions implies solutions with stronger ones (indicated by arrows). The section and theorem numbers in the table refer to the relevant parts in the full version of this work [2].

The case of $\Diamond\mathcal{P}$ and more always up than incorrect processes was proven to be the weakest for consensus [1] and thus, all weaker process state assumptions are impossible. We first focus on $\mathcal{P}$ and the unavailability of stable storage. We argue that at least one always up process is necessary (Sect. 4.1) and sufficient (Theorem 1). Then we weaken $\mathcal{P}$ to $\Diamond\mathcal{P}$ and present a modular algorithm for the always up majority of processes case (Sect. 4.3). We then turn to the case where processes are allowed to use stable storage. We first prove two impossibility results regarding the presence of correct processes. The first one arises in the case where we have only $\Diamond\mathcal{P}$ and one always up process (Lemma 1) and the second in the case where $\mathcal{P}$ is available and one correct process is present (Lemma 2). We then give an algorithm for the remaining case (Theorem 2): It uses $\Diamond\mathcal{P}$, a majority of correct processes, and some minimal insight about the used crash-stop consensus algorithm which needs to be saved on stable storage.

References

1. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash recovery model. Distributed Computing 13(2), 99–125 (2000)
2. Freiling, F.C., Lambertz, C., Majster-Cederbaum, M.: Easy Consensus Algorithms for the Crash-Recovery Model. Tech. Report, TR-2008-002, Univ. of Mannheim (2008)

Evaluating the Quality of a Network Topology through Random Walks

A.-M. Kermarrec, E. Le Merrer, B. Sericola, and G. Trédan

IRISA/INRIA - Campus de Beaulieu, 35042 Rennes, France

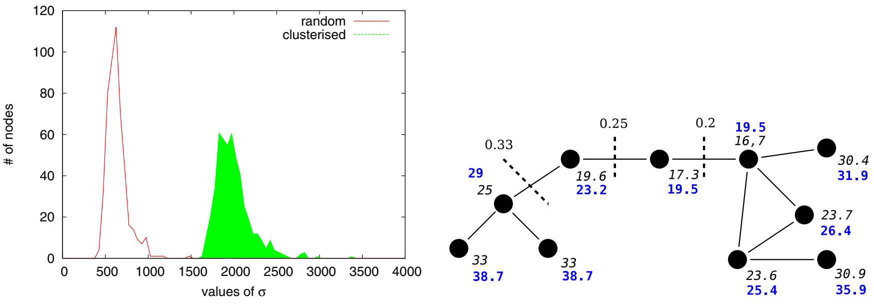
In this brief announcement we propose a distributed algorithm to assess the connectivity quality of a network, be it physical or logical. In large complex networks, some nodes may play a vital role due to their position (*e.g.* for routing or network reliability). Assessing global properties of a graph, as importance of nodes, usually involves lots of communications; doing so while keeping the overhead low is an open challenge. To that end, *centrality* notions have been introduced by researchers (see *e.g.* [Fre77]) to rank the nodes as a function of their importance in the topology. Some of these techniques are based on computing the ratio of shortest-paths that pass through any graph node. This approach has a limitation as nodes "close" from the shortest-paths do not get a better score than any other arbitrary ones. To avoid this drawback, physician Newman proposed a centralized measure of *betweenness centrality* [New03] based on random walks: counting for each node the number of visits of a random walk travelling from a node i to a target node j, and then averaging this value by all graph source/target pairs. Yet this approach relies on the full knowledge of the graph for each system node, as the random walk target node should be known by advance; this is not an option in large-scale networks. We propose a distributed solution[1] that relies on a single random walk travelling in the graph; each node only needs to be aware of its topological neighbors to forward the walk.

We consider a n node static and connected network, represented as an undirected graph $\mathcal{G} = (V, E)$, with n vertices and m edges. Each node $i \in V$, maintains its topological neighbors in $\mathcal{G}$, its degree is noted d_i.

Our approach relies on the local observation, at each node, of the variations of *return times* of a random walk on the topology. Intuitively, the more regular the visits on nodes, the more "well knitted" the network, as the walk is not periodically stuck in poorly connected parts of $\mathcal{G}$. The algorithm we propose is very simple: each graph node logs and computes the standard deviation of the return times of a permanent *unbiased random walk*, running on the topology. An unbiased random walk has a *stationary distribution* π_i, for all $i \in V$, that is $1/n$. A biased (or simple) random walk, puts mass on high degree nodes, as $\pi_i = d_i/2m$. We unbias the random walk by using the Metropolis-Hastings method [SRD+06], in order to capture on nodes passage times that are only dependant of the connectivity of the graph. Every node i in $\mathcal{G}$ joins the detection process on the first passage of the walk, by creating an array Ξ_i that logs every return time. A simple solution to capture irregularity of visits on nodes is to proceed

[1] Detailed report can be found at [KMST08].

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 509–511, 2008.



(a) (b)

as follows: after the third return, a node i computes the standard deviation σ_i of the return times recorded in Ξ_i (*i.e.* the time needed for the walk, starting at i, to return to i).

Figure 1(a) presents σ values resulting on nodes after 5.10^5 walk steps, on *(i)* an *Erdös-Rényi random graph* (two nodes are connected with proba. $p = \frac{2 \ln n}{n}$) usually seen as an "healthy" graph, and *(ii)* on a *barbell graph*, consisting of two distinct cliques connected by only a link, as a model of pathological topology. $n = 1000$ in both cases. The thiner the spike, the more homogeneously the σ values are distributed on the nodes, as a result of our algorithm. The clusterized graph's values are concentrated around 4 times the average value for the random graph, indicating that visits of the walk on the nodes are far more irregular, due to the topology characteristics. Figure 1(b) plots on a micro network, with the standard deviation of random walk visits on every node (italic values), the theoretical values [KMST08] (bold), and the three lowest *conductance* values for this graph (over dashed lines). We observe that the importance of nodes in the topology is effectively correlated to the inverse of their value order: smallest σ values are effectively related to the smallest conductance values.

Distributed auto-detection of network connectivity issues could be set up, as on following examples. Once σ values on nodes allow a ranking with respect to their importance (small σ are most critical nodes), nodes may compare their σ (*e.g.* periodical *gossip* communications), to deduce vital nodes, with respect to the network topology. Another solution is for nodes to exchange passage times; suppose nodes a and b, exchange their sets Ξ_a and Ξ_b. The ratio $r_{a \to b} = \frac{\sigma(\Xi_a \cup \Xi_b)}{\sigma(\Xi_a)}$ can be exploited as a distributed cluster detector: if a and b are located in two different clusters, then the standard deviation of the union of passage dates is small, so that $r_{a \to b}$ is low. On the other side, if nodes a and b are in the same cluster, the walk is likely to hit both at very close periods, so that $r_{a \to b}$ converges to 1. a and b are thus both able to identify their respective relative position in a fully decentralized way.

Algorithm time complexity is related to graph *cover time*, as each node needs at least three visits to start an estimation process. Cover time upper bound for biased random walk is known [Fei95]: $\frac{4}{27}n^3 + o(n^3)$, for the degenerated *lollipop* graph.

References

[Fei95] Feige. A tight upper bound on the cover time for random walks on graphs. RSA 6 (1995)

[Fre77] Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry 40(1), 35–41 (1977)

[KMST08] Kermarrec, A.-M., Le Merrer, E., Sericola, B., Trédan, G.: Rr-6534 inria - evaluating topology quality through random walks (2008)

[New03] Newman, M.E.J.: A measure of betweenness centrality based on random walks (September 2003)

[SRD$^+$06] Stutzbach, D., Rejaie, R., Duffield, N., Sen, S., Willinger, W.: On unbiased sampling for unstructured peer-to-peer networks. In: IMC 2006, pp. 27–40. ACM, New York (2006)

Brief Announcement: Local-Spin Algorithms for Abortable Mutual Exclusion and Related Problems*

Robert Danek and Hyonho Lee

Department of Computer Science
University of Toronto
{rdanek,hlee}@cs.toronto.edu

Introduction. A mutual exclusion (ME) algorithm consists of a *trying protocol (TP)* and *exit protocol (EP)* that surround a *critical section (CS)* and satisfy the following properties: **mutual exclusion**: at most one process is allowed to use the CS at a given time; **lockout freedom**: any process that enters the TP eventually enters the CS; and **bounded exit**: a process can complete the EP in a bounded number of its own steps. A First-Come-First-Served (FCFS) ME algorithm [1] additionally requires processes to enter the CS in roughly the order in which they start the TP. Once a process has started executing the TP of a ME algorithm, it has committed itself to entering the CS, since the correctness of the algorithm may depend on every process properly completing its TP and EP.

Abortable ME [2,3] is a variant of ME in which a process may change its mind about entering the CS, e.g., because it has been waiting too long. A process can withdraw its request by performing a bounded section of code, called an *abort protocol (AP)*.

We discuss novel algorithms for abortable ME and FCFS abortable ME. These algorithms are local-spin, i.e., they access only local variables while waiting and perform only a bounded number of remote memory references (RMRs) in the TP, EP and AP. Using these algorithms, we obtain new local-spin algorithms for two other additional problems: group mutual exclusion (GME) [4] and k-exclusion [5].

Summary of Results. All our algorithms use only atomic reads and writes. We call these *RW algorithms*. Our main result is the first RW local-spin abortable ME algorithm. It has $O(\log N)$ RMR complexity per operation and $O(N \log N)$ (total) space complexity for N processes. It is a surprisingly simple modification of the RW local-spin ME algorithm of Yang and Anderson [6]: we allow a process waiting in an unbounded loop in the TP to abort by executing the EP.

We also have a transformation that converts any abortable ME algorithm with $O(T)$ RMR complexity and $O(S)$ space complexity to an FCFS abortable ME algorithm that has $O(N+T)$ RMR complexity and $O(S+N^2)$ space complexity. Given an abortable ME algorithm, we add code to the beginning of its TP: a process p builds a "predecessor" set, which includes all processes that must enter

* Research supported in part by the National Sciences and Engineering Research Council of Canada.

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 512–513, 2008.

the CS before it. Process p then waits for its predecessors to finish the CS, during which time it can abort. We also add code to the end of the EP and AP: p signals to other processes that may have p in their predecessor set. This transformation combined with the modified Yang and Anderson algorithm yields the first RW local-spin FCFS abortable ME algorithm. It has $O(N)$ RMR complexity and $O(N^2)$ space complexity. This also uses only bounded registers, so it yields a positive solution to an open problem mentioned by Jayanti [3].

Danek and Hadzilacos [7] presented a number of transformations using only reads and writes that convert any FCFS abortable ME algorithm that has $O(T)$ RMR complexity and $O(S)$ space complexity into a local-spin GME algorithm that has $O(N+T)$ RMR complexity and $O(S+N^2)$ space complexity. Together with our FCFS abortable algorithm, this leads to the first RW local-spin GME algorithm. It has $O(N)$ RMR complexity and $O(N^2)$ space complexity.

Lastly, we convert any abortable ME algorithm that has $O(T)$ RMR complexity and $O(S)$ space complexity to a k-exclusion algorithm that has $O(k \cdot T)$ RMR complexity and $O(k \cdot S)$ space complexity, but is not fault-tolerant. The transformation uses k instances of an abortable ME algorithm.

When a process enters the TP of the k-exclusion algorithm, it performs all k instances of the abortable mutual exclusion algorithm concurrently (for example, repeatedly performing one step of each in round-robin order) until it enters the CS of one of the instances. When the process enters the CS of the jth abortable ME algorithm, it finishes or aborts its execution of all other instances before entering the CS of the k-exclusion algorithm. When the process finishes the CS of the k-exclusion algorithm, it performs the EP of the jth abortable ME algorithm.

Applied to our abortable ME algorithm, this yields the first RW local-spin k-exclusion algorithm. It has $O(k \cdot \log N)$ RMR complexity.

Acknowledgments. We thank Faith Ellen and Vassos Hadzilacos for their numerous helpful suggestions during the writing of this paper.

References

1. Lamport, L.: A new Solution of Dijkstra's Concurrent Programming Problem. Communications of the ACM 17(8), 453–455 (1974)
2. Scott, M.L.: Non-blocking Timeout in Scalable Queue-based Spin Locks. In: The 21st Annual Symposium on Principles of Distributed Computing (July 2002)
3. Jayanti, P.: Adaptive and Efficient Abortable Mutual Exclusion. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (July 2003)
4. Joung, Y.J.: Asynchronous group mutual exclusion. Distributed Computing 13(4), 189–206 (2000)
5. Anderson, J.H., Moir, M.: Using local-spin k-exclusion algorithms to improve wait-free object implementations. Distributed Computing 11(1), 1–20 (1997)
6. Yang, J.H., Anderson, J.H.: A Fast, Scalable Mutual Exclusion Algorithm. Distributed Computing 9(1), 51–60 (1995)
7. Danek, R., Hadzilacos, V.: Local-spin group mutual exclusion algorithms. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 71–85. Springer, Heidelberg (2004)

Data Failures

Simona Orzan[1] and Mohammad Torabi Dashti[2]

[1] TU/e, The Netherlands
[2] ETH Zurich, Switzerland

To improve the theoretical understanding of the byzantine model and enable a modular design of algorithms, we propose to decompose the byzantine behaviour into a *data failure* behaviour and a *communication failure* behaviour. We argue that the two failure types are orthogonal and we point out how they generate a range of several new interesting failure models, which are less difficult than byzantine, but different than the already well understood crash model. Such intermediate models are relevant and subject to recent studies, e.g. [2].

A formal algorithm model. Let V be a set of variable names. Every process owns a local variable set $D \in V^*$, an input buffer ib, modelled as a list of messages $v \hookleftarrow i$ (value v sent by process i), and an output buffer ob, which is a list of messages $v \hookrightarrow i$ (value v has to be sent to process i). A *local algorithm* is a finite sequence of guarded actions: $action ::= g \Rightarrow \mathsf{in}(x, j) \mid g \Rightarrow \mathsf{out}(v, i) \mid g \Rightarrow x := x' \mid g \Rightarrow \mathsf{loop}\ actblock$,where $actblock ::= action \mid action; actblock$, and g is a predicate on the variables in D. $\mathsf{in}(x, j)$ transforms the current local state $(D, ib + [v \hookleftarrow i], ob)$ into $(D[v/x, i/j], ib - [v \hookleftarrow i], ob)$, $\mathsf{out}(v, i)$ transforms it into $(D, ib, ob + [v \mapsto i])$ and $x := x'$ into $(D[x/x'], ib, ob)$. We consider, w.l.o.g., all variables of type `Nat`.. These specifications generate, sets of traces or possible runs of the algorithms described. A *distributed algorithm* for N processes is a tuple $\mathcal{A} : \langle A_1 \ldots A_N \rangle$ with A_i algorithms written in the language above. A *problem* is a tuple $(\boldsymbol{x}, \boldsymbol{y}, Req)$, with $\boldsymbol{x}$ and $\boldsymbol{y}$ the input and output vectors, and Req a list of consistent requirements on $\boldsymbol{x}, \boldsymbol{y}$ expressed as LTL formulae.

Orthogonal failures. We define two types of deviations from the expected behaviour: a *data failure* is inserting assignments ($g \Rightarrow x := x'$) actions in an algorithm description; a *communication failure* is inserting or deleting one or more $g \Rightarrow \mathsf{in}$ or $g \Rightarrow \mathsf{out}$ actions in an algorithm description. These two aspects are mutually independent and their combination give rise to interesting failure models, as discussed further. When both types of failures are considered simultaneously, the full power of a byzantine model is reached:

Theorem 1 (byzantine decomposition). *Any byzantine behaviour can be simulated by a combination of data and communication failures.*

A map of failure models. The *crash* failure is a communication failure where, from one point on, all out actions are ignored; the *general omission* or *crash-recovery* failure is a communication failure where some of the in and out actions are ignored; the *muteness* failure, as in,e.g. [1] is a communication failure where, from one point on, all in and out actions are ignored, except actions of the form $\mathsf{out}(heartbeat, H)$ where H implements a failure detector. These are all

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 514–515, 2008.

concerned with communication failures, but do not consider any form of data tempering. The new data failure dimmension generates the new models DF, DCF, DCRF. An overview:

COMMUNICATION ↓ DATA→	*no failure*	*edit failure*
no failure	ideal sync	**DF**
in/out *permanently ignored*	crash	**DCF**
some in/out *ignored*	crash-recovery	**DCRF**
some in/out *ignored or/and added*	**random-com**	BYZ

DF, the pure data failure model. In DF, processes are prone to data failures, but communication works flawlessly. This allows designing of detection mechanisms to deal specifically with data corruption.

It is impossible to achieve consensus when processes follow their algorithm, but may crash. We can show that this impossibility also holds in the complement model, when all processes are crash-free but their data is succeptible to errors.

Theorem 2 (general impossibility). *Let $P = (\boldsymbol{x}, \boldsymbol{y}, Req)$ be a non-trivial problem, i.e. at least one formula in Req depends on $\boldsymbol{x}$ and/or $\boldsymbol{y}$. In the DF model, there exist no terminating distributed algorithm $\mathcal{A}$ to solve problem P.*

So, also in superseeding models, in particular the byzantine one, problems cannot be solved without extra assumptions. Indeed, the classical solutions for multiparty computation problems rely on the famous $t < n/3$ assumption and several works on distributed consensus use byzantine failure detectors [1].

Data failure detection in DF. Algorithm design for communication failure environments essentially relies on failure detectors to enforce some level of synchrony. In DF, we are looking for similar mechanisms that could enforce some level of data reliability. Options are,e.g., encryption, check-sums, verification functions. A *corruption pattern* is a function $CP : \Omega \to 2^V$ mapping time t to the set of variables that got corrupted until time t in all local memories. A *process verifier* is a function $PV : \mathcal{P}roc \times \Omega \to 2^{\mathcal{P}roc}$, with $PV(p, t) = \{p_1 \dots p_k\}$ being a set of processes that own at least one corrupted variable until moment t. The perfect verifier always returns to p the whole set of corrupted messages or the whole set of corrupted processes, respectively. Using data failure detection, the impossibility of Theorem 2 can be circumvented:

Theorem 3 (DC solvable with a verifier and protected variables). *In DF, two-party distributed consensus is solvable in the presence of a perfect message verifier and a variable protection mechanism.*

References

1. Doudou, A., Garbinato, B., Guerraoui, R.: Encapsulating failure detection: from crash to byzantine failures. In: Blieberger, J., Strohmeier, A. (eds.) Ada-Europe 2002. LNCS, vol. 2361. Springer, Heidelberg (2002)
2. Widder, J., Gridling, G., Weiss, B., Blanquart, J.-P.: Synchronous consensus with mortal byzantines. In: Proceedings DSN, pp. 102–112 (2007)

Reliable Broadcast Tolerating Byzantine Faults in a Message-Bounded Radio Network

Marin Bertier, Anne-Marie Kermarrec, and Guang Tan

IRISA/INRIA-Rennes, France
{Marin.Bertier,Anne-Marie.Kermarrec,Guang.Tan}@irisa.fr

Abstract. We consider the problem of reliable broadcast tolerating Byzantine faults in a message-bounded radio network. Following the same model as in [4] and with a different assumption on the message bounds of nodes, we investigate the possibility of reliable broadcast given the communication range, message bounds of honest and dishonest nodes, and the maximum number of dishonest nodes per neighborhood.

1 Introduction

In the considered scenario of sensor network broadcast, there is a base station serving as message source. The task is to have the correct message delivered from the base station to all nodes in the network via multi-hop radio communications, despite some malicious nodes that can alter the message or cause collisions. Assuming a non-collision setting, Koo [3] first studies this problem and shows the maximum number of dishonest (bad) nodes per neighborhood, t, that can be tolerated by a broadcast protocol. In subsequent work [1], Bhandari et al. further prove that Koo's bound is a critical threshold. In [4], Koo et al. remove the non-collision assumption and show that the maximum t in a collision network remains the same as in a non-collision setting. One of their key assumptions is that a bad node can cause only a bounded number of collisions. This assumption allows a simple treatment for a good node to avoid collisions: if a single bad node can cause at most β collisions, then a good node can simulate a collision-free transmission by repeatedly sending a message $\beta t + 1$ times.

We study a similar problem to that of [4], but with a different assumption on the message bounds of nodes. We assume that *each of the nodes – both good and bad – has a bound on the total number of messages it can send, including correct, incorrect, and disruptive messages.* This tries to capture the fact that many network devices are extremely constrained in energy, thus a message budget for a node to perform a broadcast task is a reasonable assumption. Our aim is to explore the relations between the three quantities: m, m_f, and t, with respect to the possibility of broadcast. Here $m \in \mathbb{N}$ is the message bound for a good node, $m_f \in \mathbb{N}$ is the message bound for a bad node, and t is the maximum number of bad nodes in a single neighborhood. In particular, we consider the problem from the following perspective: Given m_f and t, how large m should be for reliable broadcast to be possible? We show that reliable broadcast can be achieved in a

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 516–517, 2008.

much more energy-efficient way than the naive scheme [4] in which every good node needs to send the message at least $tm_f + 1$ times.

We consider the network model described in [3]. Nodes are located on an infinite grid. All nodes have an integer transmission radius r. When no collision occurs, a message broadcast by a node (x, y) is heard by all nodes within its *neighborhood*, which is defined as the square of side length $2r$ centered at (x, y). Any single neighborhood contains $t < r(2r+1)$ bad nodes. A bad node can alter the message and cause collisions. When two nodes i and j perform a local broadcast at the same time, their common neighbor nodes can receive an arbitrary message (or no message at all). m_f is assumed to be known in advance by all nodes. The base station is always correct.

2 Our Results

Let $m_0 = \lceil \frac{2tm_f+1}{r(2r+1)-t} \rceil$. Our first result is the following theorem.

Theorem 1. *If $m < m_0$, then reliable broadcast is impossible.*

The second result is the following theorem.

Theorem 2. *If $m \geq 2m_0$, then reliable broadcast is achievable.*

We have designed a simple protocol to achieve reliable broadcast for $m \geq 2m_0$. If nodes are allowed to have heterogeneous message bounds, then the average message budget of nodes can be substantially reduced. The malicious behavior under consideration is the replacement of good nodes with bad nodes by the adversary, subject to the constraint of t and that the bad nodes still have the same message bound m_f.

Theorem 3. *If $\Theta(r^3)$ good nodes have message bound $m' = \frac{2tm_f+1}{\lceil (r(2r+1)-t)/2 \rceil} (\leq 2m_0)$ and the other good nodes have $m = m_0$, then there exists a protocol that achieves reliable broadcast on some configuration of m' and m.*

We have constructed a message bound configuration and a simple protocol to validate this. The result of Theorem 3 should be compared with the scheme suggested in [4] which requires every good node to have $m = 2tm_f + 1$ message bound, which is $\frac{1}{2}[r(2r+1) - t]$ times our bound.

References

1. Bhandari, V., Vaidya, N.H.: On reliable broadcast in a radio network. In: Proc. of ACM Symposium on Principles of Distributed Computing (PODC) (2005)
2. Bhandari, V., Vaidya, N.H.: Reliable broadcast in a wireless grid network with probabilistic failures. Technical Report, CSL, UIUC (October 2005)
3. Koo, C.-Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: Proc. of ACM Symposium on Principles of Distributed Computing (PODC) (2004)
4. Koo, C.-Y., Bhandari, V., Katz, J., Vaidya, N.H.: Reliable broadcast in radio networks: The bounded collision case. In: Proc. of ACM Symposium on Principles of Distributed Computing (PODC) (2006)

Brief Announcement: Eventual Leader Election in the Infinite Arrival Message-Passing System Model

Sara Tucci-Piergiovanni and Roberto Baldoni

Sapienza Università di Roma, Italy

Emerging distributed systems have a dynamic structure that is self-defined at any time by entities that autonomously decide to locally run the same distributed application. As extreme, a distributed system might thus also cease its existence when no entity is currently active while at some later moment new entities arrive and connect to each other to form again the system. Therefore the set of entities that over time might form the system is potentially infinite. This infinite arrival model is the key distinguishing factor between dynamic systems and traditional distributed systems where, on the contrary, the set of system entities is fixed since the deployment of application components is controlled and managed. In this model not only the set of correct processes is not known in advance, but no finite set containing the set of correct processes is known in advance. This actually is a higher level of uncertainty to be mastered in dynamic systems. This makes the study of possible implementations of failure detectors, as Ω, of paramount importance and at the same time makes the problem of realizing such failure detector far from being trivial. The uncertainty posed by infinite arrival models brings to two different issues (1) discovering the finite set of processes currently running and (2) dealing with a possible infinite set of non-correct processes that may wake up at any time, covering with their up times the whole computation.

By assuming only temporary partitions, each process can witness its presence in the system by periodically sending a heartbeat and its identifier, to let other up processes know it and consider it as part of the system. At first glance, it may seem that discovering the finite set of processes currently running is the hard part of the problem, herein solved by assumption, and that among this set it is possible to eventually select a unique leader following well-known solutions employed in the crash-failure model. However, in the crash-failure model the objective is to assure that any correct process is eventually able to univocally select one correct process among an initial set containing a finite number of non-correct processes. By eventually suspecting as crashed those processes whose heartbeats/messages stops to arrive, all the complexity lies in avoiding to falsely suspect at least one correct process infinitely often. Solving this issue means that eventually and permanently at least one correct process will be considered as alive. If more than one correct process is considered as alive, processes can be totally and locally ordered at each process sorting their identifiers. By establishing this total order it is possible to apply at each process a local deterministic rule that independently chooses the same process (e.g. the one with

G. Taubenfeld (Ed.): DISC 2008, LNCS 5218, pp. 518–519, 2008.

the lowest identifier) as leader. In contrast to this crash-failure model with a finite set of processes, the infinite arrival model implies that heartbeats may arrive from correct and non-correct processes over the entire computation. A list of alive processes built by sorting process identifiers will continually include correct processes and up but non-correct processes since for any identifier assigned to a correct process, an infinite set of non-correct processes with a lower identifier or higher identifier (e.g. processes running on nodes with a lower, respectively higher, IP address) may arrive over the entire computation. Without a selection of the only set of correct processes, any choice on the flat mixed set may lead to elect a non-correct leader infinitely often, violating the specification of Ω.

The technical report [1] proposes an implementation of the failure detector in a message passing system where infinitely many processes may arrive and depart over time and where the number of processes which may be simultaneously up is finite and cannot exceed a known bound C. The implementation is composed by two different algorithms. The first algorithm implements a new lower-level oracle called HB^* which provides a list called alive of length C containing processes deemed to be up in the system. The alive lists have to eventually and permanently include correct processes in the first positions. The algorithm implementing HB^* sorts processes in alive by their age. The age is a sequence number on heartbeats. A correct process can just getting older and older, i.e. its age never stops increasing. A non-correct process will reach a finite age and will turn down. By assuming unknown bounds on message losses and message delay, there will exist a point of time after which no non-correct process can be perceived as older than any correct process. Sorting by age is then a way to eventually have correct processes in the first positions of the alive lists. The oracle, however, does not guarantee an eventual total order on correct processes. Moreover, in any run the set of correct processes may be lower than C, and since the non-correct processes may continually arrive, the alive lists could always include other $C - b$ non-correct processes. The second algorithm eventually outputs a unique correct leader in the system and uses the alive lists provided by HB^*.

Actually it uses these lists and employs a mechanism to identify a subset of correct processes totally ordered. In this algorithm a known lower bound b on the number of correct processes is used, to let the algorithm safely choose, among the first b positions of alive lists, the set of leader candidates. Actually, the real number of correct processes can be greater than b, and different processes may have different processes in the first b positions. The algorithm will exchange alive lists and manipulate them by selecting the subset of processes eventually and permanently among the first b positions in all alive lists. A majority assumption on the number of correct processes w.r.t. the total number of processes concurrently running makes this selection possible.

Reference

1. Tucci Piergiovanni, S., Baldoni, R.: Eventual Leader Election in the Infinite Arrival Message-passing System Model. MIDLAB Tech Report #10-08, Sapienza Universitá di Roma (2008), http://www.dis.uniroma1.it/~midlab/publications.php

Author Index