Davide Frey
Michel Raynal
Saswati Sarkar
Rudrapatna K. Shyamasundar
Prasun Sinha (Eds.)

LNCS 7730

Distributed Computing and Networking

14th International Conference, ICDCN 2013
Mumbai, India, January 2013
Proceedings

Springer

Lecture Notes in Computer Science 7730

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

Davide Frey Michel Raynal
Saswati Sarkar Rudrapatna K. Shyamasundar
Prasun Sinha (Eds.)

Distributed Computing and Networking

14th International Conference, ICDCN 2013
Mumbai, India, January 3-6, 2013
Proceedings

Springer

Volume Editors

Davide Frey
IRISA/INRIA-Rennes
35042 Rennes Cedex, France
E-mail: davide.frey@irisa.fr

Michel Raynal
Institut Universitaire de France
IRISA-ISTIC Université de Rennes 1
35042 Rennes Cedex, France
E-mail: michel.raynal@irisa.fr

Saswati Sarkar
University of Pennsylvania
Philadelphia, PA 19104, USA
E-mail: swati@ee.upenn.edu

Rudrapatna K. Shyamasundar
Faculty of Technology and Computer Science
Tata Institute of Fundamental Research
Mumbai 400005, India
E-mail: shyam@tifr.res.in

Prasun Sinha
Ohio State University
Columbus, OH 43210-1277, USA
E-mail: prasun@cse.ohio-state.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-35667-4 e-ISBN 978-3-642-35668-1
DOI 10.1007/978-3-642-35668-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012953914

CR Subject Classification (1998): C.2, D.1.3, D.2.12, C.2.4, D.4, F.2, F.1.2, H.4

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

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

Printed on acid-free paper

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

Preface

The International Conference on Distributed Computing and Networking (ICDCN) was launched 14 years ago, initially as an International Workshop (IWDC), and within a few years it gained visibility as one of the leading international conferences in this area. ICDCN is one of the very few conferences where researchers in distributed computing get to meet their networking counterparts on the same scientific platform. In addition to presenting scientific research, ICDCN organizes various forums such as specific workshops, tutorials, and panels of current importance, and it encourages strong participation from industry.

The current edition of ICDCN 2013 was the 14th edition of the series. This volume contains papers presented at ICDCN 2013 held at Tata Institute of Fundamental Research, Mumbai, during January 3–6, 2013. The conference consisted of two tracks, "Distributed Computing" and "Networking", and received in all 149 submissions.

The Distributed Track consisted of 67 submissions, each of which received at least three reviews by the track's Program Committee (PC) members as well as by selected external reviewers. Based on these reviews and discussions among PC members, 18 regular and 3 short papers were selected for inclusion in the proceedings. The Networking Track consisted of 72 papers, each of which was reviewed by at least three PC members and a few selected external referees. Based on the reviews, and the PC discussions spead over two weeks, the PC selected 9 regular papers, 2 short papers, and 7 poster papers.

The ICDCN 2013 program included five keynote speakers: Leslie Lamport (Microsoft Research), Hari Balakrishnan (CSAIL, MIT), David Culler (UC, Berkeley), Yoram Moses (Technion), and the *R. Narasimhan Memorial* keynote lecture by Sanjit Seshia (UC, Berkeley). On behalf of the entire PC, we would like to take this opportunity to thank them for accepting our invitation and contributing to the success of the conference. We are confident that the conference attendees enjoyed and were enlightened by the various keynote addresses and benefited further from productive discussions with them.

On behalf of the PC and conference organizers, we would like to express our gratitude to the PC members and the external reviwers who took part in the review process: their hard work and responsiveness made it possible to stick to our tight review schedule and arrive at a quality programme. We are sure the reviews would enable authors to further refine their works and arrive at new openings/initiatives.

It is our pleasure to thank ACM SIGCOMM for according us ACM *Incooperation status* and IEEE CS TCDP for according us "Technical cooperation status."

We are grealy indebted to Tata Institute of Fundamental Research both for the generous financial and organizational support. Thanks go the faculty of the School of Technology and Computer Science (STCS) for the support in organizing the conference. We thank the staff of STCS, in particular John Barretto, Ravi Naik, and Kishore Menon (Public Relation Officer) for their efforts in making the organization and running of the conference smooth.

We express our gratitude to our sponsors C-DAC (Centre for Advanced computing, DeITY (Department of Electronics and Information Technology of the Government of India), Quallcomm, Microsoft Research, HP Research, IBM Research, Tata Consultancy Services (TCS) as well as other organizations like IARCS and DRDO (GOI) that supported the conference in various ways.

From the practical perspective of conference management, we would like to acknowledge that the whole process of paper submission, selection, and compilation in the proceedings was greatly simplified by the friendly interface of the EasyChair conference system (http://www.easychair.org). We owe a lot to the EasyChair creators and maintainers for their commitment to the scientific community.

To conclude, we are confident that the 14th edition of ICDCN will contribute to enhancing knowledge in the broad areas of distributed computing and networking and will further add to the prestige and reputation of ICDCN. We are sure that the readers will find the proceedings informative and they will assist them in their upcoming research and development activities.

23 October 2012

Rudrapatna Shyamasundar
Michel Raynal
Davide Frey
Saswati Sarkar
Prasun Sinha

Conference Organization

General Chair

R.K. Shyamasundar	Tata Institute of Fundamental Research (TIFR)

Technical Program Committee Chairs

Distributed Computing Track Co-chairs

Michel Raynal	IRISA
Davide Frey	INRIA

Networking Track Co-chairs

Saswati Sarkar	University of Pennsylvania
Prasun Sinha	Ohio State University

Workshop Chairs

R.K. Ghosh	IIT Kanpur
Mohan J. Kumar	UTA
Maitreya Natu	TRDDC Pune

Tutorial Co-chairs

N.V. Krishna	IIT Madras
Mainak Chatterjee	University of Central Florida

Doctoral Forum Chair

Santonu Sarkar	Infosys

Publicity Co-chairs

Santonu Sarkar	Infosys
Yong Cui	Tsinghua University, Beijing
Mario Di Francesco	Aalto University and University of Texas at Arlington
Nirmalya Roy	Wichita State University

Industry Track Chair

Ankur Narang	IBM Research

Steering Committee Co-chairs

Sajal K. Das	University of Texas at Arlington
Sukumar Ghosh	University of Iowa

Webmaster

Ravikumar J. Naik	TIFR

Conference Secretary

John Barretto	TIFR

Technical Program Committee

Distributed Computing Track

Yehuda Afek	University of Tel Aviv
Mustaque Ahamad	Georgia Tech
Anish Arora	Ohio State University
Roberto Baldoni	University of Rome La Sapienza
Jiannong Cao	Polytechnic University of Hong Kong
Gregory Chockler	IBM, Haifa
Ajoy K. Datta	University of Las Vegas
Xavier Defago	JAIST, Kanazawa
Shlomi Dolev	Ben-Gurion University
Panagiota Fatourou	University of Crete
Hugues Fauconnier	LIAFA, Université de Paris 7
Christof Fetzer	Dresden University of Technology
Vijay K. Garg	University of Texas at Austin
Chryssis Georgiou	University of Cyprus
Prasad Jayanti	Dartmouth College
Petr Kuznetsov	Technical University of Berlin
Ajay Kshemkalyani	University of Chicago
Toshimitsu Masuzawa	University of Osaka
Yoram Moses	The Technion, Haifa
Gilles Muller	INRIA/LIP6, Paris
Paritosh K. Pandya	TIFR
Sergio Rajsbaum	Universidad Nacional Autonama de Mexico
R. Ramanujam	IMSc, Chennai
Matthieu Roy	LAAS-CNRS, Toulouse
Alex A. Shvartsman	University of Connecticut and MIT
Ulrich Schmidt	University of Vienna
Gadi Taubenfeld	Interdisciplinary Center at Herzliya
Paulo Verissimo	University of Lisbon
Krishnamurthy Vidyasankar	University of Newfoundland

Networking Track

Alexandre Proutiere	KTH Royal Institute of Technology
Amarjeet Singh	IIIT Delhi
Anurag Kumar	Indian Institute of Science
Arunabha Sen	Arizona State University
Arup Acharya	IBM T.J. Watson Research Center
Benyuan Liu	University of Massachusetts at Lowell
Cormac J. Sreenan	University College of Cork
Dirk Pesch	Cork Institute of Technology
George Kesidis	National Science Foundation and Penn State University
Hwangnam Kim	Korea University
Joy Kuri	Indian Institute of Science
Koushik Kar	Rensselaer Polytechnic Institute
Krishna Sivalingam	IIT Madras
Kyu-Han Kim	HP Labs, Palo Alto
Mani Srivastava	University of California at Los Angeles
Munchoon Chan	National University of Singapore
Ness Shroff	Ohio State University
Niloy Ganguli	IIT Kharagpur
Paolo Bellavista	University of Bologna
Paolo Santi	IIT, CNR
Prasanna Chaporkar	IIT, Bombay
Prithwish Basu	BBN Technologies
R. Srikant	University of Illinois at Urbana-Champaign
Samir Das	SUNY at Stony Brook
Sanjay Jha	University of New South Wales
Sotiris Nikoletseas	CTI, Patras
Srinivas Shakkotai	Texas A&M University
Subir Biswas	Michigan State University
Sudip Misra	IIT, Kharagpur
Tara Javidi	University of California at San Diego
Thyagarajan Nandagopal	National Science Foundation
Tomasso Melodia	SUNY at Buffalo
Tracy Camp	Colorado School of Mines
Vartika Bhandari	Google, New York
Vikram Srinivasan	Bell Labs India
Vinod Prabhakaran	TIFR
Vinod Sharma	Indian Institute of Science
Xinbing Wang	Shanghai Jiao Tong University
Young-Bae Ko	Ajou University
Zizhan Zheng	Ohio State University

Technical Co-sponsors

acm In-Cooperation

IEEE COMPUTER SOCIETY

Premium Sponsors

tifr

Platinum Sponsors

C-DAC

Gold Sponsors

QUALCOMM

सत्यमेव जयते

Silverplus Sponsors

Microsoft Research

Silver Sponsors

hp

IBM

TATA CONSULTANCY SERVICES
Experience certainty.

Supporting Organizations

IARCS

Indian Association for Research in Computing Science
Excellence in Computing. Provably.

DRDO

Who Builds a House without Drawing Blueprints? (Invited Keynote Talk)

Leslie Lamport

Microsoft Research

Abstract. Architects draw detailed plans before a brick is laid or a nail is hammered. Programmers and software engineers don't. Can this be why houses seldom collapse and programs often crash?

Sense and Sensibility for Wireless Networks
(Invited Talk)

Hari Balakrishnan

M.I.T.
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139
hari@mit.edu

Abstract. Truly mobile devices such as smartphones and tablets are fast becoming the dominant mode of Internet access. People use these devices while moving through wide range of locations, often in quick succession. The rapid variations in network conditions experienced poses a significant challenge for wireless network protocols, degrading performance, and making users unhappy. This talk will describe two techniques to overcome this challenge: (1) spinal codes, a new class of rateless codes, and (2) a system to use sensors on mobile devices to improve network performance.

On Using Knowledge and Timing Information for Distributed Coordination (Invited Talk)*

Yoram Moses

EE Dept., Technion Israel
moses@ee.technion.ac.il

This talk will illustrate the interplay between knowledge, action and communication in a distributed setting. Fundamental to distributed an multi-agent systems is the fact that an agent can base her actions only on her locally available information, or *knowledge*. Some of the actions, resulting in communication between the agents, modify agents' knowledge. Consequently, there is a subtle inter-dependence between knowledge and action in a distributed setting. A formal theory of knowledge in multi-agent systems has been slowly emerging over the course of the last three decades [4, 3]. The talk will focus on three aspects:

- From specifications to knowledge: If preconditions for Bob performing action b are specified, the Bob must *know* that they hold before performing b. Moreover, if Alice should perform a only after Bob does b, then Alice must have knowledge about Bob's knowledge before she can act. Thus, specifications induce sometimes subtle knowledge preconditions.
- The role of time in the emergence of knowledge: In the absence of timing information, the evolution of knowledge is governed by Lamport's *happened before* relation [5]. The presence of clocks and timing information allows the passage of time to be used in conjunction with communication in coordinating nontrivial tasks. The generalisation of Lamport's relation will be discussed, along the lines of [1].
- An example: Depending on available time, an example of how knowledge and time can be used to obtain an optimal protocol for computing spontaneous global snapshots in a setting with synchronous clocks will be discussed. This is a synchronous counterpart of [2].

References

1. Ben-Zvi, I., Moses, Y.: Beyond Lamport's *Happened-Before*: On the Role of Time Bounds in Synchronous Systems. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 421–436. Springer, Heidelberg (2010)

* This work was supported in part by Grant 1520/11 of the Israel Science Foundation.

2. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. on Computer Systems 3(1), 63–75 (1985)
3. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (2003)
4. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. Journal of the ACM 37(3), 549–587 (1990)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)

Sustainable Energy Networks - The Distributed Computing and Networking Challenge of the Real World
(Invited Talk)

David E. Culler

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
culler@eecs.berkeley.edu

After 150 years of industrial (r)evolution, we must ask how we can bring Information Technology, which has brought such advances in productivity and performance, to bear on efficiency and sustainability. The problems of energy, climate, and sustainability are not crisp, clean technology challenges; they are complex Cyber-Physical Systems challenges and fundamentally distributed. In this talk, we explore how to apply lessons of the Internet, i.e., design principles for building distributed and robust communications infrastructures, to develop an architecture for a cooperative energy network that promotes reduction in use and penetration of renewable sources. We explore how pervasive information can improve energy production, distribution and use. We investigate how design techniques for scalable, power proportional computing infrastructures can translate to the design of a more scalable and flexible electric infrastructure, encouraging efficient use, integrating local or non-dispatchable generation, and managing demand through awareness of energy availability and use over time. Our approach is to develop a cyber overlay on the energy distribution system in its physical manifestations: machine rooms, buildings, neighborhoods and regional grids. A scaled series of experimental energy networks demonstrate monitoring, negotiation protocols, and control algorithms can eb designed to integrate information and energy flows in various settings, e.g., datacenter, building, and campus. We seek to understand broadly how information enables energy efficiencies: through intelligent matching of loads to sources, via various levels of aggregation, power proportional design, and by managing how and when energy is delivered to demand, adapted in time and form to available supply. Together these offer a path to a consumer-centric grid with supply-following loads.

Table of Contents

Verifying High-Confidence Interactive Systems: Electronic Voting and Beyond

Sanjit A. Seshia

EECS Department, UC Berkeley
sseshia@eecs.berkeley.edu

Abstract. Human interaction is central to many computing systems that require a high level of assurance. We term such systems as *high-confidence interactive systems*. Examples of such systems include aircraft control systems (interacting with a pilot), automobiles with self-driving features (interacting with a driver), medical devices (interacting with a doctor), and electronic voting machines (interacting with a voter). A major challenge to verifying the correct operation of such systems is that it is difficult to formally specify the human user's view of correct operation and perception of the input/output interface. In this paper, we describe a promising approach towards addressing this challenge that combines formal verification with systematic testing by humans. We describe an illustrative application of this approach to electronic voting in the U.S., and outline directions for future work.

1 Introduction

High-confidence computer systems are those that require a high level of assurance of correct operation. Many of these systems are *interactive* — they interact with a human being — and the human operator's role is central to the operation of the system. Examples of such systems include fly-by-wire aircraft control systems (interacting with a pilot), automobiles with driver assistance systems (interacting with a driver), medical devices (interacting with a doctor, nurse, or patient), and electronic voting machines (interacting with a voter). The costs of incorrect operation in all such systems can be very severe. Given the central role of the human operator/user in these systems, correct operation necessarily involves the human-computer interface; in fact, problems in this interface are often the source of failures. For instance, the U.S. Federal Aviation Administration has attributed several incidents, including fatal crashes, to problems in the human-computer interface [3]. Similarly, human errors in medical device use account for a large portion of medical errors, and many of these errors are due to poor design of the interface [5,6]. It is therefore essential to develop techniques to ensure correct operation of such high-confidence interactive systems.

Formal methods appear to provide the perfect fit for this need. Techniques such as model checking and automatic theorem proving have made tremendous advances over the past several years, with successful applications to the verification of hardware, software, and even biological systems. However, they are only applicable to systems where all the parts are formally specifiable. This presents a problem for interactive systems, where it is difficult or even impossible to formally specify the human user's view of

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 1–10, 2013.

correct operation and perception of the input/output interface. For instance, given a bitmap image on a touch-sensitive screen, automatically recognizing which portions of the screen a human would expect to form a touchable region (e.g. button) might require non-trivial image processing. Testing by humans is well-suited to checking that the system performs according to their expectation, since it eliminates the need to model the human operator. However, a limitation of conventional testing is that it is not exhaustive and it therefore can only find bugs; it cannot guarantee their absence.

In this paper, we advocate for a new approach that enables *the principled design of high-confidence interactive systems where correctness is certified through a combination of formal verification and testing by humans*. The approach has three main steps. First, we need to identify design principles that ease the task of verification and testing. Second, given a set of verification tasks, we need scalable algorithmic methods to tackle them. Third, a rigorous testing protocol must be specified for humans that also uses a tractable number of tests (ideally, low-degree polynomial in the size of the design, since running each test involves possibly several hours of human effort). We have performed an initial application of the approach to direct-recording electronic voting for U.S. elections [7].

Two key elements of the approach are *integrating design and verification* and using *formal verification to reduce the testing burden*. We propose that systems must be designed in a component-based manner following the principles of *determinism*, *independence*, and *unambiguity of the I/O interface*. Unambiguity of the output, for example, means that the system output is a 1-1 function of a subset of "core" state variables defined by the designer (possibly along with the system input), and nothing else. Determinism ensures that these core state variables evolve as a function only of their previous values and the system input. For modularity, the core state variables are partitioned amongst different system components. Independence ensures that updating the state of one component does not change the state of other components.

The above design principles are not new in and of themselves. However, the way in which we *combine* them for design and *verify* them on an implementation is novel. We verify *determinism*, *independence*, and *unambiguity* on the system code using encodings to satisfiability modulo theories (SMT) formulas, which are then proved automatically [2]. The results of the above verification is used to reduce the amount of human-driven testing to a tractable number. Our approach uses rigorous test coverage criteria to derive "test scripts" which humans can then use to evaluate the correctness of the system through testing.

In the rest of this paper, we sketch out the approach using electronic voting as an illustrative application domain (Section 2), and outline directions for future work (Section 3).

2 Electronic Voting

We begin by describing one of our key motivating applications: electronic voting. The work described in this section is joint with several colleagues and has been published earlier [7].

2.1 Preliminaries

Electronic voting is increasingly coming into use in the U.S. and around the world. It has brought with it concerns about reliability, accuracy, and trustworthiness. Existing electronic voting systems can be complex systems, often consisting of hundreds of thousands of lines of code, and a single bug anywhere in the code could potentially cause votes to be lost, misrecorded, or altered. As a result, it is difficult for independent evaluators to be confident that these systems will record and count the votes accurately. Moreover, in order to completely verify the voting machine, it is necessary to also verify the interface to human voters, i.e., that the operation of the voting machine is consistent with the behavior expected by voters.

The kind of voting machine that we focus on here is known as a direct-recording electronic (DRE) voting machine (although the principles we use here are applicable elsewhere). A DRE voting machine is one where voters interact with the machine to make their selections and then the votes are recorded electronically. The most familiar example is a touchscreen voting machine, where the voter interacts with a graphical user interface displayed on the screen by software running on the voting machine. The voter presses at various locations on the screen to register her selections, and the voting software records the voter's selections once she is ready to cast her ballot. DREs are widely deployed throughout the US: for instance, in 2008 and 2010 DREs were used by approximately 33% of registered voters [1,9]. While DRE's are commonly thought to be large, complex machines, in preliminary work [7] we have shown that a functional DRE can be designed as a finite-state machine directly in hardware, in custom Verilog code, so that there is no operating system or runtime software environment to verify.

Before we get into the notion of correctness for a voting machine, here are some voting-related terms that are used throughout the discussion.

Contest: A single race, such as for President, that a voter will vote on.

Ballot: The physical or electronic representation of all contests that a voter will be deciding on election day.

Candidate: A choice in a particular contest. The voter will typically make one or more selections from among two or more candidates for each contest on the ballot.

Voting Session: A voter's interaction with the machine from the time they are given a new ballot until the time their entire ballot is stored in non-volatile memory, i.e., until the time they cast the ballot. A session in U.S. elections typically comprises voting on several contests.

Cast: Casting a vote refers to the action taken at the end of a voting session that causes the selections made in all contests to be irrevocably written to non-volatile memory. Making a selection in a particular contest and moving on to the next contest is *not* considered casting a vote.

Selection State: The state representing the set of all candidates currently selected in a particular contest.

Button: A (usually rectangular) region on the screen. Touching anywhere within this region activates a particular functionality of the machine. The corresponding part of the screen image is often designed to provide the appearance of a physical button.

Given the above terms, consider the notion of correctness for a single voting session: given a series of inputs (button presses) from the human voter, the machine must record votes in accordance with the *expectation of the human voter*.

A human voter's expectation is difficult to *completely* specify formally. However, it is possible to *specify at least part of it formally*, as a state machine describing how the voting machine must update its internal state: for instance, how the set of candidates currently selected should be updated when the voter presses a button, or how the current contest is updated when the voter presses a "next" or "previous" buttong to navigate between contests, or that the ballot is irrevocably cast when the voter presses the "cast" button. This state machine serves to formalize our assumptions about the "mental model" of the user.

However, the specification machine does not specify all human expectations – for instance, what kinds of screen images should be produced by the voting machine. For example, if there is a rectangular region on the screen that displays "Thomas Jefferson" in some readable font, a human might expect that pressing that portion of the screen would select Jefferson, causing Jefferson's name to be highlighted and eventually causing a vote to be recorded for Jefferson if no other selection is subsequently made in this contest. However, because it involves semantic interpretation of the contents of a particular screen image by a human it is not clear how to specify this expected behavior in a precise, mathematical fashion. For instance, given a bitmap image, mechanically recognizing which portions of the screen a human would expect to correspond to a touchable region might require non-trivial image processing; moreover, mechanically determining that the touchable region should be associated with Thomas Jefferson might require computer vision algorithms and other complex computation. Formalizing these kinds of human expectations in a formal logic could be horribly messy, and probably error-prone as well.

For this reason, one might consider using a representative panel of human "test voters" in the validation process. In particular, we ask human voters to cast test votes on the voting machine during pre-election testing. We ask them to check that the machine seems to be working correctly and recording their votes accurately. We assume that if the machine behaves in a way inconsistent with their expectations, they will notice and complain. Consequently, if the voting machine passes all of these tests, then at least we know that the voting machine has behaved in a way consistent with human expectations during those tests.

We propose that *such human-driven testing can be supported by formal verification*. For instance, we can formally verify that the voting machine (as implemented in code or in hardware) behaves *deterministically*. This ensures that the voting machine will behave the same way on election day as it did in pre-election testing.

However, this verification alone is not enough to provide useful guarantees in practice, because the number of tests needed to exhaustively exercise all possible machine behaviors is astronomically large. For instance, in an election with N contests and k choices in each contest, the number of different ways to vote (assuming voters are only

allowed to vote for a single candidate in each contest) is k^N, an exponential function of N. Taking into account the possibility to change one's selections in a contest as many times as one likes, the number of ways to interact with the voting machine becomes infinitely large. Clearly, we cannot exhaustively try all of these possibilities in pre-election testing: we need something more selective.

The burden of testing can be reduced by a combination of principled design and formal verification. In the above scenario, we can ensure that only $O(kN)$ tests are needed. Roughly speaking, if the state and behavior for each contest is *independent* of the state of all other contests, it suffices to choose a test suite that attains 100% transition coverage in each individual contest and of navigation between contests, rather than 100% coverage of the whole voting machine's statespace. This can be achieved with $O(k)$ tests per contest, since the state space in a single contest is only of size $O(k)$ (whereas the statespace for the entire voting machine has size $O(k^N)$ and thus would require exponentially many tests to fully cover). Such independence properties can be verified using formal verification.

Each contest can be viewed as a separate *logical component* of the voting machine. Ideally, the design should be structured into logical components in a manner so as to ease the formal verification of *independence* of one logical component on another.

Finally, we also need to verify that the input/output interface of the voting machine is not ambiguous; e.g., that the machine cannot output the same output screen for two different internal states (selection state and contest number). One way of formalizing this is to show that the output bitmap generated by the voting machine code is an injective (1-1) function of the selection state and contest number. Such a *injectivity* property can also be verified by formal verification.

To summarize, verifying that a voting machine meets human expectations must involve the following steps:

- Formalizing part of the human mental model as a finite-state machine;
- Designing the voting machine using *logical components* so that it satisfies the properties of *determinism*, *independence*, and *unambiguity* of input/output;
- Formally verifying that the design actually satisfies the above properties, and
- Testing of the input/output interface by humans, where ideally each logical component (contest) can be tested independently so that the overall number of tests grows polynomially with the number of such components.

For the first step, it is relatively easy to formalize correct operation of the voting machine (informally described above) as a finite-state machine $\mathcal{P}$. We term this state machine the *specification voting machine* — it is intended to capture the typical mental model that a voter has for U.S. elections. Details of this model may be found in our paper [7]; in essence, it specifies the initial state (start in the first contest with no selections), how one can navigate between contests and select (and deselect) candidates within a contest, and what happens when the "cast" button is pressed to finalize one's votes. The important point for the rest of the paper is that the operation can be formalized as a (finite) state machine.

2.2 Verifying Independence, Determinism, and Injectivity

We now describe how SMT solving can be used for verification of the three key properties: independence, determinism, and injectivity. In order to perform formal verification, the implementation (code) of the voting machine is automatically transformed into a finite-state transducer model. For our Verilog implementation [7], this is a trivial step.

Independence and determinism both involve checking that a variable v depends only on some specified set $W = \{w_1, \ldots, w_n\}$ of variables, and *nothing else*. In other words, we must verify that v can be expressed as a deterministic function of the other variables: $v = f(w_1, \ldots, w_n)$, for some function f, or in shorthand, $v = f(W)$. Put another way, we want to check that v deterministically depends on W, and only W, i.e., for every other variable $x \notin W$, v is conditionally independent of x given W. We verify this kind of property by formulating it as a Boolean satisfiability (SAT) problem (for completely bit-level designs) or as a satisfiability modulo theories (SMT) problem for designs specified at higher levels of abstraction.

The first step is to encode a step of the transducer as a Boolean formula. We begin by introducing some notation. We assume there is a set S of state variables, so that each valuation of values to these variables corresponds to a state of the system. Similarly, let I be a set of input variables, and O a set of output variables. For each state variable s, let the variable s' denote the previous value of s; let S' denote the set of these variables. Then we can write the transition relation as a function δ, which expresses the state as a function of the previous state and the input via the relation $S = \delta(S', I)$. (This is shorthand for $s_i = \delta_i(s'_1, \ldots, s'_k, i_1, \ldots, i_\ell)$ for $i = 1, \ldots, k$, assuming $S = \{s_1, \ldots, s_k\}$ and $I = \{i_1, \ldots, i_\ell\}$.) Similarly, we assume the output function is modeled as a function ρ, via the relation $O = \rho(S)$. Thus, we can model a step of the transducer from state S' to S by the formula

$$\phi(S, S', I, O) \equiv \quad S = \delta(S', I) \wedge O = \rho(S).$$

Now suppose we wish to check that state or output variable v is a deterministic function of a set W of state or input variables. Let S_1, S_2 be two copies of the state variables, I_1, I_2 be two copies of I, and O_1, O_2 be two copies of O. Consider the formula

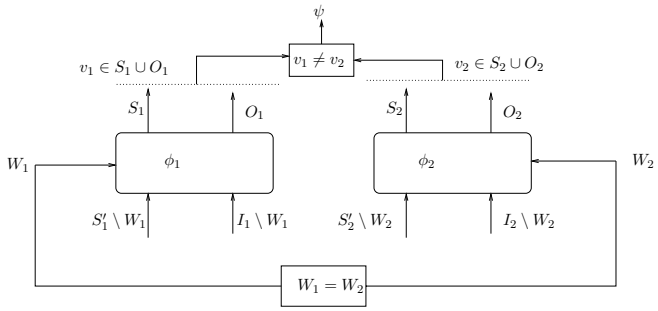


Fig. 1. Satisfiability problem for checking that v deterministically depends on W and nothing else

$$\psi(S_1, S_1', I_1, O_1, S_2, S_2', I_2, O_2) \equiv$$
$$\phi(S_1, S_1', I_1, O_1) \wedge \phi(S_2, S_2', I_2, O_2) \wedge$$
$$v_1 \neq v_2 \wedge \forall w \in W \,.\, w_1 = w_2.$$

Effectively, we make two copies of our model of the system. We then check whether it is possible for v to take on two different values in the two copies, while all variables in W take on the same value in both copies; the answer reveals whether v depends deterministically upon W. In particular, v can be expressed as a deterministic function of W ($v = f(W)$) if and only if ψ is unsatisfiable. Figure 2.2 illustrates this idea. This approach to checking dependence is similar to the technique of using self-composition for checking information flow [8]. The key idea is to formulate non-interference as a 2-safety property.

The property of injectivity can be easily formalized in logic. In general, the outputs O of the transducer is computed as a function $\rho(S)$, where, as above, S is the state of the system. However, for injectivity, we wish to show that ρ is a 1-1 function of a subset of "relevant state variables". We do this in two steps. First, for a candidate set of state variables W, we check (as shown above) that ρ is a function only of variables in W. Then to prove that ρ is an injective function, we additionally need to prove that the following formula is valid:

$$\rho(W_1) = \rho(W_2) \implies (W_1 = W_2)$$

In other words, if two output screens are identical, the relevant state of the system is the same in the two cases.

All of these checks have been performed for the voting machine we designed [7] using the Beaver SMT solver for finite-precision bit-vector arithmetic [4].

2.3 Systematic Human-Driven Testing

A key component of the approach is a systematic protocol for testing the interactive system by human "test users". The main steps are:

1. Prove determinism, independence, and unambiguity (injectivity) properties on the *implementation*, as described above;
2. Define coverage criteria that a test suite must satisfy, and
3. Prove that the two items above taken together ensure correctness of the interactive system.

We now briefly sketch the above approach using our voting machine case study.

A *test input* (or just *test*) is a sequence of button presses involving navigating between contests or selecting candidates that ends in the `cast` button being pressed. Let $\tau_{\mathcal{A}}$ denote the input-output trace exhibited by the implementation machine $\mathcal{A}$ on test input T, and let $\tau_{\mathcal{P}}$ be the trace exhibited by $\mathcal{P}$ on T. We ensure by design and formal verification that $\mathcal{A}$ and $\mathcal{P}$ are deterministic, meaning that for any T, there exists exactly one $\tau_{\mathcal{A}}$ and exactly one $\tau_{\mathcal{P}}$. Denote by I an *input/output interpretation function* that

formalizes (i) how a human voter might map regions on the screen to input buttons, and (ii) how the human voter might map the bitmap of an output screen to their perception of the relevant state of the machine (i.e., the current contest and selection state). If $I(\tau_{\mathcal{A}}) = \tau_{\mathcal{P}}$, we say that $\mathcal{A}$ is *correct on test* T or that test T *passes*.

Intuitively, at each step, the tester will check the output screen to make sure that the voting machine appears to have responded correctly, according to their expectations about correct behavior (e.g., after selecting a candidate, the candidate should be highlighted or otherwise appear to be selected). After casting their ballot, the tester will inspect the cast vote record produced by the voting machine (e.g., on a paper readout) and check that it appears to be correct (i.e., it is consistent with the selections the tester has made during this test, according to their interpretation of the test inputs). If any of these checks fail, the human tester will judge the machine $\mathcal{A}$ to have failed; otherwise, the human tester will pass the machine.

A *test suite* $\mathcal{T}$ is a set of complete tests. We say that $\mathcal{T}$ passes if every $T \in \mathcal{T}$ passes.

We assume that if any test fails, the voting system will not be used in an election. Therefore, we wish to identify a condition on $\mathcal{T}$ so that if every test in $\mathcal{T}$ passes, then we can be assured that $\mathcal{A}$ is trace-equivalent to $\mathcal{P}$ after application of the input-output interpretation function. We identify such a sufficient condition on $\mathcal{T}$ below. The condition relies upon the following formally verified properties:

P0: The output function of the voting machine is a injective function of the contest number and selection state of the current contest.
P1: The voting machine is a deterministic transducer.
P2: The state of a contest is updated independently of the state of other contests.
P3: If a navigation button is pressed, the selection state remains unchanged.
P4: If a selection button is pressed, the current contest number stays unchanged.

In addition, we require another property of $\mathcal{A}$ (to be formally verified on the implementation):

P5: The electronic cast vote record that is produced when we cast the ballot is an accurate copy of the selection state for each contest.

All of these properties have been formally verified on the implementation used in our paper [7].

Coverage Criteria. We say that a test suite $\mathcal{T}$ satisfies our coverage criteria if the resulting set of traces of $\mathcal{P}$ satisfies the following conditions:

C0: (*Initial State Coverage*) There is a test in which, from the initial output screen z_0, $\mathcal{P}$ receives the `cast` input.
C1: (*Transition Coverage*)
- **(a)** (*Selection transitions*) For every contest i, every selection state s_i within contest i, and every input button b corresponding to a selection, there is some trace where $\mathcal{P}$ receives b in a state (i, s) where the ith component of s is s_i.
- **(b)** (*Navigation transitions*) For every contest i, and every input button corresponding to navigation between contests, there is some trace where $\mathcal{P}$ receives b in a state of the form (i, s).

C2: (*Output Screen Coverage*) For every contest i and every selection state s_i of $\mathcal{P}$ within contest i, there is some trace of $\mathcal{P}$ where *the last transition* within contest i ended at s_i and then at some point thereafter $\mathcal{P}$ receives the `cast` input.

The main correctness theorem we obtain in our paper [7] for the voting machine described therein is that the tests pass iff the machine is trace-equivalent w.r.t. the mental model $\mathcal{P}$:

Theorem 1. *Consider a test suite $\mathcal{T}$ that satisfies coverage criteria C0–C2. Then, $\mathcal{T}$ passes if and only if $\mathcal{A}$ is correct (i.e., $\mathrm{Tr}(\mathcal{P}) = \{I(\tau) : \tau \in \mathrm{Tr}(\mathcal{A})\}$).*

2.4 Extensions

The basic approach outlined in the preceding sections is well-suited for finite-state interactive systems, such as the voting machine. However, even in the domain of electronic voting machines, there is more to be done by including advanced features of voting, such as a summary screen that lists selections made in multiple contests, straight-party voting, where one can cast a vote for all candidates of the same party, instant runoff voting, where one can rank candidates rather than select them, etc. For some of these, we have already developed some initial ideas that can be used to extend the basic approach.

3 Conclusions and Future Work

Even as computing systems are increasingly integrated into our everyday lives, human interaction and operation remains central to their working. In this paper, we describe how a combination of formal verification and systematic testing by humans can help in improving the assurance of these systems. As an illustrative example, we described our work on verification of an electronic voting machine for U.S. elections [7].

A particularly compelling next step is to consider interactive *cyber-physical systems* — systems that tightly integrate computational processes with the physical world, possibly involving networking — that also have humans playing central roles in their operation. Modern automotive, avionics and medical systems are good examples. The technical challenge in these systems vis-a-vis electronic voting arises from the combination and close interaction of continuous and discrete state and dynamics. While the essence of the approach described in this paper, including the properties of independence, determinism, and umabiguity, should remain relevant, extensions are required to deal with the complexity in the state space. Our ongoing work is developing new techniques for such systems.

Another direction for future work involves systems that have multiple humans involved — i.e., teams of human operators or even multiple competing human agents interacting with computing systems, possibly over a network. Altogether, many more advances are needed before we can achieve the goal of high-assurance distributed cyber-physical systems with multiple humans in the loop.

Acknowledgements. This paper describes work conducted jointly with Susmit Jha, Cynthia Sturton, and David Wagner. The author gratefully acknowledges the support of an Alfred P. Sloan Research Fellowship and the National Science Foundation through grants CNS-0644436 and CCF-1116993.

References

1. Alexander, K., Smith, P.: Verifying the vote in 2008 presidential election battleground states (November 2008), http://www.calvoter.org/issues/votingtech/pub/pres2008_ev.html
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, vol. 4, ch. 8. IOS Press (2009)
3. Federal Aviation Administration (FAA). The interfaces between flight crews and modern flight systems (1995), http://www.faa.gov/avr/afs/interfac.pdf
4. Jha, S., Limaye, R., Seshia, S.A.: Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 668–674. Springer, Heidelberg (2009)
5. Kohn, L.T., Corrigan, J.M., Donaldson, M.S. (eds.): To err is human: Building a safer health system. Technical report, A report of the Committee on Quality of Health Care in America, Institute of Medicine. National Academy Press, Washington, DC (2000)
6. Obradovich, J.H., Woods, D.D.: Users as designers: How people cope with poor HCI design in computer-based medical devices. Human Factors 38(4), 574–592 (1996)
7. Sturton, C., Jha, S., Seshia, S.A., Wagner, D.: On voting machine design for verification and testability. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (November 2009)
8. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. Technical Report UCB/CSD-05-1396, EECS Department, University of California, Berkeley (June 2005)
9. Verified Voting Foundation. America's voting systems in 2010, http://verifiedvoting.org

Fast Distributed PageRank Computation

Atish Das Sarma[1], Anisur Rahaman Molla[2],
Gopal Pandurangan[3], and Eli Upfal[4]

[1] eBay Research Labs, eBay Inc., CA, USA
atish.dassarma@gmail.com
[2] Division of Mathematical Sciences, Nanyang Technological University, Singapore 637371
anisurpm@gmail.com
[3] Division of Mathematical Sciences, Nanyang Technological University, Singapore 637371 and Department of Computer Science, Brown University, Providence, RI 02912, USA
gopalpandurangan@gmail.com
[4] Department of Computer Science, Brown University, Providence, RI 02912, USA
eli@cs.brown.edu

Abstract. Over the last decade, PageRank has gained importance in a wide range of applications and domains, ever since it first proved to be effective in determining node importance in large graphs (and was a pioneering idea behind Google's search engine). In distributed computing alone, PageRank vectors, or more generally random walk based quantities have been used for several different applications ranging from determining important nodes, load balancing, search, and identifying connectivity structures. Surprisingly, however, there has been little work towards designing provably efficient fully-distributed algorithms for computing PageRank. The difficulty is that traditional matrix-vector multiplication style iterative methods may not always adapt well to the distributed setting owing to communication bandwidth restrictions and convergence rates.

In this paper, we present fast random walk-based distributed algorithms for computing PageRank in general graphs and prove strong bounds on the round complexity. We first present an algorithm that takes $O(\log n/\epsilon)$ rounds with high probability on any graph (directed or undirected), where n is the network size and ϵ is the reset probability used in the PageRank computation (typically ϵ is a fixed constant). We then present a faster algorithm that takes $O(\sqrt{\log n}/\epsilon)$ rounds in undirected graphs. Both of the above algorithms are scalable, as each node processes and sends only small (polylogarithmic in n, the network size) number of bits per round and hence work in the CONGEST distributed computing model. For directed graphs, we present an algorithm that has a running time of $O(\sqrt{\log n/\epsilon})$, but it requires a polynomial number of bits to processed and sent per node in a round. To the best of our knowledge, these are the first fully distributed algorithms for computing PageRank vectors with provably efficient running time.

Keywords: PageRank, Distributed Algorithm, Random Walk, Monte Carlo Method.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 11–26, 2013.

1 Introduction

In the last decade, PageRank has emerged as a very powerful measure of relative importance of nodes in a network. The term PageRank was first introduced in [14,7] where it was used to rank the importance of webpages on the web. Since then, PageRank has found a wide range of applications in a variety of domains within computer science such as distributed networks, data mining, web algorithms, and distributed computing [8,5,6,12]. Since PageRank is essentially the steady state distribution (or the top eigenvector of the Laplacian) corresponding to a slightly modified random walk process, it is a easily defined quantity. However, the power and applicability of PageRank arises from its basic intuition of being a way to naturally identify "important" nodes, or in certain cases, similarity between nodes.

While there has been recent work on performing random walks efficiently in distributed networks [19,4], surprisingly, little theoretically provable results are known towards efficient distributed computation of PageRank vectors. This is perhaps because the traditional method of computing PageRank vectors is to apply iterative methods (i.e., do matrix-vector multiplications) till (near)-convergence. While such techniques may not adapt well in certain settings, when dealing with a global network with only local views (as is common in distributed networks such as Peer-to-Peer (P2P) networks), and particularly, very large networks, it becomes crucial to design far more efficient techniques. Therefore, PageRank computation using Monte Carlo methods is more appropriate in a distributed model where only limited sized messages are allowed through each edge in each round.

To elaborate, a naive way to compute PageRank of nodes in a distributed network is simply scaling iterative PageRank algorithms to distributed environment. But this is firstly not trivial, and secondly expensive even if doable. As each iteration step needs computation results of previous steps, there needs to be continuous synchronization and several messages may need to be exchanged. Further, the convergence time may also be slow. It is important to design efficient and localized distributed algorithms as communication overhead is more important than CPU and memory usage in distributed page ranking. We take all these concerns into consideration and design highly efficient fully decentralized algorithms for efficiently computing PageRank vectors in distributed networks.

Our Contributions. In this paper, to the best of our knowledge, we present the first provably efficient fully decentralized algorithms for estimating PageRank vectors under a variety of settings. Our algorithms are scalable since, each node processes and sends only polylogarithmic in n (the network size) number of bits per round. Thus our algorithms work in the well-studied CONGEST distributed computing model [16], where there is a restriction on the number of bits (typically, polylogarithmic in n) that can be sent per edge per round. Specifically, our contributions are as follows:

- We present an algorithm, Simple-PageRank-Algorithm (cf. Algorithm 1), that computes the PageRank accurately in $O(\frac{\log n}{\epsilon})$ rounds with high

probability[1], where n is the number of nodes in the network and ϵ is the random reset probability in the PageRank random walk [2,4,19]. Our algorithms work for any arbitrary network (directed as well as undirected).

- We present an improved algorithm, Improved-PageRank-Algorithm (cf. Algorithm 2), that computes the PageRank accurately in *undirected graphs* and terminates with high probability in $O(\frac{\sqrt{\log n}}{\epsilon})$ rounds. We note that though PageRank is usually applied for directed graphs (e.g., for the World Wide Web), however, it is sometimes also applied in connection with undirected graphs as well ([10,1,17,11,21]) and is non-trivial to compute (cf. Section 2.2). In particular, it can be applied for distributed networks when modeled as undirected graphs (as is typically the case, e.g., in P2P network models). We note that the Improved-PageRank-Algorithm as well as the Simple-PageRank-Algorithm require only polylogarithmic in n number of bits to be processed and sent per round and works in the CONGEST model.
- We present an improved algorithm for directed graphs (which is a modified version of the Improved-PageRank-Algorithm) that computes PageRank accurately and terminates with high probability in $O(\sqrt{\frac{\log n}{\epsilon}})$ rounds, but it requires a polynomial number of bits to be processed and sent per node in a round. Assuming ϵ is a constant (which is typically the case), this algorithm as well as the Improved-PageRank-Algorithm yields a sub-logarithmic (in n) running time. Thus, in many networks, this running time can be substantially smaller than even the network diameter (e.g., in constant-degree networks, the diameter is $\Omega(\log n)$).

2 Background and Related Work

2.1 Distributed Computing Model

We model the communication network as an unweighted, connected n-node graph $G = (V, E)$. Each node has limited initial knowledge. Specifically, we assume that each node is associated with a distinct identity number (e.g., its IP address). At the beginning of the computation, each node v accepts as input its own identity number (which is of length $O(\log n)$ bits) and the identity numbers of its neighbors in G. The node may also accept some additional inputs as specified by the problem at hand (e.g., the number of nodes in the network). A node v can communicate with any node u if v knows the id of u.[2] (Initially, each node knows only the ids of its neighbors in G.) We assume that the communication occurs in synchronous *rounds*, i.e., nodes run at the same processing speed and

[1] Throughout, "with high probability (whp)" means with probability at least $1 - 1/n^{\Omega(1)}$, where n is the number of nodes in the networks.

[2] This is a typical assumption in the context of P2P and overlay networks, where a node can establish communication with another node if it knows the other node's IP address. We sometimes call this *direct* communication, especially when the two nodes are not neighbors in G. Note that our algorithm of Section 3 uses no direct communication between non-neighbors in G.

any message that is sent by some node v to its neighbors in some round r will be received by the end of r. To ensure scalability, we restrict the number of bits that are processed and sent per round by each node to be polylogarithmic in n, the network size. In particular, in each round, each node is allowed to send a message of size B bits (where B is polylogarithmic in n) through each communication link. This is a widely used standard model (called the CONGEST(B) model) to study distributed algorithms (e.g., see [16,15]) and captures the bandwidth constraints inherent in real-world computer networks. We assume B to be polylogarithmic in n. We relax this restriction in Section 5, where we allow polynomial (in n) number of bits to be sent across a link per round; thus our algorithm presented in this Section works in the LOCAL model [16], which is another standard model where there is no restriction on the amount of communication per link per round.

There are several measures of efficiency of distributed algorithms; here we will focus on the running time, i.e. the number of rounds of distributed communication. (Note that the computation that is performed by the nodes locally is free, i.e., it does not affect the number of rounds.)

2.2 PageRank

We formally define the PageRank of a graph $G = (V, E)$. Let ϵ be a small constant which is fixed (ϵ is called the *reset* probability, i.e., with probability ϵ, it starts from a node chosen uniformly at random among all nodes in the network). The PageRank of a graph (e.g., see [2,4,19,5]) is the *stationary distribution* vector π of the following special type of random walk: at each step of the walk, with probability ϵ it starts from a randomly chosen node and with remaining probability $1-\epsilon$, it follows a randomly chosen outgoing (neighbor) edge from the current node and moves to that neighbor.[3] Therefore the PageRank transition matrix on the state space (or vertex set) V can be written as

$$P = (\frac{\epsilon}{n})J + (1-\epsilon)Q \tag{1}$$

where J is the matrix with all entries 1 and Q is the transition matrix of a simple random walk on G defined as $Q_{ij} = 1/k$, if j is one of the $k > 0$ outgoing links of i, otherwise 0. Computing PageRank and its variants efficiently in various computation models has been of tremendous research interest in both academia and industry. For a detailed survey of PageRank see e.g., [5,12]. We note that PageRank is well-defined in both directed and undirected graphs. Note that it is difficult to compute analytically (and no such analytical formulas are known for general graphs) the PageRank distribution and hence various computational methods have been used to estimate the PageRank distribution. In fact, this is true for general *undirected* graphs as well [10].

There are mainly two broad approaches to computing PageRank (e.g., see [3]). One is to using linear algebraic techniques, (e.g., the Power Iteration [14]) and the

[3] We sometime use the terminology "PageRank random walk" for this special type of random walk process.

other approach is Monte Carlo [2]. In the Monte Carlo method, the basic idea is to approximate PageRank by directly simulating the corresponding random walk and then estimating the stationary distribution with the performed walk's distribution. In [2] Avrachenkov et al., proposed the following Monte Carlo method for PageRank approximation: Perform K random walks (according to the PageRank transition probability) starting from each node v of the graph G. For each walk, terminate the walk with its first reset instead of moving to a random node. Then, the frequencies of visits of all these random walks to different nodes will approximate the PageRank. Our distributed algorithms are based on the above method.

Monte Carlo methods are efficient, light weight and highly scalable [2]. It has proved to be a useful technique in designing algorithms for PageRank and its variants in important computational models like data streaming [19] and MapReduce [3]. The works in [20,18] study distributed implementation of PageRank in peer-to-peer networks but uses iteration methods.

3 A Distributed Algorithm for PageRank

We present a Monte Carlo based distributed algorithm for computing PageRank distribution of a network [2]. The main idea of our algorithm (formal pseudocode is given in Algorithm 1) is as follows. Perform K (K will be fixed appropriately later) random walks starting from each node of the network in parallel. In each round, each random walk independently goes to a random (outgoing) neighbor with probability $1-\epsilon$ and with the remaining probability (i.e., ϵ) terminates in the current node. (Henceforth, we call this random walk as *'PageRank random walk'*. This random walk can be shown to be equivalent to one based on the PageRank transition matrix P (defined in Section 2.2) [2].) Since, ϵ is the probability of termination of a walk in each round, the expected length of every walk is $1/\epsilon$ and the length will be at most $O(\log n/\epsilon)$ with high probability. Let every node v count the number of visits (say, ζ_v) of all the walks that go through it. Then, after termination of all walks in the network, each node v computes (estimates) its PageRank π_v as $\tilde{\pi}_v = \frac{\zeta_v \epsilon}{nK}$. Notice that $\frac{nK}{\epsilon}$ is the (expected) total number of visits over all nodes of all the nK walks. The above idea of counting the number of visits is a standard technique to approximate PageRank (see e.g., [2,4]).

We show in the next section that the above algorithm approximates PageRank vector π accurately (with high probability) for an appropriate value of K. The main technical challenge in implementing the above method is that performing many walks from each node in parallel can create a lot of congestion. Our algorithm uses a crucial idea to overcome the congestion. We show that (cf. Lemma 1) that there will be no congestion in the network even if we start a polynomial number of random walks from every node in parallel. The main idea is based on the Markovian (memoryless) properties of the random walks and the process that terminates the random walks. To calculate how many walks move from node i to node j, node i only needs to know the number of walks that reached it. It does not need to know the sources of these walks or the transitions that they took before reaching node i. Thus it is enough to send the *count* of

the number of walks that pass through a node. The algorithm runs till all the walks are terminated. It is easy to see that it finishes in $O(\log n/\epsilon)$ rounds with high probability (this is because the maximum length of any walk is $O(\log n/\epsilon)$ whp). Then every node v outputs its PageRank as the ratio between the number of visits (denoted by ζ_v) to it and the total number of visits ($\frac{nK}{\epsilon}$) over all nodes of all the walks. We show that our algorithm computes approximate PageRank accurately in $O(\log n/\epsilon)$ rounds with high probability (cf. Theorem 1).

Algorithm 1. SIMPLE-PAGERANK-ALGORITHM

Input (for every node): Number of walks $K = c \log n$ from each node (where $c = \frac{2}{\delta' \epsilon}$ and δ' is defined in Section 3.2), reset probability ϵ.
Output: PageRank of each node.

[Each node v starts $c \log n$ walks. All walks keep moving in parallel until they terminate. The termination probability of each walk is ϵ, so the expected length of each walk is $1/\epsilon$.]

1: Initially, each node v in G creates $c \log n$ messages (called coupons) $C_1, C_2, \ldots, C_{c \log n}$. Each node also maintains a counter ζ_v (for counting visits of random walks to it).
2: **while** there is at least one (alive) coupon **do**
3: This is i-th round. Each node v holding at least one coupon does the following: Consider each coupon C held by v which is received in the $(i-1)$-th round. Generate a random number $r \in [0, 1]$.
4: **if** $r < \epsilon$ **then**
5: Terminate the coupon C.
6: **else**
7: Select an outgoing neighbor uniformly at random, say u. Add one coupon counter number to T_u^v where the variable T_u^v indicates the number of coupons (or random walks) chosen to move to the neighbor u from v in the i-th round.
8: **end if**
9: Send the coupon's counter number T_u^v to the respective outgoing neighbors u.
10: Every node u adds the total counter number ($\sum_{v \in N(u)} T_u^v$—which is the total number of visits of random walks to u in i-th round) to ζ_u.
11: **end while**
12: Each node outputs its PageRank as $\frac{\zeta_v \epsilon}{cn \log n}$.

3.1 Analysis

Our algorithm computes the PageRank of each node v as $\tilde{\pi}_v = \frac{\zeta_v \epsilon}{nK}$ and we say that $\tilde{\pi}_v$ approximates original PageRank π_v. We first focus on the correctness of our approach and then analyze the running time.

3.2 Correctness of PageRank Approximation

The correctness of the above approximation follows directly from the main result of [2] (see Algorithm 4 and Theorem 1) and also from [4] (Theorem 1). In

particular, it is mentioned in [2,4] that the approximate PageRank value is quite good even for $K = 1$. It is easy to see that the expected value of $\tilde{\pi}_v$ is π_v (e.g., [2]). In [4] (Theorem 1), it shows that $\tilde{\pi}_v$ is sharply concentrated around π using a Chernoff bound technique ([13]). They show,

$$\Pr[|\ \tilde{\pi}_v - \pi_v\ | \leq \delta \pi_v] \leq e^{-nK\pi_v\delta'} \tag{2}$$

where δ' is a constant depending on ϵ and δ. From the above bound (cf. Equation 2), we see that for $K = \frac{2\log n}{\delta' n \pi_{min}}$, we get a sharp approximation of PageRank vector with high probability. Since the PageRank of any node is at least ϵ/n (i.e. the minimum PageRank value, $\pi_{min} \geq \epsilon/n$), so it gives $K = \frac{2\log n}{\delta'\epsilon}$. For simplicity we assume the constant $c = \frac{2}{\delta'\epsilon}$. Therefore, it is enough if we perform $c \log n$ PageRank random walks from each node. Now we focus on the running time of our algorithm.

3.3 Time Complexity

From the above section we see that our algorithm is able to compute the PageRank vector π in $O(\log n/\epsilon)$ rounds with high probability if we perform $c \log n$ walks from each node in parallel without any congestion. The lemma below guarantees that there will be no congestion even if we do a polynomial number of walks in parallel.

Lemma 1. *There is no congestion in the network if every node starts at most a polynomial number of random walks in parallel.*

Proof. It follows from our algorithm that each node only needs to count the number of visits of random walks to itself. Therefore nodes do not require to know from which source node or rather from where it receives the random walk coupons. Hence it is not needed to send the ID of the source node with the coupon. Recall that in our algorithm, in each round, every node currently holding at least one random walk coupon (could be many) does the following. For each coupon, either the walk is terminated with probability ϵ or with remaining probability $1 - \epsilon$, any outgoing edge is chosen uniformly at random to send the coupon. Any particular outgoing edge may be chosen for more than one coupon. Instead of sending each coupon separately through that edge, the algorithm simply sends the count, i.e., number of coupons, to the chosen outgoing neighbor. Since we consider *CONGEST* model, a polynomial in n number of coupon's count (i.e., we can send count of up to a polynomial number) can be sent in one message through each edge without any congestion. □

Theorem 1. *The algorithm* Simple-PageRank-Algorithm *(cf. Algorithm 1) computes PageRank in* $O(\frac{\log n}{\epsilon})$ *rounds with high probability.*

Proof. The algorithm stops when all the walks terminate. Since the termination probability is ϵ, so in expectation after $1/\epsilon$ steps, a walk terminates and with high probability (via the Chernoff bound) the walk terminates in $O(\log n/\epsilon)$ rounds and by union bound [13], all walks (they are only polynomially many)

terminate in $O(\log n/\epsilon)$ rounds whp. Since all the walks are moving in parallel and there is no congestion (cf. Lemma 1), all the walks in the network terminate in $O(\log n/\epsilon)$ rounds whp. Hence the algorithm stops in $O(\log n/\epsilon)$ rounds whp. The correctness of the PageRank approximation follows from [2,4] as discussed earlier in Section 3.2. □

4 A Faster Distributed PageRank Algorithm (for Undirected Graphs)

We present a faster algorithm for PageRank computation. First we present an algorithm for *undirected* graphs and in Section 5 we modify it slightly to work for directed graphs. Our algorithm's time complexity for the undirected graphs holds in the CONGEST model, whereas for directed graphs a slightly better time complexity applies only in the LOCAL model.

We use a similar Monte Carlo method as described in Section 3 to estimate PageRank. This says that the PageRank of a node v is the ratio between the number of visits of PageRank random walks to v itself and the sum of all the visits over all nodes in the network. In the previous section (cf. Section 3) we show that in $O(\log n/\epsilon)$ rounds, one can approximate RageRank accurately by walking in a naive way on general graph. We now outline how to speed up our previous algorithm (cf. Algorithm 1) using an idea similar to the one used in [9]. In [9], it is shown how one can perform *a* standard (simple) random walk in an undirected graph[4] of length L in $\tilde{O}(\sqrt{LD})$ rounds whp (D is the diameter of the network). The high level idea of their algorithm is to perform 'many' short walks in parallel and later 'stitch' them to get the desired longer length walk. To apply this idea in our case, we modify our approach accordingly as speeding up (*many*) PageRank random walks is different from speeding up *one* (standard) random walk. We show that our improved algorithm (cf. Algorithm 2) approximates PageRank in $O(\frac{\sqrt{\log n}}{\epsilon})$ rounds whp.

4.1 Description of Our Algorithm

In Section 3, we showed that by performing $\Theta(\log n)$ walks (in particular we are performing $c \log n$ walks, where $c = \frac{2}{\delta' \epsilon}$, δ' is defined in Section 3.2) of length $\log n/\epsilon$ from each node, one can approximate the PageRank vector π accurately (with high probability). In this section we focus on the problem of how efficiently one can perform $\Theta(n \log n)$ walks ($\Theta(\log n)$ from each node) each of length $\log n/\epsilon$ and count the number of visits of these walks to different nodes. Throughout, by "random walk" we mean the "PageRank random walk" (cf. Section 3).

The main idea of our algorithm is to first perform 'many' short random walks in parallel and then 'stitch' those short walks to get the longer walk of length $\log n/\epsilon$ and subsequently 'count' the number of visits of these random walks to different nodes. In particular, our algorithm runs in three phases. In the first

[4] In each step, an edge is taken from the current node x with probability proportional to $1/d(x)$ where $d(x)$ is the degree of x.

phase, each node v performs $d(v)\eta$ ($d(v)$ is degree of v) independent 'short' random walks of length λ in parallel. (The value of the parameters η and λ will be fixed later in the analysis.) This is done naively by forwarding $d(v)\eta$ 'coupons' having the ID of v from v (for each node v) for λ steps via random walks. The intuition behind performing $d(v)\eta$ short walks is that the PageRank of an undirected graph is proportional to the degree distribution [10]. Therefore we can easily bound the number of visits of random walks to any node v (cf. Lemma 2). At the end of this phase, if node u has k coupons with the ID of a node v, then u is a destination of k walks starting at v. Note that just after this phase, v has no knowledge of the destinations of its own walks, but it can be known by direct communication from the destination nodes. The destination nodes (at most $d(v)\eta$) have the ID of the source node v. So they can contact the source node via *direct* communication. We show that this takes at most constant number of rounds as only polylogarithmic number of bits are sent (since η will be at most $O(\log^3 n/\epsilon)$, shown later). It is shown that the first phase takes $O(\frac{\lambda}{\epsilon})$ rounds with high probability (cf. Lemma 3).

In the second phase, starting at source node s, we 'stitch' some of the λ-length walks prepared in first phase (note that we do this for every node v in parallel as we want to perform $\Theta(\log n)$ walks from each node). The algorithm starts from s and randomly picks one coupon distributed from s in Phase 1. We now discuss how to sample one such coupon randomly and go to the destination vertex of that coupon. One simple way to do this is as follows: In the end of Phase 1, each node v knows the destination node's ID of its $d(v)\eta$ short walks (or coupons). When a coupon needs to be sampled, node s chooses a random coupon number (from the unused set of coupons) and informs the destination node (which will be the next stitching point) holding the coupon C (by direct communication, since s knows the ID of the destination node at the end of the first phase). Let C be the sampled coupon and v be the destination node of C. The source s then sends a 'token' to v and s deletes the coupon C (so that C will not be sampled again next time at s, otherwise, randomness will be destroyed). The process then repeats. That is, the node v currently holding the token samples one of the coupons it distributed in Phase 1 and forwards the token to the destination of the sampled coupon, say v'. Nodes v, v' are called 'connectors' — they are the endpoints of the short walks that are stitched. A crucial observation is that the walk of length λ used to distribute the corresponding coupons from s to v and from v to v' are independent random walks. Therefore, we can stitch them to get a random walk of length 2λ. We therefore can generate a random walk of length $3\lambda, 4\lambda, \ldots$ by repeating this process. We do this until we have completed a length of at least $(\log n/\epsilon - \lambda)$. Then, we complete the rest of the walk by doing the naive random walk algorithm. We show that Phase 2 finishes in $O(\frac{\log n}{\lambda\epsilon})$ rounds with high probability (cf. Lemma 5).

In the third phase we count the number of visits of all the random walks to a node. As we have discussed, we have to create many short walks of length λ from each node. All short walks may not be used to make the long walk of length $\log n/\epsilon$. We show a technique to count all the used short walks' visits to different nodes. Remember that after completion of Phase 2, all the $\Theta(n \log n)$

Algorithm 2. IMPROVED-PAGERANK-ALGORITHM

Input (for every node): Length $\ell = \frac{\log n}{\epsilon}$ of each walk, reset probability ϵ, short walk length $\lambda = \sqrt{\log n}$ and number of walks $K = c \log n$ (where $c = \frac{2}{\delta' \epsilon}$ and δ' is defined in Section 3.2).
Output: PageRank of each node.

Phase 1: (Each node v performs $d(v)\eta = d(v)\log^3 n/\epsilon$ random walks of length $\lambda = \sqrt{\log n}$. At the end of this phase, there are $d(v)\log^3 n/\epsilon$ (not necessarily distinct) nodes holding a 'coupon' containing the ID of v.)

1: **for** each node v **do**
2: Construct $d(v)\eta = d(v)\log^3 n/\epsilon$ messages containing its ID and also the desired walk length of $\lambda = \sqrt{\log n}$. We will refer to these messages created by node v as 'coupons created by v'.
3: **end for**
4: **for** $i = 1$ to λ **do**
5: This is the i-th round. Each node v does the following: Consider each coupon C held by v which is received in the $(i-1)$-th round. If the coupon C's desired walk length is at most i, then v keeps this coupon (v is the desired destination). Else, {v generates a random number $r \in [0, 1]$. If $r < \epsilon$, terminate the coupon C and keep the coupon as then v itself is the destination. Else, pick a neighbor u uniformly at random for the coupon C and forward C to u after incrementing counter}. Note that v does this for every coupon simultaneously in the i-th round.
6: **end for**
7: Each destination node sends its ID to the source node, as it has the source node's ID now.

Phase 2: (Stitch short walks by token forwarding. Stitch $\Theta(\ell/\lambda)$ walks, each of length λ)

1: The source node s creates a message called "token" which contains the ID of s. (Note that for simplicity we are showing the stitching from one source node but this has to be done for each node in the network in parallel.)
2: The algorithm will forward the token around and keep track of a set of connectors, denoted by CON. Initially, $CON = \{s\}$.
3: **while** Length of walk completed is at most $\ell - \lambda$ **do**
4: Let v be the node that is currently holding the token.
5: v samples one of the coupons distributed by v uniformly at random from the unused set of coupons. Let v' be the destination node of the sampled coupon, say C.
6: v sends the token to v' and deletes the coupon C.
7: $CON = CON \cup \{v'\}$
8: **end while**
9: Walk naively until ℓ steps are completed (this is at most another λ steps).
10: A node say w, holding the token having the ID of s is final destination of $\ell = \log n/\epsilon$ length PageRank random walk. $CON = CON \cup \{w\}$

Phase 3: (Counting the number of visits of short walks to a node)

1: Each node v maintains a counter ζ_v to keep track of the number of visits of walks.
2: **for** each walk completed in Phase 2 **do**
3: Start from each connector node in CON except the source node s.
4: Trace the random walk in reverse (in parallel) up to the source node of the corresponding short walk. (Recall that each connector node is the destination of some short walk).
5: Count the number of visits during this reverse tracing and add to ζ_v.
6: **end for**
7: Each node v outputs its PageRank π_v as $\frac{\zeta_v \epsilon}{cn \log n}$.

long walks ($\Theta(\log n)$ from each node) have been stitched. During stitching (i.e., in Phase 2), each connector node (which is also end point of the short walk) should remember the source node of the short walk. Now start from the each connector node and do a walk in reverse direction (i.e., retrace the short walk backwards) to the source node in parallel. During the reverse walk, simply count the visit to nodes. It is easy to see that this will take at most $O(\lambda)$ rounds with high probability (cf. Lemma 6). Now we analyze the running time of our algorithm Improved-PageRank-Algorithm. The compact pseudo code is given in Algorithm 2.

4.2 Analysis

First we are interested in the value of η i.e., how many coupons (short walks) do we need from each node to successfully answer all the stitching requests. Notice that it is possible that $d(v)\eta$ coupons are not enough (if η is not chosen suitably large): We might forward the token to some node v many times in Phase 2 and all coupons distributed by v in the first phase may be deleted. (In other words, v is chosen as a connector node many times, and all its coupons have been exhausted.) If this happens then the stitching process cannot progress. To fix this problem, we use an easy upper bound of the number of visits to any node v of a random walk of length ℓ in an undirected graph: $d(v)\ell$ times. Therefore each node v will be visited as a connector node at most $O(d(v)\ell)$ times with high probability. This implies that each node does not have to prepare too many short walks.

The following lemma bounds the number of visits to every node when we do $\Theta(\log n)$ walks from each node, each of length $\log n/\epsilon$ (note that this is the maximum length of a long walk, whp).

Lemma 2. *If we perform $\Theta(\log n)$ random walks of length $\log n/\epsilon$ from each node, then no node v is visited more than $O(\frac{d(v)\log^3 n}{\epsilon})$ times with high probability.*

Proof. Suppose we perform so many long walks in parallel. In other words, we can say that each node performing one walk of length $\Theta(\log^2 n/\epsilon)$. The bound on the number of visits to each node follows because in each round a node v can get only at most $d(v)$ walks in expectation (since we have an undirected graph) and hence $O(d(v)\log n)$ whp (via Chernoff bound). Since long walk length is $\Theta(\log^2 n/\epsilon)$, so total number of visits is $O(d(v)\log^3 n/\epsilon)$ whp. □

It is now clear from the above lemma (cf. Lemma 2) that $\eta = O(\log^3 n/\epsilon)$ i.e., each node v has to prepare $O(d(v)\log^3 n/\epsilon)$ short walks of length λ in Phase 1. Now we show the running time of algorithm (cf. Algorithm 2) using the following lemmas.

Lemma 3. *Phase 1 finishes in $O(\frac{\lambda}{\epsilon})$ rounds with high probability.*

Proof. It is known from the Lemma 2 that in Phase 1, each node v performs $O(d(v)\log^3 n/\epsilon)$ walks of length λ. Initially each node v starts with

$O(d(v)\log^3 n/\epsilon)$ coupons (or messages) and each coupon takes a random walk according to the PageRank transition probability. Let $\eta = O(\log^3 n/\epsilon)$. We now prove that after any given number of steps j ($j \leq \lambda$), the expected number of coupons at node any v is still $d(v)\eta$. This is because at each step any node v can send (as well as receive) $d(v)$ messages in expectation. The number of messages we started at any node v is proportional to its degree $d(v)$. Therefore, in expectation the number of messages at any node remains same. Thus in expectation the number of messages, say X that want to go through an edge in any round is at most 2η (from both end points). Using Chernoff bound we get, $\Pr[X \geq 4\eta \log n] \leq 2^{-4\log n} = n^{-4}$. It follows that the number of messages that want to go through any edge in any round is at most $4\eta \log n = O(\log^4 n/\epsilon)$ with high probability. Hence there will be at most $O(\log^5 n/\epsilon)$ bits whp at any edge per round (as one message is $\log n$ bits). Since we consider CONGEST(polylog n) model, we can extend all walk's length from i to length $i+1$ in $O(1/\epsilon)$ rounds whp. Therefore, for walks of length λ it takes $O(\lambda/\epsilon)$ rounds whp as claimed. □

Lemma 4. *One time stitching in parallel from each node always finishes within* $O(1)$ *rounds.*

Proof. Each node knows all of its short walk's (or coupon's) destination address. Each time when a (source or connector) node wants to stitch, it randomly chooses one of its unused coupons (created in Phase 1). Then it contacts the destination node (holding the coupon) through *direct* communication and informs it as the next connector node (or stitching point). Since the network allows polylog n congestion, this will finish in constant rounds. □

Lemma 5. *Phase 2 finishes in* $O(\frac{\log n}{\lambda\epsilon})$ *rounds.*

Proof. Phase 2 is for stitching short walks of length λ to get the long walk of length $O(\log n/\epsilon)$. Therefore it needs to stitch approximately $O(\log n/\lambda\epsilon)$ times. Since each time stitching can be done in constant rounds (cf. Lemma 4), Phase 2 finishes in $O(\frac{\log n}{\lambda\epsilon})$ rounds. □

Lemma 6. *Phase 3 finishes in* $O(\lambda)$ *rounds with high probability.*

Proof. Each short walk is of length λ. Phase 3 is simply tracing back the short walks. So it is easy to see we can perform all the reverse walks in parallel in $O(\lambda)$ rounds (same as the time to do all the short walks in parallel in Phase 1). Due to Lemma 3 and the fact that each node can communicate a polylog n number of bits in every round, we can say that Phase 3 finishes in $O(\lambda)$ rounds with high probability. □

Now we are ready to show the main result of this section.

Theorem 2. *The* Improved-PageRank-Algorithm *(cf. Algorithm 2) computes the PageRank accurately and with high probability finishes in* $O(\frac{\sqrt{\log n}}{\epsilon})$ *rounds.*

Proof. The algorithm Improved-PageRank-Algorithm consists of three phases. We have calculated above the running time of each phase separately. Now we want to compute the overall running time of the algorithm by combining these three phases and by putting appropriate value of parameters. By summing up the running time of all three phases, we get from Lemmas 3, 5 and 6 that the total time taken to finish the Improved-PageRank-Algorithm is $O(\frac{\lambda}{\epsilon} + \frac{\log n}{\lambda\epsilon} + \lambda)$ rounds with high probability. Choosing $\lambda = \sqrt{\log n}$, gives the required bound as $O(\frac{\sqrt{\log n}}{\epsilon})$ whp. □

5 A Faster Algorithm for Directed Graphs

We extend the Improved-PageRank-Algorithm of Section 4 to directed graphs. Recall that it follows from Section 3 that it is enough to approximate PageRank vector if each node performs $c \log n$ PageRank random walk of length $\log n/\epsilon$, where $c = 2/\delta'\epsilon$ is a constant. The basic idea of the algorithm is similar as above i.e., create some short walks from each node in parallel and later stitch them to get long walks and then count the number of visits of all the long walks to different nodes. However, the main difficulty in an directed graph is to bound the number of visits of random walks to any node. This is because, in a directed graph we do not have a suitable upper bound on PageRank (unlike the case of an undirected graph). There could be large discrepancy between *indegree* and *outdegree* of a node on a directed graph (in shorthand we use *indeg* and *outdeg* respectively). Therefore, for any node whose indeg and outdeg ratio is large enough, it is very likely that many random walk coupons will pass over those nodes in every round. In the similar way, there can be a large congestion on those nodes if we want to perform a large number of short walks from each node. Hence it is difficult to derive a similar faster algorithm as Algorithm 2 in the CONGEST model for directed graphs. Hence, in this section, we adopt the LOCAL distributed computing model [16] where message size restriction is removed, i.e., nodes can communicate any number of bits in each round. (Our algorithm will need a polynomial number of bits to be processed and sent by a node in each round.) Even in the LOCAL model, it is not obvious how to perform a ℓ length random walk in less than ℓ rounds when $\ell < D$, the diameter of the network. Because in LOCAL model, a trivial solution of any distributed computation problem is to collect all the information of the network to a single node and compute the solution locally. Clearly this will take diameter (D) time. Since we are interested to performing random walks of length $\log n/\epsilon$ which can be much less than the diameter (in general), our algorithm gives a non-trivial result in LOCAL model also. We discuss below our algorithm for directed graphs using the same approach as in Section 4.

5.1 Description of Our Algorithm

It is now clear that only Phase 1 of Algorithm 2 is problematic. We want to modify the Phase 1 of the previous algorithm. First we consider an upper bound

on the number of times any node is visited if we perform $c \log n$ random walks of length $\log n/\epsilon$ from each node. We assume the trivial upper bound that any node v will be visited at most $\frac{cn \log^2 n}{\epsilon}$ times with high probability (since total $cn \log n$ walks of length $\log n/\epsilon$). This bound also trivially holds for number of visits as a connector node. This implies that we have to create $\frac{cn \log^2 n}{\epsilon}$ short walks of length λ from each node in Phase 1. It is easy to see that this can be done in $O(\lambda)$ rounds in the LOCAL model (cf. Lemma 7). The other two phases of the algorithm namely, Phase 2 (stitching short walks) and Phase 3 (counting number of visits) can be done by the same approach as in Section 4. We note that Phase 2 and Phase 3 can be done in almost the same running time without considering *direct* communication in LOCAL model.

5.2 Analysis

Lemma 7. *Phase 1 takes $O(\lambda)$ rounds for performing $\frac{cn \log^2 n}{\epsilon}$ walks of length λ from each node v.*

Proof. We are interested in performing $\frac{cn \log^2 n}{\epsilon}$ random walks of length λ from each node. In the LOCAL model, every node can send or receive any number of messages through an edge in each round. Congestion is not an issue here. Therefore at any round i, each node holding any number of coupons can forward them to randomly chosen outgoing neighbors (in parallel). This will take one round only. Thus by walking in naive way for λ rounds in parallel, all short walks can extend their length to λ, i.e. every coupon will reach to the destination node after λ rounds. So it will finish in $O(\lambda)$ rounds. □

Lemma 8. *Phase 2 finishes in $O(\frac{\log n}{\lambda \epsilon})$.*

Proof. Since Phase 2 is the same as in Algorithm 2, the proof follows from the Lemma 5 above. □

Lemma 9. *Phase 3 finishes in $O(\lambda)$ rounds with high probability.*

Proof. The Phase 3 is also same as in Algorithm 2. The proof follows from the Lemma 6 above. □

Theorem 3. *The algorithm computes the PageRank accurately on directed graph and with high probability finishes in $O(\sqrt{\frac{\log n}{\epsilon}})$ rounds in LOCAL model.*

Proof. The algorithm for computing PageRank on directed graph also comprises of three phases. Combining the running time of these three phases from above we get the total time taken to finish the algorithm: $O(\lambda + \frac{\log n}{\epsilon\lambda} + \lambda)$ rounds with high probability. Choosing $\lambda = \sqrt{\frac{\log n}{\epsilon}}$, gives the required bound as $O(\sqrt{\frac{\log n}{\epsilon}})$. □

6 Conclusion

We presented fast distributed algorithms for computing PageRank, a measure of fundamental interest in networks. Our algorithms are Monte-Carlo and based on the idea of speeding up random walks in a distributed network. Our faster algorithms take time only sub-logarithmic in n which can be useful in large-scale, resource-constrained, distributed networks, where running time is especially crucial. Since they are based on random walks, which are lightweight, robust, and local, they can be amenable to self-organizing and dynamic networks.

References

1. Andersen, R., Chung, F., Lang, K.: Local graph partitioning using pagerank vectors. In: FOCS, pp. 475–486 (2006)
2. Avrachenkov, K., Litvak, N., Nemirovsky, D., Osipova, N.: Monte carlo methods in pagerank computation: When one iteration is sufficient. SIAM J. Number. Anal. 45(2), 890–904 (2007)
3. Bahmani, B., Chakrabarti, K., Xin, D.: Fast personalized pagerank on mapreduce. In: SIGMOD Conference, pp. 973–984 (2011)
4. Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized pagerank. PVLDB 4, 173–184 (2010)
5. Berkhin, P.: A survey on pagerank computing. Internet Mathematics 2(1), 73–120 (2005)
6. Bianchini, M., Gori, M., Scarselli, F.: Inside pagerank. ACM Trans. Internet Technol. 5(1), 92–128 (2005)
7. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Seventh International World-Wide Web Conference (WWW 1998), pp. 107–117 (1998)
8. Cook, M.: Calculation of pagerank over a peer-to-peer network (2004)
9. Das Sarma, A., Nanongkai, D., Pandurangan, G., Tetali, P.: Efficient distributed random walks with applications. In: PODC, pp. 201–210 (2010)
10. Grolmusz, V.: A note on the pagerank of undirected graphs. CoRR, abs/1205.1960 (2012)
11. Iván, G., Grolmusz, V.: When the web meets the cell: using personalized pagerank for analyzing protein interaction networks. Bioinformatics 27(3), 405–407 (2011)
12. Langville, A.N., Meyer, C.D.: Survey: Deeper inside pagerank. Internet Mathematics 1(3), 335–380 (2003)
13. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
14. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab (1999)
15. Pandurangan, G., Khan, M.: Theory of communication networks. In: Algorithms and Theory of Computation Handbook, 2nd edn. CRC Press (2009)
16. Peleg, D.: Distributed computing: a locality-sensitive approach. SIAM, Philadelphia (2000)
17. Perra, N., Fortunato, S.: Spectral centrality measures in complex networks. Phys. Rev. E 78, 36107 (2008)

18. Sankaralingam, K., Sethumadhavan, S., Browne, J.C.: Distributed pagerank for p2p systems. In: Proceedings of the 12th International Symposium on High Performance Distributed Computing, pp. 58–68 (June 2003)
19. Sarma, A.D., Gollapudi, S., Panigrahy, R.: Estimating pagerank on graph streams. In: PODS, pp. 69–78. ACM (2008)
20. Shi, S., Yu, J., Yang, G., Wang, D.: Distributed page ranking in structured p2p networks. In: Proceedings of the 2003 International Conference on Parallel Processing (2003)
21. Wang, J., Liu, J., Wang, C.: Keyword Extraction Based on PageRank. In: Zhou, Z.-H., Li, H., Yang, Q. (eds.) PAKDD 2007. LNCS (LNAI), vol. 4426, pp. 857–864. Springer, Heidelberg (2007)

Dealing with Undependable Workers in Decentralized Network Supercomputing*

Seda Davtyan[1], Kishori Konwar[2], Alexander Russell[1], and Alexander Shvartsman[1]

[1] Department of Computer Science & Engineering, University of Connecticut, Storrs, CT 06269, USA
{seda,acr,aas}@engr.uconn.edu
[2] University of British Columbia, Vancouver, BC V6T 1Z3, Canada
kishori@interchange.ubc.ca

Abstract. Internet supercomputing is an approach to solving partitionable, computation-intensive problems by harnessing the power of a vast number of interconnected computers. This paper presents a new algorithm for the problem of using network supercomputing to perform a large collection of independent tasks, while dealing with undependable processors. The adversary may cause the processors to return bogus results for tasks with certain probabilities, and may cause a subset F of the initial set of processors P to crash. The adversary is constrained in two ways. First, for the set of non-crashed processors $P - F$, the *average* probability of a processor returning a bogus result is inferior to $\frac{1}{2}$. Second, the adversary may crash a subset of processors F, provided the size of $P - F$ is bounded from below. We consider two models: the first bounds the size of $P - F$ by a fractional polynomial, the second bounds this size by a poly-logarithm. Both models yield adversaries that are much stronger than previously studied. Our randomized synchronous algorithm is formulated for n processors and t tasks, with $n \leq t$, where depending on the number of crashes each live processor is able to terminate dynamically with the knowledge that the problem is solved with high probability. For the adversary constrained by a fractional polynomial, the time complexity of the algorithm is $O(\frac{t}{n^{\varepsilon}} \log n \log\log n)$, its work is $O(t \log n \log\log n)$ and message complexity is $O(n \log n \log\log n)$. For the poly-log constrained adversary, the time complexity is $O(n)$, work is $O(t\, poly \log n)$, and message complexity is $O(n\, poly \log n)$. All bounds are shown to hold with high probability.

1 Introduction

Cooperative network supercomputing is becoming increasingly popular for harnessing the power of the global Internet computing platform. A typical Internet supercomputer, e.g., [1, 2], consists of a master computer and a large number of computers called workers, performing computation on behalf of the master. Despite the simplicity and benefits of a single master approach, as the scale of such

* This work is supported in part by the NSF Award 1017232.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 27–41, 2013.

computing environments grows, it becomes unrealistic to assume the existence of the infallible master that is able to coordinate the activities of multitudes of workers. Large-scale distributed systems are inherently dynamic and are subject to perturbations, such as failures of computers and network links, thus it is also necessary to consider fully distributed peer-to-peer solutions.

Interestingly, computers returning bogus results is a phenomenon of increasing concern. While this may occur unintentionally, e.g., as a result of over-clocked processors, workers may in fact wrongly claim to have performed assigned work so as to obtain incentives associated with the system, e.g., higher rank. To address this problem, several works, e.g., [3, 6, 7, 10], study approaches based on a reliable master coordinating unreliable workers. The drawback in these approaches is the reliance on a reliable, bandwidth-unlimited master processor.

In our recent work [4, 5] we began to address this drawback of centralized systems by removing the assumption of an infallible and bandwidth-unlimited master processor. We introduced a decentralized approach, where a collection of worker processors cooperates on a large set of independent tasks without the reliance on a centralized control. Our prior algorithm is able to perform all tasks with high probability (*whp*), while dealing with undependable processors under an assumption that the average probability of live (non-crashed) processors returning incorrect results remains inferior to $\frac{1}{2}$ during the computation. There the adversary is only allowed to crash a constant fraction of processors, and the correct termination of the n-processor algorithm strongly depends on the availability of $\Omega(n)$ live processors.

The goal of this work is to develop a new n-processor algorithm that is able to deal with much stronger adversaries, e.g., those that can crash all but a fractional polynomial in n, or even a poly-log in n, number of processors. One of the challenges here is to enable an algorithm to terminate efficiently in the presence of any allowable number of crashes. Of course, to be interesting, such a solution must be efficient in terms of its work and communication complexities.

Contributions. We consider the problem of performing t tasks in a distributed system of n workers *without* centralized control. The tasks are independent, they admit at-least-once execution semantics, and each task can be performed by any worker in constant time. We assume that tasks can be obtained from some repository (else we can assume that the tasks are initially known). The fully-connected message-passing system is synchronous and reliable. We deal with failure models where crash-prone workers can return incorrect results. We present a randomized decentralized algorithm and analyze it for two different adversaries of increasing strength: constrained by a fractional polynomial, and poly-log constrained. In each of these settings, we assume that all surviving processors return bogus results with the average probability inferior to $\frac{1}{2}$. In more detail our contributions are as follows.

1. Given the initial set of processors P, with $|P| = n$, we formulate two adversarial models, where the adversary can crash a set F of processors, subject to model constraints: *a*) For the first model, constrained by a fractional

polynomial, we have $|P - F| = \Omega(n^{\varepsilon})$, for a constant $\varepsilon \in (0, 1)$. *b*) For the second, poly-log constrained model, we have $|P - F| = \Omega(log^{c} n)$, for a constant $c \geq 1$. The adversary may assign arbitrary constant probability to processors, provided that processors in P return bogus results with the *average* probability inferior to $\frac{1}{2}$. The adversary is additionally constrained, so that the *average* probability of returning bogus results for processors in $P - F$ must remain inferior to $\frac{1}{2}$.

2. We provide a randomized algorithm for n processors and t tasks that works in synchronous rounds, where each processor, starting as a "worker," performs a random task and shares its cumulative knowledge of results with *one* randomly chosen processor. Once a processor accumulates a "sufficient" number of results, it becomes "enlightened." Enlightened processors then "profess" their knowledge by multicasting it to exponentially growing random subsets of processors. When a processor receives a "sufficient" number of such messages, it halts. We note that workers become enlightened without any synchronization, and using only the local knowledge. The values that control "sufficient" numbers of results and messages are established in our analysis and are used as *compile-time* constants.

We consider the protocol, by which the "enlightened" processors "profess" their knowledge and reach termination, to be of independent interest. The protocol's message complexity does not depend on crashes, and the processors can terminate without explicit coordination. This addresses one of the challenges associated with termination when $P - F$ can vary broadly in both models.

3. We analyze the quality and performance of the algorithm for the two adversarial models. For each model we show that all live workers obtain the results of all tasks *whp*, and that these results are correct *whp*. Complexity results for the algorithm also hold *whp*:

a) For the polynomially constrained model we show that work complexity is $O(t \log n \log \log n)$ and message complexity is $O(n \log n \log \log n)$.

b) For the poly-log constrained model we show that work is $O(t\, poly(\log n))$ and message complexity is $O(n\, poly(\log n))$. For this model we note that trivial solutions with all workers doing all tasks may be work-efficient, but they do not guarantee that the results are correct.

Prior Work. Earlier approaches explored ways of improving the quality of the results obtained from untrusted workers in the settings where a bandwidth-unlimited and infallible master is coordinating the workers. Fernandez et al. [7, 6] and Konwar et al. [10] consider a distributed master-worker system where the workers may act maliciously by returning wrong results. Works [7, 6, 10] design algorithms that help the master determine correct results *whp*, while minimizing work. The failure models assume that some fraction of processors can exhibit faulty behavior. Another recent work by Christoforou et al. [3] pursues a game-theoretic approach. Paquette and Pelc [11] consider a model of a fault-prone system in which a decision has to be made on the basis of unreliable information and design a deterministic strategy that leads to a correct decision *whp*.

As already mentioned, our prior work [4] introduced the decentralized approach that eliminates the master, and provided a synchronous algorithm that is able to perform all tasks *whp*, while dealing with incorrect behaviors under an assumption that the average probability of non-crashed processors returning incorrect results remains inferior to $\frac{1}{2}$. There the adversary was allowed to crash fail only a linear in n number of processors, and the analysis for live processors easily followed from the case when no crashes were allowed. That algorithm requires $\Omega(n)$ live processors to terminate correctly. While our new algorithm uses a similar approach to performing tasks, it uses a completely different approach once processors become "enlightened" and commence "professing" their knowledge, ultimately leading to termination.

A related problem, called Do-All, deals with the setting where a set of processors must perform a collection of tasks in the presence of adversity [8, 9]. Here the termination condition is that all tasks must be performed, but the processors need not learn the results of the computation.

Document Structure. In Section 2 we give the models of computation and adversity, and measures of efficiency. Section 3 presents our algorithm. In Section 4 we carry out the analysis of the algorithm and derive complexity bounds (for paucity of space proofs of some lemmas are given in the appendix found at `http://www.engr.uconn.edu/~sad06005/APX/ICDCN13.pdf`). We conclude in Section 5 with a discussion.

2 Model of Computation and Definitions

System Model. There are n processors, each with a unique identifier (id) from set $P = [n]$. We refer to the processor with id i as processor i. The system is synchronous and processors communicate by exchanging reliable messages. Computation is structured in terms of synchronous *rounds*, where in each round a processor can send and receive messages, and perform local polynomial computation, where the local computation time is assumed to be negligible compared to message latency. The duration of each round depends on the algorithm and need not be constant (e.g., it may depend on n). Messages received by a processor in a given step include all messages sent to it in the previous step.

Tasks. There are t tasks to be performed, each with a unique id from set $\mathcal{T} = [t]$. We refer to the task with id j as $Task[j]$. The tasks are (a) similar, meaning that any task can be done in constant time by any processor, (b) independent, meaning that each task can be performed independently of other tasks, and (c) idempotent, meaning that each task admits at-least-once execution semantics and can be performed concurrently. For simplicity, we assume that the outcome of each task is a binary value. The problem is most interesting when there are at least as many tasks as there are processors, thus we consider $t \geq n$.

Models of Adversity. Processors are undependable in that a processor may compute the results of tasks incorrectly and it may crash. Following a crash, a processor performs no further actions. Otherwise, each processor adheres to

the protocol established by the algorithm it executes. We refer to non-crashed processors as *live*. We consider an oblivious adversary that decides prior to the computation what processors to crash and when to crash them. The maximum number of processors that can crash is established by the adversarial models (specified below).

For each processor $i \in P$, we define p_i to be the probability of processor i returning incorrect results, independently of other processors, such that, $\frac{1}{n}\sum_i p_i < \frac{1}{2} - \zeta$, for some $\zeta > 0$. That is, the average probability of processors in P returning incorrect results is inferior to $\frac{1}{2}$. We use the constant ζ to ensure that the average probability of incorrect computation does not become arbitrarily close to $\frac{1}{2}$ as n grows arbitrarily large. The individual probabilities of incorrect computation are unknown to the processors.

For an execution of an algorithm, let F be the set of processors that adversary crashes. The adversary is constrained in that the *average* probability of processors in $P - F$ computing results incorrectly remains inferior to $\frac{1}{2}$. We define two adversarial models:

Model $\mathcal{F}_{fp}$, *adversary constrained by a fractional polynomial* :
$|P - F| = \Omega(n^{\varepsilon})$, for a constant $\varepsilon \in (0, 1)$.

Model $\mathcal{F}_{pl}$, *poly-log constrained adversary* :
$|P - F| = \Omega(log^{c} n)$, for a constant $c \geq 1$.

Measures of Efficiency. We assess the efficiency of algorithms in terms of *time*, *work*, and *message* complexities. We use the conventional measures of time complexity and work complexity. We assess message complexity as the number of point-to-point messages sent during the execution of an algorithm. Lastly, we use the common definition of *an event* $\mathcal{E}$ *occurring with high probability* (*whp*) to mean that $\mathbf{Pr}[\mathcal{E}] = 1 - O(n^{-\alpha})$ for some constant $\alpha > 0$.

3 Algorithm Description

We now present our decentralized solution, called algorithm DAKS (for Decentralized Algorithm with Knowledge Sharing), that employs no master and instead uses a gossip-based approach. We start by specifying in detail the algorithm for n processors and $t = n$ tasks, then we generalize it for t tasks, where $t \geq n$.

The algorithm is structured in terms of a main loop. The principal data structures at each processor are two arrays of size linear in n: one accumulates knowledge gathered from the processors, and another stores the results. All processors start as *workers*. In each iteration, any worker performs one randomly selected task and sends its knowledge to just one other randomly selected processor. When a worker obtains "enough" knowledge about the tasks performed in the system, it computes the final results, stops being a worker, and becomes "enlightened." Such processors no longer perform tasks, and instead "profess" their knowledge to other processors by means of multicasts to exponentially increasing random sets of processors. The main loop terminates when a certain number of messages is received from enlightened processors.

```
   Procedure for processor i;
     external n, k_m, k_t /* n is number of processors and tasks */
              /*  k_m, k_t are constants */
     Task[1..n] /* set of tasks */
     R_i[1..n] init ∅^n /* set of collected results */
     Results_i[1..n] init ⊥ /* array of results */
     prof_ctr init 0 /* number of profess messages received */
     r init 0 /* round number */
     ℓ init 0 /* specifies the number of profess messages to be sent per iteration */
     worker init true /* indicates whether the processor is still a worker */
     while prof_ctr < k_m log n do
       Send:
1:       if worker then
2:         Let q be a randomly selected processor from P
3:         Send ⟨share, R_i[ ]⟩ to processor q
4:       else
5:         Let D be a set of 2^ℓ log n randomly selected processors from P
6:         Send ⟨profess, R_i[ ]⟩ to processors in D
7:         ℓ ← ℓ + 1
       Receive:
8:       Let M be the set of received messages
9:       prof_ctr ← prof_ctr + |{m : m ∈ M ∧ m.type = profess}|
10:        for all j ∈ T do
11:              R_i[j] ← R_i[j] ∪ (⋃_{m∈M} m.R[j]) /* update knowledge */
       Compute:
12:      r ← r + 1
13:      if worker then
14:          Randomly select j ∈ T and compute the result v_j for Task[j]
15:        R_i[j] ← R_i[j] ∪ {⟨v_j, i, r⟩}
16:        if min_{j∈T}{|R_i[j]|} ≥ k_t log n then /* i has enough results */
17:          for all j ∈ T do
18:            Results_i[j] ← u such that triples ⟨u, _, _⟩ form a plurality in R_i[j]
19:          worker ← false /* worker becomes enlightened */
   end
```

Fig. 1. Algorithm DAKS for $t = n$; code at processor i for $i \in P$

The pseudocode for algorithm DAKS is given in Figure 1. We now give the details.

Local Knowledge and State. Every processor i maintains the following:

- Array of results $R_i[1..n]$, where element $R_i[j]$, for $j \in \mathcal{T}$, is a set of results for $Task[j]$. Each $R_i[j]$ is a set of triples $\langle v, i, r\rangle$, where v is the result computed for $Task[j]$ by processor i in round r (here the inclusion of r ensures that the results computed by processor i in different rounds are preserved).
- The array $Results_i[1..n]$ stores the final results.
- The $prof\_ctr$ stores the number of messages received from enlightened processors.

– r is the round (iteration) number that is used by *workers* to timestamp the computed results.
– ℓ is the exponent that controls the number of messages multicast by enlightened processors.

Control Flow. The algorithm iterations are controlled by the main while-loop, and we use the term *round* to refer to a single iteration of the loop. The loop contains three stages, viz., *Send*, *Receive*, and *Compute*.

Processors communicate using messages m that contain pairs $\langle type, R[\,]\rangle$. Here $m.R[\,]$ is the sender's array of results. When a processor is a worker, it sends messages with $m.type = \mathsf{share}$. When a processor becomes enlightened, it sends messages with $m.type = \mathsf{profess}$. The loop is controlled by the counter $prof\_ctr$ that keeps track of the received messages of type profess. We next describe the stages in detail.

***Send* stage:** Any worker chooses a target processor q at random and sends its array of results $R[\,]$ to processor q in a share message. Any enlightened processor chooses a set $D \subseteq P$ of processors at random and sends the array of results $R[\,]$ to processors in D in a profess message. The size of the set D is $2^\ell \log n$, where initially $\ell = 0$, and once a processor is enlightened, it increments ℓ by 1 in every round.

***Receive* stage:** Processor i receives messages (if any) sent to it in the preceding *Send* stage. The processor increments its $prof\_ctr$ by the number of profess messages received. For each task j, the processor updates its $R_i[j]$ by including the results received in all messages.

***Compute* stage:** Any worker randomly selects task j, computes the result v_j, and adds the triple $\langle v_j, i, r\rangle$ for round r to $R_i[j]$. For each task the worker checks whether "enough" results were collected. Once at least $k_t \log n$ results for each task are obtained, the worker stores the final results in $Results_i[\,]$ by taking the plurality of results for each task, and becomes enlightened. (In Section 4 we reason about the compile-time constant k_t, and establish that $k_t \log n$ results are sufficient for our claims.) Enlightened processors rest on their laurels in subsequent *Compute* stages.

Reaching Termination. We note that a processor must become enlightened before it can terminate. Processors can become enlightened at different times and without any synchronization. Once enlightened, they profess their knowledge by multicasting it to exponentially growing random subsets D of processors. When a processor receives sufficiently many such messages, i.e., $k_m \log n$, it halts, again without any synchronization, and using only the local knowledge. We consider this protocol to be of independent interest. In Section 4 we reason about the compile-time constant k_m, and establish that $k_m \log n$ profess messages are sufficient for our claims; additionally we show that the protocol's efficiency can be assessed independently of the number of crashes.

Extending the Algorithm for $t \geq n$. We now modify the algorithm to handle arbitrary number of tasks t such that $t \geq n$. Let $\mathcal{T}' = [t]$ be the set of unique

task identifiers, where $t \geq n$. We segment the t tasks into chunks of $\lceil t/n \rceil$ tasks, and construct a new array of chunk-tasks with identifiers in $\mathcal{T} = [n]$, where each chunk-task takes $\Theta(t/n)$ time to perform by any live processor. We now use algorithm DAKS, where the only difference is that each *Compute* stage takes $\Theta(t/n)$ time to perform a chunk-task.

4 Algorithm Analysis

Here we analyze the performance of algorithm DAKS in our two failure models. We start by giving several lemmas relevant to all models, then detail the analysis for each model. We start by stating the Chernoff bound.

Lemma 1 (Chernoff Bounds). *Let $X_1, X_2, \cdots, X_n$ be n independent Bernoulli random variables with $\mathbf{Pr}[X_i = 1] = p_i$ and $\mathbf{Pr}[X_i = 0] = 1 - p_i$, then it holds for $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ that for all $\delta > 0$, (i) $\mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\frac{\mu\delta^2}{3}}$, and (ii) $\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}}$.*

The following lemma shows that if $\Theta(n \log n)$ profess messages are sent by the enlightened processors, then every live processor either terminates or becomes enlightened *whp*.

Lemma 2. *Let r be the first round by which the total number of* profess *messages is $\Theta(n \log n)$. Then by the end of this round every live processor either halts or becomes enlightened whp.*

Proof. Let $\tilde{n} = kn \log n$ be the number of profess messages sent by round r, where $k > 1$ is a sufficiently large constant. We show that *whp* every live processor received at least $(1-\delta)k \log n$ profess messages, for some constant $\delta \in (0, 1)$.

Let us assume that there exists processor q that receives less than $(1-\delta)k \log n$ of such messages. We prove that *whp* such a processor does not exist.

Since $\tilde{n}$ profess messages are sent by round r, there were $\tilde{n}$ random selections of processors from set P in line 5 of algorithm DAKS, possibly by different enlightened processors. We denote by i an index of one of the random selections in line 5. Let X_i be a Bernoulli random variable such that $X_i = 1$ if processor q was chosen by an enlightened processor and $X_i = 0$ otherwise.

We define the random variable $X = \sum_{i=1}^{\tilde{n}}$ to estimate the total number of times processor q is selected by round r. In line 5 every enlightened processor chooses a destination for the profess message uniformly at random, and hence $\mathbf{Pr}[X_i = 1] = \frac{1}{n}$. Let $\mu = \mathbb{E}[X] = \sum_{i=1}^{\tilde{n}} X_i = \frac{1}{n} k\, n \log n = k \log n$, then by applying Chernoff bound, for the same δ chosen as above, we have:

$$\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}} \leq e^{-\frac{(k \log n)\delta^2}{2}} \leq \frac{1}{n^{\frac{b\delta^2}{2}}} \leq \frac{1}{n^{\alpha}}$$

where $\alpha > 1$ for some sufficiently large b. Hence, by letting $k_m = (1-\delta)k$, we have $\mathbf{Pr}[X \leq k_m \log n] \leq \frac{1}{n^{\alpha}}$ for some $\alpha > 1$. Now let us denote by $\mathcal{E}_q$ the fact that $prof\_ctr_q \geq k_m \log n$ by the end of round r, and let $\bar{\mathcal{E}}_q$ be the complement

of that event. By Boole's inequality we have $\mathbf{Pr}[\cup_q \bar{\mathcal{E}}_q] \leq \sum_q \mathbf{Pr}[\bar{\mathcal{E}}_q] \leq \frac{1}{n^\beta}$, where $\beta = \alpha - 1 > 0$. Hence each processor $q \in P$ is the destination of at least $k_m \log n$ profess messages *whp*, i.e.,

$$\mathbf{Pr}[\cap_q \mathcal{E}_q] = \mathbf{Pr}[\overline{\cap_q \bar{\mathcal{E}}_q}] = 1 - \mathbf{Pr}[\cap_q \bar{\mathcal{E}}_q] \geq 1 - \frac{1}{n^\beta} ,$$

and hence, it halts.

It follows that every live processor becomes enlightened. Indeed, even if there exists a processor that does not receive enough profess messages ($\geq k_m \log n$) to halt, it receives at least one such message with extremely high probability, and hence becomes enlightened. □

Recall that the constant k_m from the proof of Lemma 2 is used as a compile-time constant in algorithm DAKS.

Lemma 3. *Once a processor $q \in P - F$ becomes enlightened, every live processor halts in additional $O(\log n)$ rounds whp.*

Proof. According to Lemma 2 if $\Theta(n \log n)$ profess messages are sent then every processor halts *whp*. Given that processor q does not crash it takes q at most $\log n$ rounds to send $\Theta(n \log n)$ profess messages (per line 5 in Figure 1), regardless of the actions of other processors. Hence, *whp* every live processor halts in $O(\log n)$ rounds. □

We denote by L the number of rounds required for a processor from the set $P - F$ to become enlightened. We next analyze the value of L for models $\mathcal{F}_{fp}$ and $\mathcal{F}_{pl}$. The compile-time constant k_t appearing in algorithm DAKS is computed as $\max\{k_1, k_2, k_3\}$, where k_2 and k_3 are from the proofs of Lemmas 4 and 8. The constant $k_1 = (1 - \delta)\lambda k$ is from the proof of Lemma 5 of [4].

4.1 Analysis for Model $\mathcal{F}_{fp}$

In model $\mathcal{F}_{fp}$ we have $|F| \leq n - n^\varepsilon$. Let F_r be the actual number of crashes that occur prior to round r. For the purpose of analysis we divide an execution of the algorithm into two epochs: epoch $\mathfrak{a}$ consists of all rounds r where $|F_r|$ is at most linear in n, so that when the number of live processors is at least $c'n$ for some suitable constant c'; epoch $\mathfrak{b}$ consists of all rounds r starting with first round r' (it can be round 1) when the number of live processors drops below some $c'n$ and becomes $c''n^\varepsilon$ for some suitable constant and c''. Note that either epoch may be empty.

For a small number of failures in epoch $\mathfrak{a}$, we anchor the analysis to our work [4]. Here $|F_r|$ is bounded as in model $\mathcal{F}$ of [4] with at most hn processor crashes for a constant $h \in (0, 1)$, and the new algorithm incurs costs exactly as algorithm A of [4] does. (To avoid a complete restatement, we refer the kind reader to that earlier paper, and focus here on the new analysis).

Next we consider epoch $\mathfrak{b}$. If the algorithm terminates in round r', the first round of the epoch, the cost remains the same as for algorithm A [4]. If it

does not terminate, it incurs additional costs associated with the processors in $P - F_{r'}$, where $|P - F_{r'}| \leq c''n^{\varepsilon}$. We analyze the costs for epoch $\mathfrak{b}$ in the rest of this section. The final message and work complexities will be at most the worst case complexity for epoch $\mathfrak{a}$ plus the additional costs for epoch $\mathfrak{b}$ incurred while $|P - F| = \Omega(n^{\varepsilon})$ per model $\mathcal{F}_{fp}$.

First we show that *whp* it will take $L = O(n^{1-\varepsilon} \log n \log\log n)$ rounds for a worker from the set $P - F$ to become enlightened in epoch $\mathfrak{b}$.

Lemma 4. *In $O(n^{1-\varepsilon} \log n)$ rounds of epoch $\mathfrak{b}$ every task is performed $\Theta(\log n)$ times whp by processors in $P - F$.*

Proof. Let us assume that after $\tilde{r} = kn^{1-\varepsilon} \log n$ rounds of algorithm DAKS, where k is a sufficiently large constant and $0 < \varepsilon < 1$ is a constant, there exists a task τ that is performed less than $(1-\delta)k \log n$ times among all live workers, for some $\delta > 0$. We prove that *whp* such a task does not exist.

According to our assumption at the end of round $\tilde{r}$ for some task τ, we have $|\cup_{j=1}^{n} R_j[\tau]| < k_2 \log n$, where $k_2 = (1-\delta)\log n$. Let X_i be a Bernoulli random variable such that $X_i = 1$ if the task was chosen to be performed in line 14 of the algorithm by a processor in $P - F$, and $X_i = 0$ otherwise.

Let us next define the random variable $X = X_1 + \cdots + X_{\tilde{r}cn^{\varepsilon}}$ to count the total number of times task τ is performed by the end of $\tilde{r}$ rounds by workers in $P - F$.

Note that according to line 14 any worker picks a task uniformly at random. To be more specific let x be an index of one of $\tilde{r}cn^{\varepsilon}$ executions of line 14 by processors in $P - F$. Observe that for any x, $\mathbf{Pr}[X_x = 1] = \frac{1}{n}$ given that the workers choose task τ uniformly at random. Let $\mu = \mathbb{E}[X] = \sum_{x=1}^{\tilde{r}cn^{\varepsilon}} \frac{1}{n} = \frac{\tilde{r}cn^{\varepsilon}}{n} = kc \log n$, then by applying Chernoff bound, for the same $\delta > 0$ chosen as above, we have:

$$\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}} \leq e^{-\frac{(kc\log n)\delta^2}{2}} \leq \frac{1}{n^{\frac{b\delta^2}{2}}} \leq \frac{1}{n^{\alpha}}$$

where $\alpha > 1$ for some sufficiently large b. Now let us denote by $\mathcal{E}_\tau$ the fact that $|\cup_{i=1}^{n} R_i(\tau)| > k_2 \log n$ by the round $\tilde{r}$ of the algorithm and we denote by $\bar{\mathcal{E}}_\tau$ the complement of that event. Next by Boole's inequality we have $\mathbf{Pr}[\cup_\tau \bar{\mathcal{E}}_\tau] \leq \sum_\tau \mathbf{Pr}[\bar{\mathcal{E}}_\tau] \leq \frac{1}{n^{\beta}}$, where $\beta = \alpha - 1 > 0$. Hence each task is performed at least $\Theta(\log n)$ times by workers in $P - F$ *whp*, i.e.,

$$\mathbf{Pr}[\cap_\tau \mathcal{E}_\tau] = \mathbf{Pr}[\overline{\cup_\tau \bar{\mathcal{E}}_\tau}] \geq 1 - \frac{1}{n^{\beta}}.$$

We now focus only on the set of live processors $P - F$ with $|P - F| \geq cn^{\varepsilon}$. Our goal is to show that in $O(n^{1-\varepsilon} \log n \log\log n)$ rounds of algorithm DAKS at least one processor from $P - F$ becomes enlightened.

We first show that any triple $\tau = \langle x, y, z \rangle$ generated by a processor in $P - F$ is known to all processors in $P - F$ in $O(n^{1-\varepsilon} \log n \log\log n)$ rounds of algorithm DAKS.

We denote by $S(r) \subseteq P - F$ the set of processors that know a certain triple τ by round r, and let $s(r) = |S(r)|$. Next lemma shows that after $r_1 = O(n^{1-\varepsilon} \log n \log\log n)$ rounds $s(r_1) = \Theta(\log^3 n)$.

Lemma 5. *After* $r_1 = O(n^{1-\varepsilon} \log n \log\log n)$ *rounds of epoch* $\mathfrak{b}$, $s(r_1) = \Theta(\log^3 n)$ *whp.*

Proof. Consider a scenario when a processor $p \in P - F$ generates a triple τ. Then the probability that processor p sends triple τ to at least one other processor $q \in P - F$, where $p \neq q$, in $n^{1-\varepsilon} \log n$ rounds is at least $1 - (1 - \frac{cn^{\varepsilon}}{n})^{n^{1-\varepsilon} \log n} \geq 1 - e^{-b \log n} > 1 - \frac{1}{n^{\alpha}}$ s.t. $\alpha > 0$ for some appropriately chosen b and for a sufficiently large n. Similarly, it is straightforward to show that the number of live processors that learn about τ doubles every $n^{1-\varepsilon} \log n$ rounds, hence *whp* after $(n^{1-\varepsilon} \log n) \cdot 3 \log\log n = O(n^{1-\varepsilon} \log n \log\log n)$ rounds the number of processors in $P - F$ that learn about τ is $\Theta(\log^3 n)$. □

In the next lemma we reason about the growth of $s(r)$ after round r_1.

Lemma 6. *Let* r_2 *be the first round after round* r_1 *in epoch* $\mathfrak{b}$ *such that* $r_2 - r_1 = O(n^{1-\varepsilon} \log n)$. *Then* $s(r_2) \geq \frac{3}{5}|P - F|$ *whp.*

Next we calculate the number of rounds required for the remaining $\frac{2}{5}|P - F|$ processors in $P - F$ to learn τ. Let $U_d \subset P - F$ be the set of workers that do not learn τ after $O(n^{1-\varepsilon} \log n \log\log n)$ rounds of algorithm DAKS. According to Lemma 6 we have $|U_d| \leq \frac{2}{5}|P - F|$.

Lemma 7. *Once every task is performed* $\Theta(\log n)$ *times in epoch* $\mathfrak{b}$ *by processors in* $P - F$ *then at least one worker from* $P - F$ *becomes enlightened in* $O(n^{1-\varepsilon} \log n \log\log n)$ *rounds, whp.*

(The proofs of Lemmas 6 and 7 are in the Appendix.)

Theorem 1. *In epoch* $\mathfrak{b}$ *algorithm* DAKS *performs all* n *tasks correctly, and the results are known at every live processor after* $O(n^{1-\varepsilon} \log n \log\log n)$ *rounds.*

Proof sketch. According to algorithm DAKS (line 18) every live processor computes the result of every task t by taking a plurality among all the results. According to our assumption in every execution of the algorithm the average probability of workers computing incorrectly is inferior to $\frac{1}{2}$ among live processors. On the other hand Lemma 4 shows that for every task processors in $P - F$ calculate $\Theta(\log n)$ results in $O(n^{1-\varepsilon} \log n)$ rounds *whp*. Lemmas 3, 6 and 7 show that *whp* in $O(n^{1-\varepsilon} \log n \log\log n)$ rounds of algorithm DAKS at least $\Theta(n \log n)$ triples generated by processors in $P - F$ will be known to all live processors. The claim then follows along the lines of the proof of Theorem 1 of [4]. □

According to Lemma 7, after $O(n^{1-\varepsilon} \log n \log\log n)$ rounds of epoch $\mathfrak{b}$ at least one processor in $P - F$ becomes enlightened. Furthermore, once a processor in $P - F$ becomes enlightened, according to Lemma 3 after $O(\log n)$ rounds of the algorithm every live processor becomes enlightened and then terminates, *whp*. Next we assess work and message complexities.

Theorem 2. *For* $t = n$ *algorithm* DAKS *has work and message complexity* $O(n \log n \log\log n)$.

Proof. To obtain the result we combine the costs associated with epoch $\mathfrak{a}$ with the costs of epoch $\mathfrak{b}$. As reasoned earlier, the worst case costs for epoch $\mathfrak{a}$ are captured in Theorems 2 and 4 of [4]. Specifically, both complexity bounds are $\Theta(n \log n)$.

For epoch $\mathfrak{b}$ (if it is not empty), where $|P - F| = O(n^{\varepsilon})$, algorithm DAKS terminates after $O(n^{1-\varepsilon} \log n \log\log n)$ rounds *whp* and there are $\Theta(n^{\varepsilon})$ live processors, thus its work is $O(n \log n \log\log n)$. In every round if a processor is a *worker* it sends a share message to one randomly chosen processor. If a processor is enlightened then it sends profess messages to a randomly selected subset of processors. In every round $\Theta(n^{\varepsilon})$ share messages are sent. Since *whp* algorithm DAKS terminates in $O(n^{1-\varepsilon} \log n \log\log n)$ rounds, $\Theta(n \log n \log\log n)$ share messages are sent. On the other hand according to Lemma 2 if during the execution of the algorithm $\Theta(n \log n)$ profess messages are sent then every processor terminates *whp*. Hence, the message complexity is $O(n \log n \log\log n)$.

The worst case costs of the algorithm correspond to executions with non-empty epoch $\mathfrak{b}$. In this case the costs from epoch $\mathfrak{a}$ are asymptotically absorbed into the costs of epoch $\mathfrak{b}$ computed above. □

Last, we consider the complexities of algorithm DAKS for t tasks such that $t \geq n$. The following result is trivially obtained from the analysis for $t = n$ by multiplying the time and work complexities by the size of the chunk $\Theta(t/n)$; the message complexity is unchanged.

Theorem 3. *For $t \geq n$ algorithm* DAKS *has time complexity $O(\frac{t}{n^{\varepsilon}} \log n \log\log n)$, work complexity $O(t \log n \log\log n)$ and message complexity $O(n \log n \log\log n)$.*

Proof sketch. For epoch $\mathfrak{a}$ the algorithm has time $\Theta(\log n)$, work $\Theta(t \log n)$, and message complexity is $\Theta(n \log n)$ For epoch $\mathfrak{b}$, just as for the case of $t = n$, the algorithm takes $O(n^{1-\varepsilon} \log n \log\log n)$ iterations for at least one processor from set $P - F$ to become enlightened *whp*, except that each iteration now takes $\Theta(t/n)$ time. This yields time complexity $O(\frac{t}{n^{\varepsilon}} \log n \log\log n)$. Work complexity is then $O(t \log n \log\log n)$. The message complexity remains the same at $O(n \log n \log\log n)$ as the number of messages does not change. The final assessment is obtained by combining the costs of epoch $\mathfrak{a}$ and epoch $\mathfrak{b}$. □

4.2 Failure Model $\mathcal{F}_{pl}$

In model $\mathcal{F}_{pl}$ we have $|F| \leq n - poly \log n$, thus, $|P - F| = \Omega(poly \log n)$. We first note that when a large number of crashes make $|P - F| = \Theta(poly \log n)$, one may attempt a trivial solution where all live processors perform all t tasks. While this approach has efficient work, it does not guarantee that workers compute correct results; in fact, since the overall probability of live workers producing bogus results can be close to $\frac{1}{2}$, this may yield on the average just slightly more than $t/2$ correct results.

For executions in $\mathcal{F}_{pl}$, let $|P - F|$ be at least $a \log^{c} n$, for specific constants a and c satisfying the model constraints. Let F_r be the actual number of crashes

that occur prior to round r. For the purpose of analysis we divide an execution of the algorithm into two epochs: epoch $\mathfrak{b}'$ consists of all rounds r where $|F_r|$ remains bounded as in model $\mathcal{F}_{fp}$ (for reference, this epoch combines epoch $\mathfrak{a}$ and epoch $\mathfrak{b}$ from the previous section); epoch $\mathfrak{c}$ consists of all rounds r starting with first round r'' (it can be round 1) when the number of live processors drops below $a_1 n^{\varepsilon}$ and becomes $a_2 \log^{a_3} n$ for some suitable constants a_1, a_2, and a_3 (here $a_2 \geq a$ and $a_3 \geq c$). Note that either epoch may be empty.

In epoch $\mathfrak{b}'$ the algorithm incurs costs exactly as in model $\mathcal{F}_{fp}$.

Next we consider epoch $\mathfrak{c}$. If the algorithm terminates in round r'', the first round of the epoch, the costs remain the same as the costs analyzed for $\mathcal{F}_{fp}$ in the previous section.

If it does not terminate, it incurs additional costs associated with the processors in $P - F_{r''}$, where $|P - F_{r''}| \leq b \log^c n$. We analyze the costs for epoch $\mathfrak{c}$ in the rest of this section. The final message and work complexities will be at most the worst case complexity for epoch $\mathfrak{b}'$ plus the additional costs for epoch $\mathfrak{c}$ incurred while $|P - F| = \Omega(poly \log n)$ per model $\mathcal{F}_{pl}$.

The following lemma shows that within some $O(n)$ rounds in epoch $\mathfrak{c}$ every task is chosen for execution $\Theta(\log n)$ times by processors in $P - F$ *whp*.

Lemma 8. *In $O(n)$ rounds of epoch $\mathfrak{c}$ every task is performed $\Theta(\log n)$ times whp by processors in $P - F$.*

Proof. Let us assume that after $\tilde{r}$ rounds of algorithm DAKS, where $\tilde{r} = kn$ (k is a sufficiently large constant), there exists a task τ that is performed less than $(1 - \delta)k \log n$ times by the processors in $P - F$, for some $\delta > 0$. We prove that *whp* such a task does not exist.

According to our assumption at the end of round $\tilde{r}$ for some task τ, we have $|\cup_{j=1}^{n} R_j[\tau]| < k_3 \log n$, where $k_3 = (1 - \delta) \log n$. Let X_i be a Bernoulli random variable such that $X_i = 1$ if the task was chosen to be performed in line 14 of the algorithm by processors in $P - F$, and $X_i = 0$ otherwise.

Let us next define the random variable $X = X_1 + \cdots + X_{\tilde{r}a \log^c n}$ to count the total number of times task τ is performed by the end of $\tilde{r}$ rounds by workers in $P - F$.

Note that according to line 14 any worker picks a task uniformly at random. To be more specific let x be an index of one of $\tilde{r}a \log^c n$ executions of line 14 by processors in $P - F$. Observe that for any x, $\mathbf{Pr}[X_x = 1] = \frac{1}{n}$ given that the workers choose task τ uniformly at random. Let $\mu = \mathbb{E}[X] = \sum_{x=1}^{\tilde{r}a \log^c n} \frac{1}{n} = ka \log^c n > k_3 \log n$, then by applying Chernoff bound, for the same $\delta > 0$ chosen as above, we have:

$$\mathbf{Pr}[X \leq (1 - \delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}} \leq e^{-\frac{(ka \log^c n)\delta^2}{2}} \leq \frac{1}{n^{\frac{b \log^{c-1} n \delta^2}{2}}} \leq \frac{1}{n^{\alpha}}$$

where $\alpha > 1$ for some sufficiently large b. Now let us denote by $\mathcal{E}_\tau$ the fact that $|\cup_{i=1}^{n} R_i(\tau)| > k_3 \log n$ by the round $\tilde{r}$ of the algorithm, and we let $\bar{\mathcal{E}_\tau}$ be the complement of that event. Next by Boole's inequality we have $\mathbf{Pr}[\cup_\tau \bar{\mathcal{E}_\tau}] \leq$

$\sum_\tau \mathbf{Pr}[\bar{\mathcal{E}_\tau}] \leq \frac{1}{n^\beta}$, where $\beta = \alpha - 1 > 0$. Hence each task is performed at least $\Theta(\log n)$ times *whp*, i.e., $\mathbf{Pr}[\cap_\tau \mathcal{E}_\tau] = \mathbf{Pr}[\overline{\cup_\tau \bar{\mathcal{E}_\tau}}] \geq 1 - \frac{1}{n^\beta}$. □

Next we show that once each task is done a logarithmic number of times, then every processor in $P - F$ will acquire a sufficient collection of triples in at most a linear number of rounds to become enlightened.

Lemma 9. *Once every task is performed $\Theta(\log n)$ times by processors in $P - F$ then at least one worker in $P - F$ becomes enlightened whp after $O(n)$ rounds in epoch $\mathfrak{c}$.*

(The proof of Lemma 9 is in the Appendix.)

Theorem 4. *Algorithm* DAKS *performs all n tasks correctly, and the results are known at every live processor after $O(n)$ rounds of epoch $\mathfrak{c}$ whp.*

Proof sketch. The proof of this theorem is similar to the proof of Theorem 1. This is because by Lemma 8 in $O(n)$ rounds $\Theta(\log^c n)$, where $c \geq 1$ is a constant, triples are generated by processors in $P - F$. According to Lemmas 3 and 9 in $O(n)$ rounds of the algorithm every live worker will become enlightened. □

According to Lemma 9, after $O(n)$ rounds of epoch $\mathfrak{c}$ at least one processor in $P - F$ becomes enlightened. Furthermore, once a processor in $P - F$ becomes enlightened, according to Lemma 3 after $O(\log n)$ rounds of the algorithm every live processor becomes enlightened and then terminates, *whp*. Next we assess work and message complexities (using the approach in the proof of Theorem 2).

Theorem 5. *Algorithm* DAKS *has work and message complexity $O(n\, poly \log n)$.*

Proof. To obtain the result we combine the costs associated with epoch $\mathfrak{b}'$ with the costs of epoch $\mathfrak{c}$. As reasoned earlier, the worst case costs for epoch $\mathfrak{b}'$ are given in Theorem 2.

For epoch $\mathfrak{c}$ (if it is not empty), where $|P - F| = \Theta(poly \log n)$, algorithm DAKS terminates after $O(n)$ rounds *whp* and there are $\Theta(poly \log n)$ live processors, thus its work is $\Theta(n\, poly \log n)$. In every round if a processor is a *worker* it sends a share message to one randomly chosen processor. If a processor is enlightened then it sends profess messages to a randomly selected subset of processors. In every round $\Theta(poly \log n)$ share messages are sent. Since *whp* algorithm DAKS terminates in $O(n)$ rounds, $\Theta(n\, poly \log n)$ share messages are sent. On the other hand according to Lemma 2 if during the execution of the algorithm $\Theta(n \log n)$ profess messages are sent then every processor terminates *whp*. Hence, the message complexity is $\Theta(n\, poly \log n)$.

The worst case costs of the algorithm correspond to executions with non-empty epoch $\mathfrak{c}$. In this case the costs from epoch $\mathfrak{b}'$ are asymptotically absorbed into the costs of epoch $\mathfrak{c}$ computed above. □

Last, we consider algorithm DAKS for t tasks such that $t \geq n$.

Theorem 6. *For $t \geq n$ algorithm* DAKS *has time complexity $O(t)$, work complexity $O(t\, poly \log n)$ and message complexity $O(n\, poly \log n)$.*

Proof sketch. The result is obtained (as in Theorem 3) by combining the costs from epoch $\mathfrak{b}'$ (ibid.) with the costs of epoch $\mathfrak{c}$ derived from the analysis for $t = n$ (Theorem 5) by multiplying the time (number of rounds) and work complexities by the size of the chunk $\Theta(t/n)$; the message complexity is unchanged. □

5 Conclusion

We presented a synchronous decentralized algorithm that can perform a set of tasks using a distributed system of undependable, crash-prone processors. Our randomized algorithm allows the processors to compute the correct results and make the results available at every live participating processor, *whp*. We provided time, message, and work complexity bounds for two adversarial strategies, viz., (a) all but $\Omega(n^{\varepsilon})$ processors can crash, and (b) all but a poly-logarithmic number of processors can crash. Future work considers the problem in synchronous and asynchronous decentralized systems, with more virulent adversarial settings in both. Lastly, it is interesting to derive strong lower bounds on the message, time, and work complexities in various models.

References

[1] Distributed.net, http://www.distributed.net/
[2] Seti@home, http://setiathome.ssl.berkeley.edu/
[3] Christoforou, E., Fernandez, A., Georgiou, C., Mosteiro, M.: Algorithmic mechanisms for internet supercomputing under unreliable communication. In: NCA, pp. 275–280 (2011)
[4] Davtyan, S., Konwar, K.M., Shvartsman, A.A.: Robust Network Supercomputing without Centralized Control. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 435–450. Springer, Heidelberg (2011)
[5] Davtyan, S., Konwar, K.M., Shvartsman, A.A.: Decentralized network supercomputing in the presence of malicious and crash-prone workers. In: Proc. of 31st ACM Symp. on Principles of Distributed Computing, pp. 231–232 (2012)
[6] Fernandez, A., Georgiou, C., Lopez, L., Santos, A.: Reliably executing tasks in the presence of malicious processors. Technical Report Numero 9 (RoSaC-2005-9), Grupo de Sistemas y Comunicaciones, Universidad Rey Juan Carlos (2005), http://gsyc.escet.urjc.es/publicaciones/tr/RoSaC-2005-9.pdf
[7] Fernandez, A., Georgiou, C., Lopez, L., Santos, A.: Reliably executing tasks in the presence of untrusted entities. In: SRDS, pp. 39–50 (2006)
[8] Georgiou, C., Shvartsman, A.A.: Cooperative Task-Oriented Computing; Algorithms and Complexity, 1st edn. Morgan & Claypool Publishers (2011)
[9] Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997)
[10] Konwar, K.M., Rajasekaran, S., Shvartsman, M.M.A.A.: Robust Network Supercomputing with Malicious Processes. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 474–488. Springer, Heidelberg (2006)
[11] Paquette, M., Pelc, A.: Optimal decision strategies in byzantine environments. Parallel and Distributed Computing 66(3), 419–427 (2006)

Decentralized Erasure Coding for Efficient Data Archival in Distributed Storage Systems

Lluis Pamies-Juarez[1], Frederique Oggier[1], and Anwitaman Datta[2]

[1] School of Mathematical and Physical Sciences
[2] School of Computer Engineering
Nanyang Technological University, Singapore
{lpjuarez,frederique,anwitaman}@ntu.edu.sg

Abstract. Distributed storage systems usually achieve fault tolerance by replicating data across different nodes. However, redundancy schemes based on *erasure codes* can provide a storage-efficient alternative to replication. This is particularly suited for *data archival* since archived data is rarely accessed. Typically, the migration to erasure-encoded storage does not leverage on the existing replication based redundancy, and simply discards (garbage collects) the excessive replicas. In this paper we propose a new *decentralized erasure coding process* that achieves the migration in a *network-efficient* manner in contrast to the traditional coding processes. The proposed approach exploits the presence of data that is already replicated across the system and distributes the redundancy generation among those nodes that store part of this replicated data, which in turn reduces the overall amount of data transferred during the encoding process. By storing additional replicated blocks at nodes executing the distributed encoding tasks, the necessary network traffic for archiving can be further reduced. We analyze the problem using symbolic computation and show that the proposed decentralized encoding process can reduce the traffic by up to 56% for typical system configurations.

Keywords: archival, migration, erasure codes, distributed storage.

1 Introduction

Large data centers such as Google file-system (GFS) [9], Amazon S3 [2] or Hadoop file-system (HDFS) [3] handle extremely big volume of data by scaling-out, i.e., by realizing a distributed storage system comprising of hundreds or even thousands of commodity storage servers. To ensure that the stored data survives failures of some of the storage nodes, all data needs to be redundantly stored. A common and simple form of redundancy is to store multiple copies (replicas) of each data across the system. Storing erasure coded data is a more sophisticated alternative, which achieves significantly better trade-off in terms of storage-overhead and fault-tolerance [14,17]. Many recent systems such as Microsoft Azure [10], Facebook's HDFS-RAID [4,16] and the new version of the Google File System [8] among others have thus embraced erasure codes based storage systems. Typical parameter choices of erasure codes used in these deployed systems incur overall overhead of 1.3×–2× the size of the original data [4,7,10].

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 42–56, 2013.

This translates to a reduction of up to 50% in infrastructural costs with respect to a (3-way) replication based storage system.

Even though erasure coding based systems have significantly lower storage overhead, newly inserted data is usually first replicated across different storage nodes. The reasons for initially using replication are twofold. Replication is easy and fast to achieve by pipelining the data through the involved storage nodes,[1] without incurring any computation related costs or latency. This allows high throughput data insertion, achieving immediate fault tolerance. Furthermore, replication allows task schedulers to exploit data locality and achieve load-balancing [9], helping improve performance of applications manipulating the stored data.

Thus the use of erasure code based redundancy is primarily relegated to archival of data which is no longer frequently accessed [7]. Such a pragmatic design choice reduces the storage footprint significantly, and has immediate impact on the infrastructural and operational costs of a data center. However, this dual redundancy based approach, namely, the use of replication for newly inserted data and that of erasure codes for subsequent archival suffers from an important problem: all archived data passes through two independent redundancy generation processes, each of them having their own associated costs.

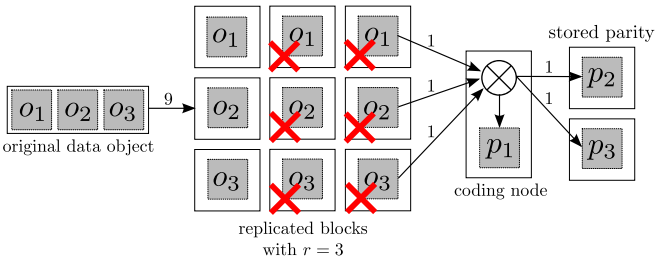
In this work, we explore a new decentralized erasure coding process leveraging on the existence of replicas from the first (data insertion) phase to reduce the network overheads during the second (migration from replication to erasure coded storage) phase. The basic idea is illustrated with a toy example shown in Figure 1. However, in order to amplify the reduction of the network traffic w.r.to the traditional archiving process, the replicas created during the insertion of new data have to be placed in some specific manner. Specifically, if multiple distinct blocks of a replicated object are collocated in a specific manner within the subset of nodes performing the decentralized encoding, then such locality can be exploited to further reduce the network traffic. Note that collocation of random object blocks may not be amenable to such benefits.

Symbolic computation based analysis show that for typical erasure code configurations used in in-production storage systems, the proposed decentralized erasure coding process can reduce the network traffic by up to 24% or 56%, depending on the collocation of the replicated blocks.

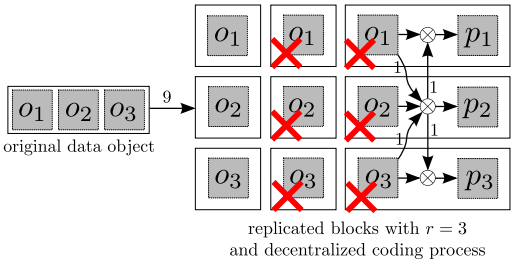
The main contributions of this paper are threefold.

1. We introduce a new *decentralized erasure coding process* to reduce the traffic required to archive data in replicated storage systems.
2. We provide a *generic code construction* that exploits this decentralized coding process.
3. We show how the traffic required during the decentralized coding process can by further reduced by adopting smart *replica placement strategies* during the data insertion phase.

[1] New inserted data can be stored in a first node while it is concurrently forwarded to and stored in a second node, and from this second node to a third, and so on [3,9].



(a) Traditional archiving process.



(b) Decentralized coding process.

Fig. 1. Example of how to generate erasure code redundancy from a replicated system using (a) a traditional encoding process, and (b) a decentralized coding process. White squares represent storage nodes and arrow labels denote the number of blocks transferred over the network. We can see how (a) requires a total of 5 network transfers while (b) only needs 4 network transfers. The "X" symbol denotes the replicas that are discarded once the archival finishes, and the symbol $\otimes$ denotes an encoding operation.

The rest of the paper is organized as follows. In Section 2 we provide some background on erasure codes for distributed storage systems. In Section 3 we present our new decentralized erasure coding process, and in Section 4 we evaluate its fault tolerance and its encoding traffic savings. In Section 5 we discuss some related works. Finally, in Section 6 we draw our conclusions and outline some planned future work.

2 Background

When a data object is stored for the first time in a distributed storage system such as GFS [9] or HDFS [3], it is initially split into blocks of size B and each of them is replicated over r different storage nodes (usually $r = 3$). The size of these blocks is set to relatively large values of B, e.g., $B = 64$MB in GFS and HDFS, which allow to amortize data access latencies, as well as to exploit local data caching [9].

At a later time, when an object does not need to be frequently accessed, it can be archived using an erasure code, reducing its storage footprint, and hence its associated storage costs. This encoding process takes k blocks of data, each of size B, and computes m parity blocks (or redundancy blocks) of the same

size, which are stored in m other different storage nodes. Since in most cases it is unlikely that data objects were split exactly into k blocks during the insertion process, the k blocks used in the encoding process might belong to different data objects. For example, in some systems files from the same directory are jointly encoded [5].

We can formally define the erasure encoding process as follows. Let the vector $\mathbf{o} = (o_1, \ldots, o_k)$ denote a data object of $k \times q$ bits. That is, each symbol o_i, $i = 1, \ldots, k$ is a string of q bits. Operations are typically performed using finite field arithmetic, that is, the two bits $\{0, 1\}$ are seen as forming the finite field $\mathbb{F}_2$ of two elements, while o_i, $i = 1, \ldots, k$ then belong to the binary extension field $\mathbb{F}_{2^q}$ containing 2^q elements. Then, the encoding of the object $\mathbf{o}$ is performed using an $(n \times k)$ generator matrix G such that $G \cdot \mathbf{o}^T = \mathbf{c}^T$, in order to obtain an n-dimensional codeword $\mathbf{c} = (c_1, \ldots, c_n)$ of size $n \times q$ bits. When the generator matrix G has the form $G = [I_k, G']^T$ where I_k is the identity matrix and G' is a $k \times m$ matrix, the codeword $\mathbf{c}$ becomes $\mathbf{c} = [\mathbf{o}, \mathbf{p}]$ where $\mathbf{o}$ is the original object, and $\mathbf{p}$ is a parity vector containing $m \times q$ parity bits. The code is then said to be systematic, in which case the k parts of the original object remain unaltered after the coding process. The data can then still be read without requiring a decoding process by accessing these systematic pieces.

Finally, an optimal erasure code in terms of the trade-off between storage overhead and fault tolerance is called a maximum distance separable (MDS) code, and has the property that the original object can be reconstructed from any k out of the total $n = k+m$ stored blocks, tolerating the loss of any arbitrary $m = n-k$ blocks. The notation "(n, k) code" is often used to emphasize the code parameters. Examples of the most widely used MDS codes are the Reed-Solomon codes [13].

While the efficacy of the use of erasure codes for fault tolerant storage has long been understood and leveraged, the migration process from a replication based storage to erasure coding based storage has been identified as a significant challenge relatively recently [5], given the tremendous growth in the volume of data that is continuously being generated and needs to be processed and archived. Next, we explain how the network traffic can be significantly reduced during the migration by embracing a decentralized coding process leveraging on the replicas.

3 Decentralizing the Data Archival Process

In this section we first introduce the decentralized erasure coding problem statement in 3.1. In 3.2 we provide a motivating example of how the decentralized archiving process works, and in 3.3 we provide a general code construction using a decentralized erasure coding process.

3.1 Problem Statement

When a data object $\mathbf{o} = (o_1, \ldots, o_k)$ is newly inserted in the system, each data block o_i is replicated across r different storage nodes. Once the object $\mathbf{o}$ is no longer in frequent use, it is archived using a systematic (n, k) erasure code. The

migration from replicas to encoding goes as follows: a node obtains an entire copy of $\mathbf{o}$ by downloading k different blocks, encodes them to generate a parity vector $\mathbf{p} = (p_1, \ldots, p_m)$, and finally uploads each parity block p_i to a different storage node. The encoding process thus requires the transfer of $n = k + m$ blocks, which can be reduced to $n - 1$ if the coding node keeps one of the parity blocks and uploads $m - 1$ blocks. Once the m parity blocks are stored, the number of replicas of each stored block o_i can be safely reduced to $r = 1$, discarding the remaining $r - 1$ block replicas. After this, a whole replica of the original object $\mathbf{o}$ remains unaltered and stored over k different storage nodes, becoming then the systematic part of the final codeword, which is formed of the blocks $c_1 = o_1, c_2 = o_2, \ldots, c_k = o_k$. An example of this process is depicted in Figure 1a for a simple (6,3) code.

The problem with this traditional coding process is that redundant data is transferred within the network twice (once to obtain and store kr replicated blocks, and once to compute and upload the m parity blocks) which might seem a waste of resources. Indeed, out of the kr replicated blocks generated during the first storage phase, only k of them are used to compute the coded parity blocks. This observation motivates the design of a new decentralized erasure coding process that reuses the original replicated data to reduce the total number of blocks transferred during the data archival process. In Figure 1b we showed a simple toy example of how to perform a decentralized encoding and save a block transfer as compared to the classical encoding process.

Iterative Encoding: Unfortunately, the simple decentralized encoding depicted in Figure 1b cannot be easily adopted by codes with large n and k values. To solve this problem we propose an iterative encoding process that splits the coding in ν different steps and involves up to m nodes that store some of the rk block replicas. The coding can be described in two logical phases: (i) at each step, a node generates a temporary redundant block and forwards it to the next node, and (ii) after ν steps each of the m nodes locally combines the stored replicas with the temporary blocks it received to generate and store one of the m parity blocks $p_1, \ldots, p_m$.

Replica Collocation: Traditionally, distributed storage systems allocate the rk replicated blocks of each data object among different nodes at random, which guarantees with high probability that the different replicated blocks are stored in rk different nodes. Random replica placement balances the amount of data stored per node and guarantees high resiliency in face of correlated node failures. However, in the case of the previous iterative coding process, having only one block replica per node increases the number of steps ν required to obtain an MDS erasure code. To minimize the number of steps required to achieve MDS codes, and thus minimize encoding traffic, we propose to collocate ℓ out of the total $(r-1)k$ unused block replicas within the m coding nodes. By doing so, the coding nodes will have more information about the original data and would be able to reduce the number of encoding steps.

For this decentralized erasure coding process to be relevant, the benefits that they provide in terms of network resources should not be at the expense of fault tolerance. We will show that the fault tolerance of the proposed decentralized erasure codes depends on the values of ℓ and ν: the larger these values are, the more likely it is to achieve the MDS property. However, large values of ℓ impose strict placement policies which might complicate load balancing, while large values of ν increase the number of transferred blocks and thus reduce the benefit in terms of communication costs. It is then important to understand the trade-off between these two parameters to find high fault tolerance codes (preferably MDS codes) requiring low communication costs during the archiving process and flexible initial replica placement policies.

The drawback of collocation of ℓ replicas within m storage nodes is that a high collocation might reduce the tolerance of the storage system to correlated node failures. For this reason it is very important to keep low collocation rates (small value of ℓ). We also note that a traditional erasure coding process can also exploit the presence of nodes storing multiple blocks from a single object, and thus reduce the number of blocks downloaded in order to obtain the whole data object $\mathbf{o}$. Thus, in Section 4.3 we compare the required traffic of traditional erasure encoding process and that of the decentralized coding process, and find that for the same amount of collocation, decentralized coding always achieve the same or less traffic than the traditional coding, demonstrating its efficacy. Furthermore, in the cases where both coding schemes require the same traffic, a decentralized coding is preferred over a centralized one since it avoids the network and computing bottlenecks of having a single coding node.

3.2 A Motivating Example

To understand the proposed decentralized erasure coding process, we first provide as example the encoding of a (10,6) erasure code, which provides $m = 10 - 6 = 4$ blocks of redundancy (parity blocks). We have a data object $\mathbf{o} = (o_1, o_2, \ldots, o_6)$ to be stored with a replica placement policy that stores $r = 3$ replicas of $\mathbf{o}$, that is three replicas of every o_i, $i = 1, \ldots, 6$ (for a total of 18 data blocks). We assume that one of the replicas of $\mathbf{o}$ is stored in $k = 6$ different nodes, which will finally constitute the systematic part of the codeword, $c_1 = o_1, \ldots, c_k = o_k$. From the $(r-1)k = 12$ replicas left, we select a subset of ℓ of them to be stored in the $m = 4$ coding nodes that will carry out the decentralized encoding process. The assignment of these ℓ replicas is as follows:

$$\mathcal{N}_1 = \{o_1, o_2, o_3\}$$
$$\mathcal{N}_2 = \{o_4, o_5, o_6\}$$
$$\mathcal{N}_3 = \{o_1, o_2\}$$
$$\mathcal{N}_4 = \{o_3, o_4\}$$

where $\mathcal{N}_j$ denotes the set of blocks stored in node j. Note that only $\ell = 10$ out of the available $(r-1)k = 12$ blocks are replicated in the m coding nodes, while the remaining two can be flexibly stored in other nodes, e.g., to balance the amount

of data stored per node. Note also that no node stores any repeated block, since this would reduce fault tolerance.

To describe the decentralized encoding process we use an iterative encoding process of $\nu = 7$ steps, in which every $\psi_i, \xi_j \in \mathbb{F}_{2^q}$ are predetermined values that define the actual code instance. During step 1, node 1 which has $\mathcal{N}_1$ generates

$$x_1 = o_1\psi_1 + o_2\psi_2 + o_3\psi_3$$

and sends it to node 2, which uses $\mathcal{N}_2$ and x_1 to compute

$$x_2 = o_4\psi_4 + o_5\psi_5 + o_6\psi_6 + x_1\psi_7$$

during step 2. After two more steps, we get:

$$x_3 = o_1\psi_8 + o_2\psi_9 + x_2\psi_{10}$$
$$x_4 = o_3\psi_{11} + o_4\psi_{12} + x_3\psi_{13},$$

and node 4 forwards x_4 to node 1, since $\nu = 7 > m = 4$, which creates

$$x_5 = o_1\psi_{14} + o_2\psi_{15} + o_3\psi_{16} + x_4\psi_{17}$$

before sending x_5 to node 2. For the last two iterations, both node 2 and node 3 use respectively $\mathcal{N}_2$, x_1 and x_5, and $\mathcal{N}_3$, x_2 and x_3 together, to compute

$$x_6 = o_4\psi_{18} + o_5\psi_{19} + o_6\psi_{20} + x_1\psi_{21} + x_5\psi_{22}$$
$$x_7 = o_1\psi_{23} + o_2\psi_{24} + x_2\psi_{25} + x_6\psi_{26}.$$

After this phase, node 1 to 4 are locally storing:

$$\mathcal{N}_1 = \{o_1, o_2, o_3, x_4\}$$
$$\mathcal{N}_2 = \{o_4, o_5, o_6, x_1, x_5\}$$
$$\mathcal{N}_3 = \{o_1, o_2, x_2, x_6\}$$
$$\mathcal{N}_4 = \{o_3, o_4, x_3, x_7\}$$

from which they compute the final m parity blocks:

$$p_1 = o_1\xi_1 + o_2\xi_2 + o_3\xi_3 + x_4\xi_4$$
$$p_2 = o_4\xi_5 + o_5\xi_6 + o_6\xi_7 + x_1\xi_8 + x_5\xi_9$$
$$p_3 = o_1\xi_{10} + o_2\xi_{11} + x_2\xi_{12} + x_6\xi_{13}$$
$$p_4 = o_3\xi_{14} + o_4\xi_{15} + x_3\xi_{16} + x_7\xi_{17}.$$

The final codeword is $\mathbf{c} = [\mathbf{o}, \mathbf{p}] = (o_1, \ldots, o_6, p_1, \ldots, p_4)$. There is a total of ν blocks transmitted during the encoding process (those forwarded during the iterative phase). In this example, $\nu = 7$, and the encoding process requires two block transmissions less than the classical encoding process, which requires $n - 1 = 9$ blocks, thus achieving a 22% reduction of the traffic.

We will in fact analytically show (see Section 4) that this decentralized encoding obtains a (10,6) MDS code. It means that there is a set of values for the coefficients ψ_i and ξ_i, which can be determined (for example, by exhaustive search), guaranteeing maximum fault tolerance of the code. In fact, the larger the field $\mathbb{F}_{2^q}$ is, the more likely it is to obtain MDS codes, allowing to use random ψ_i and ξ_i values in practical settings [1].

3.3 General Code Construction

We now provide a general technique to generate the parity vector $\mathbf{p} = (p_1, \ldots, p_m)$ in a decentralized manner by using m storage nodes that altogether store ℓ out of the total $(r-1)k$ block replicas.

We first define how the ℓ replicated blocks are allocated among the m coding nodes, i.e., the content of the set $\mathcal{N}_j$ for each node j. For the sake of simplicity, we assume that the ℓ replicas are deterministically assigned in a sequential manner as illustrated in the example used in 3.2, trying to even out the number of blocks assigned to each node. A formal description of this allocation is provided in Algorithm 1. Note that for practical small values of ℓ Algorithm 1 avoids the replication of the same block in a single node.

Algorithm 1. Replica placement policy

```
1: i ← 1
2: for j = 1, ..., m do
3:     α ← ⌊ℓ/m⌋
4:     if j ≤ (ℓ mod m) then
5:         α ← α + 1
6:     end if
7:     N_j = {o_l : l = (j mod k), j = i, ..., i + α}
8:     i ← i + α
9: end for
```

This assignment policy imposes some restrictions on the location of the different replicated blocks. These restrictions might become a drawback in systems trying to uniformly distribute the storage load among all nodes in the system. The problem can be specially important in the extreme case when $\ell = (r-1)k$, where all replicas need to be specifically stored in the m coding blocks. However, smaller values of ℓ provide some flexibility on where to assign the $(r-1)k - \ell$ remaining replicas. We will explore the effects that different ℓ values have in the fault tolerance of the erasure code in Section 4.

Remark 1. In the case of $\ell = k$, there is no replica assignment policy and a random placement can be used.

Given the previous replica assignment policy, the decentralized encoding process is split in two different phases: the *iterative encoding* and the *local encoding*.

The iterative encoding consists of ν sequential encoding steps, where at each step, each node generates and forwards a temporary redundant block. For each step i, where $i = 1, \ldots, \nu$, node $j = (i \bmod m)$ which stores the set of blocks $\mathcal{N}_j = \{z_1, z_2, \ldots\}$ locally computes a temporary block $x_i \in \mathbb{F}_{2^q}$ as follows:

$$x_i = z_1\psi_1 + z_2\psi_2 + \cdots + z_{|\mathcal{N}_j|}\psi_{|\mathcal{N}_j|}, \tag{1}$$

where $\psi_i \in \mathbb{F}_{2^q}$ are predetermined values. Once x_i is computed, node j sends x_i to the next node $l = (i+1 \bmod m)$, who stores locally the new temporary block:

$\mathcal{N}_l = \mathcal{N}_l \cup \{x_i\}$. After that, node l computes x_{i+1} as defined in (1) and forwards it the next node. The iterative process is similarly repeated a total of ν times.

After this iterative encoding phase, each node $i = 1, \ldots, m$ executes a local encoding process where the stored blocks $\mathcal{N}_i$ (including the temporary blocks generated during the iterative encoding phase) are combined to generate the final parity block p_i (for predetermined values of $\xi_i \in \mathbb{F}_{2^q}$) as follows:

$$p_i = z_1\xi_1 + z_2\xi_2 + \cdots + z_{|\mathcal{N}_i|}\xi_{|\mathcal{N}_i|}. \tag{2}$$

Finally, we describe the overall distributed encoding algorithm (including the iterative encoding and the local encoding) in Algorithm 2. Note that values ψ_l and ξ_l (lines 7 and 17) are picked at random. In a sufficiently large field (e.g., when $q = 16$) this random choice will not introduce additional dependencies other than the ones introduced by the iterative encoding process itself [1].

Algorithm 2. Decentralized redundancy generation

```
1: l ← 1
2: j ← 1
3: x ← 0
4: for i = 1, ..., ν do                    ▶ Generation of the ν temporary blocks.
5:     x ← 0
6:     for z ∈ N_j do                      ▶ Coding operation as described in (1).
7:         x ← x + ψ_l · z
8:         l ← l + 1
9:     end for
10:    j ← (i + 1) mod m
11:    N_j ← N_j ∪ {x}                     ▶ Each union (∪) represents a block transfer.
12: end for
13: l ← 1
14: for i = 1, ..., m do                   ▶ Generation of the final m parity blocks.
15:    p_i ← 0
16:    for x ∈ N_i do                      ▶ Coding operation as described in (2).
17:        p_i ← p_i + ξ_l · x
18:        l ← l + 1
19:    end for
20: end for
```

4 Evaluation

In this section we evaluate the effects that the number of collocated replicas, ℓ, and the number of steps, ν, have in the fault tolerance of the code obtained by the decentralized coding strategy. To do it, we symbolically compute different iterated codings as it is specified in Algorithm 2. We do it for different values of n, k, ℓ and ν, however, the block values o_i and the value of coefficients ψ_i and ξ_i are not specified, which means that after the iterative coding phase we obtain a symbolic codeword $\mathbf{c} = (c_1, \ldots, c_n)$. We use this codeword then to enumerate

all the possible $\binom{n}{k}$ k-subsets of blocks in $\mathbf{c}$ and measuring how many of them contain k linearly independent blocks. We refer to the fault tolerance of the code, π, as the percentage of k-subsets that do not contain linear dependencies between its blocks. When all the $\binom{n}{k}$ k-subsets are free of linear dependencies we say that the code is MDS, and has maximum fault tolerance, i.e., $\pi = 1$.

Since we aim to evaluate the effects of the parameters ℓ and ν, we select three different (n,k) instances commonly used by in-production distributed storage systems: (i) a (6,3) code, suggested in the new Google FS [8], (ii) a (10,6) code used in the Microsoft Azure Storage service [10], and (iii) a (14,10) code used in Facebook's HDFS-RAID implementation [4,15]. Respectively, these codes have storage footprints of $2\times$, $1.\dot{6}\times$ and $1.4\times$ the size of the original stored data, which represents a diverse spectrum of code parameters. For each of them we evaluate the fault tolerance of a code generated with a decentralized erasure code that uses ℓ collocated blocks and ν coding steps, for different values of ℓ and ν.

4.1 Fault Tolerance Analysis

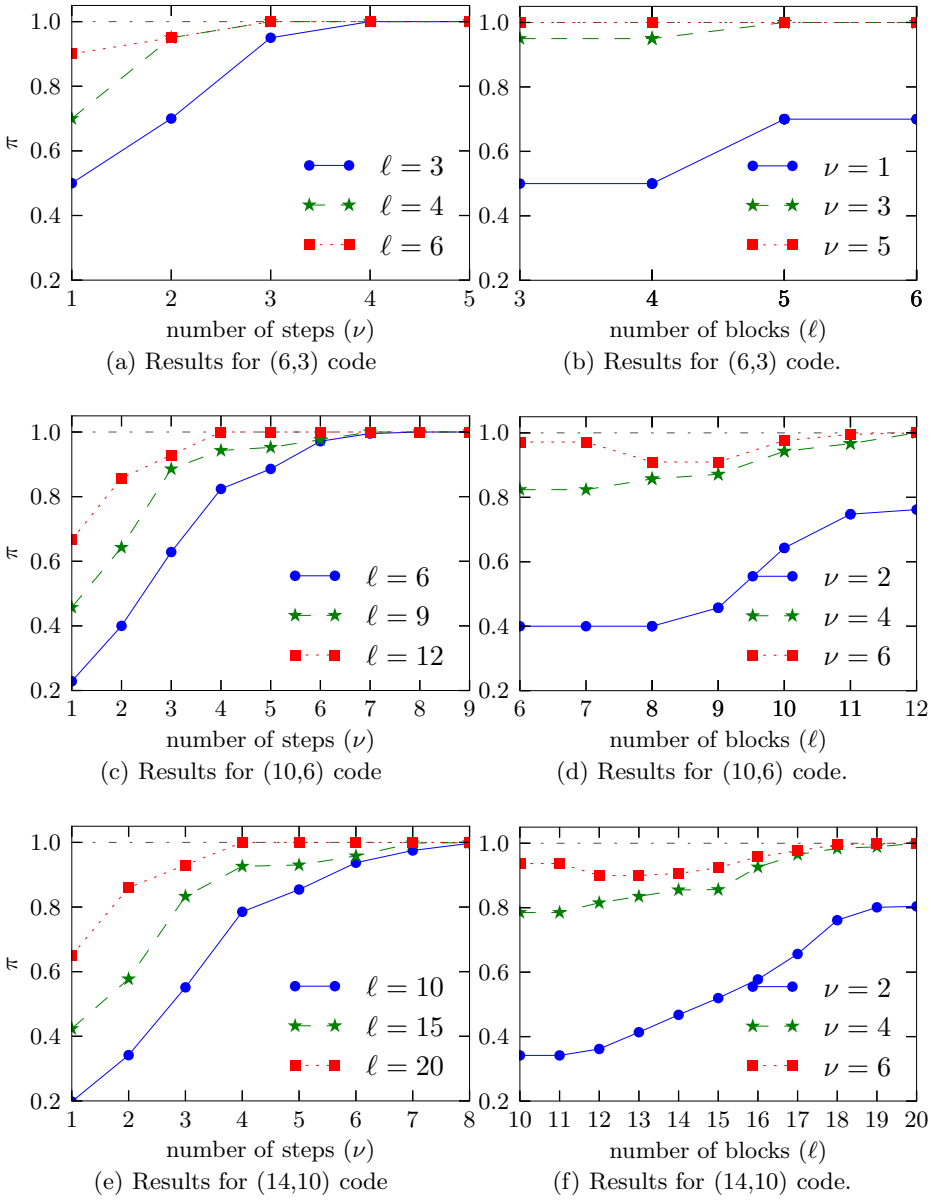
We divide the fault tolerance analysis in two experiments, one aiming to evaluate the effects of ν, and another one to evaluate the effects of ℓ.

In figures 2a, 2c, and 2e we show the fault tolerance of the code π as a function of the number of steps, ν. For each of the three different codes we depict the effects of ν for three different values of ℓ: $\ell = k$, $\ell = 1.5$ and $\ell = 2k$. We can see how the proportion of linearly independent k-subsets increases as more encoding iterations are executed. Achieving the maximum fault tolerance (when the fraction of linearly independent k-subsets is one) requires less iterations for high replica collocation values ℓ.

Similarly, in figures 2b, 2d, and 2f we show the fault tolerance as a function of the number of blocks stored within the m coding nodes, ℓ. For each code we also show the results for three different values of ν, which aim to show the fault tolerance when all (i) only a few coding nodes execute the iterative encoding process, (ii) when all coding nodes execute it exactly once, and (iii) when some coding nodes execute it more than once. In general we can see how increasing the number of initially collocated replicas ℓ increases the fault tolerance of the code. However, for small values of ν there are cases where increasing ℓ might slightly reduce the fault tolerance. Finally, we want to note that in those cases where $\nu \leq m$ (only a few coding nodes execute the iterative encoding), the code produced by the decentralized coding can never achieve maximum fault tolerance. To achieve maximum fault tolerance all the m coding nodes need to execute at least one coding step.

4.2 Obtaining an (6,3) MDS Code

We propose a simple example to understand why the iterative encoding process allows to obtain MDS codes. Suppose we have a (6,3) erasure code, whose codewords are of the general form



(a) Results for (6,3) code
(b) Results for (6,3) code.
(c) Results for (10,6) code
(d) Results for (10,6) code.
(e) Results for (14,10) code
(f) Results for (14,10) code.

Fig. 2. Fault tolerance achieved by our decentralized erasure coding process as a function of the number of encoding steps, ν, and the number of co-located block replicas, ℓ. The fault tolerance π is expressed as the proportion of k-subsets of the codeword $\mathbf{c}$ that do not contain linear dependencies. When this value is one, the code is MDS and has maximum fault tolerance.

$$\mathbf{c} = (o_1,\ o_2,\ o_3,\ \alpha_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3,\ \beta_1 o_1 + \beta_2 o_2 + \beta_3 o_3,\ \gamma_1 o_1 + \gamma_2 o_2 + \gamma_3 o_3)$$

for some fixed $\alpha_i, \beta_i, \gamma_i \in \mathbb{F}_{2^q}$, $i = 1, 2, 3$, where $\mathbf{o} = (o_1, o_2, o_3)$ is the object to be encoded. We assume that every α_i, β_i, γ_i is nonzero, so that it is invertible. Note that if any of them were to be zero, then the code cannot be MDS.

Let us assume a replica placement policy using $\ell = 3$ that allocates these ℓ replicas within the m coding blocks as follows:

$$\mathcal{N}_1 = \{o_1\},\ \ \mathcal{N}_2 = \{o_2\},\ \ \mathcal{N}_3 = \{o_3\}.$$

Then, an iterative encoding process of $\nu = 4$ steps allows to compute the generic parity blocks as given above:

1. Node 1 sends o_1 to node 2,
2. Node 2 uses o_2 and $x_1 = o_1$ to compute $x_2 = o_1\gamma_1 + o_2\gamma_2$.
3. Node 3 receives x_2 and sends $x_3 = \gamma_2^{-1}\alpha_2 x_2 + \alpha_3 o_3 = \gamma_2^{-1}\alpha_2\gamma_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3$.
4. Node 1 forwards x_3 to node 2.

After this phase, node 1 to 3 are locally storing:

$$\begin{aligned}
\mathcal{N}_1 &= \left\{o_1, x_3 = \gamma_2^{-1}\alpha_2\gamma_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3\right\}\\
\mathcal{N}_2 &= \left\{o_1, o_2, x_3 = \gamma_2^{-1}\alpha_2\gamma_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3\right\}\\
\mathcal{N}_3 &= \{o_3, x_2 = o_1\gamma_1 + o_2\gamma_2\}
\end{aligned}$$

from which they compute the final m parity blocks:

$$\begin{aligned}
p_1 &= o_1(\gamma_2^{-1}\alpha_2\gamma_1 + \alpha_1) + x_3\\
p_2 &= \alpha_3^{-1}\beta_3 x_3 + (\alpha_3^{-1}\beta_3\gamma_2^{-1}\alpha_2\gamma_1 + \beta_1)o_1 + (\alpha_3^{-1}\beta_3\alpha_2 + \beta_2)o_2\\
p_3 &= \gamma_3 o_3 + x_2.
\end{aligned}$$

The final codeword is $\mathbf{c} = [\mathbf{o}, \mathbf{p}] = (o_1, o_2, o_3, p_1, p_2, p_3)$. Thus any (6,3) MDS code can be obtained through this iterative encoding.

4.3 Reduction of the Encoding Traffic

Finally, we aim to evaluate the traffic savings that the decentralized erasure coding provides on the data archival process. For that we take the results presented in Figure 2 and measure the encoding traffic of the MDS codes obtained when $\ell = k$ and $\ell = 2k$. For each ℓ value the decentralized encoding traffic corresponds to the minimum value of ν required to achieve the MDS property. Additionally, for the same value of collocated replicas ℓ we also evaluate the encoding traffic that a traditional erasure would require, considering that the coding node stores more than one object block.

In Figure 3 we depict the encoding traffic comparison between a centralized coding process, denoted by RS[2], and a decentralized MDS coding process, denoted by DE. In the case of the (6,3) code, there are only traffic savings when

[2] The acronym refers to the classical Reed-Solomon coding process.

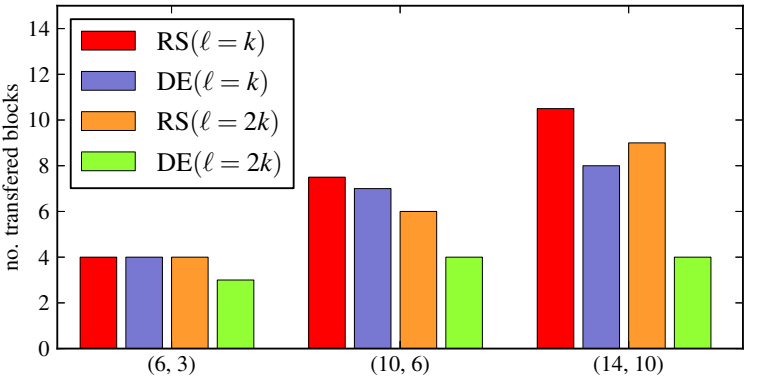


Fig. 3. Comparison of the number of transferred blocks during the encoding of a classical Reed-Solomon code (RS) and the decentralized coding (DE) for two different replica collocation values: $\ell = k$ and $\ell = 2k$. All DE codes are MDS codes optimized to minimize the number of coding steps ν.

the $m = 3$ coding nodes store all the $(r-1)k$ replicas. In this case the decentralized coding saves one block transfer. In the case of the (10,6) the decentralized coding process always requires less network traffic, even for low replica collocation levels, and these traffic savings are amplified for the (14,10) code. In this last case the savings range from a 24% in the case of the low replica collocation ($\ell = k$), up to 56% for high collocation values ($\ell = 2k$).

5 Related Work

Despite widespread use of erasure coding for archiving data in distributed storage systems, the study of the actual migration process from replication based redundancy to erasure code based redundancy is in its infancy.

The most relevant related work is that of Fan et al. [5], who propose to distribute the task of erasure coding using the Hadoop infrastructure, as MapReduce tasks. Any individual object is however encoded at a single node, and hence the parallelism achieved in their approach is only at the granularity of individual data objects. Besides, their data archiving process relies on the traditional erasure code redundancy generation process, which does not exploit previously existing replicas.

In [12] we recently proposed RapidRAID codes, a family of pipelined erasure code that aim to speedup the archival process in distributed storage systems. Although such fast data archival is also achieved by a decentralized coding process, the RapidRAID approach differs fundamentally from the one presented in this paper in two ways. First, RapidRAID codes are non-systematic codes that require decoding operations to read the archived data, complicating the access to archived data. Second, RapidRAID codes are aimed to reduce the encoding

time, but achieves no reduction of traffic as compared to what is required by a traditional coding processes. The presented work in contrast is specifically aimed at traffic reduction, and any collateral benefits in terms of speed-up of the process is left for future study.

Finally, a somewhat unrelated line of work worth mentioning are codes designed for storage in sensor networks [6, 11]. However, in such a setting, the (disjoint) data generated by k sensors is jointly stored over $n > k$ storage sensors based on erasure coding redundancy. This is achieved using network coding techniques by creating random linear combinations of the already distributed data. Such a technique has however not been explored for the specific migration problem studied in this work.

6 Conclusions

In this paper we introduce a new decentralized erasure coding process to reduce the network traffic required to archive replicated data in distributed storage systems. The decentralized process distributes the coding tasks among those nodes storing data block replicas of the object to be archived. These nodes collaboratively generate and store all the parity data.

Additionally, we provide a formal definition of the decentralized erasure coding process and symbolically analyze the fault tolerance of the obtained codes for different parameters. We show that in already deployed systems where the placement of the replicated data can not be changed, our decentralized coding process can reduce the redundancy generation traffic by 20% upto 38% for typical code configurations used in current systems. However, when the replica placement of newly inserted data can be manipulated to co-locate more block replicas in some specific manner in the nodes participating in the coding process, the redundancy generation traffic can be reduced by 40% upto 70%.

The design of decentralized erasure coding process to archive replicated data is a relatively unexplored problem that needs to be further studied. We identify two problems that we plan to address in future works. Specifically, we aim to look methodically at the effects that different replica placement policies have on the network traffic required during the archiving process. We also want to generalize the idea to decentralize the redundancy generation of existing standard erasure codes such as Reed-Solomon codes. Ultimately, we want to develop a holistic theory, which explores any possible trade-offs between the network traffic generated by and the speed of the archival process, subject to various other system design choices such as the initial replica placement.

Acknowledgments. We would like to thank Dr. Punarbasu Purkayastha for his valuable comments on this research. L. Pamies-Juarez and F. Oggier's research is supported by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03. A. Datta's work is supported by A*Star TSRP grant number 1021580038 and NTU/MoE Tier-1 grant number RG 29/09.

References

1. Acedański, S., Deb, S., Médard, M., Koetter, R.: How good is random linear coding based distributed networked storage. In: Workshop on Network Coding, Theory, and Applications, NetCod (2005)
2. Amazon.com. Amazon S3, http://aws.amazon.com/s3
3. Apache.org. HDFS, http://hadoop.apache.org/hdfs/
4. Apache.org. HDFS-RAID, http://wiki.apache.org/hadoop/HDFS-RAID
5. Fan, B., Tantisiriroj, W., Xiao, L., Gibson, G.: DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing. Technical Report Technical Report CMU-PDL-11-112, Carnegie Mellon Univsersity, Parallel Data Laboratory (2011)
6. Dimakis, A., Prabhakaran, V., Ramchandran, K.: Decentralized erasure codes for distributed networked storage. IEEE/ACM Transactions on Networking 14 (2006)
7. Fan, B., Tantisiriroj, W., Xiao, L., Gibson, G.: DiskReduce: RAID for Data-Intensive Scalable Computing. In: The 4th Annual Workshop on Petascale Data Storage, PDSW (2009)
8. Ford, D., Labelle, F., Popovici, F.I., Stokely, M., Truong, V.A., Barroso, L., Grimes, C., Quinlan, S.: Availability in Globally Distributed Storage Systems. In: The 9th USENIX Conference on Operating Systems Design and Implementation, OSDI (2010)
9. Ghemawat, S., Gobioff, H., Leung, S.: The Google File System. In: Proceedings of the ACM Symposium on Operating Systems Principles, SOSP (2003)
10. Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., Yekhanin, S.: Erasure Coding in Windows Azure Storage. In: Proceedings of the USENIX Annual Technical Conference, ATC (2012)
11. Kamra, A., Misra, V., Feldman, J., Rubenstein, D.: Growth codes: maximizing sensor network data persistence. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM (2006)
12. Pamies-Juarez, L., Datta, A., Oggier, F.E.: RapidRAID: Pipelined Erasure Codes for Fast Data Archival in Distributed Storage Systems. CoRR, abs/1207.6744 (2012)
13. Reed, I., Solomon, G.: Polynomial Codes Over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics 8(2), 300–304 (1960)
14. Rodrigues, R., Zhou, T.H.: High Availability in DHTs: Erasure Coding vs. Replication. In: van Renesse, R. (ed.) IPTPS 2005. LNCS, vol. 3640, pp. 226–239. Springer, Heidelberg (2005)
15. Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A.G., Vadali, R., Chen, S., Borthakur, D.: Novel Codes for Cloud Storage (2012), https://mhi.usc.edu/files/2012/04/Sathiamoorthy-Maheswaran.pdf
16. Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., Liu, H.: Data warehousing and analytics infrastructure at facebook. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010 (2010)
17. Weatherspoon, H., Kubiatowicz, J.D.: Erasure Coding Vs. Replication: A Quantitative Comparison. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 328–337. Springer, Heidelberg (2002)

Transport Protocol with Acknowledgement-Assisted Storage Management for Intermittently Connected Wireless Sensor Networks

Ying Li, Radim Bartos, and James Swan

Department of Computer Science
University of New Hampshire
Durham, NH 03824
{yws2,rbartos,jmswan}@cs.unh.edu
http://www.cs.unh.edu

Abstract. The benefits of hop-by-hop transport protocols and in-network storage in intermittently connected networks are well known. However, due to the extreme limitation on the storage capability of wireless sensor networks (WSNs), the hop-by-hop transport protocols that are based on in-network storage without storage management are inappropriate to apply directly to intermittently connected WSNs. The lack of storage management leads to mote storage overflow when using in-network storage, which in turn degrades the performance of the network.

In this paper, we propose a hop-by-hop transport protocol that provides not only end-to-end reliability and congestion control but also an innovative storage management mechanism. The proposed protocol enhances the network delivery rate without sacrificing the delivery speed even in high contention scenarios.

Keywords: WSNs, transport protocol, storage management, intermittent connection.

1 Introduction

Wireless sensor networks (WSNs) have been extensively studied in the last decade, not only because of the wide range of applications but also the challenges involved in this kind of network. It is well known that the motes used in WSNs are usually limited in energy, storage, computation capability and communication range [1,21]. In addition, communication contention and collisions easily happen in such wireless environments. Moreover, signal attenuation can make the network intermittently connected, in which case there might be periods when there is no path from the source to the destination.

In such a wireless scenario, it is inappropriate to directly use the traditional reliable transport protocol TCP [4]. In TCP, packet loss is a sign of network congestion [18], which is not always true in wireless communication. In WSNs,

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 57–71, 2013.

many factors other than network congestion can cause packet loss, such as motes running out of memory, energy exhaustion and signal attenuation. Therefore, the congestion control mechanism of TCP fails to determine the traffic problem correctly, causing the protocol to perform poorly. Moreover, TCP is mainly operated on the sender side. In WSNs, motes have limited energy supplies that could be drained quickly due to heavy communication.

In intermittently connected networks, end-to-end congestion control and loss recovery are less efficient than in connected networks. In contrast, hop-by-hop congestion control and loss recovery allow for a more immediate reaction after a traffic problem is detected. In-network storage can reduce the number of end-to-end retransmissions when using hop-by-hop transport protocols. Both delay-tolerant networks (DTN) [2] and multi-hop wireless communication [14] have realized the benefits of hop-by-hop transport and in-network storage. The framework of traditional DTNs can be thought of as an overlay network on top of the Internet [3]. The DTN *bundle layer* between the application layer and transport layer provides a persistent storage for custody transfer [2]. In the cache and forward (CNF) architecture, both CNF routers and cache and carry (CNC) routers have persistent storages. Unlike DTN and CNF, the storage capability of motes in WSNs is very limited. Typically the amount of the storage in one mote is less than 512 KB. Table 1 lists the storage capability of several motes available today [12]. From the table, we can see that storage management is necessary for hop-by-hop transfer and in-network storage in WSNs. Although there are many routing protocols providing storage management such as CHARON [15], RED&FED [20] and TBR [13], they do not guarantee transmission reliability.

Table 1. Mote's extra nonvolatile storage size

Mote	Storage size(KB)
WeC	32
Renè	32
Renè2	32
Dot	32
Mica	512
Mica2Dot	512
Mica2	512
Telos	128

In this paper, we propose a hop-by-hop transport protocol, Transport Protocol with **Ack**nowledgement-as**sis**ted Storage Management *(Acksis)*. Acksis provides an innovative storage management feature in order to enhance the network delivery rate without losing delivery speed. We disable the link-layer ACKs in Acksis in order to reduce energy spent on overhead. We also take advantage of in-network storage and overhearing to reduce the number of end-to-end retransmissions when packet loss happens.

The rest of the paper is organized as follows: in Section 2 we study several existing transport protocols. Section 3 describes Acksis in detail. In Section 4 we explain the simulator set up and analyze the performance. The conclusion is drawn in Section 5.

2 Related Works

STCP [4], ART [17], Flush [5], RCRT [10] and Hop [8] are transport protocols that have been designed in recent years to provide both end-to-end reliability and congestion control. STCP and RCRT implement end-to-end congestion control, while ART, Flush and Hop implement hop-by-hop congestion control. Hop-by-hop congestion control is more appropriate for intermittently connected networks than end-to-end congestion control, since end-to-end congestion control is less effective when there typically is no contemporaneous path in a network. However, the end-to-end reliability of STCP, Flush and RCRT relies on the source to provide loss recovery, which is not effective in intermittently connected scenarios.

Hop-by-hop reliability takes advantage of in-network storage such that loss recovery can happen through the use of cached data in the intermediate motes instead of only relying on the source to retransmit. This is more efficient and less expensive than relying on the source to provide loss recovery. However, in WSNs the mote storage capability is limited which makes the in-network storage less desirable than in networks where there is plenty of storage. GARUDA [11], PSFQ [19] and ART provide packet level reliability in a hop-by-hop manner but only for downstream data. The motivation is that the amount of data sent from the sink to other motes is small, so storage management is not a significant problem. Hop, DTNS [9], RMST [16] and RBC [22] provide upstream transport reliability, but are more suitable for networks where storage capability is not a problem.

Table 2 provides an attribute comparison of the mentioned transport protocols. It is easily to see that Hop is the only protocol that provides packet-level, end-to-end reliability in both directions and hop-by-hop congestion control. In Hop, data is packed into blocks, which are forwarded to the next hop through backpressure routing. In-network storage is used by Hop to achieve end-to-end reliability.

In-network storage and hop-by-hop transport resolve the problems of reliability and congestion control in intermittently connected networks more efficiently than end-to-end recovery and congestion control. The storage limitation of motes makes existing protocols, such as Hop, less attractive for intermittently connected WSNs. Storing every packet that is received or overheard until packets out of date works under the condition that traffic is light, contention and collisions are infrequent, noise is low, and the storage capability is not a bottleneck. If traffic load is high, contention and collision happen frequently or transmission is lossy, motes risk encountering storage overflow, making end-to-end reliability difficult to guarantee.

Table 2. Attribute comparison of transport protocols mentioned in Section 2

Protocols	Reliability			Congestion Control	Storage Management
	Type	Level	Direction	Type	
STCP	end-to-end	custom	upstream	end-to-end	clean buffer for continuous flows
Flush	end-to-end	packet	upstream	hop-by-hop	N/A
RCRT	end-to-end	packet	upstream	end-to-end	N/A
ART	hop-by-hop	packet (downstream)/event (upstream)	both	hop-by-hop	N/A
Hop	hop-by-hop	packet	both	hop-by-hop	N/A
GARUDA	hop-by-hop	packet	downstream	N/A	N/A
PAFQ	hop-by-hop	packet	downstream	N/A	N/A
DTNS	hop-by-hop	packet	upstream	N/A	clean buffer, no control on buffer size
RMST	hop-by-hop	packet	upstream	N/A	N/A
RBC	hop-by-hop	packet	upstream	N/A	N/A

3 Transport Protocol with Acknowledgement-Assisted Storage Management

Hop-by-hop transport and in-network storage are suitable for intermittently connected WSNs. Hop-by-hop congestion control can react rapidly after congestion is detected. In-network storage can significantly reduce the number of end-to-end recoveries under the condition that the storage is plentiful. In WSNs, this assumption is not appropriate, since the storage capability of motes is very limited. Moreover, because of the requirement for low cost motes, to add extra storages to motes is not an optimal solution. Hence, storage limitations must be taken into consideration for in-network storage in WSNs. Otherwise, the network could be overloaded, and its performance will be degraded.

Acksis implements end-to-end reliability and congestion control in a hop-by-hop manner. It employs overhearing and in-network storage to reduce end-to-end retransmissions when packet loss happens. It also takes advantage of backpressure and acknowledgement messages to achieve congestion control and to manage mote storage in order to minimize storage overflow and increase delivery rate without sacrificing delivery speed.

3.1 Protocol Overview

In Acksis, packets are grouped into blocks to send. The block size is pre-calculated through the number of motes in a network and the minimum storage size of motes

in the network, which will be explained in Subsection 3.3. An example of Acksis communication is shown in Figure 1. Figure 2 is the state diagram of storage management of Acksis.

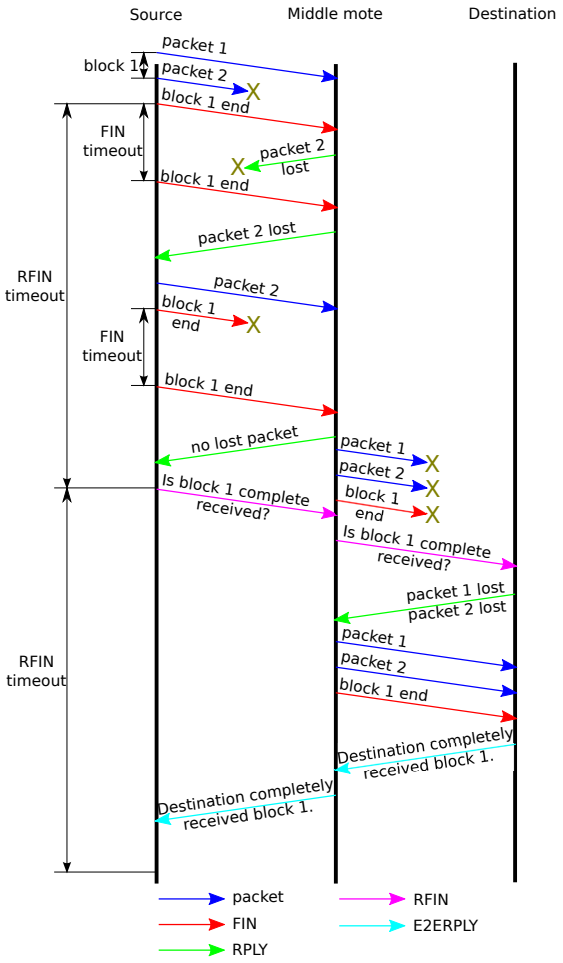


Fig. 1. One example of Acksis communication

There are four acknowledgement messages in Acksis:

FIN: sent by a block B's sender to B's receiver after the last packet of B is sent, to indicate the end of the block to the receiver.

RPLY: sent by B's receiver to B's sender after the receiver receives the FIN of B to guarantee hop-by-hop reliability. If there is packet loss, indicate the lost packet number in the message.

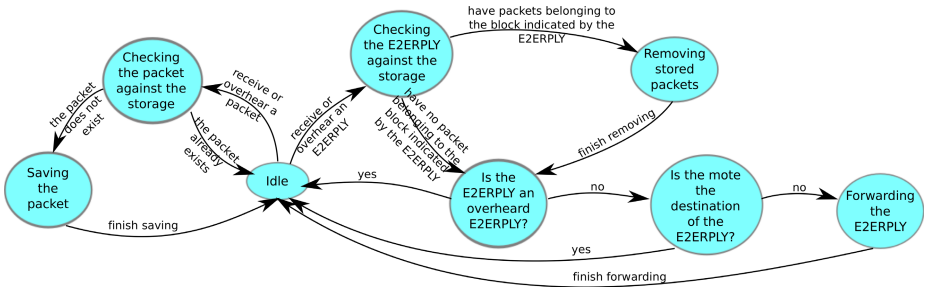


Fig. 2. State diagram of storage management of Acksis

E2ERPLY: sent by B's destination to B's source after B has been received completely to notify that B has been received by the destination successfully. Motes execute storage management when they receive or overhear an E2ERPLY message, which will be interpreted in Subsection 3.4.

RFIN: sent by B's source if it does not receive E2ERPLY of B after a certain period of time to guarantee end-to-end reliability.

Figure 1 shows an example packet exchange between motes using Acksis. During this communication process, motes execute storage management when they receive or overhear a packet or E2ERPLY message according to the state machine in Figure 2. In the rest of this subsection, we outline how Acksis works.

Initially, a sender sends a block B to a receiver followed by a FIN message. The receiver stores packets of B that have not been cached in its storage while increasing its backlog value by one per packet. Any mote that overhears the transmission will store overheard packets that are not already stored. After receiving the FIN message of B, the receiver checks for lost packets in B before sending an RPLY message to the sender. When the sender receives the RPLY message, it retransmits any lost packets. This process continues until the receiver completely receives B. If there are no lost packets, the sender decreases its backlog value by the block size.

If several motes send FIN messages to the same receiver simultaneously, then the receiver serves the first arriving FIN message. A mote does not acknowledge more FIN messages until it forwards one more block to a downstream mote. If there is no RPLY message received before timeout, the sender resends the FIN message. The sender cannot send more blocks if the current one has not been completely received by the receiver.

When B reaches its destination, the destination sends out an E2ERPLY message to B's source. When an intermediate mote receives the E2ERPLY message, it deletes the cached B from its storage, and forwards the E2ERPLY message to the next hop. If the E2ERPLY message reaches B's source before timeout, the source deletes B from its storage and is ready to send the next block. Otherwise, the source sends an RFIN message. Motes that overhear the E2ERPLY message remove B from their storage, but do not forwarded the E2ERPLY message further.

When a mote receives the RFIN message for B, it searches the storage to see whether all packets within B are cached. If there are missing packets, the mote sends an RPLY message to the sender of the RFIN message to ask for a retransmission. The process continues until the mote receives the whole block B. It then forwards the RFIN message to the next hop. If there are no missing packets, the RFIN message is forwarded to the next mote directly. If the mote is the destination, an E2ERPLY message for B will be sent out.

The main differences between Acksis and Hop [8]:

- In Acksis, the block size is pre-calculated through the number of motes in a network and the minimum storage size of motes in the network, which will be illustrated in Subsection 3.3.
- In Acksis, motes take advantage of the received and overheard E2ERPLY message to manage the storage.

3.2 Congestion Control and Flow Control

Acksis employs three mechanisms to provide congestion control in a hop-by-hop manner.

1. **Backpressure.** Each mote maintains a backlog table of its neighbors' queue lengths, and periodically sends its queue length to its neighbors to update their backlog tables. When a mote receives one uncached packet, its queue length is increased by one. After completely forwarding a block to the next hop, its queue length is decreased by the size of the block. The next hop is the neighbor with the shortest queue length.
2. **RPLY block.** A mote cannot send more blocks if the current one has not been completely received by the receiver.
3. **FIN constrain.** A mote replies to one FIN message in a first come first serve manner if it receives more than one simultaneously.

Backpressure is a method to signal the congestion level. The longer a neighbor's queue length is, the more congested the neighbor is. Leveraging of backpressure can alleviate the congestion by directing traffic to the paths with lighter traffic load. Moreover, by picking routing paths through backpressure, a mote can react to the congestion rapidly, which is more appropriate to intermittently connected networks. The **RPLY block** and **FIN constrain** can alleviate the congestion by withholding sending blocks until the congestion is relieved.

In Acksis, the flow control is achieved through **E2ERPLY lock** ensuring that a source cannot inject more blocks into the network until it receives the E2ERPLY message for the block it just sent. There are two reasons leading to the source failing to receive the E2ERPLY before the timeout: network congestion and signal attenuation. Both are valid reasons not to inject new blocks into the network. Waiting for an E2ERPLY message before injecting new blocks can help to relieve the overall network congestion.

3.3 Block Size Calculation

In this study, the block size in Acksis is determined on the basis of the number of motes in the network and the minimum storage size of motes. The size is pre-calculated and shipped with motes when the network is deployed initially. The block size equals the floor of the minimum storage size of motes divided by the number of motes in the network:

$$BlockSize = \left\lfloor \frac{Min(StorageSize(i))}{count(S)} \right\rfloor, \forall i \in S \quad (1)$$

where S is the set of motes in the network, and i is an individual mote.

The idea of the block size calculation is based on a worst case consideration in this study for simplicity, but do not exclude other methods in future work. A mote can only receive or overhear transmissions from one-hop neighbors, since only two motes can be in the same communication range in this pre-planned network. Due to E2ERPLY lock, each source can inject one block into the network before it receives an E2ERPLY for the block it sent. The worst case is that a mote needs to cache all blocks from the rest of motes in the network. If there is not enough space available, the storage of the mote will overflow causing unnecessary retransmissions. Equation 1 reduces the probability of overflow by letting each mote reserve space for the transmissions from other motes in the network.

3.4 Storage Management

The goal of Acksis storage management is to save storage space for valuable packets without causing unnecessary retransmissions, and control storage size to minimize overflow. Simply saving every packet a mote received or overheard may cause the storage overflow resulting in packet loss. Simply dropping the oldest packets can also cause unnecessary retransmissions. Without storage size control, blindly transmitting packets to motes without sufficient available space causes overflow and wasted energy. All of these negatively affect the network performance.

Acksis solves the storage management problem through three mechanisms:

1. **Block size calculation.** This is explained in Subsection 3.3.
2. **Duplicate checking.** Checking the storage before caching a packet. If the packet has not been cached yet, it is cached. Otherwise, it is dropped.
3. **Buffer cleaning.** Checking the storage after receiving or overhearing an E2ERPLY message. If there are packets in the block indicated by the E2ERPLY message cached in the storage, they are deleted.

Mechanism 1 avoids transmitting packets to motes without sufficient free storage in a worst case scenario, which is elaborated in detail in the Subsection 3.3. Mechanism 2 saves space by avoiding repeatedly caching packets, which greatly reduces the opportunity of storage overflow. Mechanism 3 deletes the packets

that already arrive at the destination from storage in order to save spaces for valuable packets. Together these mechanisms significantly reduce the packet loss caused by storage overflow.

4 Simulator and Experiments

4.1 Simulator and Experiment Setup

TinyOS [6] is a popular operation system designed for low-power wireless devices such as motes used in WSNs. TOSSIM [7] is a discrete event simulator for TinyOS sensor networks. TOSSIM models the network in a weighted, directed graph, $G = \{V, A\}$. Weights of A specify the receive signal power between two nodes.

In this study, we implement the Acksis protocol and Hop protocol on TinyOS. The Hop protocol is designed for a regular mesh wireless network where the storage is not extremely limited. So it stores every packet motes receive or overhear. In order to investigate the performance of acknowledgement-assisted storage management of Acksis, we implement a modified Hop allowing motes to check their storage before caching packets and store the packets that have not been cached yet. This eliminates performance improvements of duplicate checking and allows us to focus on the improvements due to the buffer cleaning of Acksis protocol. We also want to compare Acksis with protocols designed for WSN, so we chose Collection Tree Protocol (CTP) as an example, which is a common WSN communication protocol. All four protocols are tested in TOSSIM. In the following experiment analysis, *Hop* is the original Hop protocol that motes store everything they receive or overhear without duplicate checking. *Modified Hop* has the duplicate checking, storing packets that are not already cached by the motes. *CTP* sends sequential packets without interval. *CTP(2s)* has two seconds sending interval between two consequent packets. When comparing Acksis, Hop and Modified Hop, these protocols use the same block size and same beacon timer for motes to exchange backlog tables with neighbors, in order to focus on the performance of storage management.

We ran the experiments on a 7×7 grid with the receive signal power over each edge of -60.0 dBm which is a typical value for wireless networks. Since mote storage is limited and we want to investigate the performance of the acknowledgement-assisted storage management of Acksis, we set the data packet queue size of each mote to 50.

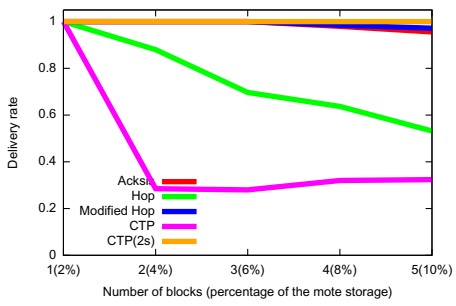
We use two metrics to evaluate the performance: *delivery rate* and *delivery speed*. Delivery rate is the number of received distinct packets at the destination divided by the total number of packets sent by the sources. Delivery speed is the number of packets sent by sources divided by the time to receive all of them at the destination. If there are packet losses, the time is the sum of the actual time to receive the last received packet and the estimation time to receive the rest of the packets. The higher the loss rate is, the longer the estimation time is. Unlike throughput, delivery speed is not the overall transmission capability of the network, but the speed to deliver a certain amount of data. Since TOSSIM

does not consider energy consumption, we will inspect this metric in future work. Each data point used for plotting is the average value of 100 runs of the experiment.

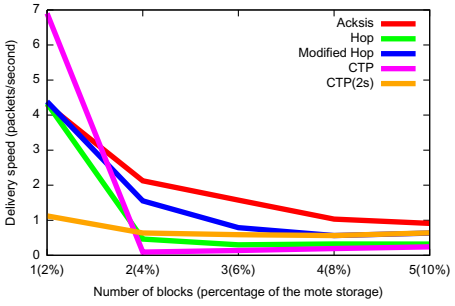
4.2 Experiment Results and Performance Evaluation

Single Source. In the single source experiment, the protocols are tested in light contention and collisions scenario. In this experiment we set the source to be the top left mote of the grid and the destination to be the right bottom mote of the grid. The block size of Acksis is one, which is calculated through Equation 1. Hop and Modified Hop use the same block size. The source sends out a flow of packets. The runtime of each run is 30 minutes simulator time. Figure 3 shows the delivery rates and delivery speeds of the protocols.

In Figure 3 (a), the delivery rate of Acksis and Modified Hop are very close and high, and decrease slowly as the flow length increases. However, the delivery rate of Hop is much lower than the other two, and decreases quickly with the flow length increasing. The reason for the quick decrease in the delivery rate of Hop is that the longer flows need more rounds to get delivered completely. Therefore the probability of retransmission also increases, which leads to the waste of mote storage for the duplicate packets. The longer the queue is, the higher the probability of storage overflow, which results in the delivery rate of Hop decreasing quickly. Collection Tree Protocol is designed for light traffic scenarios, and has no end-to-end reliability guarantee. So it can reach 100 percent delivery rate when it has two seconds sending interval. But when it has no sending interval, its delivery rate decreases quickly with the flow length increase.



(a) Acksis and Modified Hop have much higher delivery rates than Hop. CTP has the lowest delivery rate. CTP(2s) has a perfect delivery rate.



(b) Acksis has higher speeds than Hop and Modified Hop. CTP has the highest delivery speed initially, but declines quickly. CTP(2s) has comparatively low delivery speed.

Fig. 3. Delivery rate and delivery speed comparison in a low contention and collision scenario

In Figure 3 (b), the delivery speed of Acksis, Hop and Modified Hop decrease with flow length increasing. However Acksis has higher delivery speed than the other two. This is because Acksis deletes useless packets to save space, which keeps the queue length comparatively shorter than Hop and Modified Hop. So the queue time of packets in Acksis is comparatively less than the other two. CTP has good delivery speed initially, but due to its high loss rate the delivery speed decreases quickly. CTP(2s) has a fairly low delivery speed because of the two seconds sending interval.

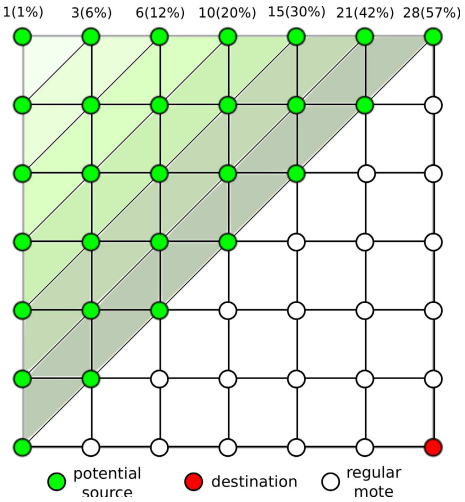
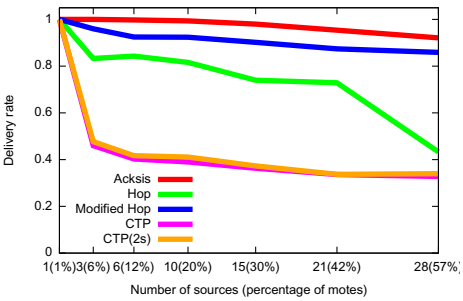


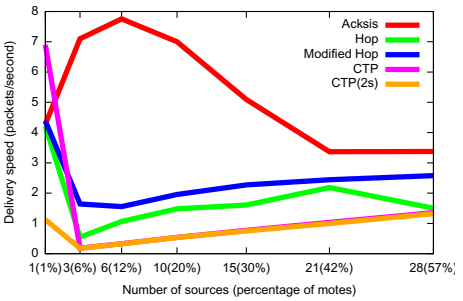
Fig. 4. Network layout for the experiment with multiple sources

Multiple Sources. In the multiple sources experiment, the protocols are tested in a high contention and collision scenario. In this experiment, we increase the number of sources from left to right diagonally. Figure 4 shows how the sources increase in this experiment. Initially, only the top left green mote transmits, which accounts for 1% of all motes. Next, the three green motes on the left of the 3(6%) diagonal start transmitting. The sources continue to increase until all the green motes in the grid start transmitting, which accounts for 57% of all motes. All sources send simultaneously. The runtime of each run is 30 minute simulator time. The block size for Acksis, Hop and Modified is one. Each source sends a single packet. So in this experiment, contention and collision increase as the percentage of sources increase. Figure 5 shows the delivery rates and delivery speeds of the protocols under the multiple sources scenario.

In Figure 5 (a), the delivery rate of Acksis is higher than Hop and Modified Hop. This is because Acksis has comparatively higher delivery speed than the other two, which is shown in Figure 5 (b). The higher the delivery speed, the more packets can be delivered during a certain period. As discussed in the previous experiment, the queue lengths of motes implementing Acksis are comparatively

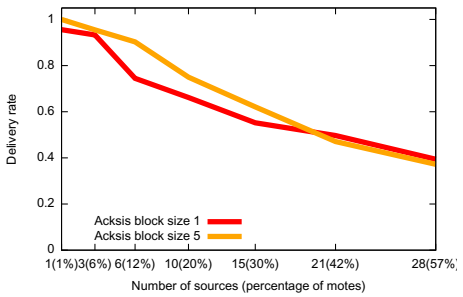


(a) Acksis has better delivery rates than Hop and Modified Hop. CTP and CTP(2s) have the lowest and very close delivery rates.

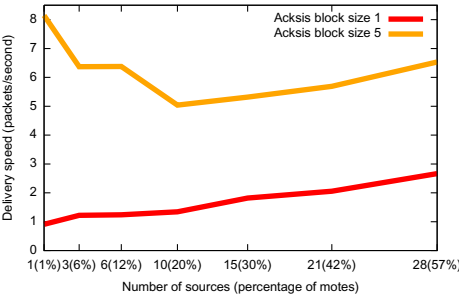


(b) Acksis has higher delivery speeds than Hop and Modified Hop. CTP has the highest delivery speed initially, but declines quickly. CTP(2s) has the lowest delivery speed all the time.

Fig. 5. Delivery rate and delivery speed comparison in a high contention and collision scenario



(a) Acksis with block size 1 has better delivery rates when more than 40% motes are sources.



(b) Acksis with block size 1 has a lower delivery speeds because it takes more rounds to send one flow than Acksis with block size 5.

Fig. 6. Delivery rates and delivery speeds of different block sizes

shorter than that of the motes implementing Hop and Modified Hop. As a result, the queue time of packets in Acksis is shorter than Hop and Modified Hop, and the delivery speed of Acksis is higher than the other two protocols. When the number of sources increases, the delivery rates of Acksis and Modified Hop get close, since the delivery speed of the two protocols also gets close. The reason for this change is the flow control mechanism. When there are more motes injecting packets into the network, the E2ERPLY buffer cleaning will not change the queue length much. Hence, the difference in mote queue length in these two protocols is insignificant, so that the delivery speeds of the two protocols are getting closer as the number of sources increase. The delivery rate and delivery

speed of Hop decreases quickly when more than 40% of the motes are sources. In Hop, motes store every packet received or overheard which causes storage overflows that degrades the performance. Due to a lack of end-to-end reliability and the capability to deal with heavy traffic, both CTP and CTP(2s) have the lowest delivery rates and delivery speeds.

Different Block Sizes. The block size does affect the performance of Acksis. In the different block sizes experiment, Acksis with different block sizes is tested under a high contention and collision scenario. In this experiment, the sources are selected in the same fashion as in multiple sources experiment. But, instead of each source sending a single packet, a flow of five packets needs to be sent by each source. Acksis with block size 5 sends one flow at one time, while Acksis with block size one needs five rounds to send one flow. The runtime of the experiment is 60 minute simulator time. Figure 6 shows the delivery rates and delivery speeds of each condition.

In Figure 6 (a), the delivery rates of Acksis with both sizes decrease as the number of sources increase. When more than 40% of motes are sources, Acksis with block size one has a better delivery rate than Acksis with block size five, since storages overflow happens when the block size is five.

In Figure 6 (b), Acksis with block size five has higher delivery speeds than Acksis with block size one. Because when sending the same flow, Acksis with block size five needs fewer rounds than Acksis with block size one.

This experiment exposes the weakness of the block size calculation in Acksis: the network resources are not sufficiently utilized when the traffic load is light. We are starting to address this problem by implementating an optimal block size calculation and will evaluate it in future work.

5 Conclusions and Future Work

This paper proposes a hop-by-hop transport protocol, Acksis, for intermittently connected WSNs, which supplies end-to-end reliability, congestion control and storage management. Acksis can provide a higher delivery rate and delivery speed even in a high contention and collision scenario than other hop-by-hop transport protocols with in-network storage that do not employ sufficient storage management. This is achieved by the storage management mechanisms of Acksis: block size calculation, checking duplicates before caching a packet and removing cached packets that have been confirmed by their destinations.

There are further opportunities for improvement of Acksis. In order to avoid overflow even in the high traffic load situations, the block size is pre-calculated, which leads to mote storage underutilization in low traffic load situations. Because the small block size needs more rounds to completely deliver a flow, the delivery speed is reduced. In our future work, we optimize block size decision making to imporve the network resource utilization. This optimization can also improve the scalability of Acksis, and help to ameliorate congestion control by allowing more blocks from a single source in flight. In addition, we will evaluate

the performance of Acksis on a delay-tolerant simulator and compare it with more WSN communication protocols, since it is hard to model the intermittent connectivity in TOSSIM. The network transmission capability of Acksis will also be inspected.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks 38(4), 393–422 (2002), http://www.sciencedirect.com/science/article/pii/S1389128601003024
2. Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., Weiss, H.: Delay-tolerant networking architecture (2007), http://www.ietf.org/rfc/rfc4838.txt
3. Fall, K.: A delay-tolerant network architecture for challenged internets. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM 2003, pp. 27–34. ACM, New York (2003), http://doi.acm.org/10.1145/863955.863960
4. Iyer, Y., Gandham, S., Venkatesan, S.: STCP: a generic transport layer protocol for wireless sensor networks. In: Proceedings of the 14th International Conference on Computer Communications and Networks (ICCCN 2005), pp. 449–454 (October 2005)
5. Kim, S., Fonseca, R., Dutta, P., Tavakoli, A., Culler, D., Levis, P., Shenker, S., Stoica, I.: Flush: a reliable bulk transport protocol for multihop wireless networks. In: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, SenSys 2007, pp. 351–365. ACM, New York (2007), http://doi.acm.org/10.1145/1322263.1322296
6. Levis, P., Gay, D.: TinyOS Programming. Cambridge University Press (2009)
7. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: accurate and scalable simulation of entire TinyOS applications. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, SenSys 2003, pp. 126–137. ACM, New York (2003), http://doi.acm.org/10.1145/958491.958506
8. Li, M., Agrawal, D., Ganesan, D., Venkataramani, A.: Block-switched networks: a new paradigm for wireless transport. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, pp. 423–436. USENIX Association, Berkeley (2009), http://dl.acm.org/citation.cfm?id=1558977.1559006
9. Marchi, B., Grilo, A., Nunes, M.: DTSN: Distributed transport for sensor networks. In: IEEE Symposium on Computers and Communications (ISCC 2007), pp. 165–172 (July 2007)
10. Paek, J., Govindan, R.: RCRT: Rate-controlled reliable transport protocol for wireless sensor networks. ACM Transactions on Senensor Networks 7, 20:1–20:45 (2010), http://doi.acm.org/10.1145/1807048.1807049
11. Park, S.J., Vedantham, R., Sivakumar, R., Akyildiz, I.F.: A scalable approach for reliable downstream data delivery in wireless sensor networks. In: Proceedings of the 5th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc 2004, pp. 78–89. ACM, New York (2004), http://doi.acm.org/10.1145/989459.989470
12. Polastre, J., Szewczyk, R., Sharp, C., Culler., D.: The mote revolution: Low power wireless sensor network devices. In: IEEE HotChips 16 (August 2004)

13. Prodhan, A.T., Das, R., Humayun, K., Shoja, G.C.: TTL based routing in opportunistic networks. Journal of Network and Computer Applications 34, 1660–1670 (2011), http://dx.doi.org/10.1016/j.jnca.2011.05.005
14. Saleem, A.B.: Performance Evaluation of The Cache and Forward Link Layer Protocol in Multihop Wireless Subnetworks. Master's thesis, Rutgers University (2008)
15. Soares, J.M., Franceschinis, M., Rocha, R.M., Zhang, W., Spirito, M.A.: Opportunistic data collection in sparse wireless sensor networks. EURASIP Journal on Wireless Communications Networking, 6:1–6:20 (January 2011), http://dx.doi.org/10.1155/2011/401802
16. Stann, F., Heidemann, J.: RMST: reliable data transport in sensor networks. In: Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications, pp. 102–112 (May 2003)
17. Tezcan, N., Wang, W.: ART: an asymmetric and reliable transport mechanism for wireless sensor networks. International Journal of Sensor Network 2, 188–200 (2007), http://dl.acm.org/citation.cfm?id=1359004.1359009
18. Tian, Y., Xu, K., Ansari, N.: TCP in wireless environments: problems and solutions. IEEE Communications Magazine 43(3), S27–S32 (2005)
19. Wan, C.Y., Campbell, A.T., Krishnamurthy, L.: PSFQ: a reliable transport protocol for wireless sensor networks. In: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications, WSNA 2002, pp. 1–11. ACM, New York (2002), http://doi.acm.org/10.1145/570738.570740
20. Wang, Y., Wu, H.: Delay/fault-tolerant mobile sensor network (DFT-MSN): A new paradigm for pervasive information gathering. IEEE Transactions on Mobile Computing 6(9), 1021–1034 (2007)
21. Yick, J., Mukherjee, B., Ghosal, D.: Wireless sensor network survey. Computer Networks 52(12), 2292–2330 (2008), http://www.sciencedirect.com/science/article/B6VRG-4S8TBBT-1/2/b242d2fd1f6d2cf5c6fce0a24c4cb029
22. Zhang, H., Arora, A., Choi, Y.R., Gouda, M.G.: Reliable bursty convergecast in wireless sensor networks. In: Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc 2005, pp. 266–276. ACM, New York (2005), http://doi.acm.org/10.1145/1062689.1062724

Iterative Approximate Byzantine Consensus under a Generalized Fault Model*

Lewis Tseng[1,3] and Nitin Vaidya[2,3]

[1] Department of Computer Science
[2] Department of Electrical and Computer Engineering, and
[3] Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{ltseng3,nhv}@illinois.edu

Abstract. In this work, we consider a *generalized* fault model [7,9,5] that can be used to represent a wide range of failure scenarios, including correlated failures and non-uniform node reliabilities. Under the generalized fault model, we explore iterative approximate Byzantine consensus (IABC) algorithms [15] in arbitrary directed networks. We prove a *tight* necessary and sufficient condition on the underlying communication graph for the existence of IABC algorithms.

We propose a new IABC algorithm for the generalized fault model, and present a *transition matrix*-based proof to show the correctness of the proposed algorithm. While transition matrices have been used to prove correctness of non-fault-tolerant consensus [6], this paper is the first to extend the technique to Byzantine fault-tolerant algorithms.

Keywords: iterative consensus, graph property, generalized fault model.

1 Introduction

Dolev et al. [3] introduced the notion of *approximate Byzantine consensus* by relaxing the requirement of exact consensus [12]. The goal in approximate consensus is to allow the fault-free nodes to agree on values that are approximately equal to each other (and not necessarily exactly identical). The fault model assumed in much of the work on Byzantine consensus allows up to f Byzantine faulty nodes in the network. We will refer to this fault model as the "f-total" fault model [11,10,3,12]. In prior work, other fault models have been explored as well. For instance, in the "f-local" fault model, up to f neighbors of each node in the network may be faulty [8,1,11]. In this paper, we consider a *generalized* fault model (to be described in the next section), which is similar to the fault model presented in [7,9,5]. The generalized fault model specifies a "fault

* This research is supported in part by National Science Foundation award CNS 1059540 and Army Research Office grant W-911-NF-0710287. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 72–86, 2013.

domain", which is a collection of feasible fault sets. For example, in a system consisting of four nodes, namely, nodes $1, 2, 3$ and 4, the fault domain could be specified as $\mathcal{F} = \{ \{1\}, \{2, 3, 4\} \}$. Thus, in this case, either node 1 may be faulty, or any subset of nodes in $\{2, 3, 4\}$ may be faulty. However, node 1 may not be faulty together with another node in the same execution. This fault model is general in the sense that the other fault models, such as f-total, and f-local models, are special cases of the generalized fault model.

In this work, we consider "iterative" algorithms for achieving approximate Byzantine consensus in synchronous point-to-point networks that are modeled as arbitrary directed graphs. The *iterative approximate Byzantine consensus* (IABC) algorithms [15] of interest have the following properties:

- *Initial state* of each node is equal to a real-valued input provided to that node.
- *Memory-less*: the computation of new state at each node is based only on local information, i.e., node's own state and states from neighboring nodes.
- *Validity* condition: After each iteration of an IABC algorithm, the state of each fault-free node must remain in the convex hull of the states of the fault-free nodes at the end of the previous iteration.
- *Convergence* condition: For any $\epsilon > 0$, there exists an iteration r such that the states of the fault-free nodes are guaranteed to be within ϵ in every iteration $r' \geq r$.

1.1 Main Contributions

This paper is a generalization of our recent work on IABC algorithms under the f-total fault model [15]. There are two contributions of this paper:

- We identify a necessary (Section 4) condition on the underlying communication graph for the existence of a correct IABC algorithm under the generalized fault model. Moreover, we show that the necessary condition is also sufficient by introducing a new IABC algorithm for the generalized fault model (Section 5) that uses only "local" information.
- We present a *transition matrix* representation [14] of the new IABC algorithm (Section 6). This representation is then used to prove the correctness of the proposed algorithm (Section 6.4). Transition matrices have been used previously to prove correctness of non-fault-tolerant consensus [6]. However, this paper is the first to develop transition matrix representation for Byzantine fault-tolerant consensus. We make the following observation: for a given evolution of the state vector corresponding to the state of the fault-free nodes, many alternate state transition matrices may potentially be chosen to emulate that evolution correctly. However, for any given state evolution, we can suitably "design" the transition matrices so that the classical results on matrix products can be applied to prove convergence of our algorithm in all networks that satisfy the necessary condition.

2 Models

Communication Model: The system is assumed to be synchronous. The communication network is modeled as a simple directed graph $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \ldots, n\}$ is the set of n nodes, and $\mathcal{E}$ is the set of directed edges between the nodes in $\mathcal{V}$. We assume that $n \geq 2$, since the consensus problem for $n = 1$ is trivial. Node i can reliably transmit messages to node j if and only if the directed edge (i, j) is in $\mathcal{E}$. Each node can send messages to itself as well; however, for convenience, we exclude self-loops from set $\mathcal{E}$. That is, $(i, i) \notin \mathcal{E}$ for $i \in \mathcal{V}$. With a slight abuse of terminology, we will use the terms *edge* and *link* interchangeably in our presentation.

For each node i, let N_i^- be the set of nodes from which i has incoming edges. That is, $N_i^- = \{ j \mid (j, i) \in \mathcal{E} \}$. Similarly, define N_i^+ as the set of nodes to which node i has outgoing edges. That is, $N_i^+ = \{ j \mid (i, j) \in \mathcal{E} \}$. Nodes in N_i^- and N_i^+ are, respectively, said to be incoming and outgoing neighbors of node i. Since we exclude self-loops from $\mathcal{E}$, $i \notin N_i^-$ and $i \notin N_i^+$. However, we note again that each node can indeed send messages to itself.

Generalized Byzantine Fault Model: A Byzantine faulty node may misbehave arbitrarily. Possible misbehavior includes transmitting incorrect and mismatching (or inconsistent) messages to different neighbors. The faulty nodes may collaborate with each other. Moreover, the faulty nodes are assumed to have a complete knowledge of the execution of the algorithm, including the states of all the nodes, the algorithm specification, and the network topology.

The generalized fault model we consider is similar to fault models presented in [7,9,5]. The generalized fault model is characterized using *fault domain* $\mathcal{F} \subseteq 2^{\mathcal{V}}$ as follows: Nodes in set F may fail during an execution of the algorithm only if there exists set $F^* \in \mathcal{F}$ such that $F \subseteq F^*$. Set F is then said to be a *feasible* fault set.

Definition 1. *Set $F \subseteq \mathcal{V}$ is said to be a feasible fault set, if there exists $F^* \in \mathcal{F}$ such that $F \subseteq F^*$.*

Thus, each set in $\mathcal{F}$ specifies nodes that may all potentially fail during a single execution of the algorithm. This feature can be used to capture the notion of correlated failures. For example, consider a system consisting of four nodes, namely, nodes 1, 2, 3, and 4. Suppose that

$$\mathcal{F} = \{ \{1\}, \{2\}, \{3, 4\} \}.$$

This definition of $\mathcal{F}$ implies that during an execution either (i) node 1 may fail, (ii) node 2 may fail, or (iii) any subset of $\{3, 4\}$ may fail, and no other combination of nodes may fail (e.g., nodes 1 and 3 cannot both fail in a single execution). In this case, the reason that the set $\{3, 4\}$ is in the fault domain may be that the failures of nodes 3 and 4 are correlated.

The generalized fault model is also useful to capture variations in node reliability [7,9,5]. For instance, in the above example, nodes 1 and 2 may be more reliable than nodes 3 and 4. Therefore, while nodes 3 and 4 may fail in the same execution, nodes 1 and 2 are less likely to fail together in the same execution. Therefore, $\{1, 2\} \notin \mathcal{F}$.

Local knowledge of $\mathcal{F}$: To implement our IABC Algorithm presented in Section 5, it is sufficient for each node i to know $N_i^- \cap F$, for each feasible fault set F. In other words, each node only needs to know the set of its incoming neighbors that may fail in the same execution of the algorithm. Thus, the iterative algorithm can be implemented using only "local" information regarding $\mathcal{F}$.

3 Iterative Approximate Byzantine Consensus (IABC) Algorithms

In this section, we describe the structure of the IABC algorithms of interest, and state the validity and convergence conditions that they must satisfy.

Each node i maintains state v_i, with $v_i[t]$ denoting the state of node i at the end of the t-th iteration of the algorithm. Initial state of node i, $v_i[0]$, is equal to the initial input provided to node i. At the start of the t-th iteration ($t > 0$), the state of node i is $v_i[t-1]$. The IABC algorithms of interest will require each node i to perform the following three steps in iteration t, where $t > 0$. Note that the faulty nodes may deviate from this specification.

1. *Transmit step:* Transmit current state, namely $v_i[t-1]$, on all outgoing edges and self-loop (to nodes in N_i^+ and node i itself).
2. *Receive step:* Receive values on all incoming edges and self-loop (from nodes in N_i^- and itself). If the node does not receive the message from an incoming neighbor, the message value is assumed to be equal to some *default* value. Denote by $r_i[t]$ the vector of values received by node i from its incoming neighbors and itself. The size of vector $r_i[t]$ is $|N_i^-| + 1$.
3. *Update step:* Node i updates its state using a transition function Z_i as follows. Z_i is a part of the specification of the algorithm, and takes the vector $r_i[t]$ as the input.

$$v_i[t] = Z_i \ (\ r_i[t] \ ) \tag{1}$$

The following conditions must be satisfied by an IABC algorithm when the set of faulty nodes (in a given execution) is F:

- *Validity:* $\forall t > 0$, and all fault-free nodes $i \in \mathcal{V} - F$,
  $v_i[t] \geq \min_{j \in \mathcal{V}-F} \ v_j[t-1]$ and $v_i[t] \leq \max_{j \in \mathcal{V}-F} \ v_j[t-1]$.[1]
- *Convergence:* for all *fault-free* nodes $i, j \in \mathcal{V} - F$, $\lim_{t \to \infty} \ (v_i[t] - v_j[t]) = 0$

[1] For sets X and Y, $X - Y$ contains elements that are in X but not in Y. That is, $X - Y = \{i \mid i \in X, \ i \notin Y\}$.

An IABC algorithm is said to be correct if it satisfies the above validity and convergence conditions in the given graph $G(\mathcal{V}, \mathcal{E})$. For a given fault domain $\mathcal{F}$ for graph $G(\mathcal{V}, \mathcal{E})$, the objective here is to identify the necessary and sufficient conditions for the existence of a correct IABC algorithm.

4 Necessary Condition

In this section, we develop a necessary condition for the existence of a correct IABC algorithm. The necessary condition will be proved to be also sufficient in Section 6.

4.1 Preliminaries

To facilitate the statement of the necessary condition, we first introduce the notions of "source component" and "reduced graph" using the following three definitions.

Definition 2. Graph Decomposition: *Let H be a directed graph. Partition graph H into strongly connected components, $H_1, H_2, \cdots, H_h$, where h is a non-zero integer dependent on graph H, such that*

- *every pair of nodes within the same strongly connected component has directed paths in H to each other, and*
- *for each pair of nodes, say i and j, that belong to two different strongly connected components, either i does not have a directed path to j in H, or j does not have a directed path to i in H, or both.*

Construct a graph H^d wherein each strongly connected component H_k above is represented by vertex c_k, and there is an edge from vertex c_k to vertex c_l if and only if the nodes in H_k have directed paths in H to the nodes in H_l. H^d is called the decomposition graph of H.

It is known that for any directed graph H, the corresponding decomposition graph H^d is a directed acyclic graph (DAG) [2].

Definition 3. Source Component: *Let H be a directed graph, and let H^d be its decomposition graph as per Definition 2. Strongly connected component H_k of H is said to be a source component if the corresponding vertex c_k in H^d is not reachable from any other vertex in H^d.*

Definition 4. Reduced Graph: *For a given graph $G(\mathcal{V}, \mathcal{E})$ and a feasible fault set F, a reduced graph $G_F(\mathcal{V}_F, \mathcal{E}_F)$ is obtained as follows:*

- *Node set is obtained as $\mathcal{V}_F = \mathcal{V} - F$.*
- *For each node $i \in \mathcal{V}_F$, a feasible fault set $F_x(i)$ is chosen, and then the edge set $\mathcal{E}_F$ is obtained as follows:*

- *remove from $\mathcal{E}$ all the links incident on the nodes in F, i.e., all the incoming and outgoing links of nodes in F, and*
- *for each $j \in F_x(i) \cap \mathcal{V}_F \cap N_i^-$, remove link (j, i) from $\mathcal{E}$.*

Feasible fault sets $F_x(i)$ and $F_x(j)$ chosen for $i \neq j$ may or may not be identical.

Note that for a given $G(\mathcal{V}, \mathcal{E})$ and a given F, multiple reduced graphs G_F may exist, depending on the choice of F_x sets above.

4.2 Necessary Condition

For a correct IABC algorithm to exist, the network graph $G(\mathcal{V}, \mathcal{E})$ must satisfy the necessary condition stated in Theorem 1 below.

Theorem 1. *Suppose that a correct IABC algorithm exists for $G(\mathcal{V}, \mathcal{E})$. Then, any reduced graph G_F, corresponding to any feasible fault set F, must contain exactly one source component.*

Proof Sketch: A complete proof is presented in our technical report [13].The proof is by contradiction. Let us assume that a correct IABC algorithm exists, and for some feasible fault set F, and feasible sets $F_x(i)$ for each $i \in \mathcal{V} - F$, the resulting reduced graph contains two source components. Let L and R denote the nodes in the two source components, respectively. Thus, L and R are disjoint and non-empty. Let $C = (\mathcal{V} - F - L - R)$ be the remaining nodes in the reduced graph. C may or may not be non-empty. Assume that the nodes in F (if non-empty) are all faulty, and all the nodes in L, R, and C (if non-empty) are fault-free. Suppose that each node in L has initial input equal to m, each node in R has initial input equal to M, where $M > m$, and each node in C has an input in the range $[m, M]$. As elaborated in [13], the faulty nodes can behave in such a manner that, in each iteration, nodes in L and R are forced to maintain their updated state equal to m and M, respectively, so as to satisfy the validity condition. This ensures that, no matter how many iterations are performed, the convergence condition cannot be satisfied. □

5 Algorithm 1

We will prove that there exists an IABC algorithm – particularly *Algorithm 1* below – that satisfies the validity and convergence conditions provided that the graph $G(\mathcal{V}, \mathcal{E})$ satisfies the necessary condition in Theorem 1. This implies that the necessary condition in Theorem 1 is also sufficient. *Algorithm 1* has the three-step structure described in Section 3. This algorithm is a generalization – to accommodate the generalized fault model – of iterative algorithms that were analyzed in prior work [3,12,11], including in our own prior work as well [15]. The key difference from previous algorithms is in the *Update* step below. Now, we describe the steps need to be followed by all the fault-free nodes in iteration t ($t > 0$).

Algorithm 1.

1. *Transmit step:* Transmit current state $v_i[t-1]$ on all outgoing edges and self-loop.
2. *Receive step:* Receive values on all incoming edges and self-loop. These values form vector $r_i[t]$ of size $|N_i^-|+1$ (including the value from node i itself). When a fault-free node expects to receive a message from an incoming neighbor but does not receive the message, the message value is assumed to be equal to its own state, i.e., $v_i[t-1]$.
3. *Update step:* Sort the values in $r_i[t]$ in an increasing order (breaking ties arbitrarily). Let D be a vector of nodes arranged in an order "consistent" with $r_i[t]$: specifically, $D(1)$ is the node that sent the smallest value in $r_i[t]$, $D(2)$ is the node that sent the second smallest value in $r_i[t]$, and so on. The size of vector D is also $|N_i^-|+1$.
   From vector $r_i[t]$, eliminate the smallest f_1 values, and the largest f_2 values, where f_1 and f_2 are defined as follows:
   - f_1 is the largest number such that there exists a feasible fault set $F' \subseteq N_i^-$ containing nodes $D(1), D(2), ..., D(f_1)$. Recall that $i \notin N_i^-$.
   - f_2 is the largest number such that there exists a feasible fault set $F'' \subseteq N_i^-$ containing nodes $D(|N_i^-|-f_2+2), D(|N_i^-|-f_2+3), ..., D(|N_i^-|+1)$.

   F' and F'' above may or may not be identical.
   Let $N_i^*[t]$ denote the set of nodes from whom the remaining $|N_i^-|+1-f_1-f_2$ values in $r_i[t]$ were received, and let w_j denote the value received from node $j \in N_i^*[t]$. Note that $i \in N_i^*[t]$. Hence, for convenience, define $w_i = v_i[t-1]$ to be the value node i receives from itself. Observe that if $j \in N_i^*[t]$ is fault-free, then $w_j = v_j[t-1]$.
   Define

$$v_i[t] \;=\; Z_i(r_i[t]) \;=\; \sum_{j \in N_i^*[t]} a_i \, w_j \tag{2}$$

   where

$$a_i = \frac{1}{|N_i^*[t]|} = \frac{1}{|N_i^-| + 1 - f_1 - f_2}$$

   The "weight" of each term on the right-hand side of (2) is a_i, and these weights add to 1. Also, $0 < a_i \leq 1$. Although f_1, f_2 and a_i may be different for each iteration t, for simplicity, we do not explicitly represent this dependence on t in the notations.

Observe that $f_1 + f_2$ nodes whose values are eliminated in the *Update* step above are all in N_i^-. Thus, the above algorithm can be implemented by node i if it knows which of its incoming neighbors may fail in a single execution of the algorithm; node i does not need to know the entire fault domain $\mathcal{F}$ as such.

The main difference between the Algorithm 1 and IABC algorithms in prior work is in the choice of the values eliminated from vector $r_i[t]$ in the *Update* step.

The manner in which the values are eliminated ensures that the values received from nodes $D(f_1+1)$ and $D(|N_i^-|-f_2+1)$ (i.e., the smallest and largest values that survive in $r_i[t]$) are within the convex hull of the state of fault-free nodes, even if nodes $D(f_1+1)$ and $D(|N_i^-|-f_2+1)$ may not be fault-free. This property is useful in proving algorithm correctness (as discussed below).

6 Sufficiency: Correctness of Algorithm 1

We will show that Algorithm 1 satisfies validity and convergence conditions, provided that $G(\mathcal{V},\mathcal{E})$ satisfies the condition below, which matches the necessary condition stated in Theorem 1.

Sufficient Condition: *Any reduced graph G_F corresponding to any feasible fault set F contains exactly one* source component*.*

In the rest of this section, we assume that $G(\mathcal{V},\mathcal{F})$ satisfies the above condition. We first introduce some standard matrix tools to facilitate our proof. Then, we develop a transition matrix representation of the *Update* step in Algorithm 1, and show how to use these tools to prove the correctness of Algorithm 1 in $G(\mathcal{V},\mathcal{F})$.

When presenting matrix products, for convenience of presentation, we adopt the "backward" product convention below, where $a \leq b$,

$$\Pi_{i=a}^{b}\mathbf{A}[i] = \mathbf{A}[b]\mathbf{A}[b-1]\cdots\mathbf{A}[a] \tag{3}$$

6.1 Matrix Preliminaries

In the discussion below, we use boldface upper case letters to denote matrices, rows of matrices, and their elements. For instance, $\mathbf{A}$ denotes a matrix, $\mathbf{A}_i$ denotes the i-th row of matrix $\mathbf{A}$, and $\mathbf{A}_{ij}$ denotes the element at the intersection of the i-th row and the j-th column of matrix $\mathbf{A}$.

Definition 5. *A vector is said to be stochastic if all the elements of the vector are non-negative, and the elements add up to 1. A matrix is said to be row stochastic if each row of the matrix is a stochastic vector.*

For a row stochastic matrix $\mathbf{A}$, coefficients of ergodicity $\delta(\mathbf{A})$ and $\lambda(\mathbf{A})$ are defined as follows [16]:

$$\delta(\mathbf{A}) = \max_j \max_{i_1,i_2} |\mathbf{A}_{i_1 j} - \mathbf{A}_{i_2 j}|$$

$$\lambda(\mathbf{A}) = 1 - \min_{i_1,i_2} \sum_j \min(\mathbf{A}_{i_1 j}, \mathbf{A}_{i_2 j})$$

It is easy to show that $0 \leq \delta(\mathbf{A}) \leq 1$ and $0 \leq \lambda(\mathbf{A}) \leq 1$, and that the rows of $\mathbf{A}$ are all identical if and only if $\delta(\mathbf{A}) = 0$. Also, $\lambda(\mathbf{A}) = 0$ if and only if $\delta(\mathbf{A}) = 0$.

The next result from [4] establishes a relation between the coefficient of ergodicity $\delta(\cdot)$ of a product of row stochastic matrices, and the coefficients of ergodicity $\lambda(\cdot)$ of the individual matrices defining the product.

Lemma 1. *For any p square row stochastic matrices* $\mathbf{A}(1), \mathbf{A}(2), \ldots \mathbf{A}(p)$,

$$\delta(\mathbf{A}(p)\mathbf{A}(p-1)\cdots\mathbf{A}(1)) \;\leq\; \Pi_{i=1}^{p}\ \lambda(\mathbf{A}(i)).$$

Lemma 1 is proved in [4]. It implies that if, for all i, $\lambda(\mathbf{A}(i)) \leq 1-\gamma$ for some γ, where $0 < \gamma \leq 1$, then $\delta(\mathbf{A}(p)\mathbf{A}(p-1)\cdots\mathbf{A}(1))$ will approach zero as p approaches ∞. We now define a *scrambling* matrix [4,16].

Definition 6. *A row stochastic matrix* $\mathbf{A}$ *is said to be a scrambling matrix if*

$$\lambda(\mathbf{A}) < 1$$

The following lemma follows easily from the above definition of $\lambda(\cdot)$.

Lemma 2. *If any column of a row stochastic matrix* $\mathbf{A}$ *contains only non-zero elements that are all lower bounded by some constant* γ*, where* $0 < \gamma \leq 1$*, then* $\mathbf{A}$ *is a scrambling matrix, and* $\lambda(\mathbf{A}) \leq 1-\gamma$.

6.2 Transition Matrix Representation

In our discussion below, $\mathbf{M}[t]$ is a square matrix, $\mathbf{M}_i[t]$ is the i-th row of the matrix, and $\mathbf{M}_{ij}[t]$ is the element at the intersection of the i-th row and j-th column of $\mathbf{M}[t]$.

For a given execution of Algorithm 1, let F denote the actual set of faulty nodes in that execution. Let $|F| = \psi$. Without loss of generality, suppose that nodes 1 through $(n-\psi)$ are fault-free, and if $\psi > 0$, nodes $(n-\psi+1)$ through n are faulty. Denote by $v[0]$ the column vector consisting of the initial states of all the fault-free nodes. Denote by $v[t]$, where $t \geq 1$, the column vector consisting of the states of all the fault-free nodes at the end of the t-th iteration. The i-th element of vector $v[t]$ is state $v_i[t]$. The size of vector $v[t]$ is $(n-\psi)$.

We will show that the iterative update of the state of a fault-free node i ($1 \leq i \leq n-\psi$) performed in (2) in Algorithm 1 can be expressed using the matrix form below.

$$v_i[t] = \mathbf{M}_i[t]\ v[t-1] \tag{4}$$

where $\mathbf{M}_i[t]$ is a stochastic row vector of size $n-\psi$. That is, $\mathbf{M}_{ij}[t] \geq 0$, for $1 \leq j \leq n-\psi$, and $\sum_{1\leq j\leq n-\psi} \mathbf{M}_{ij}[t] = 1$.[2] By "stacking" (4) for different i, $1 \leq i \leq n-\psi$, we will represent the *Update* step of Algorithm 1 at all the fault-free nodes together using (5) below.

$$v[t] = \mathbf{M}[t]\ v[t-1] \tag{5}$$

where $\mathbf{M}[t]$ is a $(n-\psi)\times(n-\psi)$ row stochastic matrix, with its i-th row being equal to $\mathbf{M}_i[t]$ in (4). $\mathbf{M}[t]$ is said to be a transition matrix.

[2] In addition to t, the row vector $\mathbf{M}_i[t]$ may depend on the state vector $v[t-1]$ as well as the behavior of the faulty nodes in F. For simplicity, the notation $\mathbf{M}_i[t]$ does not explicitly represent this dependence.

By repeated application of (5), we can represent the *Update* step of Algorithm 1 at the t-th iterations ($t \geq 1$) as:

$$v[t] = \left(\Pi_{k=1}^{t} \mathbf{M}[k] \right) v[0] \tag{6}$$

Recall that we adopt the "backward" product convention as presented in (3).

In the rest of this section, we will first "construct" transition matrices $\mathbf{M}[k]$ ($1 \leq k \leq t$) that satisfy certain desirable properties. Then, we will identify a connection between these transition matrices and the sufficiency condition stated above, and use this connection to establish convergence property for Algorithm 1. The validity property also follows from the transition matrix representation.

6.3 Construction of the Transition Matrix

We will construct a transition matrix with the property described in Lemma 3 below.

Lemma 3. *The* Update *step of Algorithm 1 at the fault-free nodes can be expressed using row stochastic transition matrix* $\mathbf{M}[t]$*, such that there exists a feasible fault set* $F_x(i)$ *for each* $i \in \mathcal{V} - F$*, and for all* $j \in \{i\} \cup ((\mathcal{V}_F - F_x(i)) \cap N_i^-)$*,*

$$\mathbf{M}_{ij}[t] \ \geq \ \beta$$

where β *is a constant (to be defined later), and* $0 < \beta \leq 1$*.*

Proof. We prove the correctness of Lemma 3 by constructing $\mathbf{M}_i[t]$ for $1 \leq i \leq n - \psi$ that satisfies the conditions in Lemma 3. Recall that F is the set of faulty nodes, and $|F| = \psi$. As stated before, without loss of generality, nodes 1 through $n - \psi$ are assumed to be fault-free, and the remaining ψ nodes faulty.

Consider a fault-free node i performing the *Update* step in Algorithm 1. In the *Update* step, recall that the smallest f_1 and the largest f_2 values are eliminated from $r_i[t]$, where the choice of f_1 and f_2 is described in Algorithm 1. Let us denote by $\mathcal{S}$ and $\mathcal{L}$, respectively, the set of nodes[3] from whom the smallest f_1 and the largest f_2 values were received by node i in iteration t. Define sets $\mathcal{S}_g$ and $\mathcal{L}_g$ to be subsets of $\mathcal{S}$ and $\mathcal{L}$ that contain all the fault-free nodes in $\mathcal{S}$ and $\mathcal{L}$, respectively. That is, $\mathcal{S}_g = \mathcal{S} \cap (\mathcal{V} - F)$ and $\mathcal{L}_g = \mathcal{L} \cap (\mathcal{V} - F)$.

Construction of $\mathbf{M}_i[t]$ differs somewhat depending on whether sets $\mathcal{S}_g, \mathcal{L}_g$ and $N_i^*[t] \cap F$ are empty or not. We divide the possibilities into 6 cases. Due to space limitation, here we present the construction for one of the cases (named Case I). The construction for the remaining cases is presented in [13].

In Case I, $\mathcal{S}_g \neq \emptyset, \mathcal{L}_g \neq \emptyset$, and $N_i^*[t] \cap F \neq \emptyset$. Let $m_{\mathcal{S}}$ and $m_{\mathcal{L}}$ be defined as shown below. Recall that the nodes in $\mathcal{S}_g$ and $\mathcal{L}_g$ are all fault-free, and therefore, for any node $j \in \mathcal{S}_g \cup \mathcal{L}_g$, $w_j = v_j[t-1]$ (in the notation of Algorithm 1).

$$m_{\mathcal{S}} = \frac{\sum_{j \in \mathcal{S}_g} v_j[t-1]}{|\mathcal{S}_g|} \qquad \text{and} \qquad m_{\mathcal{L}} = \frac{\sum_{j \in \mathcal{L}_g} v_j[t-1]}{|\mathcal{L}_g|}$$

[3] Although $\mathcal{S}$ and $\mathcal{L}$ may be different for each t, for simplicity, we do not explicitly represent this dependence on t in the notations $\mathcal{S}$ and $\mathcal{L}$.

Now, consider any node $k \in N_i^*[t]$. By the definition of sets $\mathcal{S}_g$ and $\mathcal{L}_g$, $m_{\mathcal{S}} \leq w_k \leq m_{\mathcal{L}}$. Therefore, we can find weights $S_k \geq 0$ and $L_k \geq 0$ such that $S_k + L_k = 1$, and

$$w_k = S_k\ m_{\mathcal{S}} + L_k\ m_{\mathcal{L}} \tag{7}$$

$$= \frac{S_k}{|\mathcal{S}_g|} \sum_{j \in \mathcal{S}_g} v_j[t-1] + \frac{L_k}{|\mathcal{L}_g|} \sum_{j \in \mathcal{L}_g} v_j[t-1] \tag{8}$$

Clearly, at least one of S_k and L_k must be $\geq 1/2$. We now define elements $\mathbf{M}_{ij}[t]$ of row $\mathbf{M}_i[t]$:

- For $j \in N_i^*[t] \cap (\mathcal{V} - F)$: In this case, j is either a fault-free incoming neighbor of i, or i itself. For each such j, define $\mathbf{M}_{ij}[t] = a_i$. This is obtained by observing in (2) that the contribution of such a node j to the new state $v_i[t]$ is $a_i\ w_j = a_i\ v_j[t-1]$.
  The elements of $\mathbf{M}_i[t]$ defined here add up to
  $$|N_i^*[t] \cap (\mathcal{V} - F)|\ a_i$$
- For $j \in \mathcal{S}_g \cup \mathcal{L}_g$: In this case, j is a fault-free node in $\mathcal{S}$ or $\mathcal{L}$.
  For each $j \in \mathcal{S}_g$,
  $$\mathbf{M}_{ij}[t] \ = \ a_i \sum_{k \in N_i^*[t] \cap F} \frac{S_k}{|\mathcal{S}_g|}$$
  and for each node $j \in \mathcal{L}_g$,
  $$\mathbf{M}_{ij}[t] \ = \ a_i \sum_{k \in N_i^*[t] \cap F} \frac{L_k}{|\mathcal{L}_g|}$$
  To obtain these two expressions, we represent value w_k sent by each faulty node k in $N_i^*[t]$, i.e., $k \in N_i^*[t] \cap F$, using (8). Recall that this node k contributes $a_i w_k$ to (2). The above two expressions are then obtained by summing (8) over all the faulty nodes in $N_i^*[t] \cap F$, and replacing this sum by equivalent contributions by nodes in $\mathcal{S}_g$ and $\mathcal{L}_g$.
  The elements of $\mathbf{M}_i[t]$ defined here add up to
  $$a_i \sum_{k \in N_i^*[t] \cap F} (S_k + L_k) = |N_i^*[t] \cap F|\ a_i.$$
- For $j \in (\mathcal{V} - F) - (N_i^*[t] \cup \mathcal{S}_g \cup \mathcal{L}_g)$: These fault-free nodes have not yet been considered above. For each such node j, define $\mathbf{M}_{ij}[t] = 0$.

With the above definition of $\mathbf{M}_i[t]$, it should be easy to see that $\mathbf{M}_i[t]\, v[t-1]$ is, in fact, identical to $v_i[t]$ obtained using (2). Thus, the above construction of $\mathbf{M}_i[t]$ results in the contribution of the faulty nodes in $N_i^*[t]$ to (2) being replaced by an equivalent contribution from fault-free nodes in $\mathcal{L}_g$ and $\mathcal{S}_g$.

Properties of $\boldsymbol{M}_i[t]$: First, we show that $\mathbf{M}[t]$ is row stochastic. Observe that all the elements of $\mathbf{M}_i[t]$ are non-negative. Also, all the elements of $\mathbf{M}_i[t]$ above add up to

$$|N_i^*[t] \cap (\mathcal{V} - F)|\ a_i + |N_i^*[t] \cap F|\ a_i = |N_i^*[t]|\ a_i = 1$$

because $a_i = 1/|N_i^*[t]|$ as defined in Algorithm 1. Thus, $\mathbf{M}_i[t]$ is a stochastic row vector.

Recall that from the above discussion, for $k \in N_i^*[t]$, one of S_k and L_k must be $\geq 1/2$. Without loss of generality, assume that $S_s \geq 1/2$ for some node $s \in N_i^*[t] \cap F$. Consequently, for each node $j \in \mathcal{S}_g$, $\mathbf{M}_{ij}[t] \geq \frac{a_i}{|\mathcal{S}_g|} S_s \geq \frac{a_i}{2|\mathcal{S}_g|}$. Also, for each fault-free node j in $N_i^*[t]$, $\mathbf{M}_{ij}[t] = a_i$. Thus, if β is chosen such that

$$0 < \beta \leq \frac{a_i}{2|\mathcal{S}_g|} \tag{9}$$

and $F_x(i)$ is defined to be equal to $\mathcal{L}$, then the condition in the lemma holds for node i. That is, $\mathbf{M}_{ij}[t] \geq \beta$ for $j \in \{i\} \cup ((\mathcal{V}_F - F_x(i)) \cap N_i^-)$.

All Cases Together: Using similar constructions in other cases as well (presented in [13]) and a suitable choice of β (presented in [13]), we can obtain a row stochastic matrix $\mathbf{M}[t]$, and for each $i \in \mathcal{V} - F$ identify a feasible fault set $F_x(i)$, such that $\mathbf{M}_{ij}[t] \geq \beta$ for all $j \in \{i\} \cup ((\mathcal{V}_F - F_x(i)) \cap N_i^-)$. Thus, Lemma 3 can be proved correct. □

6.4 Validity and Convergence of Algorithm 1

Correspondence between $\mathbf{M}[t]$ and a Reduced Graph: Let R_F denote the set of all the reduced graphs of $G(\mathcal{V}, \mathcal{E})$ corresponding to a feasible fault set F. Let $\tau = |R_F|$. τ depends on F and the underlying network, and is finite.

In discussion below, let us denote a reduced graph by an italic upper case letter, and the corresponding "adjacency matrix" (defined below) using the same letter in boldface upper case. Thus, $\mathbf{H}$ denotes the adjacency matrix for graph $H \in R_F$.

Non-zero elements of adjacency matrix $\mathbf{H}$ are defined as follows: (i) for $1 \leq i, j \leq n - \psi$, $\mathbf{H}_{ij} = 1$ if and only if $(j, i) \in H$, and (ii) $\mathbf{H}_{ii} = 1$ for $1 \leq i \leq n - \psi$. That is, non-zero elements of row $\mathbf{H}_i$ correspond to the incoming links at node i, and the self-loop at node i. Thus, the adjacency matrix for any reduced graph in R_F has a non-zero diagonal.

Based on the sufficient condition stated at the start of Section 6 and Lemma 3, we can show the following key lemmas.

Lemma 4. *For any $H \in R_F$, $\mathbf{H}^{\mathbf{n}-\psi}$ has at least one non-zero column.*

Proof. $G(\mathcal{V}, \mathcal{E})$ satisfies the sufficient condition stated at the start of Section 6. Therefore, there exists at least one non-faulty node k in the reduced graph H that has directed paths to all the nodes in H (consisting of the edges in H). Since

the length of the path from k to any other node in H is at most $n-\psi-1$, the k-th column of matrix $\mathbf{H}^{n-\psi}$ will be non-zero.[4] □

Definition 7. *For matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ of identical size, and a scalar γ, $\gamma \boldsymbol{B} \leq \boldsymbol{A}$ provided that $\gamma \boldsymbol{B}_{ij} \leq \boldsymbol{A}_{ij}$ for all i, j.*

Lemma 5. *For any $t \geq 1$, there exists a graph $H \in R_F$ such that $\beta \mathbf{H} \leq \mathbf{M}[t]$.*

Proof. Observe that the i-th row of the transition matrix $\mathbf{M}[t]$ corresponds to the state update (in Algorithm 1) performed at fault-free node i. Recall from Lemma 3 that $\mathbf{M}_{ij}[t] \geq \beta$ for $j \in \{i\} \cup ((\mathcal{V}_F - F_x(i)) \cap N_i^-)$, where $F_x(i)$ is a feasible fault set.

Let us obtain a reduced graph H by choosing $F_x(i)$ for each i as defined in Lemma 3. Then from the definition of adjacency matrix $\mathbf{H}$, Lemma 5 then follows. □

Correctness of Algorithm 1: The rest of the proof below is inspired by related work on non-fault-tolerant consensus [6].

Let $\mathbf{H}[t]$ denote the matrix $\mathbf{H}$ corresponding to $\mathbf{M}[t]$ as defined in Lemma 5.

Lemma 6. *For any $z \geq 1$, in the product below of $\mathbf{H}[t]$ matrices for consecutive $\tau(n-\psi)$ iterations, at least one column is non-zero.*

$$\mathit{\Pi}_{t=z}^{z+\tau(n-\psi)-1} \mathbf{H}[t]$$

Proof. Since the above product consists of $\tau(n-\psi)$ adjacency matrices corresponding to graphs in $R_{\mathcal{F}}$, at least one of the adjacency matrices corresponding to the τ distinct graphs in $R_{\mathcal{F}}$, say matrix $\mathbf{H}_*$, will appear in the above product at least $n-\psi$ times.

Now observe that: (i) By Lemma 4, $\mathbf{H}_*^{n-\psi}$ contains a non-zero column, say the k-th column is non-zero, and (ii) all the $\mathbf{H}[t]$ matrices in the product contain a non-zero diagonal. These two observations together imply that the k-th column in the above product is non-zero. □

Let us now define a sequence of matrices $\mathbf{Q}(i)$, $i \geq 1$, such that each of these matrices is a product of $\tau(n-\psi)$ of the $\mathbf{M}[t]$ matrices. Specifically,

$$\mathbf{Q}(i) = \mathit{\Pi}_{t=(i-1)\tau(n-\psi)+1}^{i\tau(n-\psi)} \mathbf{M}[t] \tag{10}$$

From (6) and (10) observe that

$$v[k\tau(n-\psi)] = \left(\mathit{\Pi}_{i=1}^{k} \mathbf{Q}(i) \right) v[0] \tag{11}$$

[4] That is, all the elements of the column will be non-zero. Also, such a non-zero column will exist in $\mathbf{H}^{n-\psi-1}$, too. We use the loose bound of $n-\psi$ to simplify the presentation.

Lemma 7. *For $i \geq 1$, $\mathbf{Q}(i)$ is a scrambling row stochastic matrix, and*

$$\lambda(\mathbf{Q}(i)) \leq 1 - \beta^{\tau(n-\psi)}.$$

Proof. $\mathbf{Q}(i)$ is a product of row stochastic matrices ($\mathbf{M}[t]$); therefore, $\mathbf{Q}(i)$ is row stochastic. From Lemma 5, for each $t \geq 1$,

$$\beta\, \mathbf{H}[t] \;\leq\; \mathbf{M}[t]$$

Therefore,

$$\beta^{\tau(n-\psi)}\; \Pi_{t=(i-1)\tau(n-\psi)+1}^{i\tau(n-\psi)}\; \mathbf{H}[t] \;\leq\; \Pi_{t=(i-1)\tau(n-\psi)+1}^{i\tau(n-\psi)}\; \mathbf{M}[t] \;=\; \mathbf{Q}(i)$$

By using $z = (i-1)(n-\psi)+1$ in Lemma 6, we conclude that the matrix product on the left side of the above inequality contains a non-zero column. Therefore, $\mathbf{Q}(i)$ on the right side of the inequality also contains a non-zero column.

Observe that $\tau(n-\psi)$ is finite, and hence, $\beta^{\tau(n-\psi)}$ is non-zero. Since the non-zero terms in $\mathbf{H}[t]$ matrices are all 1, the non-zero elements in $\Pi_{t=(i-1)\tau(n-\psi)+1}^{i\tau(n-\psi)}\mathbf{H}[t]$ must each be ≥ 1. Therefore, there exists a non-zero column in $\mathbf{Q}(i)$ with all the elements in the column being $\geq\; \beta^{\tau(n-\psi)}$. Therefore, by Lemma 2, $\lambda(\mathbf{Q}(i)) \leq 1 - \beta^{\tau(n-\psi)}$, and $\mathbf{Q}(i)$ is a scrambling matrix. □

Theorem 2. *Suppose that $G(\mathcal{V}, \mathcal{E})$ satisfies the sufficient condition stated above. Algorithm 1 satisfies both the validity and convergence conditions.*

Proof. Since $v[t] = \mathbf{M}[t]\, v[t-1]$, and $\mathbf{M}[t]$ is a row stochastic matrix, it follows that Algorithm 1 satisfies the validity condition.

Using Lemma 1 and the definition of $\mathbf{Q}(i)$, and using the inequalities $\lambda(\mathbf{M}[t]) \leq 1$ and $\lambda(\mathbf{Q}(i)) \leq (1 - \beta^{\tau(n-\psi)}) < 1$, we get

$$\lim_{t\to\infty} \delta(\Pi_{i=1}^{t}\mathbf{M}[i]) \;=\; \lim_{t\to\infty} \delta\left(\left(\Pi_{i=(\lfloor \frac{t}{\tau(n-\psi)} \rfloor)\tau(n-\psi)+1}^{t}\mathbf{M}[i]\right)\left(\Pi_{i=1}^{\lfloor \frac{t}{\tau(n-\psi)} \rfloor}\mathbf{Q}(i)\right)\right)$$

$$\leq \lim_{t\to\infty} \Pi_{i=1}^{\lfloor \frac{t}{\tau(n-\psi)} \rfloor}\lambda(\mathbf{Q}(i)) \;=\; 0$$

Thus, the rows of $\Pi_{i=1}^{t}\mathbf{M}[i]$ become identical in the limit. This observation, and the fact that $v[t] = (\Pi_{i=1}^{t}\mathbf{M}[i])v[0]$ together imply that the states of the fault-free nodes satisfy the convergence condition. □

7 Summary and Discussion

This paper considers a *generalized* fault model [7,9,5], which can be used to specify more complex failure patterns, such as correlated failures or non-uniform node reliabilities. Under this fault model, we prove a necessary condition for the existence of synchronous iterative approximate Byzantine consensus algorithms in arbitrary directed graphs. Then, we show the condition is also sufficient by providing a new IABC algorithm.

We present a transition matrix-based proof to show the correctness of the proposed algorithm. While transition matrices have been used to prove correctness of non-fault-tolerant consensus [6], this paper is the first to extend the technique to Byzantine consensus.

There are two main open problems: (i) for arbitrary graph, finding the global fault-tolerance parameter f, and (ii) analyzing the communication and computation complexity of Algorithm 1.

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 (2005)
2. Dasgupta, S., Papadimitriou, C., Vazirani, U.: Algorithms. McGraw-Hill Higher Education (2006)
3. Dolev, D., Lynch, N.A., Pinter, S.S., Stark, E.W., Weihl, W.E.: Reaching approximate agreement in the presence of faults. J. ACM 33, 499–516 (1986)
4. Hajnal, J., Bartlett, M.S.: Weak ergodicity in non-homogeneous markov chains. In: Proceedings of the Cambridge Philosophical Society (1958)
5. Hirt, M., Maurer, U.: Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In: PODC 1997 (1997)
6. Jadbabaie, A., Lin, J., Morse, A.: Coordination of groups of mobile autonomous agents using nearest neighbor rules. IEEE Transactions on Automatic Control 48(6), 988–1001 (2003)
7. Junqueira, F.P., Marzullo, K.: Synchronous consensus for dependent process failures. In: ICDCS 2003 (2003)
8. Koo, C.-Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: Proc. ACM Symp. on Principles of Distributed Computing, PODC 2004 (2004)
9. Kuznetsov, P.: Understanding non-uniform failure models. Bulletin of the European Association for Theoretical Computer Science (BEATCS) 106, 53–77 (2012)
10. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. on Programming Languages and Systems (1982)
11. LeBlanc, H., Zhang, H., Sundaram, S., Koutsoukos, X.: Consensus of multi-agent networks in the presence of adversaries using only local information. In: HiCoNs 2012 (2012)
12. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
13. Tseng, L., Vaidya, N.H.: Iterative approximate byzantine consensus under a generalized fault model. Technical report, CSL, UIUC (2012)
14. Vaidya, N.H.: Matrix representation of iterative approximate byzantine consensus in directed graphs. Technical report, CSL, UIUC (2012)
15. Vaidya, N.H., Tseng, L., Liang, G.: Iterative approximate byzantine consensus in arbitrary directed graphs. In: PODC 2012 (2012)
16. Wolfowitz, J.: Products of indecomposable, aperiodic, stochastic matrices. In: Proc. of the American Mathematical Society, vol. 14, pp. 733–737 (1963)

A Scalable Byzantine Grid

Alexandre Maurer and Sébastien Tixeuil

UPMC Sorbonne Universités, France
{Alexandre.Maurer,Sebastien.Tixeuil}@lip6.fr

Abstract. Modern networks assemble an ever growing number of nodes. However, it remains difficult to increase the number of channels per node, thus the maximal degree of the network may be bounded. This is typically the case in grid topology networks, where each node has at most four neighbors. In this paper, we address the following issue: if each node is likely to fail in an unpredictable manner, how can we preserve some global reliability guarantees when the number of nodes keeps increasing unboundedly ?

To be more specific, we consider the problem or reliably broadcasting information on an asynchronous grid in the presence of Byzantine failures – that is, some nodes may have an arbitrary and potentially malicious behavior. Our requirement is that a constant fraction of correct nodes remain able to achieve reliable communication. Existing solutions can only tolerate a fixed number of Byzantine failures if they adopt a worst-case placement scheme. Besides, if we assume a constant Byzantine ratio (each node has the same probability to be Byzantine), the probability to have a fatal placement approaches 1 when the number of nodes increases, and reliability guarantees collapse.

In this paper, we propose the first broadcast protocol that overcomes these difficulties. First, the number of Byzantine failures that can be tolerated (if they adopt the worst-case placement) now increases with the number of nodes. Second, we are able to tolerate a constant Byzantine ratio, however large the grid may be. In other words, the grid becomes scalable. This result has important security applications in ultra-large networks, where each node has a given probability to misbehave.

Keywords: Byzantine failures, Networks, Broadcast, Fault tolerance, Distributed computing, Protocol, Random failures.

1 Introduction

As modern networks grow larger and larger, their components become more likely to fail. Indeed, some nodes can be subject to crashes, attacks, bit flips, etc. Many models of failures and attacks have been studied so far, but the most general one is the *Byzantine* model [11]: the failing nodes behave arbitrarily. In other words, we must anticipate the most malicious strategy they could adopt. This encompasses all other possible types of failures, and has important security applications.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 87–101, 2013.

In this paper, we study the problem of reliably broadcasting information in a network despite the presence of Byzantine failures. This is a difficult problem, as a single Byzantine node, if not neutralized, can potentially lie to the entire network. Our objective is to design a broadcast protocol that prevent or limit the diffusion of malicious messages.

Related works. Many Byzantine-robust protocols are based on *cryptography* [3,5]: the nodes use digital signatures or certificates. Therefore, the correct nodes can verify the validity of received informations and authenticate the sender across multiple hops. However, this approach may not be as general as we want, as the malicious nodes are supposed to ignore some cryptographic secrets: therefore, their behavior is not *completely* arbitrary. Besides, cryptographic operations require the presence of a trusted infrastructure that deals with public and private keys: if this infrastructure fails, the whole network fails. Yet, we would like to consider that *any* component can fail. For these reasons, we focus on cryptography-free solutions.

Cryptography-free solutions have first been studied in completely connected networks [11,1,12,13,17]: a node can directly communicate with any other node, which implies the presence of a channel between each pair of nodes. Therefore, these approaches are hardly scalable, as the number of channels per node can be physically limited. We thus study solutions in multihop networks, where a node must rely on other nodes to broadcast informations.

A notable class of algorithms tolerates Byzantine failures with either space [15,18,21] or time [14,9,8,7,6] locality. Yet, the emphasis of space local algorithms is on containing the fault as close to its source as possible. This is only applicable to the problems where the information from remote nodes is unimportant (such as vertex coloring, link coloring or dining philosophers). Also, time local algorithms presented so far can hold at most one Byzantine node and are not able to mask the effect of Byzantine actions. Thus, the local containment approach is not applicable to reliable broadcast.

It has been shown that, for agreement in the presence of up to k Byzantine nodes, it is necessary and sufficient that the network is $(2k+1)$-connected, and that the number of nodes in the system is at least $3k+1$ [4]. Also, this solution assumes that the topology is known to every node, and that nodes are scheduled according to the synchronous execution model. Both requirements have been relaxed [19]: the topology is unknown and the scheduling is asynchronous. Yet, this solution retains $2k+1$ connectivity for reliable broadcast and $k+1$ connectivity for detection (the nodes are aware of the presence of a Byzantine failure). In sparse networks such as a grid (where a node has at most four neighbors), both approaches can cope only with a single Byzantine node, independently of the size of the grid.

Another existing approach is based, not on connectivity, but on the fraction of Byzantine neighbors per node. Broadcast protocols have been proposed for nodes organized on a grid [10,2]. However, the wireless medium typically induces much more than four neighbors per node, otherwise the broadcast does not work. Both approaches are based on a local voting system, and perform correctly if

every node has strictly less than a 1/4 fraction of Byzantine neighbors. This result was later generalized to other topologies [20], assuming that each node knows the global topology. Again, in weakly connected networks, this constraint on the proportion of Byzantine nodes in any neighborhood may be difficult to assess.

All aforementioned results rely on strong connectivity and Byzantine proportions assumptions in the network. In other words, tolerating more Byzantine failures requires to increase the number of channels per node, which may be difficult or impossible when the size of the network increases. To overcome this difficulty, an alternate approach has been proposed [16]. The idea is to make a small concession to the problem: we now aim at reliable communication, not between *all* correct nodes, but between *most* correct nodes. In other words, we now accept that a small minority of correct nodes can be fooled by the Byzantine nodes. This is not unrealistic, as we already accepted the idea that some nodes can fail unpredictably (being hit by Byzantine failures). This approach has been shown very efficient when the Byzantine failures are randomly distributed. This is the case, for instance, in a peer-to-peer overlay (the malicious nodes do not choose their localization when they join the overlay), or if we consider that each node has a given probability of failure.

All existing approaches have the same weak point: if the number of channels per node (degree) is bounded, a fixed number of Byzantine nodes can destabilize the whole network. Indeed, if they adopt a sufficiently close formation, they can pretend to be the source node, and lie to *any* other node – thus, we cannot even ensure that *most* correct nodes communicate reliably. Besides, if each node has a given probability to be Byzantine, the probability that such a fatal formation exists approaches 1 when the number of nodes increases. Therefore, these approaches are hardly scalable when the maximal degree is bounded.

Our contribution. In this paper, we propose the first broadcast protocol that overcomes these difficulties on a specific degree-bounded topology: the grid, where each node has at most four neighbors. For this protocol, the diameter of the grid can only have discrete values, but can be as large as we want. As in [16], our requirement is that a constant fraction of correct nodes achieves reliable communication. We show that the number of Byzantine failures that can be tolerated (if they adopt the worst-case placement) increases with the number of nodes: in other words, for the first time, this number is not limited by the maximal degree or the connectivity of the network. Besides, if we assume a constant rate of Byzantine failures (each node has the same probability to be Byzantine), the expected reliable fraction of the network is always the same, however large the grid may be. This may have applications in large-scale networks, where each node has a given probability to fail: we can now increase the size of the network indefinitely, and yet preserve the same reliability guarantees.

The paper is organized as follows. In Section 2, we describe the network topology (a sequence of grid networks that may be as large as we want) and the broadcast protocol to execute on it. In Section 3, we adopt the point of view of an omniscient observer that knows the positions of Byzantine nodes, and give a

methodology to determine a *reliable node set* - that is, a set of nodes that always communicate reliably, in any possible execution. At last, in Section 4, we use the aforementioned methodology to prove the claims.

2 Our Algorithm

In this section, we define a class of grid networks and the broadcast protocol to execute on.

2.1 Hypotheses

The network is constituted by a set of processes, called *nodes*. Some pairs of nodes are linked by a communication channel – we call them *neighbors* – and can exchange messages. Each node of the network has a unique identifier, which is its position on the grid. A node, upon receiving a message from a neighbor, knows the identifier of this neighbor. The network is asynchronous: any message sent is eventually received, but it can be at any time.

2.2 Network Topology

Let $N = 10$. Our broadcast protocol is defined for the networks G_k, $\forall k \geq 1$, G_k being a $N^k \times N^k$ *grid*. These networks may be as large as needed.

Definition 1 (Grid network). *An $M \times M$ grid is a network such that:*

- *Each node has a unique identifier (i, j) with $0 \leq i < M$ and $0 \leq j < M$.*
- *Two nodes (i_1, j_1) and (i_2, j_2) are neighbors if and only if one of these two conditions is satisfied:*
  - *$i_1 = i_2$ and $|j_1 - j_2| = 1$.*
  - *$j_1 = j_2$ and $|i_1 - i_2| = 1$.*

According to our hypotheses, each node knows its identifier (i, j) on the grid, and the identifier (i, j) of its neighbors. Each node of G_k also knows N and k.

2.3 Informal Description of the Protocol

Our broadcast protocol (BP) is defined by induction: we use an existing BP on G_1, then use the BP of G_k to define the BP of G_{k+1}. The idea is to associate a cluster of G_{k+1} to each node of G_k. Let $G(p)$ be the cluster associated to a node p (we call it *macro-node*). This is illustrated in Figure 1. The goal of a macro-node $G(p)$ is to simulate the behavior of p, so that we obtain a macroscopic BP in G_{k+1}. Then, when a node u of $G(p)$ wants to broadcast a message m in G_{k+1}:

1. First, u broadcasts m in $G(p)$ with a local BP.
2. Then, $G(p)$ broadcasts m in G_{k+1} with the macroscopic BP.

The interest of this inductive definition lies in its Byzantine-resilience properties. These properties are studied in Section 3.

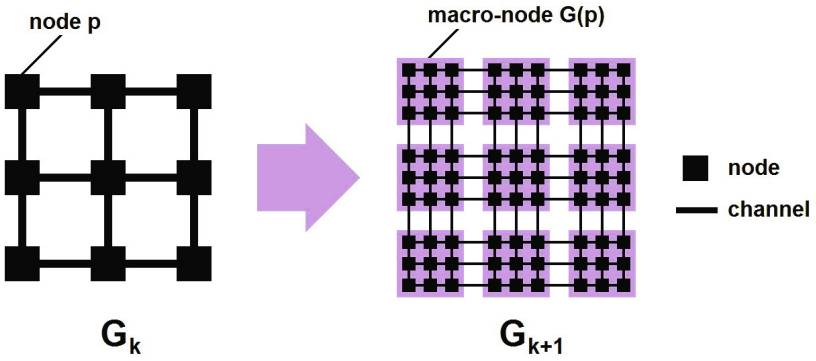


Fig. 1. Association of a macro-node of G_{k+1} to each node of G_k

2.4 Complete Description of the Protocol

The BP executed on G_1 is the *Control Zone Protocol* (CZP) proposed in [16]. Let us give the methodology to construct the BP of G_{k+1} with the BP of G_k. For this purpose, we first give an algorithm to communicate between two macro-nodes (*macro-channel*), then use it to construct the macroscopic BP.

Macro-node. To each node p of G_k, we associate a cluster $G(p)$ of G_{k+1}, called macro-node. Let (i, j) be the identifier of p. Then, $G(p)$ is the $N \times N$ grid such that the node $(0, 0)$ of $G(p)$ corresponds to the node (Ni, Nj) of G_{k+1}.

Macro-channel. Let p and q be two neighbor nodes in G_k. We give an algorithm to tranfer messages from $G(p)$ to $G(q)$, as if they were two neighbor nodes linked by a channel.

First, we execute the CZP on both $G(p)$ and $G(q)$, to enable local broadcast inside each macro-node. The following algorithm enables to send a message m, known by the nodes of $G(p)$, to the nodes of $G(q)$. Let $Border(p)$ (resp. $Border(q)$) be the set of nodes of $G(p)$ (resp. $G(q)$) having a neighbor in $G(q)$ (resp. $G(p)$).

1. The nodes of $Border(p)$ send m to their neighbor in $Border(q)$.
2. The nodes of $Border(q)$, upon receiving m from their neighbor in $Border(q)$, broadcast m in $G(q)$ with the CZP.
3. The nodes of $G(q)$, upon receiving strictly more than $N/2$ distinct messages (v_i, m) trough the CZP with $v_i \in Border(q)$, accept m.

We associate a dynamic set Sen_q to each node of $G(p)$ (storing the message to send), and a dynamic set Rec_p to each node of $G(q)$ (storing the messages received). We execute this algorithm for each pair of neighbor macro-nodes. This mechanism is illustrated in Figure 2.

Macroscopic BP. For each node p of G_k, all nodes of $G(p)$ execute the same algorithm than p, with the two following modifications:

1. When the algorithm requires to send a message m to a neighbor q, add m to Sen_q.
2. When a message m is added to the set Rec_q, consider that m was received from q.

Now, let s be a node of $G(p)$ that wants to broadcast a message m in G_{k+1}. First, s broadcasts (s, m) in $G(p)$ with the CZP. Then, upon receiving (s, m), the nodes of $G(p)$ broadcast (s, m) with the macroscopic BP. Thus, the nodes receiving (s, m) know that s broadcast m: we now have a BP on G_{k+1}.

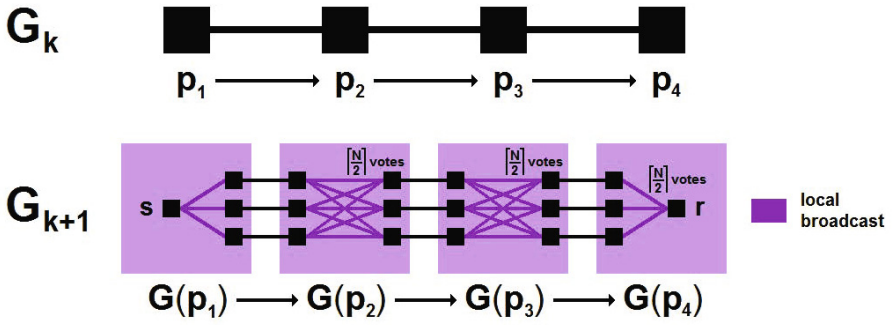


Fig. 2. Principle of the protocol

3 Construction of a Reliable Node Set

In this section, we now assume that some nodes are Byzantine, and behave arbitrarily instead of following the aforementioned protocol. We adopt the point of view of an omniscient external observer, knowing the positions of Byzantine nodes, and give a methodology to determine a *reliable node set* - that is, a set of nodes that communicate reliably in any possible execution. This methodology is used in Section 4 to prove the claims. Notice that we never require that a node determines such a set: this is just a global view of the system.

Notion of reliable node set. The nodes following the aforementioned protocol are called *correct*. The correct nodes do not know the positions of Byzantine nodes.

Definition 2 (Reliable node set). *For a given broadcast protocol (BP), a set of correct nodes is* reliable *if, for each pair of nodes s and r of this set:*

1. *If s broadcasts m, r eventually accepts (s, m).*
2. *If r accepts (s, m), r necessarily broadcast m.*

In other words, a reliable node set behaves like a network without Byzantine failures. The item (1) guarantees that the nodes always manage to communicate. The item (2) guarantees that no node of the reliable set can be fooled - for instance, if a Byzantine node broadcasts (s, m') to make the network believe that s broadcast m'.

Construction of a reliable node set. Let $Corr$ be a set of correct nodes of G_k. Let us define a function Rel_k such that $Rel_k(Corr)$ returns a reliable node set for our BP. For this purpose, we first introduce some new elements.

In [16], we gave a methodology to determine a reliable node set for the CZP on an $N \times N$ grid, for a given set $Corr_0$ of correct nodes. Let Rel_{CZP} be a function such that $Rel_{CZP}(Corr_0)$ returns a reliable node set for the CZP.

At last, we introduce the notion of *correct macro-node*. In broad outline, a correct macro-node behaves like a correct node in the macroscopic BP. This intuitive idea is the key element of the next theorem.

Definition 3 (Correct macro-node). *Let there be an $N \times N$ grid with a distribution $Corr_0$ of correct nodes. This grid (or macro-node) is said* correct *if each side of the grid (up, down, right and left), among its N nodes, has strictly more than $3N/4$ nodes in $Rel_{CZP}(Corr_0)$.*

The underlying idea of this definition is the following: the reliable node sets of two adjacent correct macro-nodes are always connected by a majority of channels (strictly more than $N/2$). Therefore, the messages exchanged between these two reliable sets always receive a majority of votes. This idea is illustrated in Figure 3, and used in the proof below.

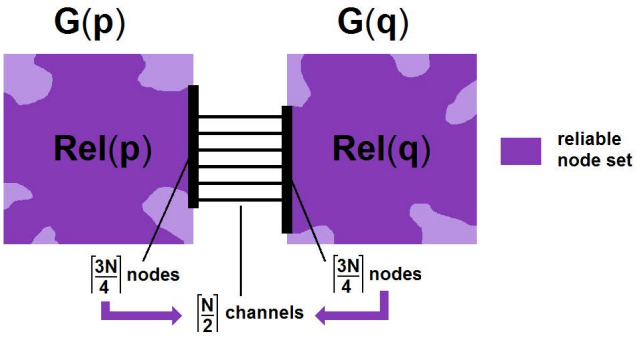


Fig. 3. Reliable communication between 2 correct macro-nodes

We can now define the function Rel_k by induction, $\forall k \geq 1$:

- $Rel_1 = Rel_{CZP}$
- $Rel_{k+1}(Corr) = \bigcup_{p \in Rel_k(Corr')} Rel_{CZP}(Corr(p))$, where ...

- $Corr$ is a distribution of correct nodes on G_{k+1}.
- $Corr(p)$ is the corresponding distribution on $G(p)$.
- $Corr'$ is the set of nodes p of G_k such that $G(p)$ is a correct macro-node.

In the following, we refer to $Rel_{CZP}(Corr(x))$ by $Rel(x)$.

Theorem 1. *$\forall k \geq 1$, if $Corr$ is a distribution of correct nodes on G_k, then $Rel_k(Corr)$ is a reliable node set for our BP.*

Proof. The main idea of the proof is to show an equivalence between the execution on G_{k+1} and a virtual execution on G_k (this, of course, does not mean that G_k must actually exist for G_{k+1} to work).

The proof is by induction. The property is true at rank 1 by definition. Now, let us suppose that the property is true at rank k, and show that it is true at rank $k+1$. Let $Corr$ be a distribution of correct nodes on G_{k+1}, and let s and r be two nodes of $Rel_{k+1}(Corr)$. Let us suppose that s broadcasts m in G_{k+1}. Then, to show that $Rel_{k+1}(Corr)$ is a reliable node set, we show that the items (1) and (2) of Definition 2 are satisfied.

1. We call *accumulative* a distributed algorithm where each node holds a given number of dynamic sets $S_1, S_2, S_3 \ldots$, can only add elements to these sets ($S_i \leftarrow S_i \cup \{x\}$), and eventually executes an action when a given collection of elements has joined these sets: $(X_1 \subseteq S_1) \wedge (X_2 \subseteq S_2) \wedge \ldots$. The CZP is accumulative, and so is our BP, as it is an inductive combination of accumulative algorithms. In other words, the order of reception of messages is unimportant in our BP.

   Let p and q be the nodes of G_k such that s belongs to $G(p)$ and r belongs to $G(q)$. By definition of Rel_{k+1}, p and q belong to $Rel_k(Corr')$. Let us suppose that $Corr'$ is a distribution of correct nodes on G_k. Then, $Rel_k(Corr')$ is a reliable node set on G_k. Therefore, if p broadcasts (s, m), there exists a sequence of message receptions such that q eventually accepts (s, m). Let $(R_1, R_2, \ldots, R_M)$ be this sequence, R_i being a triplet (q_i, m_i, p_i) such that q_i receives m_i from p_i, with $p_1 = p$ and $q_M = q$. Let us prove the following property $\mathcal{P}_i$ by induction, $\forall i \in \{1, \ldots, M\}$: all the nodes of $Rel(q_i)$ eventually add m_i to Rec_{p_i}.

   – First, let us show that $\mathcal{P}_1$ is true. According to our BP, s initially broadcasts (s, m) in $G(p)$. Therefore, as $p = p_1$, all the nodes of $Rel(p_1)$ eventually accept (s, m). Then, as they execute the same alogorithm than p_1, they add m_1 to their set Sen_{q_1}.

     Let $Border(q_1)$ be the set of nodes of $G(q_1)$ having a neighbor in $G(p_1)$. As $G(q_1)$ and $G(p_1)$ are two correct macro-nodes, according to Definition 3, strictly more than $N/2$ nodes of $Rel(p_1)$ have a neighbor in $Rel(q_1)$. Therefore, strictly more than $N/2$ nodes of $Border(q_1) \cap Rel(q_1)$ eventually receive m_1, and broadcast it in $G(q_1)$. So all the nodes of $Rel(q_1)$ eventually receive strictly more than $N/2$ messages (v_x, m_1) with $v_x \in Border(q_1)$ and add m_1 to Rec_{p_1}. Thus, $\mathcal{P}_1$ is true.

– Now, let us suppose that $\mathcal{P}_j$ is true $\forall j \leq i$. Then, as the order of reception of messages is unimportant, all the nodes of $Rel(p_{i+1})$ eventually behave as p_{i+1}, and add m_{i+1} to $Sen_{q_{i+1}}$.
Thus, by a perfectly similar demonstration, $\mathcal{P}_{i+1}$ is true.

Then , as $r \in Rel(q)$, according to $\mathcal{P}_M$: r eventually receives the same messages as $q = q_M$ and accepts (s, m). Thus, the item (1) of Definition 2 is satisfied. This is illustrated in Figure 4.

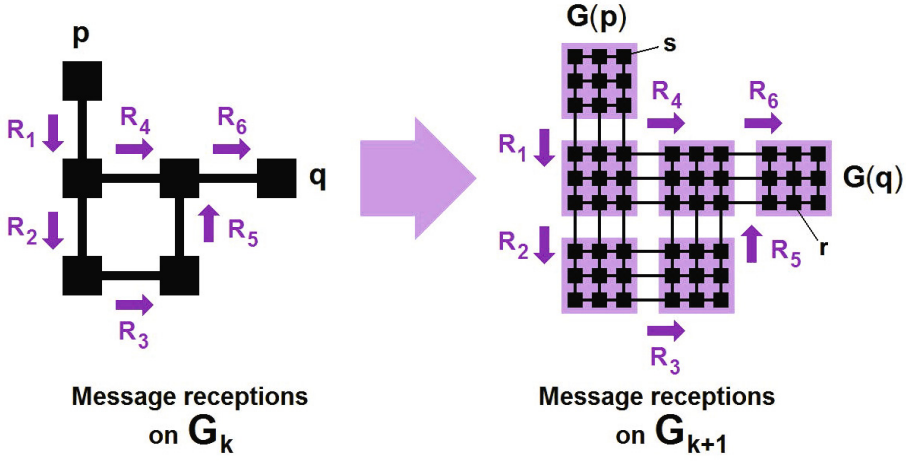


Fig. 4. Illustration of the proof (1) : what occurs in $Rel_k(Corr')$ eventually occurs in $Rel_{k+1}(Corr)$

2. The proof is by contradiction. Let us suppose the opposite: r accepts a message (s, m), yet s did not broadcast m. Let p_0 be the node of G_k such that $r \in Rel(p_0)$. If we also have $s \in Rel(p_0)$, it is impossible that r accepts (s, m), as $Rel(p_0)$ is a reliable node set. So s necessarily belongs to another macro-node. Similarly than above, let us suppose that $Corr'$ is a distribution of correct nodes on G_k. Then, as $Rel_k(Corr')$ is a reliable node set on G_k, r necessarily received a message that p_0 cannot receive in G_k. Let us show that this is impossible.

   Let u be the first node of $Rel_{k+1}(Corr)$ (possibly r), belonging to a macro-node $G(q)$, to receive a message m' that q cannot receive in G_k. Let $G(p)$ be the macro-node sending this message. If $G(p)$ is not correct (in the sense of Definition 3), then p does not belong to $Corr'$, is assumed to be Byzantine on G_k, and can actually send m' to q – so $G(p)$ is necessarily correct. It implies that u received strictly more than $N/2$ messages (v_i, m') with $v_i \in Border(q)$. As $G(p)$ and $G(q)$ are two correct macro-node, strictly more than $N/2$ nodes of $Rel(p)$ have a neighbor in $Rel(q)$. So at least one of the nodes v_i belongs to $Rel(q)$ *and* received m' from a neighbor $v \in Rel(p)$. As $Rel(p)$ is a reliable node set, the only possibility is that v received a message

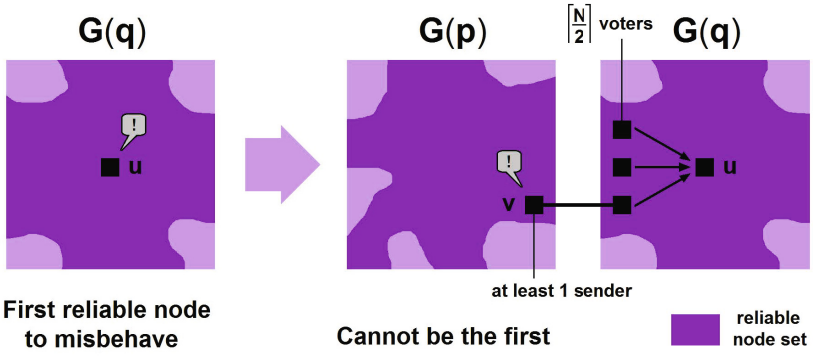


Fig. 5. Illustration of the proof (2) : a node of $Rel_{k+1}(Corr)$ cannot misbehave

that p cannot receive in G_k. So u is not the first node in this situation, which contradicts the initial statement. Thus, the item (2) of Definition 2 is satisfied. This is illustrated in Figure 5.

We now have a methodology to determine a reliable node set for a given distribution of Byzantine nodes on G_k, $\forall k \geq 1$. In the next section, we use this methodology to prove the claims.

4 Proof of the Claims

In this section, we finally prove the claims of the paper: the number of Byzantine failures that can be tolerated increases with the number of nodes (if they adopt the worst-case placement), and a constant rate of Byzantine failures can be tolerated, however large the grid may be. As in [16], our requirement to *tolerate* Byzantine failures is that a constant fraction of the network communicates reliably.

4.1 Worst-Case Placement

Let us give a minimal number of Byzantine failures that can be tolerated when they adopt an arbitrary placement (possibly the worst).

Theorem 2. *$\forall k \geq 1$, on a grid G_k with at most 2^{k-1} Byzantine failures (arbitrarily placed), the fraction of the network achieving reliable communication is at least $1 - \frac{4}{N^2}$.*

Proof. The proof is by induction. For $k = 1$, we can test all possible placements of a single Byzantine failure (as $N = 10$) and show that the property is true. Now, let us suppose that the property is true at rank k. Let there be 2^k Byzantine failures arbitrarily placed on G_{k+1}. Then, at most 2^{k-1} macro-nodes of G_{k+1}

contain more than 2 Byzantine failures. Again, by testing all possible cases, we can show that an $N \times N$ grid with at most 1 Byzantine failure is always correct in the sense of Definition 3. So at most 2^{k-1} macro-nodes are not correct. Therefore, as the property is true at rank k, the reliable node set covers at least a $1 - \frac{4}{N^2}$ fraction of macro-nodes (and in this worst case, all these macro-nodes have only correct nodes). Thus, according to the definition of Rel_{k+1}, the property is true at rank $k+1$. This is illustrated in Figure 6.

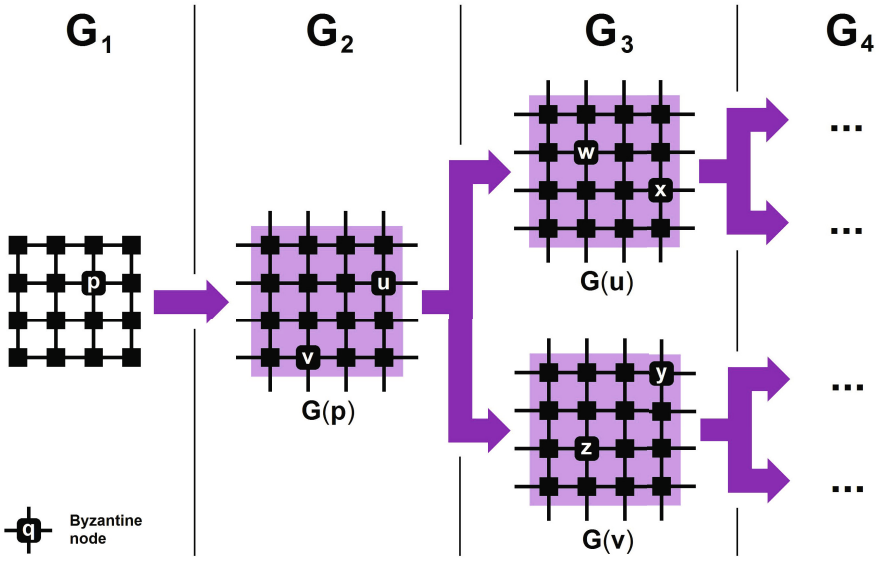


Fig. 6. Worst-case placement of 2^{k-1} Byzantine nodes on G_k

So we can always tolerate 2^{k-1} failures on G_k. As the parameter k sets the size of the grid, this number increases with the number of nodes. To our knowledge, this is the first time that this number is not limited by the connectivity or the maximal degree of the network.

4.2 Random Distribution

Let us assume a constant rate of Byzantine failures (each node has the same probability λ to be Byzantine) and give the expected reliable fraction of the network. Let $\mu = 1 - \lambda$ be the probability that a node is correct.

Theorem 3. *$\forall k \geq 1$, let $F_k(\mu)$ be the expected reliable fraction of G_k. Then, if $\mu \geq 1 - 10^{-5}$, we have $F_k(\mu) \geq 1 - 10^{-4}$.*

Proof. Let there be an $N \times N$ grid where each node has the same probability μ_0 to be correct. We call $P(\mu_0)$ the probability that the two following events occur:

1. The grid is *correct* in the sense of Definition 3.
2. A node, chosen uniformly at random, belongs to $Rel_{CZP}(Corr_0)$, $Corr_0$ being the distribution of correct nodes on the grid.

We want to prove the following property by induction: $F_k \geq \prod_{i=1}^{i=k} P^i(\mu)$, P^i being the i^{th} application of the function P. The property is true at rank 1, as $F_1(\mu) \geq P(\mu)$.

Now, let us suppose that the property is true at rank k. Let $Corr$ be the distribution of correct nodes on G_{k+1}. Let u be a randomly chosen node of G_{k+1}, and let p be the node of G_k such that u belongs to the macro-node $G(p)$. According to Theorem 1, to have $u \in Rel_{k+1}(Corr)$, it is necessary and sufficient that (1) $u \in Rel(p)$ and (2) $p \in Rel_k(Corr')$. The first event occurs with probability $P_1 \geq P(\mu)$, and if so, the second event occurs with probability $P_2 \geq F_k(P(\mu))$. Thus, $F_{k+1}(\mu) \geq P(\mu)F_k(P(\mu)) = \prod_{i=1}^{i=k+1} P^i(\mu)$: the property is true at rank $k+1$. This is illustrated in Figure 7.

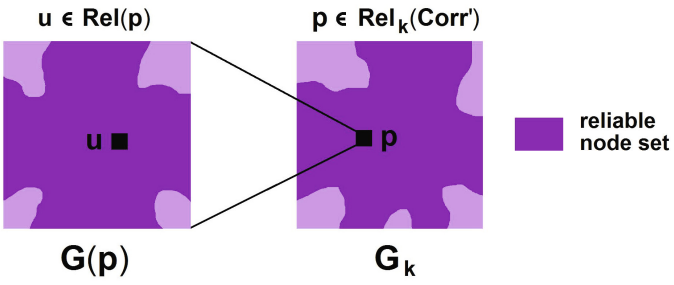


Fig. 7. Sufficient condition for u to be in $Rel_{k+1}(Corr)$

Now, let us give a lower bound of $P(\mu_0)$. We consider two disjoint cases:

1. The case where all the nodes of the $N \times N$ grid are correct, which occurs with probability $\mu_0^{N^2}$. In this case, $Rel_{CZP}(Corr_0)$ covers the whole grid, and the grid is correct in the sense of Definition 3.
2. The case where one single node is Byzantine, which occurs with probability $N^2(1-\mu_0)\mu_0^{N^2-1}$. As $N = 10$, we evaluate $Rel_{CZP}(Corr_0)$ for the 100 possible placements of the single Byzantine node. In 64 cases, this set contains 99 nodes. In 32 cases, it contains 98 nodes. In 4 cases, it contains 96 nodes. Thus, the probability that a randomly chosen correct node belongs to this set is $\alpha = \frac{64 \times 99 + 32 \times 98 + 4 \times 96}{100 \times 99} \geq \frac{199}{200}$. In all cases, the grid is correct in the sense of Definition 3. This is illustrated in Figure 8.

Fig. 8. Different cases for the placement of 1 Byzantine node on an $N \times N$ grid

So $P(\mu) \geq g(\mu) = \mu^{N^2} + \alpha N^2(1-\mu)\mu^{N^2-1}$. This function is convex $\left(\frac{\partial^2 g(\mu)}{\partial \mu^2} \leq 0\right)$ for $\mu \geq \alpha$. Let $\beta = 1 - 10^{-5} \geq \alpha$. Then, $\forall \mu \geq \beta$, $g(\mu) \geq f(\gamma, \mu) = 1 - \gamma(1-\mu)$, with $\gamma = \frac{1-g(\beta)}{1-\beta}$. Then, we easily show by induction that $\forall k \geq 1$, $P^k(\mu) \geq f(\gamma^k, \mu)$. So $F_k(\mu) \geq H_k(\mu) = \prod_{i=1}^{i=k} f(\gamma^i, \mu)$.

We now have a lower bound of $F_k(\mu)$, but it may be hard to calculate when k approaches infinity. To overcome this difficulty, let i_0 be the first integer such that, $\forall i \geq i_0$, $\gamma^i \leq \frac{1}{i^2}$. So $H_k(\mu) \geq \prod_{i=1}^{i=i_0} f(\gamma^i, \mu) \prod_{i=i_0+1}^{i=k} (1 - \frac{1-\mu}{i^2})$. Then, when k approaches infinity, we can apply the Wallis formula: $\lim_{x \to \infty} H_k(\mu) \geq \prod_{i=1}^{i=i_0} f(\gamma^i, \mu) \frac{sin(\pi\sqrt{1-\mu})}{\pi\sqrt{1-\mu}} \geq 1 - 10^{-4}$ if $\mu \geq \beta$. Thus, the result, as $H_k(\mu)$ decreases with k.

Therefore, we can hold a constant rate of Byzantine failures and yet have a constant expected fraction of reliable nodes, however large the grid may be. This may have important security applications – for instance in a computationnal grid where each processor has a given probability to misbehave. This result shows that, for a given security requirement, we can increase the size of the grid indefinetely, which could be a solution to the problem of scalability.

5 Conclusion

In this paper, we have shown that Byzantine resilience was possible in a scalable degree-bounded network. If the adversary can place the Byzantine nodes arbitrarily, then for the first time, we can tolerate a number of Byzantine failures that largely exceeds the node degree. If not (random distribution), then we can tolerate a constant fraction of Byzantine nodes, even if the size of the network approaches infinity.

We have the strong conviction that this approach (slice the network into clusters, then slice each cluster into smaller clusters, etc ...) can be generalized to

less regular topologies. Indeed, the notion of a correct macro-node (see Definition 3) can be generalized to an arbitrary graph – the key idea is that, for each interface with another macro-node, we must still have a 3/4 fraction of reliable nodes. Besides, the network diameter can only have discrete values here, but we could generalize the result to any network diameter.

References

1. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. McGraw-Hill Publishing Company, New York (1998)
2. Bhandari, V., Vaidya, N.H.: On reliable broadcast in a radio network. In: Aguilera, M.K., Aspnes, J. (eds.) PODC, pp. 138–147. ACM (2005)
3. Castro, M., Liskov, B.: Practical byzantine fault tolerance. Theoretical Computer Science TCS 243(12), 363–389 (2000)
4. Dolev, D.: The Byzantine generals strike again. Journal of Algorithms 3(1), 14–30 (1982)
5. Drabkin, V., Friedman, R., Segal, M.: Efficient byzantine broadcast in wireless ad-hoc networks. In: DSN, pp. 160–169. IEEE Computer Society (2005)
6. Dubois, S., Masuzawa, T., Tixeuil, S.: The Impact of Topology on Byzantine Containment in Stabilization. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 495–509. Springer, Heidelberg (2010)
7. Dubois, S., Masuzawa, T., Tixeuil, S.: On Byzantine Containment Properties of the *min* + 1 Protocol. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 96–110. Springer, Heidelberg (2010)
8. Dubois, S., Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. IEEE Transactions on Parallel and Distributed Systems, TPDS (2011)
9. Dubois, S., Masuzawa, T., Tixeuil, S.: Maximum Metric Spanning Tree Made Byzantine Tolerant. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 150–164. Springer, Heidelberg (2011)
10. Koo, C.-Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: Chaudhuri, S., Kutten, S. (eds.) PODC, pp. 275–282. ACM (2004)
11. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
12. Malkhi, D., Mansour, Y., Reiter, M.K.: Diffusion without false rumors: on propagating updates in a Byzantine environment. Theoretical Computer Science 299(1-3), 289–306 (2003)
13. Malkhi, D., Reiter, M., Rodeh, O., Sella, Y.: Efficient update diffusion in byzantine environments. In: The 20th IEEE Symposium on Reliable Distributed Systems (SRDS 2001), Washington, Brussels, Tokyo, pp. 90–98. IEEE (2001)
14. Masuzawa, T., Tixeuil, S.: Bounding the Impact of Unbounded Attacks in Stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 440–453. Springer, Heidelberg (2006)
15. Masuzawa, T., Tixeuil, S.: Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. International Journal of Principles and Applications of Information Science and Technology (PAIST) 1(1), 1–13 (2007)
16. Maurer, A., Tixeuil, S.: Limiting byzantine influence in multihop asynchronous networks. In: IEEE International Conference on Distributed Computing Systems, ICDCS (2012)

17. Minsky, Y., Schneider, F.B.: Tolerating malicious gossip. Distributed Computing 16(1), 49–68 (2003)
18. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: 21st Symposium on Reliable Distributed Systems (SRDS 2002), pp. 22–29. IEEE Computer Society (2002)
19. Nesterenko, M., Tixeuil, S.: Discovering network topology in the presence of byzantine nodes. IEEE Transactions on Parallel and Distributed Systems (TPDS) 20(12), 1777–1789 (2009)
20. Pelc, A., Peleg, D.: Broadcasting with locally bounded byzantine faults. Inf. Process. Lett. 93(3), 109–115 (2005)
21. Sakurai, Y., Ooshita, F., Masuzawa, T.: A Self-stabilizing Link-Coloring Protocol Resilient to Byzantine Faults in Tree Networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 283–298. Springer, Heidelberg (2005)

Collaborative Detection of Coordinated Port Scans

Roberto Baldoni, Giuseppe Antonio Di Luna, and Leonardo Querzoni

Cyber Intelligence and Information Security Center and Dipartimento di Ingegneria informatica automatica e gestionale "Antonio Ruberti"
Universitá di Roma La Sapienza, Roma, Italy
{baldoni,diluna,querzoni}@dis.uniroma1.it

Abstract. In this paper we analyze the coordinated port scan attack where a single adversary coordinates a Group of Attackers (GoA) in order to obtain information on a set of target networks. Such orchestration aims at avoiding Local Intrusion Detection Systems checks allowing each host of the GoA to send a very few number of probes to hosts of the target network. In order to detect this complex attack we propose a collaborative architecture where each target network deploys local sensors that send alarms to a collaborative layer. This, in turn, correlates this data with the aim of (i) identifying coordinated attacks while (ii) reducing false positive alarms and (iii) correctly separating GoAs that act concurrently on overlapping targets. The soundness of our approach is tested on real network traces. Tests show that collaboration among networks domains is mandatory to achieve accurate detection of coordinated attacks and sharp separation between GoAs that execute concurrent attacks on the same targets.

1 Introduction

A port scan is a specific kind of malicious activity carried out by an adversary aimed at inspecting a target network for open hosts/ports. The final adversary goal is usually to identify possible vulnerabilities in the system that can be later exploited for realizing an intrusion. Detecting this kind of activity is thus fundamental for proper network protection. In order to avoid detection the adversary employs several decoying techniques aimed at obfuscating the presence of network packets generated by the port scan activity within the regular network traffic. One of such techniques consists in coordinating a group of geographically dispersed computers (i.e., the attackers) to split the scan activities. From a defender point of view this attack appears as a set of uncorrelated connections coming from independent machines. Common network intrusion detection systems (IDSs) often fail in detecting such coordinated attacks and, in the best case, report it simply as separate scans produced by independent attackers. These coordinated attacks require high technical skill, thus, from the defender viewpoint, assessing the dimension of the attack can help in understanding the motivation and skill of the adversary.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 102–117, 2013.

Recently, Gates [1] proposed an offline solution for solving the problem of accurately identifying coordinated port scans executed against a single network domain and based on adversary modeling of the desired information gain. This solution clusters several different sources identified as scanners in order to provide a coherent view of the coordinated scan. Even though results are quite encouraging, Gates's solution could fail in identifying scans executed by *groups of attackers* (GoAs) targeting machines/ports located in different networks domains or, in the best case it could detect part of the attackers if the scan activity done in the single network domain is above some thresholds. Thus, as a matter of fact, the problem of providing an integrated solution to correctly identifying coordinated port scans that carry out their activities over multiple network domains is still an open problem.

This paper focuses on such type of coordinated port scans and provides a novel approach based on the collaboration among independent network domains. Our solution deploys a local system at each network domain that monitors and stores data on failed connections and, if the number of connection errors within the network domain goes above a given threshold, triggers the construction of a *local attack graph* retrieving historical information on failed connections. Interestingly, the amount of information retrieved depends on the speed of the scan activity (e.g., if the threshold has been quickly reached, the scan activity is suspected to be fast, then failed connections within a small time window will be retrieved from storage). The local attack graph provides potential group of attackers at the collaborative layer. The latter collects all this information and builds a *collaborative graph* that identifies GoAs even if they are executing concurrent and (partially) overlapping scan activities. Contrarily to [1], this approach is on-line in the sense that it continuously monitors failed connections and, if there is a certain amount of suspect scan activities on the network domain, it launches the next steps towards the detection of GoAs. The contributions presented in this paper are:

- a formal model of the local network domain that is able both to express an estimated failure threshold for coordinated port scans and to define a local graph that isolates potential attackers into connected components;
- concurrent use of graph clustering and separation techniques, similarity-based clustering [2] and Parallel Reactive GRASP [3] respectively, at the collaborative layer to identify and separate GoAs.
- the realization of a java-based prototype of the collaborative port scan detection system and its evaluation based on real runs and GoAs of 800 hosts.

An interesting outcome of the performance evaluation is a tradeoff between GoA detection accuracy and the number of network domains participating to the collaboration. In most of the cases (around 70%) the collaborative system is also able to separate concurrent GoAs of 200 hosts each. The rest of the paper is structured as follows: Section 2 presents the related work. Section 3 provides an overview of the problem and Section 4 introduces the network domain model. Section 5 presents the collaborative layer and Section 6 the System Architecture. Section 7 reports on the results of the evaluation and Section 8 points out

limitations and future perspectives. Finally, Section 9 draws the conclusion of this work.

2 Related Work

The idea of a coordinated port scan traces back to 1999 [4] even if it was not considered by literature on security till three years later. Apart from [1] the other works that address it are:

Staniford et al. [5] in 2002 introduced the concept of *distributed scanning* and provided a solution constituted by two subsystems: a network anomaly detector (Spade) and a correlation engine (Spice). The paper does not propose any experimental results for the detection of coordinated port scan, so it is not clear how well this approach could be effective in our case study.

The solution proposed by Conti and Abdullah in [6] heavily relies on human-analysis to find suspicious patterns in the normal traffic, and it is not clear how those patterns will be modified by the normal network activity.

Robertson et al. [7] proposed a method for detecting coordinated port scans under the assumption that the attack sources are on the same subnet. Even if this assumption sounds reasonable it somewhat limits the applicability of this solution.

A real collaborative project is Dshield [8]. DShield could be seen as a world-wide sensor network, that collects data coming from firewalls and IDSs of volunteers spread all over the world. Data collected by the DShield system is sent to the Internet Storm Center, inserted in the central database and analyzed by the staff.

Using data from DShield, Yegneswaran et al. [9] have done a general analysis focused on the issues of distribution, categorization and prevalence of portscans. They found that a large proportion of the daily scans are coordinated or come from distributed sources.

In [10] have been discussed the collaborative approach to protection of critical infrastructure. However they do not address the case of coordinated portscan.

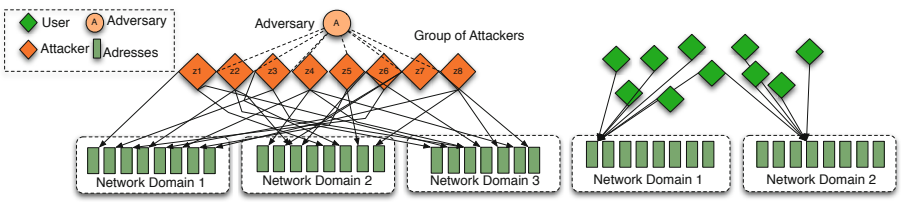


Fig. 1. (a) pattern of probes of an adversary that coordinates a set of attackers/zombies scanning addresses on multiple network domain; (b) pattern of probes of users contacting addresses of services on multiple network domain

3 Problem Overview

The scenario we consider is constituted by several network domains, possibly managed by independent administrators. An adversary is willing to obtain information on some addresses (ports) of the hosts (also called target addresses) within these networks using port scanning. Thus the adversary coordinates a set of attackers by splitting the set of addresses in order each attacker has different targets to scan. This set is also called Group of Attacker (GoA).

The adversary wants to obtain this information within a predefined timeframe, i.e. it is not willing to wait an unreasonable amount of time for a port scan to end; at the same time it wants to remain undetected, i.e. it doesn't want network administrators to be able to correctly identify that separate scan activities are originated by a single adversary. Figure 1.a shows a pattern of such coordinated port scanning while Figure 1.b shows a normal use of network addresses by non-malicious activities. This figure could represent a pattern of requests sent to two web servers.On the administrator side, the goal is to collect and correlate enough information such that scan activities originating from different sources can be correctly clustered. Ideally, each cluster should contains scan sources pertaining only to a single GoA: the result should be both complete, all machines in the GoA are included in the cluster, and accurate, no machine external to the GoA is included in the cluster. IDSs deployed within networks can identify attacker only on the basis of local information and raise alarms. Sometimes this information is not enough to correctly separate distinct attacker or merge group of source addresses belonging to a single attack. To this aim, we advocate a cooperative approach: local network domains send their alarms toward a centralized coordination node, where this data is aggregate in order to have a more complete view on the composition of GoA and thus on the number of attackers.

4 Network Domain Model

A Network N can bee seen as a set of addresses $N = \{\bigcup_{\forall j \forall c} x_j^c\}$ where c is the portnumber and j is the hostname. The state of each address at a given time t can be *closed*, *open* or *undefined* (i.e. $\{c, o, u\}$). An address x_j^c is in the o state if at time t if it replies with a `SYN-ACK` to any `SYN` packet sent to it. If the state is c it means that the address replies with `RST-ACK` to any `SYN` packet. Finally, the address is in the u state if there is no active service on it or because there is a firewall blocking its connections. The function $s_t : N \rightarrow \{c, o, u\}$ maps each address of N to the corresponding state. Two addresses x_i^k, x_j^t belong to the same host if and only if $i = j$. In the following we consider an address x_j^c as **k-active** at time t if $\exists t' \in [t-k, t] | s_{t'}(x_j^c) = o$. Otherwise x_j^c is **k-closed**. Therefore, we can define the two set of $k - active$ K_t^a and $k - closed$ K_t^c addresses.

Observer. The observer is an entity able to see and inspect any packet transmitted on the monitored network N and unable to monitor the attacker source network N_A. The observer has limited computational power and can observe the

network for a limited time window Δ_o. As an example the observer can compute on-the-fly which ports are $k - active$ and $k - closed$ if $k < \Delta_o$.

Adversary. An adversary A is an entity that wants to know the state of a set of targets S in N. The set S can be build using a sampling function $\psi_t : N \to \mathcal{P}(N)$. Commonly adopted ψ functions are [5]:

ψ_v **vertical scan:** $S = \{x_j^c \mid j = h \quad c \in P\}$. All the addresses belong to the same host.

ψ_h **horizontal scan:** $S = \{x_j^c \mid c = h \quad j \in H\}$. All the addresses share the same port but are distributed among different hosts.

ψ_s **strobe scan:** $S = \{x_j^c \mid j \in H \quad c \in P\}$. The same set of ports P is tested over a set of hosts H.

Let us remark that it is always possible for an adversary to use a sampling function ψ_k that is totally random. In the following, as in [1], we consider only adversaries that are carrying out scan activities using contiguous addresses as a target. In order to identify these specific sampling functions, we introduce the symbol $\overline{\psi}$; two addresses x_j^k, x_i^t are contiguous if $k - t = 1 \wedge i = j$ or $k = t \wedge i - j = 1$. Each adversary A has a limited amount of time useful to accomplish the scanning. This assumption is reasonable because, as an example, the results of a scan can easily become useless if the scan lasts for a too long period. We call Δ_A the maximum time window in which the adversary is supposed to accomplish the scan. If $c = |S|$ is the size of the target set we can define the minimum request rate that the adversary has to sustain during the scanning as: $\rho_A^c = \frac{c}{\Delta_A}$

Coordinated Port Scanning. A *coordinated port scan* is a port scan activity in which the target set S is probed by a set of attackers $Z = \{z_1, ..., z_n\}$ orchestrated by the adversary A. *The problem of the adversary is to carry out the coordinated port scan without being noticed by the observer which means that the number of probes sent by each attacker should be below a certain threshold.*We define the assignment function: $\phi : Z \to \mathcal{P}(S)$ This function assigns to each attacker z_j the set $\phi(z_j)$ containing the subset of addresses in S that z_j will probe during the attack and such that $S = \bigcup_{j=1}^{n} \phi(z_j)$. We can now define the average number of probes for an attacker as: $\overline{c_z} = \sum_{j=1}^{n} \frac{|\phi(z_j)|}{|Z|}$

Here we assume that $\overline{0 < c_z} < t_s$ where t_s is the minimum number of requests that results in a scan detection by the observer.

Failures and Failure Network Threshold. Let now consider an observer o using a time window size Δ_o and an adversary A carrying out a port scan activity over S. For the observer it is possible to compute the density ρ_{active} of $k - active$ ports over N. So the observer can estimate the density of $k - closed$ ports $1 - \rho_{active}$, this implies for an adversary with a fixed request rate $\rho_A'^c$ scanning the network N, that the observer should see during Δ_o a number of failures that is:

$$\overline{P} = \rho_A'^c (1 - \rho_{active}) \Delta_o \rho_t^{S_f} \geq \overline{t(\rho_A'^c, \Delta_o)} = \frac{\rho_A'^c (1 - \rho_{active}) \Delta_o}{2}$$

$\overline{t(\rho'^{c}_{A}, \Delta_o)}$ represents the **estimated failure threshold** for the network and it is a function of both the observation window and the adversary rate. In other words, this threshold is not considered for failures caused by single source attacks like in [1], but rather it is considered for all the failures happening in the observed network N, additional details on how to compute this threshold ca be found in [11].

Local Attack Graph. An attack executed against a network can be represented through a *Source-Errors* graph, namely Local Attack Graph LAG, that has some important properties. Assume A starts a coordinated port scan towards $S \subseteq N$ at time t_0 and that this attack will end at time $t_0+\Delta_A$. Without loss of generality, let us assume that this attack is an horizontal port scan. Consider F^c, that is the set of connection failures directed towards a *k-closed* port c at time $t_0 + \Delta_a$, and $F^c(x)$, the subset of these failures generated by source x.

We define the **Local Attack Graph** $LAG(V, E_{\psi_h^p})$ where $V = \{z_j | \exists r^t_{x^c_j} \in F^c(z_j)\}$ and $E_{\psi_h^p} = \{(v_j, v_i) \in E_{\psi_h^p} | \exists r^t_{x^c_k} \in F^c(v_j) \wedge \exists r^t_{x^c_t} \in F^c(v_i) \wedge near(x^c_k, x^c_t)\}$.

The *near* function is defined as:

$$near(x^c_k, x^c_t) = \begin{cases} true & \text{if } 0 < |\{x^c_k, x^c_{k+1}, \cdots, x^c_t\} \backslash K^o_{t_0+\Delta_a}| \leq d_{max} \\ false & \text{otherwise} \end{cases}$$

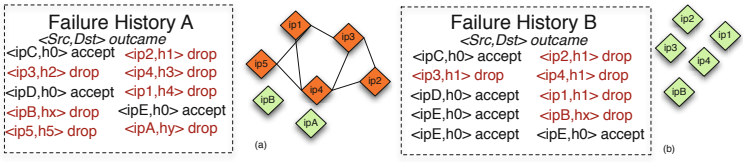


Fig. 2. (a) Local Attack Graph obtained by the Failure history A (b) Local Attack Graph obtained by the Failure history A. In both graphs d_{max} is equal to 2.

In other words, if we consider the list of failures there is an edge between two nodes that failed a connection to a host in N if their addresses are contiguous according to the *near* function. As an example, Figure 2.a and Figure 2.b show two LAGs obtained from the failure histories a and b respectively. Note that if one host fails a connection to an address that is not contiguous to other failures then these hosts appear as isolated nodes in LAG (e.g., *ipB* and *ipA* in Figure 2.a). In the case there is no contiguity at all between any pair of failures we get the graph shown in Figure 2.b where all hosts are isolated. Let us now introduce the following lemma:

Lemma 1. *Let G be the Local Attack Graph in case of coordinated port scan with sampling function ψ_h^p the following two properties hold:*

(1) all the attackers that generated at least one anomalous failure are in G.
(2) all the attackers controlled by a same adversary are in the same connected component.

As a consequence, a contiguous coordinated port scan will create connected components inside the LAG. This brings to the following observation: if a LAG shows connected components with high intra-edge density there is a suspect that a coordinated port scan is running. If we consider Figure 2.a there is a connected component of hosts that can be suspected to be a (or a part of) group of attackers. This is just a suspect because the connected component could be a false positive. Each suspect is reported as a *potential group attack* (PGA) alert constituted by a set of possible attackers $at = \{at_1, at_2, ..., at_z\}$ and for each attackers at_j the set of targets T_{at_j}. As an example considering Figure 2.a, $at = \{ip1, ip2, ip3, ip4, ip5\}$ and $T_{ip1} = \{h4\}$, $T_{ip2} = \{h1\}$, $T_{ip3} = \{h2\}$. These potential alerts are sent to the collaborative layer described below.

5 Collaborative Layer Model

The LAG can be used to detect an attack carried out by an adversary that just probes addresses in that single network. However, if we consider an adversary A carrying out an attack through GoA Z against multiple networks domains, each single local attack graph contains only a partial view of the attack. By analyzing separately the LAGs information about the port scan could go undetected. More specifically the following issues arise at the single network domain level:

(1) Detection of big attack communities: it is possible that the set of attackers available for the adversary is bigger than $|N|$ (i.e. $\overline{c_z} < 1$); in this case the information collected within each single network is not sufficient to detect the entire GoA.
(2) Separation of adversaries: assume that two adversaries owning two independent GoA Z_1, Z_2 decide to attack the same set of addresses at the same time; if $\overline{c_z} = 1$ there is no way to distinguish attackers belonging to the two GoA using only information of a single network domain, since each possible partition of the attackers set would be arbitrary.

Collaborative Graph. We can mitigate these issues by introducing a collaborative layer where a set of network domains share information about PGA. This layer collects PGA alerts from local network domains and creates a Collaborative Graph $CG_c(V, E)$ that should be able to cluster and separate GoAs. In a collaborative graph V is the set of PGA alerts and two PGAs are connected by an edge if a measure of similarity between them is above a certain threshold δ_s. The edge is labeled with the list of the intersections of attackers. To express this notion of similarity, let us introduce the **inter-alert similarity** (IAS) between two alerts a_1, a_2. IAS is defined as the ratio between the union of targets of the intersection of attackers over the union of targets. i.e.,

$$\text{IAS}(a_1, a_2) = \frac{|\cup_{\forall at_j \in a_1 \cap a_2} T_{at_j}|}{|\cup_{\forall at_i \in a_1 \cup a_2} T_{at_i}|}$$

As an example let consider Figure 5, an edge is established between potential group attack alerts $PGA1$ and $PGA2$ that share the attackers $\{b, c\}$, each attacker has one target so in this case the cardinality of the union of targets it is equals to the cardinality of the union of attackers. This value is divided for the cardinality of the union of all targets in $PGA1$ and $PGA2$, the result is above the threshold and leads to the creation of the edge between $PGA1$ and $PGA2$ whose label is $\{b, c\}$.

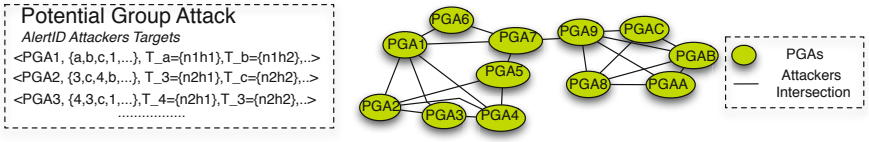


Fig. 3. Cooperative Graph obtained by a list of potential group attack alerts got by the Network Domains. The list highlights potential group attack alerts PGA1,.., PGA5.

Each time a new PGA alert gets to the collaborative layer, a node is added to the collaborative graph and its corresponding edges are created. Periodically, a procedure is executed transforming the graph in order first to aggregate and then to separate group of attackers. The final aim of this procedure is to detect entire group of attackers and to separate two groups in case of concurrent attack. Below we describe the two main steps of this procedure.

5.1 Group Attack Aggregation

Once we have a Collaborative Graph, we need a mechanism that is capable to aggregate PGA alerts coming from Local Attack Graphs.

We employ a clustering algorithm on $CG(V, E)$ that try to maximize the modularity [12], since the group of PGA alerts generated by the same attackers are likely to have higher density of intra-cluster links. The clusters obtained in this way are then analyzed: if the set C obtained by the union of all the edge-sets covers a sufficient number of targets, then the cluster is collapsed; all the nodes are deleted and substituted with a *supernode* that contains the attackers in C and the relative targets. When this happens the set C is signaled as a group of attackers.

In order to trigger the collapsing behavior we need to introduce the **intra-alert coverage** (IAC) of a set of attackers $C = \{at_1, at_2, ..., at_z\}$ over the PGA a as:

$$\text{IAC(C,a)} = \frac{|\cup_{\forall at_j \in C \cap a} T_{at_j}|}{|\cup_{\forall at_i \in a} T_{at_i}|}$$

we allow the collapse only when the set of attackers in C covers at least a certain threshold ρ_{iac} of targets over at least a threshold ρ_{alert} of PGA alerts.

5.2 Separation of Adversaries

The Group Attack aggregation step could merge two different GoAs, then we need a step that tries to check if this has been done and, in the affirmative, separate the two groups. We assume, as in [1], that the adversary will orchestrate the GoA attackers to maximize the scan coverage minimizing the intra scan overlap. So we define the Maximum Union Minimum Intersection problem as follows:

Definition 1 (definition MUMI). *(Maximum Union Minimum Intersection) Consider a set of attackers* $A = \{at_1, ..., at_z\}$ *each one associated with a set of targets* $T_{at_j} = \{t_1, ..., t_{k_{at_j}}\}$. *We want to find out the subset* A^* *of attackers that maximize* $g(X) = |\cup_{\forall at_j \in X} T_{at_j}| - k \sum_{\forall at_j, at_i \in X} |T_{at_j} \cap T_{at_i}|$ *without violating the coverage constraint* $|\cup_{\forall at_j \in A^*} T_{at_j}| \geq C$ *with* $C \leq |\cup_{\forall at_j \in A} T_{at_j}|$ *and with* $k > 0$

Theorem 1. *MUMI is not solvable in polynomial time.*

The proof of MUMI is omitted, interested reader can refer to [11].

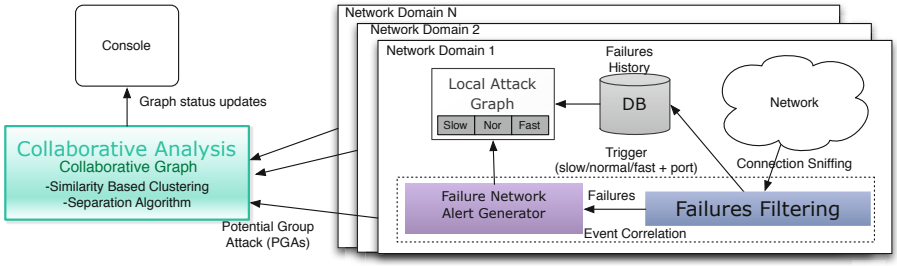


Fig. 4. The overall architecture of the collaborative port scan detection system

6 System Architecture

In this section we illustrate the details of all components present in figure 4.

Network Domain Level
The system is constituted at this level by three independent modules. Information sniffed on the local network infrastructure flows through these modules where it is analyzed and transformed in order to produce Potential Group Attacks (PGAs) for the collaborative layer.

Event correlation: this module is in charge of collecting data sniffed at the network level, filter and analyze it to detect possible anomalous activities. More in detail, connection events sniffed at the network level are sent toward a *Failures Filtering* block. This block discards all regular connections and let only

anomalous connections (failures) pass through it. Note that connection filtering is executed within this block by taking into account possible accidents that could easily raise the number of false positives. For example, the block discards failures happening on *k-active* addresses that recently become inactive as this is probably a sign of a failed local service still targeted by non-malicious connection attempts. The block also discards massive amounts of failures incoming from independent sources and directed to specific addresses as they are usually a consequence of transient misconfigurations (e.g. some changes in a DSN directory). Failures that pass the filtering stage are directed towards the DB module and the *Failure Network Alert Generator* block (FABG).

The *FABG* block receives failures and keeps track of the number of connection attempts made toward *k-closed* addresses in multiple concurrent time-windows. These statistics are separately traced per port. Each counter is compared with thresholds calculated using the function $\overline{t}$. Here we consider three kind of scan activities that we are interested in tracing (that correspond to three different time windows):

Fast : a fast attacker wants to scan S, with Δ_A limited to few minutes. This kind of attack uses a scan rate $\rho_A^c \geq 1\frac{req}{s}$, issuing a large number of errors in a small time window. To trace this kind of activity we can use a small sliding time window δ_f triggered when the errors are more than $\overline{t}(1\frac{req}{s}, \delta_f)$.

Normal : a normal attacker uses $1\frac{req}{min} \leq \rho_A^c < 1\frac{req}{s}$, with Δ_A covering a time span of few hours. To detect these attacks a time window $\delta_n > \delta_f$ is used. The value $\overline{t}(1\frac{req}{min}, \delta_f)$ is the threshold of this window.

Slow : a slow attacker uses a rate $\rho_A^c < 1\frac{req}{min}$, with Δ_A larger than ten hours. In this case the threshold is computed using $\frac{(1-\rho_{active})|N|}{2}$. At least half of all the k-closed addresses must be covered.

Every time a defined threshold is surpassed within a time window, this block triggers the *Local Attack Graph* module to create a new graph. DB: this module simply stores failures received from the Event correlation module; in this way the system keeps track of local failure histories that will be used to build *LAGs*.

Local Attack Graph: this module is in charge of building the LAG. It is triggered by the Event correlation module every time some suspicious activity (slow, normal or fast) is detected on a specific port. The LAG is built extracting historical information on past failures for that specific port from the DB and using on it the construction method detailed in Section 4. The module locally maintains data to compute the *near* function using a modified version of the interval-tree; this allows efficient maintenance under the addition and removal of active ports using logarithmic operations for update and querying. The LAG is then examined to find suspicious agglomerates created using the Louvain[1] clustering algorithm [13]. Each cluster whose size is larger than a configurable threshold represents a PGA that is sent to the collaborative layer.

[1] The Louvain algorithm has been selected since it is considered one of the fastest available method for community detection and, according to the authors, it scales well on graphs of large size.

Collaborative Layer. The collaborative layer is constituted by two modules: the *Collaborative Analysis* module and a *Console*. The former correlates and analyze PGAs incoming from the various network domains producing a Collaborative Graph that contains information on running attacks; the latter is used to visualize the CG.

```
Global Variables:
A: set of PGA alerts
CG(V,E): collaborative graph

When a new PGA a is received:
(01)  V ← V ∪ a
(02)  for each alert e ∈ A do
(03)      if (IAS(e,a)> δ_s)
(04)          E ← E ∪ {e,a'}
(05)  A ← A ∪ a

Periodically executed:
(06)  Clusters ← cluster(CG)
(07)  for each cluster c ∈ Clusters do
(08)      if c can be collapsed
(09)          g ← solve MUMI on c
(10)          for each alert a ∈ c do
(11)              remove from a every attacker at ∈ g
(12)          V ← C ∪ a new alert a' containing g
(13)          for each alert e ∈ A do
(14)          if (IAS(e,a')> δ_s)
(15)              E ← E ∪ {e,a'}
(16)          A ← A ∪ a'
```

Fig. 5. Collaborative analysis algorithm

Collaborative Analysis: this module continuously maintains the CG using the pseudocode reported in Figure 5. Every time a new PGAs is received from a network domain it is added to the CG (line 01) and the edges are updated accordingly (lines 02-04). Periodically, the module checks if alerts contained in the CG can be grouped as single GoAs. This is done by applying again the Louvain [13] clustering algorithm on the CG (line 06) that returns a set of clusters. If a cluster can be collapsed (following the policies specified in Section 5) the module solves the MUMI problem using it as input in order to extract from it the attackers actually pertaining to the GoA. The approximate solution to MUMI is obtained using a reactive greedy randomized adaptive search procedure (*RGRASP) [3] where the stopping condition is true when the ratio of new covered elements over the overlapping elements is less or equal to a threshold k, while the probability for the parameter α_j is defined as $P(\alpha_j) = \overline{sol_{\alpha_j}} / \sum_{\forall i} \overline{sol_{\alpha_i}}$ where $\overline{sol_{\alpha_j}}$ is the average of all the solutions found using α_j. We slightly modified the original RGRASP adding one step after the local search procedure: the idea is to run the randomized greedy until the stopping condition is satisfied; then, as in the original GRASP we run the local search to improve the solution found so far. The result of the search procedure is improved using one more step of randomized greedy, then this solution is used to feed a local search and so on till we reach a local maximum. The rationale in this procedure is that the local search procedure could improve the solution selected by the randomized greedy removing a conflicting attacker; this could lead to the addition of a new attacker that now does not conflict with the solution obtained after the local search. After

a solution to MUMI has been found, we create from this set of attackers a new node in the CG (this is like a new alert) that correspond to a GoA (lines 12-16). The attackers included in this GoA are removed from other alerts that remain in the CG (lines 10-11). The choice of *RGRASP has been done after an evaluation of this meta-heuristic against a greedy algorithm and the original RGRASP. Due to lack of space the comparison betwen RGRASP and *RGRASP can be found on [11].

We realized a prototype of the system in java using Jung libraries [14] for the visualization and graph manipulation. The event correlation module has been developed using Esper Complex Event Processing system [15]. The output of the system is a human-readable form of the CG that is generated by a Console module. The console could easily be substituted by more complex reactive systems that, by inspecting changes in the CG would trigger adequate automatic responses (e.g. the addition of rules to a firewall).

7 Evaluation

This section provides an evaluation of a prototype implementation of the proposed system with respect to its ability to correctly identify coordinated port scan activities.

Input data. In order to feed the system with realistic data we collected several traces from the firewall deployed at the edge of the DMZ network of a large academic institution[2]. The traces globally contain data collected during three days of activity, from november 4^{th} to november 7^{th}, 2011. The network where activity was traced includes web-servers of the institution and many other public services available from outside, (email server, ftp, etc.). Each TCP connection is associated timestamp with granularity up to seconds, and state information (Dropped or Accepted). With respect to our model a dropped connection is considered as a connection to a closed port, while an accepted connection is a connection towards an open service. Note that we had no way to clearly detect the presence of coordinated port scan activities in the traces but to run our solution on it. We decided to purposely inject such activities within the traces to simulate the presence of (multiple) adversaries. Gates [1] quotes two softwares for distributed port scan that are DSCAN and NSAT. To the best of our knowledge, these software are currently not publicly available. For this reason we wrote a coordinated port scan simulator tool that, taking a trace as input, inserts coordinated port scan activities in it. It is possible to tune the simulator specifying parameters like the number of adversaries, the scan rate, the size of the GoAs, and the targets. The tool uses a random assignment of targets that tries to balance the number of requests for each attackers minimizing c_z. In order to mimic a collaborative scenario we used the available traces as if they were extracted from different network domains.

[2] University of Rome La Sapienza.

Measures of interest. The two main characteristics of the proposed solution we want to evaluate are its capability in isolating GoAs that are larger than the target network and its effectiveness in correctly identifying multiple adversaries whose targets overlap. We define the *Intra Cluster Detection* (ICD) of GoA Z for a CG node V as the ratio $\frac{|Z \cap V|}{|A|}$. The ICD will be 1 if all the attackers in Z have been detected and inserted in the right node.

In order to measure the system ability to correctly separate concurrent attacks we can start computing the ICD for all the GoAs Z_j present in a same CG node V, i.e. $ICD(Z_j, V)$. Intuitively a bad separation of distinct GoAs within V will lead to similar values for their ICDs, while with a good separation only one GoA will sport an ICD value close to one, while all the other ICDs will be close to zero. Therefore, the sum $H(V) = \sum_{\forall Z_j \in V} -ICD(Z_j, V)Log(ICD(Z_j, V))$ is larger with a bad separation and zero if the node is homogeneous, i.e. it includes attackers from a single GoA. We normalize that value obtaining $\overline{H(V)}$ and compute the converse $D(V) = 1 - \overline{H(V)}$.

7.1 Experimental Results

Detection of GoAs

In this experiment we test the behaviour of the system considering a GoA Z that scans twenty network domains. The target dimension $|S|$ is limited to 200 addresses. We varied both the size of the GoA by tuning c_z and the parameters ρ_{iac} and ρ_{alert} used to configure the CG clustering. The two Figures 6, and 7 report results obtained setting $\rho_{iac} = 0.85, 0.7$ and $\rho_{alert} = 0.9, 0.7$ respectively. Each graph plots the values for $c_z = 2, 1, 0.6, 0.5, 0.4, 0.33, 0.28$ and 0.25; the size of the GoA goes from 100 to 800 attackers.

The results show that, for a given configuration (e.g. $\rho_{iac} = 0.7$ and $\rho_{alert} = 0.7$) the system requires information incoming from a larger number of network domains to identify GoAs ($c_z \leq 0.5$), while smaller GoAs ($c_z = 2$) can be easily identified using data incoming from few domains. The impact ρ_{iac} and ρ_{alert} have on the results is less intuitive. By increasing their values toward one, we

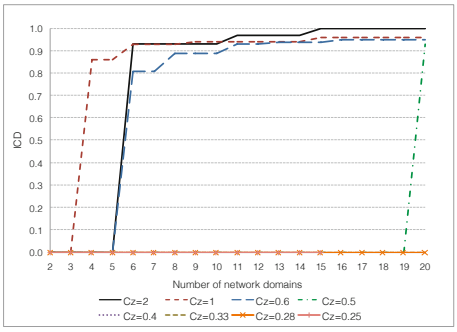


Fig. 6. Intra-Cluster Detection with $\rho_{iac} = 0.85, \rho_{alert} = 0.9$

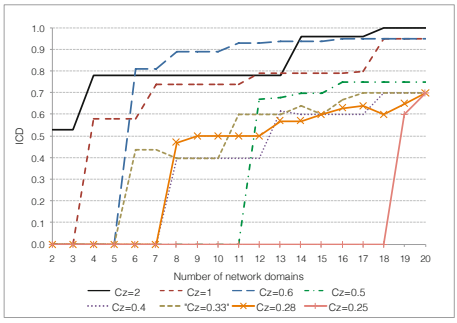


Fig. 7. Intra-Cluster Detection with $\rho_{iac} = 0.7, \rho_{alert} = 0.7$

are tuning the collaborative layer to be more "picky" in the selection of attackers that form an identified GoA; as a consequence, if the GoA is large (e.g. $c_z \leq 0.5$) it takes a larger amount of information for the system to isolate it, but, once the GoA is identified, information about almost all its attackers is already available in the CG. This accounts for the results that show how the ICD value increase for larger number of domains but at a very steep rate, quickly reaching values close to one (cfr. Figure 6). Conversely, by decreasing the values of the two ρ parameters, we choose to have a system that reacts quicker to potential attacks starting to isolate GoAs as soon as information from few network domains is available; on the other side, "early" isolation of GoAs comes at a price: most detections happens with incomplete information about the GoAs as it is shown by the slower growth rate of the curves reported in Figure 7.

Table 1. Results of experiments on GoA separation. The overlap size indicates the number of shared target network domains in the adversaries' targets. The total number of domains is 20. Each CG node is summarized as a list of cardinality for the detected GoAs.

Overlap size	CG nodes	$D(V)$
20	$V_1 = (\|Z_1\| = 198, \|Z_2\| = 0)$ $V_2 = (\|Z_1\| = 1, \|Z_2\| = 200)$	$D(V_1) = 1$ $D(V_2) = 0.95$
17	$V_1 = (\|Z_1\| = 200, \|Z_2\| = 2)$ $V_2 = (\|Z_1\| = 0, \|Z_2\| = 198)$	$D(V_1) = 0.91$ $D(V_2) = 1$
15	$V_1 = (\|Z_1\| = 199, \|Z_2\| = 15)$ $V_2 = (\|Z_1\| = 16, \|Z_2\| = 199)$	$D(V_1) = 0.62$ $D(V_2) = 0.62$
13	$V_1 = (\|Z_1\| = 200, \|Z_2\| = 16)$ $V_2 = (\|Z_1\| = 0, \|Z_2\| = 80)$	$D(V_1) = 0.62$ $D(V_2) = 1$
11	$V_1 = (\|Z_1\| = 177, \|Z_2\| = 6)$ $V_2 = (\|Z_1\| = 0, \|Z_2\| = 200)$	$D(V_1) = 0.79$ $D(V_2) = 1$
9	$V_1 = (\|Z_1\| = 98, \|Z_2\| = 158)$	$D(V_1) = 0.03$
7	$V_1 = (\|Z_1\| = 91, \|Z_2\| = 151)$	$D(V_1) = 0.04$
5	$V_1 = (\|Z_1\| = 186, \|Z_2\| = 129)$	$D(V_1) = 0.02$
1	$V_1 = (\|Z_1\| = 192, \|Z_2\| = 0)$ $V_2 = (\|Z_1\| = 0, \|Z_2\| = 193)$	$D(V_1) = 1$ $D(V_2) = 1$
0	$V_1 = (\|Z_1\| = 199, \|Z_2\| = 0)$ $V_2 = (\|Z_1\| = 0, \|Z_2\| = 198)$	$D(V_1) = 1$ $D(V_2) = 1$

Separation of GoAs

The other aspect of our system is the possibility to correctly identify two adversaries attacking concurrently the same targets. We tested this feature using 20 network domains and running different tests where we vary the amount of network domains where the targets of the two adversaries overlap. As Table 1 shows the separation is sharp in the most cases proving the effectiveness of our approach in identifying concurrent coordinated port scans.

Some difficulties arise when the target overlap happens on few domains. In these cases the system is not able to clearly separate the GoAs and create in the CG a single node containing a mix of attackers from both GoAs. Let us recall that within the MUMI problem formulation a parameter k is present that assigns a *malus* for the presence in a same CG node of two different attackers that share some targets. Tests reported in Table 1 have been performed setting $k = 2.0$. This lead to the bad behaviour reported above when the cardinality of the overlap is above half of the union of the targets. By tweaking the value of k it would be possible to take into account these specific scenarios and still obtain an effective GoA separation. However, larger k values could bring into play undesired side effects like the partitioning of a single GoA in two CG nodes.

8 Limitation, Conclusion and Future Work

In this paper we presented a new approach for collaborative detection of coordinated port scans over multiple network domains. Despite the encouraging results reported in Section 7, the proposed solution still shows some limitations that could be exploited by skilled adversaries to avoid the detection.

Target Contiguity: the system assumes that the target set of an attack contains contiguous addresses. The presence of d_{max} mitigates this problem as attackers that have probed addresses that are not strictly contiguous will be still considered as neighbors in the local graph.

Collaboration among Network Domains: the system is designed to leverage information coming from different network domains to detect coordinated port scans.The collaborative layer, in fact, is totally useless when the adversary is acting only against one network domain.

Botnet-Based GoAs: if the adversary controls a huge GoA, for example one based on a large-scale botnet, it could be capable of generating a single probe per attacker. In this way all the PGAs generated by network domains will be fully uncorrelated, hampering the possibility to identify the GoA at the collaborative layer.

Very Slow Attackers: an adversary willing to use a time-window larger than the observer maximum window will avoid detection in the network domain.

As a future direction for the evolution of this work, we are interested in devising new correlation methods, possibly based on characteristics of the scan activities that have not been considered in this work, that would make the system capable of solving some of the previous limitations.

References

1. Gates, C.: Coordinated scan detection. In: Proceedings of NDSS 2009 (2009)
2. Zhou, C.V., Leckie, C., Karunasekera, S.: A survey of coordinated attacks and collaborative intrusion detection. Computer and Security 29, 124–140 (2009-2010)
3. Prais, M., Ribeiro, C.C.: Reactive grasp: An application to a matrix decomposition problem in tdma traffic assignment. INFORMS Journal on Computing 12, 164–176 (1998)
4. hybrid, Distributed Information Gathering (2011), http://www.phrack.org/issues.html?issue=55&id=9
5. Staniford, S., Hoagland, J.A., Mcalerney, J.M.: Practical automated detection of stealthy portscans. Journal of Computer Security 10, 105–136 (2002)
6. Conti, G., Abdullah, K.: Passive visual fingerprinting of network attack tools. In: Proceedings of VizSEC/DMSEC 2004, pp. 45–54. ACM, New York (2004)
7. Robertson, S., Siegel, E.V., Miller, M., Stolfo, S.J.: Surveillance detection in high bandwidth environments. In: Proceedings of DARPA DISCEX III, pp. 229–238. IEEE Press (2003)
8. DShield: Cooperative Network Security Community - Internet Security (2009), http://www.dshield.org/indexd.html/

9. Yegneswaran, V., Barford, P., Ullrich, J.: Internet intrusions: global characteristics and prevalence. SIGMETRICS Perform. Eval. Rev. 31, 138–147 (2003)
10. Baldoni, R., Chockler, G.: Collaborative Financial Infrastructure Protection. Springer (2012)
11. Baldoni, R., Luna, G.D., Querzoni, L.: Collaborative Detection of Coordinated Port Scans, MIDLAB 1/12 - University of Rome "La Sapienza" Tech. Rep. (2012), http://www.dis.uniroma1.it/~midlab/publications.php
12. Newman, M.E.J.: Modularity and community structure in networks. Proceedings of the National Academy of Sciences 103(23), 8577–8582 (2006)
13. Blondel, V., Guillaume, J., Lambiotte, R., Mech, E.: Fast unfolding of communities in large networks. J. Stat. Mech., 10008 (2008)
14. Jung (2011), http://jung.sourceforge.net/
15. Esper (2011), http://esper.codehaus.org/

Exploiting Partial-Packet Information for Reactive Jamming Detection: Studies in UWSN Environment

Manas Khatua and Sudip Misra, *Senior Member, IEEE*

SIT, Indian Institute of Technology Kharagpur, India
{manask,smisra}@sit.iitkgp.ernet.in

Abstract. Reactive jamming in an underwater sensor network (UWSN) environment is a realistic and very harmful threat. It, typically, affects only a small part of a packet (not the entire one), in order to maintain a low detection probability. Prior works on reactive jamming detection were focused on terrestrial wireless sensor networks (TWSNs), and are limited in their ability to (a) detect it correctly, (b) distinguish the small corrupted part from the uncorrupted part of a packet, and (c) be adaptive with dynamic environment. Further, there is currently a need for a generalized framework for jamming detection that outlines the basic operations governing it. In this paper, we address these research lacunae by broadly designing such a *framework* for jamming detection, and specifically a *detection scheme* for reactive jamming. A key characteristic of this work is introducing the concept of *partial-packet* (PP) in jamming detection. The introduction of such an approach is unique – the existing works rely on holistic packet analysis, which degrades their performance – a fundamental issue that would substantially affect achieving real-time performance. We estimate the probability of high deviation in received signal strength (RSS) using a *weak estimation learning* scheme, which helps in absorbing the impact of dynamic environment. Finally, we perform CUSUM-test for reactive jamming detection. We evaluate the performance of our proposed scheme through simulation studies in UWSN environment. Results show that, as envisioned, the proposed scheme is capable of accurately detecting reactive jamming in UWSNs, with an accuracy of 100% true detection, while the average detection delay is substantially less.

Keywords: Reactive jamming, partial-packet, weak estimation, CUSUM.

1 Introduction

UWSNs [1] find many time-critical applications in scenarios such as underwater surveillance, intrusion detection, and seismic monitoring. The real-time delivery of messages is crucial in these applications. An important class of threats that can severely affect the successful functioning of such kind of challenged networks is *jamming*. The unique characteristics of UWSNs, such as low bandwidth, adverse

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 118–132, 2013.

communication channel, large and variable propagation delay, high bit error rate, and harsh environment, make these networks more vulnerable to jamming attacks. Additionally, sensor nodes, as such, have limited resources in terms of storage capacity, energy supply, and computational power, which make the designing of any jamming attack detection scheme even more challenging.

1.1 Motivation

It is shown in [2] and [3] that reactive jamming, as such, is a realistic and very harmful threat in any TWSN. The reactive jammer stays quiet untill and unless it senses any ongoing communication within its transmission range. Upon sensing any communication, the reactive jammer transmits a short-duration signal to disrupt the legitimate communication, and then again remains quiet. The UWSNs, likewise, are also vulnerable to jamming attacks [4]. Our measurement-based experimental studies reveal that reactive jamming can potentially decrease the throughput of a network, on an average, by 35.7% below normal scenarios in UWSN environments, as shown in Figure 1 (experimental setup is explained in Section 6). There exists only one work [2] on reactive jamming detection in TWSNs. However, none of the existing works was designed for UWSNs. Along with this lacuna, some of the important observations characterizing UWSN environments and reactive jamming (mentioned below) have motivated us to design an effective scheme for reactive jamming attack detection in UWSN environment.

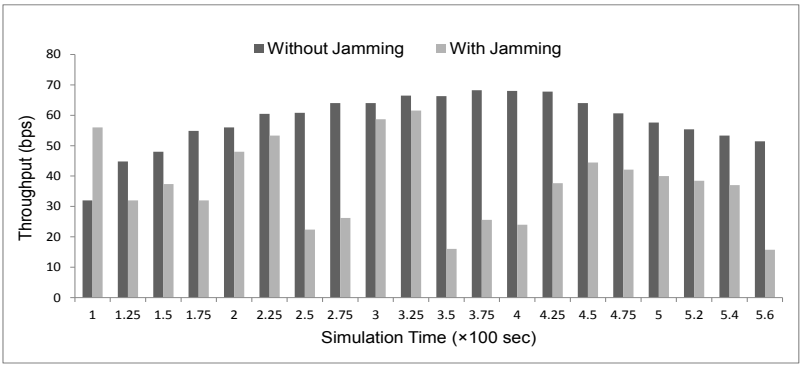


Fig. 1. Effect of reactive jamming in UWSNs

UWSNs suffer from long and variable propagation delay. The propagation speed of acoustic signal is five orders of magnitude less than the radio frequency propagation speed [4]. Therefore, ACK-based detection metrics, such as packet delivery ratio (PDR), takes much longer time for jamming detection. Moreover, accurate calculation of PDR is not feasible for on-demand reactive forwarding schemes [2]. Hence, PDR is not a suitable metric for jamming detection

in UWSNs. Underwater channels are highly influenced by artificially triggered noise sources such as pumps, gears, ships, and ambient noise sources such as tides, rain drops, fishes, and seismic activities. All these unavoidable and unpredictable noise sources make the average signal strength unpredictable in normal scenarios too, and, thus, invalidate the use of signal-to-noise (SNR) ratio for jamming detection. Global Positioning System (GPS) uses 1.5 GHz radio frequency band, and these waves do not propagate long distance in water [1]. This issue restricts the use of node-position-based jamming detection schemes in UWSNs. The underwater medium is highly dynamic in nature. Parameters such as temperature, pressure, salinity, and noise significantly vary with depth in the water column. It is inappropriate to use schemes (e.g. [5], and [6]), which rely on a predefined threshold of a metric that depends on any of these varying parameters. Rather, we argue that it is required to design suitable metrics that can reduce the impact of the dynamic nature of the medium.

Reactive jamming, typically, corrupts *only a few bits* of a packet [2]. In general, the Forward Error Correction (FEC) based schemes are not suitable for error correction in TWSNs due to high resource consumption [7]. Additionally, FEC-based schemes can be used for jamming mitigation, but not for jamming detection [2]. The corruption of only a few bits does not increase the signal strength corresponding to a packet noticeably [2], as the RSS corresponding to a packet may "dilute" the RSS changes corresponding to a few corrupted bits. So, the average RSS based schemes (e.g. [5], and [6]) are not suitable for reactive jamming detection. Furthermore, all existing schemes (e.g. [2], [5], and [6]) rely on retransmitting the whole affected packet. Such an approach leads to the retransmission of highly redundant bits, and consequently, consuming additional energy. The fact of the matter is that the algorithms used in these existing approaches cannot be used to distinguish between the corrupted and uncorrupted bits. Finally, it is observed that, the underline stream of all existing jamming detection schemes are same, even though, they followed different approaches for jamming detection in TWSNs. Hence, we tried to expose such underline stream through a well-designed framework.

1.2 Contributions

This paper addresses some of the research lacunae identified in Section 1.1, by designing a generalized *jamming detection framework*, and then, using it for reactive jamming detection studies in UWSN environments. We use the logical concept of *partial-packet* (PP) [8], [9] to monitor short-duration packet dynamics, which is a key feature of our work. We estimate the probability of high deviation in RSS using a *weak estimation learning* [10] scheme, and perform CUSUM-test [11] for reactive jamming detection. Additionally, the proposed scheme provides an opportunity of *partial-packet recovery* [8], [9] by acknowledging the sender about the corrupted PPs. In sum, our contributions in this work are as follows:

(a) *Studies on Reactive Jamming in UWSN Environment*: To the best of our knowledge, no work on reactive jamming detection in UWSN exists. We have specifically considered UWSN environments to perform a study on the

effectiveness of reactive jamming and its detection, as the UWSNs are more challenging networks [1].

(b) *Channel Observation Metrics*: We propose two new metrics for short-duration channel monitoring, *Bad Partial-Packet Ratio* and *Partial-Packet RSS*, which quantify the observed statistics with respect to PP. We define another metric, *Deviation of PPRSS*, to capture the deviation of RSS in corrupted bits. This metric correctly distinguishes the effect of poor channel from packet collision, and addresses the problem of considering signal strength decrease due to *destructive* interference (an important contribution, as the existing works assumed only constructive interference). It also absorbs the impact of dynamic environment in UWSNs by avoiding the use of static threshold for RSS.

(c) *Reactive Jamming Detection Scheme*: We propose a CUSUM-based distributed reactive jamming detection scheme, which can detect reactive jamming *correctly* (i.e. very less false detection) and in short interval of time. On the contrary, the existing schemes are limited in their fundamental ability to detect it simply correctly in different scenarios (as explained in Section 2).

(d) *Jamming Detection Framework*: We design a component-based generalized framework to provide a structural view of common operations required for jamming detection. It reduces the complexity of designing a specific scheme by identifying those task and organizing them in separate modules such that it allows implementing different algorithms for each module independently.

1.3 Organization of the Paper

The remaining paper is organized as follows. In Section 2, we briefly discuss the related works reported in some literature with respect to their suitability and correctness in UWSNs. The network model and the jamming model, used in this paper for simulation, are discussed in Section 3. In Section 4, we discuss about different packet formats and three newly defined metrics for short-duration packet monitoring. We discuss about the proposed generalized framework and CUSUM-based reactive jamming detection scheme in Section 5. The simulation results and their analysis is shown in Section 6 followed by the conclusion and future direction of work for extending it further in Section 7.

2 Related Work: Challenging Their Suitablity and Correctness for Use in UWSNs

There exist many important pieces of research works on jamming attack detection and countermeasures in TWSNs, as discussed in [7] and [12]. We noticed that the existing reactive jamming detection schemes in TWSNs use different combinations of jamming detection metrics, such as RSS, PDR, and SNR. Therefore, at first we study the effectiveness of each metric in UWSN environment as described in Section 1.1. In this section, we draw discussion on each scheme separately. Xu et al. [6,13] reported extensive studies of different jamming attack models and their effectiveness in the physical layer of wireless networks.

They proposed few jamming attack detection techniques using RSS, PDR, and carrier sensing time (CST). To improve the detection accuracy, they proposed consistency check of *signal strength* and *node location* coupled with the PDR observation. Finally, they concluded that PDR is an important metric for jamming detection, even though, any individual metric is not sufficient to detect jamming correctly. *It is already mentioned in Section 1.1 that, the use of the position of the nodes, PDR, RSS, and SNR metrics are not suitable in UWSN environments.* This issue invalidates the use of the above mentioned schemes and few others proposed in the existing pieces of literature (e.g., [5], [14], [15], and [16]), for reactive jamming detection in UWSN environments.

All the works, mentioned above, are general jamming attack detection schemes designed for TWSNs. There exist only one work [2] on reactive jamming detection in TWSNs (none for UWSNs). In that scheme, bit error is calculated as $e[i] := m_1[i] \oplus m_2[i]$, where m_1 and m_2 are two independently received packets by two nodes of "n-tuple wired chained nodes". We argue that the equation may result in an *erroneous* value for "passive monitoring", if both the packets are equally corrupted. In case of "active monitoring", periodical probe message exchange increases the overhead of the network. Further, the authors assumed for both the types of monitoring that wired communication is error free, and at least one node of n-tuple nodes must be within the jammed area. Along with these assumptions, few other limitations are as follows: (a) To implement the "limited wiring" in practice, we need either hardware support or special kind of nodes in the network. (b) By default, all signal strength based detection schemes consider that interference always increases the signal strength. However, signal strength decreases if destructive interference occurs. *Hence, we argue that detection schemes based on threshold value comparison of RSS and SNR metrics are not capable of discriminating channel fading and destructive interference.* All these discrepancies strike the *correctness* of the scheme.

3 Network and Jamming Model

In this Section, we briefly specify the network model and the jamming model that we use in this work.

3.1 Network Model

We consider security- and safety-critical applications in which timely delivery of alarm messages is crucial. Let us assume that a UWSN is deployed to provide the application requirements within a specified 3D space in an aquatic environment. The underwater sensor nodes are floated at different depths as per their hardware arrangements, such as bladder apparatus and presure gauge. Each node is assumed to have an omnidirectional antenna with equal communication ranges. They are considered not to have active mobility, although they are mobile due to the underwater current. It is further assumed that the nodes are aware of their own location coordinates, but do not know others' locations. They can localize themselves using any localization technique, as explained [17] and [18].

3.2 Jamming Model

The main objective of jamming-style denial-of-service (DoS) attacks is to block or delay the delivery of legitimate messages. In this work, we assume that only one reactive jammer exists in the network. The jammer can only send random signals to corrupt the transmission, but cannot destroy or deactivate normal nodes. The jammer cannot restrict the nodes from observing the environment or events too. The reactive jammer senses the channel for identifying any ongoing transmission. Once the data transmission is sensed, the reactive jammer sends random signals of short-duration that are capable enough of corrupting the legitimate packets. Except this intensional transmission, the reactive jammer remains quiet at all times. Thus, the reactive jammer does not need any extra information about legitimate nodes and communication protocols for jamming the network, except for channel sensing. We consider that the jammer can choose its transmission power within a finite range. The main objective of the reactive jammer is to infect the network as much as possible while it maintains the probability of detection low. The reactive jammer also has no active mobility, although it moves under the water due to underwater currents.

4 Partial-Packet Measurement

In this Section, we describe the packet formats and three new metrics designed for reactive jamming detection in UWSNs.

4.1 Message Formats

A packet of length x bytes is *logically* divided into n PPs. Each PP is augmented with one byte cyclic redundancy check (CRC) code for error detection. The size of each PP, except the last one, equals $\lceil x/n \rceil + 1$. We assume that the maximum possible value for n is 256. A traditional network layer packet is similar to a modified packet, except the CRCs augmented with it. This augmentation increases the packet size by n bytes in our design, as we use 1 byte for representing each CRC. It may be less or more depending on the type of CRC used for error detection. Note that if the maximum deliverable packet length for a network is specified, then the maximum data size of each packet is reduced by n bytes to accomodate the CRCs of n PPs. Therefore, a tradeoff is created between the network overhead and the number of PPs. We left the experimental discussion on this tradeoff as our future work.

The formats of a traditional network layer packet, partial-packet, and recovery packet are shown in Figure 2(a), 2(b), and 2(c), respectively. The *PktType*, *Src*, and *Dest* fields indicate the type, source and destination of the packet, respectively. The *PPMap* field is a bit-map indicating which PPs of the packet have been received correctly. We can use the same $PPMap$ array for the next packet as well, thereby avoiding additional storage overhead. The *CRC* field in the recovery packet is used to identify the error in it.

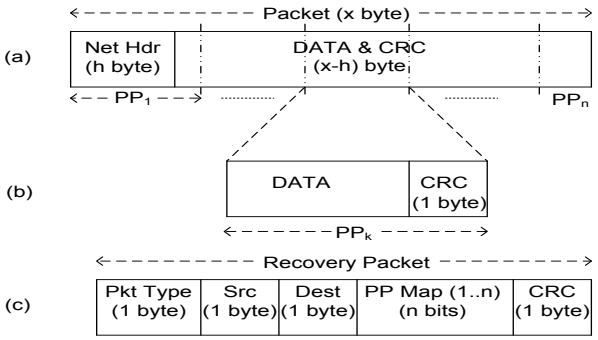


Fig. 2. Packet formats (a) Network layer packet (b) Partial-packet (c) Recovery packet

4.2 Metrics for Short-Duration Channel Monitoring

In this paper, we define three new metrics to observe the short duration packet statistics, i.e., traffic status. These metrics can help in detecting and characterizing jamming attacks in UWSN environments. Simultaneously, these metrics are carefully designed so that they do not suffer from the disadvantages mentioned in Section 1.1. The following metrics are proposed in this work.

– *Bad Partial-Packet Ratio (BPPR)*: BPPR is defined as the ratio of the number of *corrupted* PPs received to the total number of PPs received in an interval. Hence, the value of BPPR can be within the interval $[0, 1]$.

$$BPPR = \frac{number\ of\ corrupted\ PPs}{total\ number\ of\ PPs\ received} \tag{1}$$

– *Partial-Packet RSS (PPRSS)*: PPRSS is defined as the average RSS corresponding to each PP.

$$PPRSS = \frac{1}{d} \sum_{i=1}^{d} RSS[i] \tag{2}$$

where, d is the size of a PP in bits and *RSS[i]* is the received signal strength of i^{th} bit in the PP.

– *Deviation of PPRSS (DevPPRSS)*: DevPPRSS is defined as the difference between the average PPRSS of corrupted PPs and correctly received PPs.

$$DevPPRSS = \sum_{i=1}^{n} PPRSS[i] \times PPMap[i] - \sum_{i=1}^{n} PPRSS[i] \times (1 - PPMap[i]) \tag{3}$$

where, the value of *PPMap[i]* is either 0 or 1 depending on whether the i^{th} PP is erronious or not, respectively.

5 Reactive Jamming Detection

In this Section, we initially describe the proposed generalized framework for jamming detection, and then use it for designing a CUSUM-based reactive jamming detection scheme which can be executed by any node in the network.

5.1 Generalized Architecture

A well-designed architecture reduces the complexity for designing schemes. The existing works on jamming detection did not provide any common sequence of functionalities needed for the execution of any jamming detection exercise. To address this problem, in this paper, we try to expose the common sequence of steps through a well-designed component-based framework. The proposed architecture, shown in Figure 3, carries multiple advantages. For instance, using it one can design different algorithms for individual components, and any algorithm can be modified independently to improve its performance. The designed framework consists of three components:

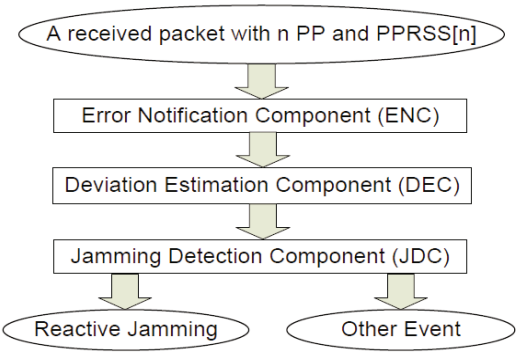


Fig. 3. A generalized framework for jamming detection

(a) *Error Notification Component* (ENC): The basic functionality of this component is to detect any error introduced within a packet, and to notify it to the next component. This component also monitors necessary information when a packet arrives at the receiver. In the proposed scheme, we use the concept of CRC for error detection in each PP. If any PP is detected as erroneous, it invokes the algorithm written in the next component; otherwise, it skips all the remaining steps. The receiver records the corresponding RSS value for each bit while it receives any packet.

(b) *Deviation Estimation Component* (DEC): This component helps to compute the values of different metrics used in any jamming detection scheme. In the proposed scheme, initially, we compute the values of metrics defined in Section 4.2.

The value of the BPPR metric varies from 0 to 1, as the number of corrupted PPs are within 0 and n. If all the PPs are received correctly, then BPPR is 0, and its value approaches unity, if the number of corrupted PPs approaches n. We, then, compute the PPRSS of each PP, and estimate the probability of high deviation in PPRSS using a *weak estimation learning* [10] scheme. Let us assume that the measured deviation is represented as Π, where Π is the absolute value of *DevPPRSS*. We determine a threshold value τ to differentiate between the high and the low values of Π. Clearly, this threshold value represents the Π parameter as a binomially distributed random variable, and its value must be either $\geq \tau$ or $< \tau$. Let us consider that Π obeys the following distribution:

$$\Pi = \begin{cases} \geq \tau \text{ with probability } \phi^0 \\ \\ < \tau \text{ with probability } \phi^1 \end{cases} \tag{4}$$

such that $\phi^0 + \phi^1 = 1$, where $\Phi = [\phi^0, \phi^1]^T$. At any iteration i, let us assume that Π takes the value π_i. The weak estimator maintains a running estimate $\Psi = [\psi_i^0, \psi_i^1]^T$ of Φ to estimate ϕ^j, where ψ_i^j is the estimate of ϕ^j at time i, for $j = 0, 1$. In this setting, the value of ψ_i^0 is updated as follows.

$$\psi_i^0 = \begin{cases} \lambda \times \psi_{i-1}^0, & \text{if } \pi_{i-1} < \tau \\ \\ 1 - \lambda \times \psi_{i-1}^1, & \text{if } \pi_{i-1} \geq \tau \end{cases} \tag{5}$$

where λ is a learning constant ($0 < \lambda < 1$), and $\psi_i^1 = 1 - \psi_i^0$. We then move to the next component to identify the cause of this packet error.

(c) *Jamming Detection Component* (JDC): This component executes an algorithm designed for jamming detection. In the proposed scheme, we design a CUSUM-based detection mechanism which is described in detail in Section 5.2. This scheme can detect the occurrence of reactive jamming quickly without possessing any *a priori* knowledge about the jammer's strategy and the time of occurrence of jamming.

5.2 CUSUM-Based Detection Scheme Design

In this Section, we present the CUSUM-based reactive jamming detection scheme (CURD) considering the following philosophy. As stated earlier, a reactive jammer, typically, corrupts only a few bits of a packet. Except this type of jammers, all others corrupt large number of bits almost equalling the number of bits present in a packet. If a packet is corrupted due to poor link or high channel fading, then the average PPRSS of both the corrupted as well as the uncorrupted PPs will be reduced and this reduction can be correctly identified by DevPPRSS. Thus, we utilize the BPPR of a packet having n PPs as the primary measuring parameter in the proposed detection scheme. We consider the reception of n PPs as an observation interval, due to its effectiveness in capturing small-duration packet statistics. As the value of n is constant during an experiment, a node

running the CURD scheme, called the *tagged node*, can easily compute the sequences of its measurement parameter. Let us consider that a node q is tagged with the CURD scheme. Our *observation measure* is the BPPR of the node q, denoted as H_t, in every n PP reception.

Let $\{H_t, t = 1, 2, ..., n\}$ be the sequence of BPPR of the tagged node. Here, H_t is a random variable, as its value varies from $(\frac{1}{n})$ to $(\frac{n}{n})$. We consider a default probability distribution of H_t, and calculate its expectation, μ. Let us assume that μ_h is the upper bound of μ. We, then, have a modified CUSUM-test statistic, which is as follows:

$$\gamma_i = [\gamma_{i-1} + (\mu_h - H_t)]^+ \tag{6}$$

where,

$$\gamma_0 = 0 \tag{7}$$

$$[\gamma_i]^+ = \begin{cases} \gamma_i, & \text{if } \gamma_i \geq 0 \\ \gamma_{i-1}, & \text{otherwise} \end{cases} \tag{8}$$

The CUSUM-test statistic indicates that, if the number of corrupted PPs continuously remains low, γ_i will quickly accumulate to a large positive value. Otherwise, γ_i maintains the previous value γ_{i-1} to overlook long- or mid-duration packet corruption. To distinguish the short-duration packet corruptions due to reactive jamming and any other network event, the indicator function consults with the estimated probability, ψ_i^0. Finally, the decision rule for i^{th} step of this CUSUM-test is defined as:

$$\delta_i = \begin{cases} 1, \text{ if } \gamma_i \geq \alpha \text{ and } \psi_i^0 \geq \beta \\ 0, \qquad \text{otherwise} \end{cases} \tag{9}$$

where, δ_i is an indicator function about the occurrence of reactive jamming, and α and β are the identification thresholds of γ_i and ψ_i^0, respectively. As soon as the detectors γ_i and ψ_i^0 cross their respective thresholds, they are reset to the default values 0 and 0.5, respectively.

6 Simulation and Evaluation

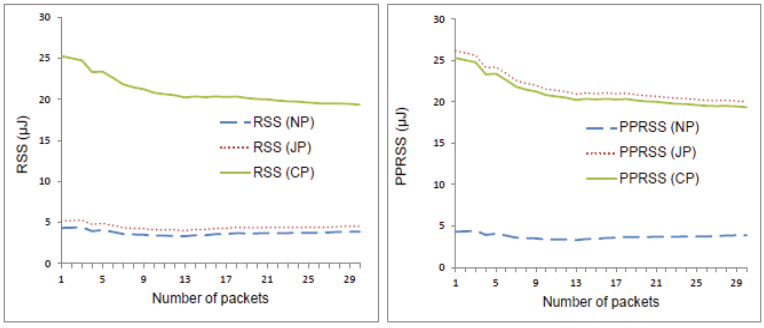
To evaluate the effectiveness of the proposed scheme, we used Aqua-Sim [19], a NS-2 based simulator for UWSNs. In the simulation, we randomly deployed 30 underwater sensor nodes in a 3D network space of 250 m^3 in an aquatic environment. One of these nodes was considered to behave like a reactive jammer. The details of the other input parameters used are listed in Table 1. We executed each simulation 20 times and computed the average value of the specified metrics for comparison. At first, we examined the usefulness of the *partial-packet* scheme used for detecting the small changes in signal strength corresponding to a few bits. Then, we studied the effectiveness of the proposed reactive jamming detection scheme using three fundamental metrics: *average detection delay*, *average missed detection ratio*, and *average residual energy*.

Table 1. Parameters and their corresponding values used in the simulation

Parameter	**UWSN**	**Jammer**
Frequency (f)	25 KHz	25 KHz
Transmitted Power (P_t)	0.2818 W	Variable ($\leq$ 0.2818 W)
Transmission Range	100 m	100 m
Propagation Model	Thorp Model [20]	Thorp Model [20]
Packet Size (x)	100 bytes	none
Routing Protocol	VBF [21]	None
MAC Protocol	BroadcastMac [19]	BroadcastMac [19]
Initial Node Energy	1000 Joules	1000 Joules
Bit Rate	10 Kbps	10 Kbps
Mobility Model	Random	Random
Node Movement	0.3 m/s	0.3 m/s
Jamming Duration (J_d)	none	20-80 bits

6.1 Effectiveness of the PPRSS Metric

In this experiment, the tagged node was configured to capture the values of RSS and PPRSS metrics corresponding to the received packets and the partial-packets, respectively. We categorized all the received packets into three types – *normal packets* (NP), *jammed packets* (JP), and *collided packets* (CP). We observed that the average RSS corresponding to the JP does not significantly increased, as compared to that of the NP, as the RSS corresponding to a packet may "dilute" the RSS changes corresponding to a few corrupted bits in the packet. Thus, RSS is not an effective metric for distinguishing the NP and the JP, as shown in Figure 4(a). On the contrary, Figure 4(b) shows that the proposed PP-based metric, PPRSS, can effectively distinguish the NP from the JP.



(a) (b)

Fig. 4. Effectiveness of PPRSS. Average (a) RSS and (b) PPRSS for NP, CP, and JP

6.2 Average Detection Delay

We performed a set of experiments with different settings of each parameter to analyze the average detection delay, $E[D]$, of the CURD scheme. We defined $E[D]$ as the average number of packets received by the tagged node for successful detection of reactive jamming, after the reactive jammer initiates its operation. Initially, we captured the distribution of γ observed by the tagged node with $\mu_h = 3/n$ and $\tau = 0.1$ mW, when $n = 50$ and $J_d = 20$ bits. We chose the weak estimation learning constant, $\lambda = 0.9$, throughout the simulation. In this experiment, if we ignore the channel condition, i.e. $\beta = 0$, the threshold value of γ_i alone indicates the jamming. On the contrary, even though the number of corrupted PPs is high enough, the identification parameter remains 0 until the probability of high deviation in RSS of those corrupted bits crosses the threshold, which is shown in Table 2. Thus, the CURD scheme can effectively distinguish the packet error due to poor channel and jamming. Similar effect was observed when we captured the distribution of ψ_i^0 with respect to the variation of β, while γ_i equals a constant value.

Table 2. Distribution of γ for different values of α and β

No. of packets received			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
β = 0	α = 0.05	γ	0	0	0	0.02	0.04	0.06	0	0.02	0.04	0.06	0.02	0.04	0.04	0.06	0.02	0.02	0.04	0.04	0.06	0
		δ	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	0
	α = 0.07	γ	0	0	0	0.02	0.04	0.06	0.06	0.08	0.02	0.04	0.06	0.08	0	0.02	0.04	0.04	0.06	0.06	0.08	0
		δ	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0
	α = 0.09	γ	0	0	0	0.02	0.04	0.06	0.06	0.08	0.1	0.02	0.04	0.06	0.06	0.08	0.1	0	0.02	0.02	0.04	0.04
		δ	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
β = 0.8	α = 0.05	γ	0	0.02	0.04	0.04	0.06	0.08	0.1	0.12	0.14	0	0	0.02	0.02	0.02	0.04	0.06	0.06	0.08	0.02	0.02
		δ	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
	α = 0.07	γ	0	0.02	0.04	0.04	0.06	0.08	0.1	0.12	0.14	0	0	0.02	0.02	0.02	0.04	0.06	0.06	0.08	0.02	0.02
		δ	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
	α = 0.09	γ	0	0.02	0.04	0.06	0.08	0.1	0.12	0.14	0.16	0.18	0.02	0.04	0.06	0.08	0.1	0.12	0.14	0.16	0.18	0.02
		δ	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0

Figure 5 shows the average detection delay under the different sets of values of μ_h and τ. The observations reveal that, irrespective of the values of α, the average detection delay is observed high, when the threshold values of ψ_i^0 is set to a high value. On the other hand, for lower values of β, the average detection delay is varied slowly with the variation of α. In conclusion, the CURD scheme provides lower detection delay, if the threshold values of both the identification parameters γ_i and ψ_i^0 are maintained at low values.

6.3 Average Missed Detection Ratio

The average missed detection ratio, R_{md}, is the ratio of the sum of the *false positives* and the *false negatives* detected by the scheme to the total number of nodes present in the network. We observed in Figure 5 that the CURD scheme waits for less duration of time for successful detection of reactive jamming, when

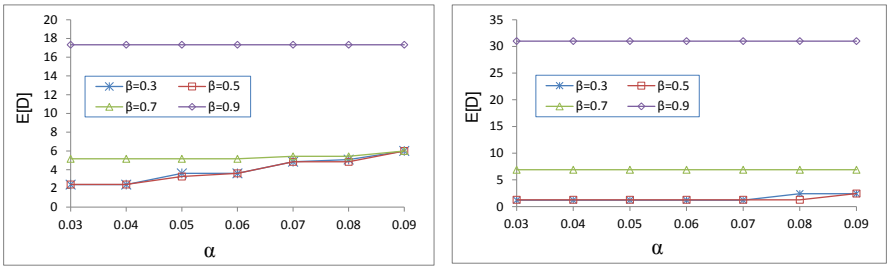


Fig. 5. Average detection delay (a) $\mu_h = \frac{3}{n}, \tau = 0.1$ mW (b) $\mu_h = \frac{6}{n}, \tau = 0.125$ mW

the values of α and β are set to 0.04 and 0.5, respectively. Therefore, in this experiment, we assumed constant values of α, β, and τ, which are 0.04, 0.5, and 0.1 mW, respectively. As the value of n has high impact on false detection (both false positive and false negative), we captured the results for different values of n to evaluate its impact. Figure 6(a) and 6(b) depict the missed detection ratio of the proposed CURD scheme with different values of μ_h, for 20 and 80 bits of reactive jamming duration, respectively. From this experiment, we infer that, irrespective of the values of μ_h, the CURD scheme provides zero missed detection ratio, i.e., 100% true detection, if we select $n = 40$ for the experimental setup whose parameters are listed in Table 1.

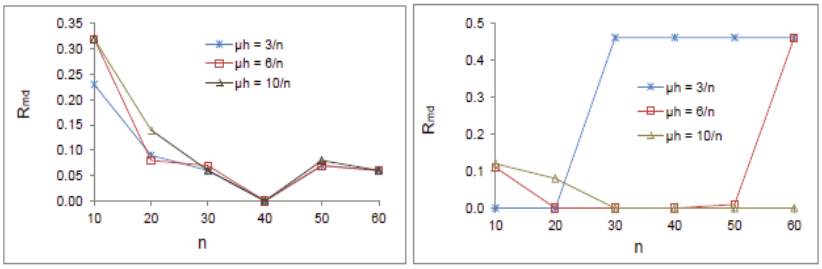
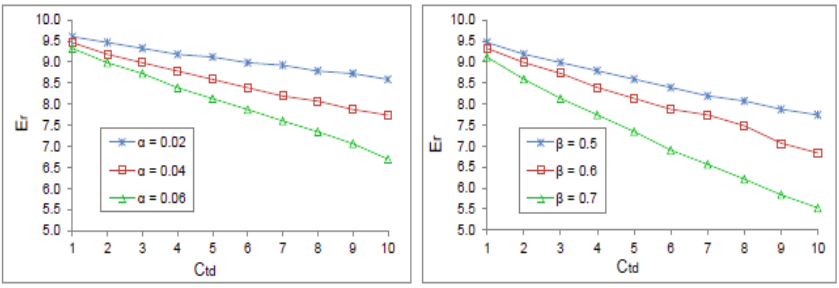


Fig. 6. Average missed detection ratio (a) $J_d = 20$ bits (b) $J_d = 80$ bits

6.4 Average Residual Energy

We measured the energy consumption rate of a tagged node in UWSN environment. Figures 7 (a) and 7 (b) show the average residual energy ($E_r^{'}$) of the tagged node for each successful detection of reactive jamming. The residual energy is defined as the ratio of the remaining energy to the initial energy of a node. The vertical axis in Figure 7 represents energy ratio, E_r, where $E_r = (E_r^{'} - 0.99) \times 1000$.



(a) (b)

Fig. 7. Average residual energy of a tagged node (a) $\beta = 0.5$ (b) $\alpha = 0.04$

As the performance of the CURD scheme depends on the selected threshold values α and β, we measured E_r with respect to the true detection count (C_{td}), for different values of α and β.

7 Conclusion

In this paper, we established that correct detection of reactive jamming in UWSN environment is possible. We developed a short-duration packet observation scheme using the concept of partial-packet, which helps to capture changes in a few bits too, and is capable of distinguishing the corrupted bits from the uncorrupted bits. The proposed metrics were designed in such a way that they absorb the effect of dynamic environment. We used a weak estimation learning scheme and a non-parametric CUSUM detector for detecting reactive jamming correctly. To show the common tasks of jamming detection in UWSNs, we stacked them up in a generalized framework for any reactive jamming detection. Finally, our validation study through simulations show that the partial-packet based reactive jamming detection scheme provides exactly 100% true detection, while the average detection delay equalling the equivalent of 3 packets reception.

In our future work, we will attempt to analytically evaluate the behaviour of the detection scheme, as the distribution of H_t follows the discrete time Markov model. We will endeavour to determine the optimal value of n, which influences detection accuracy greatly. We will try to implement the partial-packet ARQ with this detection mechanism to measure the improvement in network throughput and lifetime.

Acknowledgment. This work has been partially supported by a grant from the DIT, Govt. of India, Grant No. 13(10)/2009-CC-BT, which the authors gratefully acknowledge.

References

1. Akyildiz, I.F., Pompili, D., Melodia, T.: Underwater acoustic sensor networks: Research challenges. Ad Hoc Networks 3, 257–279 (2005)
2. Strasser, M., Danev, B., Capkun, S.: Detection of reactive jamming in sensor networks. ACM Transactions on Sensor Networks 7, 1–29 (2010)
3. Wilhelm, M., Martinovic, I., Schmitt, J.B., Lenders, V.: Short paper: Reactive jamming in wireless networks - how realistic is the threat? In: Proceedings of WiSec, Hamburg, Germany, pp. 47–52 (2011)
4. Domingo, M.: Securing underwater wireless communication networks. IEEE Wireless Communications 18, 22–28 (2011)
5. Misra, S., Singh, R., Mohan, S.V.R.: Information warfare-worthy jamming attack detection mechanism for wireless sensor networks using a fuzzy inference system. Sensors 10, 3444–3479 (2010)
6. Xu, W., Trappe, W., Zhang, Y., Wood, T.: The feasibility of launching and detecting jamming attacks in wireless networks. In: Proc. of MobiHoc, pp. 46–57 (2005)
7. Pelechrinis, K., Iliofotou, M., Krishnamurthy, S.V.: Denial of service attacks in wireless networks: The case of jammers. IEEE Communications Surveys & Tutorials 13, 245–257 (2011)
8. Ganti, R.K., Jayachandran, P., Luo, H., Abdelzaher, T.F.: Datalink streaming in wireless sensor networks. In: Proceedings of SenSys, pp. 209–222 (2006)
9. Jamieson, K., Balakrishnan, H.: PPR: Partial packet recovery for wireless networks. In: Proceedings of SIGCOMM (2007)
10. Oommen, B.J., Rueda, L.: Stochastic learning-based weak estimation of multinomial random variables and its applications to pattern recognition in non-stationary environments. Pattern Recognition 39, 328–341 (2006)
11. Poor, H., Hadjiliadis, O.: Quickest Detection. Cambridge University Press (2008)
12. Mpitziopoulos, A., Gavalas, D., Konstantopoulos, C., Pantziou, G.: A survey on jamming attacks and countermeasures in wireless sensor networks. IEEE Communications Surveys & Tutorials 11, 42–56 (2009)
13. Xu, W., Ma, K., Trappe, W., Zhang, Y.: Jamming sensor networks: Attacks and defense strategies. IEEE Network 20, 41–47 (2006)
14. Cagalj, M., Capkun, S., Hubaux, J.P.: Wormhole -based anti-jamming techniques in sensor networks. IEEE Transactions on Mobile Computing 6, 100–114 (2007)
15. Cakiroglu, M., Ozcerit, A.T.: Jamming detection mechanisms for wireless sensor networks. In: Proceedings of InfoScale, Vico Equense, Italy, pp. 1–8 (2008)
16. Li, M., Koutsopoulos, I., Poovendran, R.: Optimal jamming attack strategies and network defense policies in wireless sensor networks. IEEE Transactions on Mobile Computing 9, 1119–1133 (2010)
17. Tan, H.P., Diamant, R., Seah, W.K.G., Waldmeyer, M.: A survey of techniques and challenges in underwater localization. Ocean Engineering 38, 1663–1676 (2011)
18. Erol-Kantarci, M., Mouftah, H.T., Oktug, S.: A survey of architectures and localization techniques for underwater acoustic sensor networks. IEEE Communications Surveys & Tutorials 13, 487–502 (2011)
19. Xie, P., Zhou, Z., Peng, Z., Yan, H., Hu, T., Cui, J., Shi, Z., Pei, Y., Zhou, S.: AquaSim: an NS-2 based simulator for underwater sensor networks. In: Proceedings of OCEANS, Mississippi, USA, pp. 1–7 (2009)
20. Berkhovskikh, L., Lysanov, Y.: Fundamentals of Ocean Acoustics. Springer (1982)
21. Xie, P., Cui, J.-H., Lao, L.: VBF: Vector-Based Forwarding Protocol for Underwater Sensor Networks. In: Boavida, F., Plagemann, T., Stiller, B., Westphal, C., Monteiro, E. (eds.) NETWORKING 2006. LNCS, vol. 3976, pp. 1216–1221. Springer, Heidelberg (2006)

Fault-Tolerant Design of Wireless Sensor Networks with Directional Antennas

Shahrzad Shirazipourazad[1], Arunabha Sen[1], and Subir Bandyopadhyay[2]

[1] School of Computing, Informatics and Decision System Engineering
Arizona State University
Tempe, Arizona 85281, USA
{sshiraz1,asen}@asu.edu
[2] School of Computer Science
University of Windsor
Windsor, ON N9B 3P4, Canada
subir@uwindsor.ca

Abstract. A tree structure is often used in wireless sensor networks to deliver collected sensor data to a sink node. Such a tree can be built using directional antennas as they offer considerable advantage over the omni-directional ones. A tree is adequate for data gathering from all sensor nodes as long as no node in the tree fails. Since the connectivity of the tree is one, failure of any one node disconnects the tree and may disable the sink node from collecting data from some of the sensor nodes. In this paper we study the problem of enhancing the fault tolerance capability of a data gathering tree by adding a few additional links so that the failure of any one sensor would not disconnect the tree. Assuming that the addition of each link to the tree involves some cost, we study the problem of least-cost augmentation of the tree, so that even after failure of a single node, all the surviving nodes will remain connected to the sink node. We prove that the least-cost tree augmentation problem is NP-complete. Moreover, we provide an approximation algorithm with performance bound of two. The experimental evaluations of the algorithm demonstrate that the approximation algorithm performs even better in practice and almost always produces near-optimal solution.

1 Introduction

The primary goal of a wireless sensor network is to deliver the data collected by the sensor nodes to the sink node, possibly after some data aggregation. Generally, each sensor node has two components: the sensing component and the communication component. The sensing component gathers information from the surrounding area and the communication component transmits it to some other node for further processing. Most often, the data collection operation from all the sensor nodes is carried out by creating a tree topology that spans all the sensor nodes, with the sink node as the root [1,2].

Directional antennas offer substantial advantages over their omni-directional counterparts, as they can focus their transmission energy in a specific direction,

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 133–147, 2013.

using a narrow beam of width α. A directional antenna can be mounted on a *swivel* and can be oriented towards a target or alternately each sensor can be equipped with multiple antennas, each occupying a sector with beam width α. The transmitted signal disperses in any unguided wireless media and as a consequence, the signal strength diminishes with distance. Although attenuation is in general a complex function of the distance and the makeup of the environment through which the signal propagates, a significant cause of signal degradation is *free space loss*. Free space loss for an ideal isotropic antenna is measured as the ratio of the transmitted power to the received power and is given by $\frac{(4\pi d^2)}{\lambda^2}$. where λ is the carrier wavelength, and d is the propagation distance between transmission and reception antennas. In particular, the energy required by an antenna to reach all nodes within its transmission radius is proportional to the area covered. Thus, an omni-directional antenna with a transmission radius r will consume power proportional to πr^2 (the area of a circle with radius r) while a directional antenna with beam width α radians will consume power proportional to $\frac{\alpha}{2}r^2$. The expression is valid under the assumption that the signal is transmitted over the primary lobe and the power consumed by the remaining lobes is negligible [3]. The expressions show that with a directional antenna with beam width α, power consumption can be reduced by a factor of $\frac{\alpha}{2\pi}$. Moreover, in comparison with omnidirectional antennas, the directional antennas have significantly less interference and fading [4,5]. For additional information on antenna theory, we refer the reader to [6].

Due to such attractive features, sensors with directional antennas are being increasingly used for wireless sensor networks. Some examples include camera networks for vision-based sensing, radar networks for weather monitoring and sonar network for underwater object detection [5]. With rapid advances in the miniaturization of directional antenna technology, it is likely that the use of directional antennas in sensor platforms will proliferate. This trend is demonstrated by the increasing interest in the use of directional antennas for performance improvement in wireless networks in general and wireless sensor networks in particular [4].

Just as the directional antennas offer a number of advantages, it also introduces a few problems. When sensor nodes use omni-directional antennas, the network topology typically is a *mesh* and not a *tree*. A tree that spans over this mesh topology is utilized for the purpose of data gathering. Although it may appear that the use of omni-directional antenna for the purpose of data gathering is wasteful, as many of the network links created by such antennas are never utilized, this is not completely true. The unused links essentially introduce a certain level of *redundancy* that can be utilized when one or more of the sensor nodes fail and the data gathering spanning tree becomes disconnected.

One advantage of the directional antennas in the sensor application is that it can build the data gathering tree directly, instead of first creating a mesh network and then constructing a data gathering spanning tree for it (as is done with omni-directional antennas). The tree constructed with directional antennas is more efficient because it does not have the redundant links that are *created but*

not used by the omni-directional antennas. However, the negative aspect of this lack of redundancy is that it can no longer deal with a fault scenario, where one or more sensor nodes fail. Without any built-in redundancy, when some nodes fail, the data gathering tree is disconnected and the sink node fails to receive any data from some of the sensors.

The primary motivation of our work is to retain the advantage of both types of antennas by combining the *efficiency* of directional antennas with the *redundancy* of omni-directional ones. Specifically, our objective is to ensure that the failure of any one sensor node would not prohibit the surviving nodes from communicating with the sink node. We consider that the sensor nodes are equipped with directional antennas and nodes p and q can communicate with each other if antennas of p and q direct their beams to each other. In this case a bidirectional link is used between the nodes p and q. Fig. 1 shows a data collection tree with two sensor nodes u and v and a sink r. The sectors show the communication ranges and the lines show the wireless links. The addition of an edge between two nodes p and q in the sensor network topology corresponds to the deployment of two new directional antennas at the nodes p and q directed towards each other.

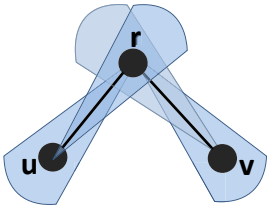


Fig. 1. A data collection tree constructed by directional antennas

Most of the prior studies on fault tolerant sensor network design [7–9] focus their attention to sensor nodes with omni-directional antennas. The primary goal of these studies is to compute transmission power to the nodes, so that the power consumption is minimized, subject to the constraint that the resulting network is k-connected. In a k-connected (i.e., $k-1$ fault-tolerant) network, there are k disjoint paths between every pair of nodes. However, for data collection in sensor networks, it is not necessary that the network be k connected. As long as every sensor node has k disjoint paths to the sink node, failure of $k-1$ sensors nodes can be tolerated. It may be noted that we assume that the sink node does not fail, i.e., it is more reliable than other sensor nodes. In [10, 11] the authors studied the problems of all-to-one and one-to-all k-fault-tolerant topology control problems. In these study also the sensor nodes are assumed to have omni-directional antennas.

In this paper we study the case of *single node failure* of sensor networks, where each node is equipped with directional antenna(s). Even for this restricted scenario, the problem turns out to be computationally hard. Assuming that the

addition of each link to the tree involves a cost, our objective is to solve the problem of *least-cost augmentation of the tree*, so that, even after the failure of a single node, all the surviving nodes will remain connected to the sink node. We will call this the *Tree Connectivity Augmentation* (TCA) problem. We prove that the TCA problem is NP-complete and we provide an approximation algorithm, with a performance bound of two. Experimental evaluation of the algorithm demonstrates that it performs even better in practice and almost always produces near-optimal solution.

In the theoretical computer science community, problems of this vein are known as the *graph augmentation problems*. Two important problems in this class are the bi-connectivity augmentation ($BICA$) and the bridge-connectivity augmentation ($BRCA$) (a bridge is defined to be an edge whose removal disconnects the graph) [12]. Although at a first glance, it may appear that TCA is the same as $BICA$ or $BRCA$, we demonstrate through the example shown in Fig. 2 that TCA is distinctly different from both $BICA$ and $BRCA$. The solid lines in Fig. 2 are the existing edges in the input graph. A few of the edges that may be added to the graph are shown in dashed lines and cost of each of these edges is 1. The cost of the edges that can be added to the graph, but not shown in dashed lines, is 10. In Fig. 2(a), the solution of the TCA is the addition of edges $\{(a, b), (c, d)\}$ with a total cost of 2. However, for $BICA$, more edges are needed and as such the total cost will be at least 12. The Fig. 2(b) shows an example in which TCA has a solution with cost 3, (addition of edges $\{(g, h), (h, i), (i, j)\}$) but $BRCA$ requires the addition of edges $\{(a, e), (g, h), (i, j), (d, f)\}$ with a total cost of 4.

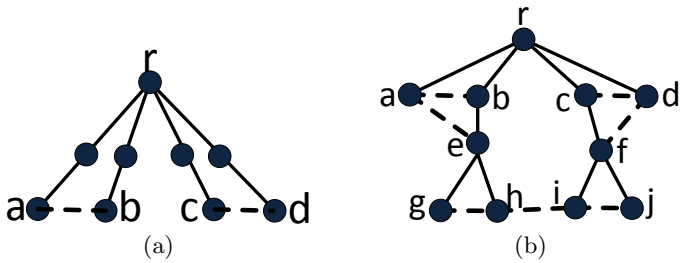


Fig. 2. Comparison of TCA in single fault model with (a) BICA (b) BRCA

Our presentation of the rest of the paper follows the following structure. In Section 2 we formally define the TCA problem and prove it to be NP-complete. In Section 3 we present an approximation algorithm for TCA with a performance bound of two. In Section 4 we report the results of our experimental evaluation of the approximation algorithm. We conclude by making a few remarks in Section 5.

2 Computational Complexity

In Section 1 we have indicated that our goal is to ensure that the data gathering tree in a sensor network with directional antennas do not get disconnected due to the failure of any one sensor node. This objective can be realized by ensuring that every node (except the root and its adjacent nodes) has an alternate path to the root (different from the path that exists in the data gathering tree). We formally state the problem below.

`Definition:` $u-$`fault tolerant graph`: A graph $G = (V, E)$ with a specified vertex $u \in V$ is said to be $u-$*fault tolerant* if after the failure of any one node $v \in V - \{u\}$, any residual node $w \in V - \{v\}$ remains connected to node u.

`Tree Connectivity Augmentation Problem (`TCA`)`

Instance: Complete undirected graph $G = (V, E)$, weight function $c(e) \in \mathbb{Z}^+$, $\forall e \in E$, a spanning tree $T_1 = (V, E_1)$ of G rooted at some node $r \in V$, and a cost budget B.
Question: Is there a set $E^{aug} \subseteq E - E_1$, such that the graph $(V, E_1 \cup E^{aug})$ is $r-$fault tolerant and $\sum_{e \in E^{aug}} c(e) \leq B$?

We show that the TCA problem is NP-complete, by a transformation from the *3-dimensional matching*, which is known to be NP-complete [13].

`3-Dimensional Matching (3DM)`
Instance: A set $M \subseteq W \times X \times Y$, where W, X and Y are disjoint sets having the same number q of elements.
Question: Does M contain a matching, that is, a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements of M' agree in any coordinate?

Theorem 1. *TCA is NP-complete.*

Proof. Let $M \subseteq W \times X \times Y$ be an instance of 3DM, with $|M| = p$ and $W = \{w_i | i = 1, 2, \ldots, q\}$, $X = \{x_i | i = 1, 2, \ldots, q\}$ and $Y = \{y_i | i = 1, 2, \ldots, q\}$. We start by creating a set of nodes having labels as follows:

- r, where r will be the root of the spanning tree T_1,
- w_i (x_i, y_i) for all $w_i \in W$ (respectively $x_i \in X$, and $y_i \in Y$),
- for each $w_i \in W$ $(x_i \in X, y_i \in Y)$, one additional node with label w_i' (respectively x_i' and y_i'),
- for each triple $(w_i, x_j, y_k) \in M$, three additional nodes with labels $a_{ijk}, a'_{ijk}, \bar{a}_{ijk}$.

We now create an instance of TCA as follows:

$$V = \{r\} \cup \{w_i, w_i', x_i, x_i', y_i, y_i' | i = 1, 2, \ldots, q\} \cup \{a_{ijk}, a'_{ijk}, \bar{a}_{ijk} | (w_i, x_j, y_k) \in M\}$$
$$E = \{(u, v) | u, v \in V \text{ and } u \neq v\}$$
$$\begin{aligned} E_1 = {} & \{(r, w_i), (w_i, w_i') | i = 1, 2, \ldots, q\} \cup \\ & \{(r, x_i), (x_i, x_i'), | i = 1, 2, \ldots, q\} \cup \\ & \{(r, y_i), (y_i, y_i') | i = 1, 2, \ldots, q\} \cup \\ & \{(r, a_{ijk}), (a_{ijk}, a'_{ijk}) | (w_i, x_j, y_k) \in M\} \cup \\ & \{(w_i', \bar{a}_{ijk}) | (w_i, x_j, y_k) \in M\} \end{aligned}$$

$B = p + q$
$c(x'_j, \bar{a}_{ijk}) = c(y'_k, a'_{ijk}) = c(\bar{a}_{ijk}, a'_{ijk}) = 1$, for all
$(w_i, x_j, y_k) \in M$. All other edges in E have weight 2.

We claim that M contains a matching M' iff there is a set E^{aug} of cost no more than B, such that the graph $(V, E_1 \cup E^{aug})$ is a $r-$fault tolerant graph.

To prove the only if part, let M contain a matching M'. We form E^{aug} by following the procedure given below:

Step i) For each triple $(w_i, x_j, y_k) \in M'$, we add edges $(x'_j, \bar{a}_{ijk})$ and (y'_k, a'_{ijk}),
Step ii) For each triple $(w_i, x_j, y_k) \in M - M'$, we add edge $(a'_{ijk}, \bar{a}_{ijk})$.

Since $|M'| = q$ and $|M - M'| = p - q$, and the cost of each edge added in Steps i and ii is 1, the total cost of the added edges in steps i and ii is $2q + p - q$. Thus, the total cost of the edges in E^{aug} is $p + q$. $E_1 \cup E^{aug}$ includes the following cycles that pass through r:

- $r, w_i, w'_i, \bar{a}_{ijk}, x'_j, x_j, \forall i, j, k : (w_i, x_j, y_k) \in M'$,
- $r, y_k, y'_k, a'_{ijk}, a_{ijk}, \forall i, j, k : (w_i, x_j, y_k) \in M'$,
- $r, a_{ijk}, a'_{ijk}, \bar{a}_{ijk}, w'_i, w_i, \forall i, j, k : (w_i, x_j, y_k) \in M - M'$.

It can be readily verified that all the nodes in $V - \{r\}$ appear in at least one of the above cycles. Therefore, there are two disjoint paths from r to each vertex.

To prove the if part, let there be a set of edges $E^{aug} \subseteq (E - E_1)$, with a cost of at most $p + q$, so that in the graph $(V, E_1 \cup E^{aug})$ every non-adjacent vertex of root r has two node disjoint paths to r. There are exactly $2p + 2q$ leaf nodes in $T_1 = (V, E_1)$ and they are not adjacent to r. Among these leaf nodes, there exists p nodes having labels of the form a''_{ijk} and $\bar{a}_{ijk}$ and q nodes having labels of the form y''_k and x''_j. To ensure that $2p + 2q$ leaf nodes have two node disjoint paths to r, at least $p + q$ edges must be added to $T_1 = (V, E_1)$. It may be noted that there are only three types of edges i) $(x'_j, \bar{a}_{ijk})$, ii) (y'_k, a'_{ijk}) and iii) $(\bar{a}_{ijk}, a''_{ijk})$ that have cost 1 and every other edge in E have a cost 2. Since the cost of the edges in E^{aug} is at most $p + q$ and $|E_{aug}| \geq p + q$, it implies that $|E^{aug}| = p + q$ and the cost of each edge in E^{aug} is 1.

In order to have two node disjoint paths from $2p + 2q$ leaf nodes to r, each node of the from y'_k and x'_j must be connected to a node of the form a'_{ijk} or $\bar{a}_{ijk}$. The total cost of this set of edges will be $2q$. Since the total cost E^{aug} is $p + q$, the cost of the edges to connect the remaining leaves of the type a'_{ijk} or $\bar{a}_{ijk}$, (i.e., the ones that were not connected to either y'_k and x'_j), must be $p - q$ and the number of such leaves must be $2(p - q)$. To connect $2(p - q)$ leaves, at least $p - q$ edges will be necessary. Since the total cost of these edges is $p - q$ and at least $p - q$ edges will be necessary, the cost of these edges must be 1. However, this will only be possible if $2(p - q)$ leaves can be grouped into $p - q$ pairs of nodes $(a'_{ijk}, \bar{a}_{i'j'k'})$, such that $i = i', j = j', k = k'$. This implies that exactly q nodes, each of the form a'_{ijk} or $\bar{a}_{ijk}$, must be connected to q nodes, each of the form y'_k and x'_j, and these $2q$ nodes (a'_{ijk} or $\bar{a}_{ijk}$) can be grouped into q pairs of nodes $(a'_{ijk}, \bar{a}_{i'j'k'})$, such that $i = i', j = j', k = k'$. Since these q pairs of ijk indices connects to all the y'_k and x'_j nodes, the corresponding subset $M' \subseteq M$ must be a matching for the instance of the 3DM problem.

3 Approximation Algorithm for the TCA

In this section we propose an approximation algorithm with a guaranteed performance bound for TCA. The input to the algorithm is a complete undirected graph $G_1 = (V, E)$ with cost function $c : E \rightarrow \mathbb{Z}^+$ defined on the edges, and $T_1 = (V, E_1)$, a spanning tree of G_1 with a specified vertex $r \in V$ as the root. The output is a set of edges $E^{aug} \subseteq E - E_1$, such that, in the graph $(V, E_1 \cup E^{aug})$, there are two node disjoint paths from every node v ($v \in V - r$) to the node r. Since the tree $T_1 = (V, E_1)$ is given as part of the input, we assume that the cost of the edges in E_1 is zero, i.e., we do not have to pay for these edges, $c(e) = 0, \forall e \in E_1$. We compute the edge set E^{aug} using a sequence of steps where, in each step, we construct a new graph/tree (undirected/directed). The sequence of construction of graphs is as follows: $[T_1 = (V, E_1)] \Rightarrow [T_2 = (V_2, E_2)] \Rightarrow [G_2 = (V_2, E_2')] \Rightarrow [T_2^d = (V_2, A_2)] \Rightarrow [G_2^d = (V_2, A_2')]$, where T_2 is a tree constructed from T_1, G_2 is a complete graph defined with the vertex set of T_2, T_2^d is a directed tree defined on T_2, and G_2^d is a completely connected directed graph defined with the vertex set of T_2^d. From G_2^d we identify a set of *arcs* (directed edges) A_2^{aug}, so that the directed graph $(V_2, A_2 \cup A_2^{aug})$ is *strongly connected* [14]. Finally, we construct E^{aug} from A_2^{aug}. We now describe, in detail, the construction rules for these graphs/trees.

`[A] Construction of` T_2: Let $V_p \subset V$ be the set of leaves in T_1 and let $V_q = V - (V_p \cup \{r\})$ be the set of all internal (non-leaf) nodes except the root. We define a new tree $T_2 = (V_2, E_2)$ using the following rules:

- $V_2 = V \cup \{v_{ij} | i, j \in V - \{r\}$ and $(i, j) \in E_1\}$.
- For each edge $(i, j) \in E_1$, we include in E_2,
  - edge (i, j), if $i = r$ or $j = r$,
  - edges (i, v_{ij}) and (v_{ij}, j), otherwise.

`[B] Construction of` G_2:
Let $G_2 = (V_2, E_2')$ be the complete graph defined on V_2. We define the cost function $c' : E_2' \rightarrow \mathbb{Z}^+ \cup \{\infty\}$ as follows. For every edge $(x, y) \in E_2'$, if $x, y \in V$, $c'(x, y) = c(x, y)$; otherwise, $c'(x, y) = \infty$ (i.e., we set the initial cost of the edges between a node $u \in V_2 - V$ and every other node in V_2 to infinity).
Next we define two functions $d(u, v)$ (distance function) and $p(u, v)$ (pointer function) for every pair of nodes u and v in G_2. We define both the functions in terms of c' and T_2.

`Definition:` $d(u, v) = min\{c'(x, y) | u$ and v are on the path from x to y in $T_2\}$.

`Definition:` $p(u, v)$ is a pointer to the edge (s, t), such that $d(u, v) = c'(s, t)$, where u and v are on the path from s to t in T_2.

`Example:` The Fig. 3(a) shows a spanning tree $T_1 = (V, E_1)$ of a complete graph $G_1 = (V, E)$ (for the sake of clarity, only three edges from the set $E - E_1$, $(a - b), (c - d)$ and $(d - e)$ are shown in Fig. 3(a)). The solid lines indicate the edges in E_1 and the dashed lines show a subset of the edges in $E - E_1$. The cost

of each edge in $E_1 = 0$. The weights associated with the dashed lines indicate the cost of these edges. All other edges in $E - E_1$, (not shown in Fig. 3(a)), have a cost of 10. Fig. 3(b) shows the tree $T_2 = (V_2, E_2)$ constructed from T_1 in Fig. 3(a). From T_2, we can construct the complete graph $G_2 = (V_2, E_2')$ following the construction rules described earlier. The solid lines in Fig. 3(b) indicate the edges in E_2 and the dashed lines show a subset of the edges in $E_2' - E_2$ (only five edges with associated weights are shown). In this example, $d(v_{ac}, v_{ad}) = c'(c,d) = c(c,d) = 1$, $d(a,b) = c'(d,e) = c(d,e) = 1$ and $p(v_{ac}, v_{ad}) = (c,d)$, $p(a,b) = (d,e)$.

We now discuss the rationale for definitions of the $d(u,v)$ and $p(u,v)$ given above. In order to have another path from a to b in Fig. 3(a), (different from the one in the tree T_2, $a-r-b$), the edge (d,e) with cost 1 or the edge (a,b) (in G_2) with cost 4 can be added to the tree. The addition of the link (d,e) will result in a cheaper path from a to b with cost 1. The goal of the distance function $d(u,v)$ is to identify this edge. The function $p(u,v)$ is defined to be a pointer to the edge selected by the function $d(u,v)$.

`Computation of` $d(u,v)$: It has been shown in [12] that $d(u,v)$ for all pairs of nodes can be computed in $O(|V|^2)$. For ease of reference, we summarize the algorithm for computing the function $d(u,v)$ presented in [12]. Initially, for every pair of nodes $u, v \in V_2, d(u,v) = c'(u,v)$ and $p(u,v) = (u,v)$. Let $l(u,v)$ be the number of edges on the path from u to v in T_2 and $s(u,v)$ be the node adjacent to v on this path. The edges $(u,v) \in E_2' - E_2$ are sorted in non-decreasing order, based on $l(u,v)$. For each edge $(u,v) \in E_2' - E_2$, we compute the distance function $d(u,v)$ as follows. If $d(u,v) < d(u, s(u,v))$ then $d(u, s(u,v)) = d(u,v)$ and $p(u, s(u,v)) = (u,v)$. If $d(u,v) < d(s(v,u), v)$, then $d(s(v,u), v) = d(u,v)$ and $p(s(v,u), v) = (u,v)$.

`[C] Construction of` $T_2^d = (V_2, A_2)$: We construct T_2^d from $T_2 = (V_2, E_2)$ by directing all edges of T_2 towards the root node r. We will use A_2 to represent the set of *arcs* (directed edges) corresponding to the undirected edges in E_2.

`[D] Construction of` $G_2^d = (V_2, A_2')$: G_2^d is a completely connected directed graph, with associated cost $c''(u,v)$ with each arc $u \to v \in A_2'$ as follows:

$$c''(u,v) = \begin{cases} \infty, & \text{if } v = r, \\ \infty, & \text{if } u \in V_q \text{ and } v \in subtree(u) \\ \text{d(u,v)}, & \text{otherwise.} \end{cases}$$

Here we define $subtree(u)$ to be the set of nodes in the subtree rooted at node u in tree T_2.

We note that each arc $u \to v \in A_2$ where $v \neq r$ has a zero cost. The rationale for assigning the arc costs in this specific way is as follows:

(a) By assigning a cost of ∞ to the edges where $v = r$, we ensure that the minimum cost arborescence on G_2^d is rooted at r,

(b) By assigning a cost of ∞ to the edges (u,v) where $u \in V_q$ and $v \in subtree(u)$ in T_2, we ensure that, in G_2^d, no node $u \in V_q$ will have a directed path from u to the nodes in $subtree(u)$, unless it first goes through some nodes not in $subtree(u)$.

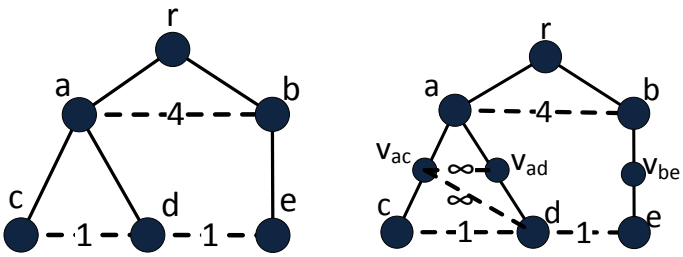


Fig. 3. (a) An example of T_1; E_1 includes the edges shown with solid lines. (b) The tree with solid lines is T_2 corresponding to T_1 in (a) and dashed lines are some of the edges in $E_2' - E_2$.

When constructing $G_2^d = (V_2, A_2')$, our goal is to identify a set of arcs A_2^{aug}, $(A_2^{aug} \subseteq A_2')$, so that the graph $(V_2, A_2 \cup A_2^{aug})$ is *strongly connected*, i.e., there exists a directed path between every pair of nodes in V_2. We obtain the arc set A_2^{aug} by computing the *least-cost arborescence* [15] in $G_2^d = (V_2, A_2')$, which we denote by $T_2^{arb} = (V_2, A_2^{arb})$. We obtain the set of arcs A_2^{aug} from A_2^{arb} by excluding those arcs with cost zero, i.e., $A_2^{aug} = A_2^{arb} - \{a \in A_2^{arb} | c''(a) = 0\}$. Finally, we construct the set of edges E^{aug} that we have to add to the input tree $T_1 = (V, E_1)$, to obtain the $r-$fault tolerant graph $(V, E_1 \cup E^{aug})$ as follows: $E^{aug} = \{p(u, v) | u \to v \in A_2^{aug}\}$.

Algorithm 1. *TCA* Algorithm

Input: $G_1 = (V, E)$, a complete graph with cost $c(e)$ for every edge $e \in E$; $T_1 = (V, E_1)$, a spanning tree of G_1 with root r.
Output: A set of edges $E^{aug} \subseteq E - E_1$, such that $(V, E_1 \cup E^{aug})$ is $r-$fault tolerant.

1: Construct $T_2 = (V_2, E_2)$, a complete graph $G_2 = (V_2, E_2')$ and the cost function $c'(.)$ from T_1 and G_1 using the technique described in [A].
2: Compute $d(u, v)$ and $p(u, v)$ for each pair of nodes $u, v \in V_2$ using the technique described in [B].
3: Compute a directed tree $T_2^d = (V_2, A_2)$ by directing all edges in E_2 toward root r using technique described in [C].
4: Compute a completely connected directed graph $G_2^d = (V_2, A_2')$ with cost c'' defined on the arcs set A_2' using technique described in [D].
5: Compute a minimum cost arborescence $T_2^d = (V_2, A_2^{arb})$ of the graph $G_2^d = (V_2, A_2')$.
6: Set $A_2^{aug} = A_2^{arb} - \{a \in A_2^{arb} | c''(a) = 0\}$.
7: Set $E^{aug} = \{p(u, v) | u \to v \in A_2^{aug}\}$.
8: Return E^{aug}.

We note that the time complexity of the *TCA* algorithm (Algorithm 1) is $O(|V|^2)$. Line 4 has $O(|V^2|)$ time complexity. Finding minimum cost arborescence also needs $O(|V|^2)$ time [15].

Theorem 2. *Algorithm TCA finds a set of edges E^{aug} such that $(V, E_1 \cup E^{aug})$ is $r-$fault tolerant.*

Proof. In order to prove that $(V, E_1 \cup E^{aug})$ is $r-$fault tolerant, we need to show that there is no node in V_q whose removal disconnects the graph (V_q is the set of all internal nodes in the tree $T_1 = (V, E_1)$ except the root r). Since the graph $(V_2, A_2 \cup A_2^{aug})$ is constructed by augmenting $T_2^d = (V_2, A_2)$ with the arcs of the T_2^{arb} (excluding the arcs that are already in A_2), it must be strongly connected. Accordingly, there must be a directed path from any node $v \in V_q$ to the nodes in $subtree(v)$. Let $w \neq v$ be a node in $subtree(v)$. Since there is no directed edge $\in A_2'$ going out of v to the nodes in $subtree(v)$ (because the cost of these edges is infinity), the first arc in a path from v to any other node in $subtree(v)$ should go through a node s where s is not in $subtree(v)$. Since the graph $(V_2, A_2 \cup A_2^{aug})$ is strongly connected, there must be a path from s to w in $(V_2, A_2 \cup A_2^{aug})$ not including v. Suppose that $c \rightarrow d \in A_2^{aug}$ is on the path from s to w which does not include v. If (c, d) is replaced by $p(c, d) = (e, f)$, ($e \in subtree(c)$ and $f \in subtree(d)$), the new path also will not include v, because v cannot be on the path from c to e or f to d in T_2. Hence, even if a node $v \in V_q$ is removed from the graph $(V, E_1 \cup E^{aug})$, the graph remains connected. Therefore, $(V, E_1 \cup E^{aug})$ is $r-$fault tolerant.

Theorem 3. *Algorithm TCA finds a set of edges E^{aug} with a total cost C^{aug}, such that $C^{aug} \leq 2C^{opt}$, where C^{opt} is the cost of the optimal solution.*

Proof: Our proof strategy is as follows. Let C^{opt} be the optimal cost of edges E^{opt} necessary to add to the input tree $T_1 = (V, E_1)$, so that the resulting graph becomes $r-$fault-tolerant and C^{aug} is the cost of edges E^{aug} selected by the TCA Algorithm. We show that there exists a subset of arcs A'' in the graph $G_2^d = (V_2, A_2')$ with three useful properties. If C'' is the cost of the arcs in $A'', A'' \subseteq A_2' - A_2$, (i.e., $C'' = \sum_{u \rightarrow v \in A''} c''(u, v)$), then (i) $C^{opt} \geq C''/2$, (ii) $C^{aug} \leq C''$, and (iii) the graph $(V_2, A_2 \cup A'')$ is strongly connected. From (i) and (ii) it follows that $C^{aug} \leq 2C^{opt}$.

We can compute the set of arcs $A'' \subseteq A_2' - A_2$ from the optimal solution $E^{opt} \subseteq E - E_1$ following the procedure described below.

Let Q be the set of nodes that are strongly connected in $(V_2, A_2 \cup A'')$. Initially, we set $A'' = \emptyset$, $Q = \{r\}$ and mark all the edges in E^{opt} as *unused*. We update Q, A'' and the marking of the edges in E^{opt} using the following procedure:
While $Q \neq V_2$ repeat the following steps:

- Select an unused edge (u, v) from E^{opt}, such that there is a node $t \in Q - V_q$ on the weakly directed path from u to v in $T_2^d = (V_2, A_2)$. (The weakly directed path from u to v in T_2^d is the path from u to v in T_2^d in which the direction of the arcs is ignored.)
- If $t \neq u$, add $t \rightarrow u$ to A'' and if $t \neq v$ add $t \rightarrow v$ to A''.
- Add all the nodes on the weakly directed path from u to v in T_2^d to Q. Since t has been selected from set Q, it is already accessible from root r in $(V_2, A_2 \cup A'')$. Therefore, by adding the new arcs to A'' in step (ii) all the

nodes on the weakly directed path from u to v in T_2^d are now accessible from root r in current augmented directed graph $(V_2, A_2 \cup A'')$.
- Change the marking of the edge (u, v) from *unused* to *used*.

We need to show that, during the execution of the iterative process, an *unused* edge $(u, v) \in E^{opt}$ and a suitable vertex $t \in Q - V_q$ exist. While there are some edges in E^{opt} which have not been used previously, there is still some node w in V_q whose deletion disconnects some node $a \in subtree(w)$ from the root in $(V_2, A_2 \cup A'')$. So, there is no directed path from r to a in $(V_2, A_2 \cup A'')$ during that iteration. Therefore, there should be an edge (u, v) in E^{opt} which creates a path from r to $a \in subtree(w)$ such that the path does not include w in $(V, E_1 \cup E^{opt})$. Also, the path from u to v in T_1 will contain more nodes from Q than just one vertex from V_q; otherwise, the removal of that vertex would disconnect the graph $(V, E_1 \cup E^{opt})$ which contradicts the fact that $(V, E_1 \cup E^{opt})$ is $r-$fault tolerant. Since there is no directed edge, in T_2^d, between the nodes in V_q, the weakly directed path from u to v in T_2^d should include a node $t \in Q - V_q$.

Let C'' be the cost of the arcs in A''; $C'' = \sum_{u \to v \in A_t''} c''(u, v)$. For every edge $(u, v) \in E^{opt}$, we have to add at most two arcs $t \to u$ and $t \to v$ to A''. Because $c''(t, u) \leq d(u, v)$ and $c''(t, v) \leq d(u, v)$, $C'' \leq 2C^{opt}$. Also we know that the graph $(V_2, A_2 \cup A'')$ is strongly connected. We can, therefore, construct an arborescence on $(V_2, A_2 \cup A'')$ rooted at r using c'' as the cost of the edges. Since the TCA algorithm gives us the minimum cost arborescence on G_2^d and $A_2 \cup A'' \subseteq A_2'$, $C^{aug} \leq C'' \leq 2C^{opt}$.

4 Experimental Results

In this section we present the experimental results of the proposed approximation algorithm for the TCA problem. In the experiments we compare the results of the approximation algorithm against the optimal solution. Moreover, we examine energy consumption by directional and omni-directional antennas deployed in a sensor network.

For every instance of our experiment, we generate the locations of the sensor nodes randomly, using a uniform distribution on a square deployment area of size 100×100 units. We take the cost of edge between the nodes u and v in the sensor network, as an indicator of the transmit power needed by the nodes to reach each other. Accordingly, we construct a complete graph $G = (V, E)$ by setting the cost of each edge $c(i, j)$ proportional to $d^2(i, j)$ where $d(i, j)$ is the Euclidean distance between nodes i and j. We assume that an omni-directional antenna will consume power proportional to r^2 where r is the radius of the coverage circle. A directional antenna with same transmit range r but with transmit beam width α degrees will consume power proportional to $\frac{\alpha}{360} r^2$ [3]. In our model, an edge represents two directional antennas transmitting signals to each other. Therefore, if the beam width is α we assume that the cost of the edge (i, j) is $\frac{2\alpha}{360} d^2(i, j)$. For each problem instance we compute the minimum spanning tree to be the initial tree T_1 and select a node randomly as the root of the tree.

In our first set of experiments our objective was to compare the results of the approximation algorithm with the optimal solution, obtained by solving an integer linear programming (ILP). We have denoted the ILP used to find the optimal solution of TCA by ILP. We used the CPLEX package to solve the ILP. Since ILP takes considerable amount of time in these experiments, we vary the number of nodes, n, from 5 to 25 in steps of 5. For each value of n, we generate 10 random instances of network layouts in a 2-dimensional plane. We set the beam width to 30 degrees. We present in Figure 4 the comparisons between the optimal augmentation cost and the augmentation cost computed by the TCA algorithm. For each n, we compute the average cost over 10 instances. We note that in these simulations, the ratio of the average cost of augmentation computed by the TCA algorithm to the average cost of optimal solution is smaller than 1.46. This indicates that in practice the performance of the TCA algorithm is significantly better than it's guaranteed worst case bound (2).

In our second set of experiments, we compare the power consumption of directional antennas versus omni-directional antennas. We vary the number of nodes, n, from 10 to 50 in steps of 10. For each value of n, we generate 50 random instances of network layouts in a 2-dimensional plane. We compute the average cost of augmentation by taking the average of costs incurred in these 50 instances. We perform the experiments for two values of beam width, 20 and 40 degrees. For each instance we execute the TCA algorithm. We assume that the transmit range of the omni-directional antenna is large enough to cover furthest neighbor in the augmented graph. Therefore, the total power consumption in network with omni-directional antennas is proportional to $\sum_{1\leq i\leq n} \max_{\{j|(i,j)\in E_1\cup E^{aug}\}} d^2(i,j)$. In the case of directional antennas, every edge in the augmented graph corresponds to two antennas, where each antenna needs $\frac{\alpha}{360}d^2(i,j)$ amount of power to reach the other. Hence, the total power consumption is proportional to $\sum_{(i,j)\in E_1\cup E^{aug}} \frac{2\alpha}{360}d^2(i,j)$. The Fig. 5 illustrates the comparison of power consumption in these two cases. We observe that the total power consumption in the network is significantly smaller when directional antennas are used instead of omni-directional ones. More specifically, when the beam width is 20 degrees, the power consumption with directional antennas is less than 9 percent of omni-directional antennas and when the beam width is 40 degrees it is less than 18 percent. We note that in order to make the network $r-$fault tolerant, we need to install additional directional antennas, which has a fixed cost associated with it. When nodes have omni-directional antenna, each node requires only one antenna. When we use directional antennas, the total number of antennas deployed in every sensor node is equal to the node degree (including both the initial set of antennas in the tree and the antennas installed to augment the tree). In Fig. 6 we illustrate the average number of antennas that is needed in the network for each value of n. The first diagram shows the total number of directional antennas needed in the network. We observe that the ratio of the total number of directional antennas to the number of omni-directional antennas in these experiments is less than 2.7. However, in the TCA problem we consider that the initial set of edges in the tree has cost zero (it is not part

of augmentation cost). Therefore, only the cost of the new (augmenting) edges (i.e., the cost of the corresponding additional antennas) that are added during augmentation phase needs to be considered. The third diagram in Fig. 6 depicts the average number of directional antennas that are added to the network during the augmentation process. This number is smaller than 75 percent of the number of omni-directional antennas. More accurately, augmenting directional antennas that are needed to make the network $r-$fault tolerant are fewer than 75 percent of number of omni-directional antennas. Therefore, the savings in power consumption with directional antennas outweighs the cost of additional directional antennas needed, particularly when the width of the antenna beam is narrow.

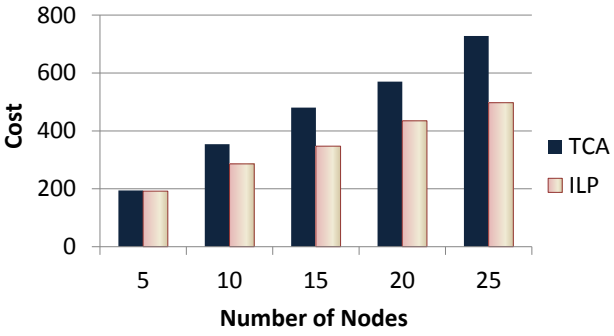


Fig. 4. Comparison of augmentation cost of TCA algorithm and ILP.

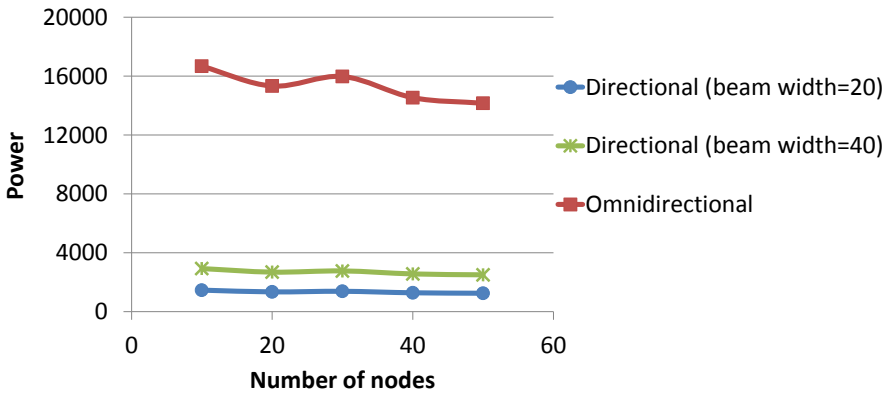


Fig. 5. Comparison between power consumption of directional antennas and omni-directional antennas

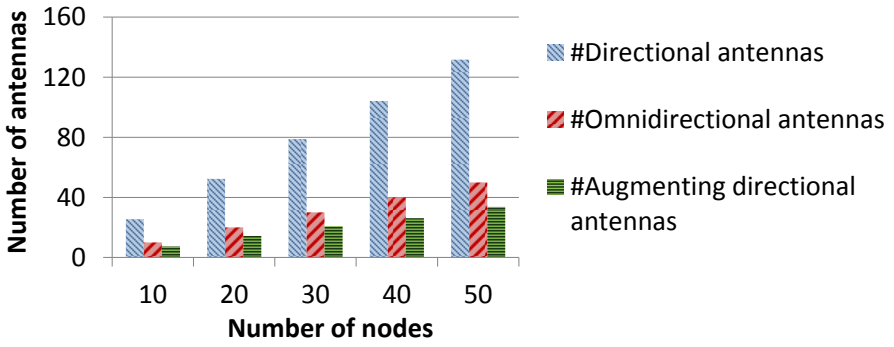


Fig. 6. Comparison between number of directional antennas and omni-directional antennas

5 Conclusions

This paper was motivated by the importance of both data collection and fault tolerance in wireless sensor networks. We have studied the problem of enhancing the fault tolerance capability of a sensor network where the sensor nodes are equipped with directional antennas using techniques for graph augmentation. We proved that the least cost tree augmentation problem to achieve resilience with respect to single faults is NP-complete. We have proposed an approximation algorithm, with a performance bound of two. Experimental evaluation of the approximation algorithm shows that it performs even better in practice. In future we plan to study the tree augmentation problem under more general topological fault models, where a fault is defined as any subgraph with diameter not exceeding d or when a fault is defined, based on the network geometry.

Acknowledgments. The research was supported in part by the DTRA grant HDTRA1-09-1-0032 and the AFOSR grant FA9550-09-1-0120.

References

[1] Incel, O., Krishnamachari, B.: Enhancing the data collection rate of tree-based aggregation in wireless sensor networks. In: Secon (2008)
[2] Li, X.Y., Wang, Y., Wang, Y.: Complexity of data collection, aggregation, and selection for wireless sensor networks. IEEE Transactions on Computers 60, 386–399 (2011)
[3] Kranakis, E., Krizanc, D., Williams, E.: Directional Versus Omnidirectional Antennas for Energy Consumption and k-Connectivity of Networks of Sensors. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 357–368. Springer, Heidelberg (2005)

[4] Yu, Z., Teng, J., Bai, X., Xuan, D., Jia, W.: Connected coverage in wireless networks with directional antennas. In: INFOCOM (2011)
[5] Wang, Y., Cao, G.: Minimizing service delay in directional sensor networks. In: INFOCOM (2011)
[6] Balanis, C.A.: Antenna Theory: Analysis and Design, 2nd edn. Wiley (1997)
[7] Bredin, J.L., Demaine, E.D., Hajiaghay, M., Rus, D.: Deploying sensor networks with guaranteed capacity and fault tolerance. In: MobiHoc (2005)
[8] Hajiaghayi, M., Immorlica, N., Mirrokni, V.: Power optimization in fault-tolerant topology control algorithms for wireless multi-hop networks. IEEE/ACM Transactions on Networking 15, 1345–1358 (2007)
[9] Pishro-Nik, H., Chan, K., Fekri, F.: Connectivity properties of large-scale sensor networks. Wireless Networks 15(7), 945–964 (2009)
[10] Wang, F., Thai, M., Li, Y., Cheng, X., Du, D.Z.: Fault-tolerant topology control for all-to-one and one-to-all communication in wireles networks. IEEE Transactions on Mobile Computing 7, 322–331 (2008)
[11] Cardei, M., Yang, S., Wu, J.: Algorithms for fault-tolerant topology in heterogeneous wireless sensor networks. IEEE Trans. Parallel Distrib. Syst. 19(4), 545–558 (2008)
[12] Frederickson, G.N., Ja'Ja', J.: Approximation algorithms for several graph augmentation problems. SIAM J. on Computing 10, 270–283 (1981)
[13] Garey, M., Johnson, D.: Computers and intractability. A guide to the theory of NP-completeness. Freeman (1979)
[14] West, D.B.: Introduction to Graph Theory, 2nd edn. Prentice Hall (2001)
[15] Tarjan, R.E.: Finding optimum branchings. Networks 7, 25–35 (1977)

Self-stabilizing Silent Disjunction in an Anonymous Network

Ajoy K. Datta[1], Stéphane Devismes[2], and Lawrence L. Larmore[1]

[1] Department of Computer Science, University of Nevada Las Vegas, USA
[2] VERIMAG UMR 5104, Université Joseph Fourier, France

Abstract. Given a fixed *input bit* to each process of a connected network of processes, the *disjunction problem* is for each process to compute an *output bit*, whose value is 0 if all input bits in the network are 0, and 1 if there is at least one input bit in the network which is 1. A uniform asynchronous distributed algorithm DISJ is given for the disjunction problem in an anonymous network. DISJ is self-stabilizing, meaning that the correct output is computed from an arbitrary initial configuration, and is silent, meaning that every computation of DISJ is finite. The time complexity of DISJ is $O(n)$ rounds, where n is the size of the network. DISJ works under the *unfair daemon*.

Keywords: anonymous, disjunction, self-stabilization, silence, unfair daemon.

1 Introduction

Given a network of processes $\mathcal{G}$, where each process has a fixed *input bit*, $Input(x)$, the *disjunction problem* is for each process to compute $Output = \bigvee_{x \in \mathcal{G}} Input(x)$, the disjunction of all input bits in the network.

A *distributed solution* to the disjunction problem is a distributed algorithm which computes an *output bit* for each process, such that all output bits are equal to *Output*. The solution given in this paper, the distributed algorithm DISJ, correctly solves the disjunction problem if the network is connected. DISJ is self-stabilizing [1,2], meaning that a correct output configuration is reached in finite time after arbitrary initialization, and is silent, meaning that eventually the computation of DISJ will halt. DISJ works under the *unfair* scheduler (daemon).

DISJ is uniform, meaning that every process has the same program, and is anonymous, meaning that processes are not required to have distinguished IDs. The *round complexity* of DISJ is $O(n)$, where n is the size of the network. We use the composite model of computation [2].

1.1 Related Work

We are not aware of closely related work in the literature. Although we use some of the same techniques in this paper that are used for leader election, the

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 148–160, 2013.

disjunction problem in an anonymous network cannot be solved by using a leader election algorithm, nor by using an algorithm to construct a spanning tree. In fact, there is no distributed algorithm which elects a leader or which constructs a spanning tree for general anonymous networks.

1.2 Outline of the Paper

In Section 2, we explain our model of computation. In Section 3, we give the formal definition of DISJ. In Section 4, we sketch the proof of the self-stabilization, silence, and time complexity of DISJ.

2 Preliminaries

We assume that we are given an anonymous network of processes. Let $N(x)$ be the set of *neighbors* of a process x.

A *self-stabilizing* [1,2] system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the system. In particular, a self-stabilizing algorithm distributed will eventually reach a *legitimate state* within finite time, regardless of its initial configuration, and will remain in a legitimate state forever. An algorithm is called *silent* if eventually all execution halts.

In the composite atomicity model of computation, each process has variables. Each process can read the values of its own and its neighbors' variables, but can write only to its own variables. We assume that each transition from a configuration to another, called a *step* of the algorithm, is driven by a *scheduler*, also called a *daemon*.

The *program* of each process consists of a finite set of *actions* of the following form: $< label >:: < guard > \longrightarrow < statement >$. The *guard* of an action in the program of a process x is a Boolean expression involving the registers of x and its neighbors. The *statement* of an action of x updates one or more variables of x. An action can be executed only if it is *enabled*, *i.e.,* its guard evaluates to true. A process is said to be enabled if at least one of its actions is enabled. A *step* $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more enabled processes executing an action. Evaluation of all guards and execution of all statements of an action are presumed to take place in one atomic step. A distributed algorithm is called *uniform* if every process has the same program.

We use the *distributed daemon*. If one or more processes are enabled, the daemon *selects* at least one of these enabled processes to execute an action. We also assume that daemon is *unfair*, *i.e.,*that it need never select a given enabled process unless it becomes the only enabled process.

We define a *computation* to be a sequence of configurations $\gamma_p \mapsto \gamma_{p+1} \ldots \mapsto \gamma_q$ such that each $\gamma_i \mapsto \gamma_{i+1}$ is a step.

We measure the time complexity of DISJ in *rounds* [2]. We say that a finite computation $\varrho = \gamma_p \mapsto \gamma_{p+1} \mapsto \ldots \mapsto \gamma_q$ is a *round* if every process which is

enabled at γ_p is either neutralized or executes an action at some step, and if the computation $\gamma_p \mapsto \gamma_{p+1} \mapsto \ldots \mapsto \gamma_{q-1}$ does not satisfy that condition. We define the *round complexity* of a computation to be the number of disjoint rounds in the computation.

3 DISJ

In this section, we give the formal definition of our algorithm, DISJ, which solves the disjunction problem in an anonymous connected network, $\mathcal{G}$. The fundamental idea of DISJ is to build a *local BFS tree* rooted at every process whose input bit is 1. Each process will join the tree rooted at the nearest process with input bit 1; ties will be broken arbitrarily. The construction of the BFS trees is by flooding.

The main difficulty with this method is the possibility that, in the initial configuration (which is arbitrary) there could be "fictitious" BFS trees. It is necessary to delete all such fictitious trees. This is an easy task if $Output = 1$, but is difficult if $Output = 0$. If a process does not have a parent in one of the trees, it will delete itself from the structure. Our problem is to ensure that a fictitious tree does not grow as fast at the leaf end as it deletes itself from the root end.

The method we use to ensure deletion of fictitious trees is derived from the *color wave* method of [3]. Each process in a tree or fictitious tree, whether true or fictitious, has a *color*, either 0 or 1. A process can only recruit a new process to the tree if its color is 1, and the recruited process will initially have color 0. Colors change in pipelined convergecast waves. Colors in a tree must alternate, and to allow the color waves to continue upward, the root of each BFS tree (a process whose input bit is 1) must "absorb" each wave. A fictitious tree will not be rooted at a process with input bit 1, and thus color waves will not be absorbed. "Color lock," the situation where the waves are maximally crowded and cannot move up, will eventually stop the growth of the fictitious tree.

We use the concept of *energy* introduced in [3]. $Energy(x)$ is a positive integer for each process x whose output bit is 1, and zero for processes with output bit 0. If $Output = 0$, $Energy(x) \leq 2n$ for all x, and the maximum value of *Energy* decreases by at least 1 during every round, and thus must eventually reach zero. At that point, every process has output bit 0. In one more round, DISJ converges.

3.1 Definition of DISJ

Recall that $Input(x)$ is the input bit of the process x, and that *Output* is the disjunction of all input bits. Define $I_i = \{x : Input(x) = i\}$, for $i = 0, 1$. Thus, $Output = 0$ if $I_1 = \emptyset$, while $Output = 1$ otherwise. If x, y are processes, let $||x, y||$ be the distance ("hop-distance") from x to y. If S is a non-empty set of processes, let $||S, y|| = \min\{||x, y|| : x \in S\}$. We let $||\emptyset, y|| = \infty$. Finally, let $L(x) = ||I_1, x||$.

Variables of DISJ. Each process x has the following variables.

1. $x.out \in \{0,1\}$, the *output bit* of x.
   When DISJ halts, $x.out = Output$ for all x. We let $O_i = \{x : x.out = i\}$, for $i = 0, 1$. During a computation of DISJ, the sets O_i can change.
2. $x.parent \in N(x) \cup \{\perp\}$, the *parent* of x.
   If $Output = 0$, then $x.parent = \perp$ for all x when DISJ halts. If $Output = 1$, then, when DISJ halts, $x.parent = \perp$ for all $x \in I_1$, while $x.parent$ is the parent pointer of x in the local BFS tree rooted at the nearest member of I_1 if $x \in I_0$.
3. $x.level \geq 0$, integer or ∞, the *level* of x.
4. $x.color \in \{0,1\}$, the *color* of x.
   If $x.out = 0$, the value of $x.color$ is irrelevant. The purpose of the color variable is to ensure that eventually $x.out = 0$ for all x, if $Output = 0$. The main difficulty of the problem in that case is eliminating processes with output bit 1. We accomplish this task by using *color waves*, which ensure that "fictitious" trees shrink faster than they grow.
5. $x.done$, Boolean. This variable is irrelevant if $x.out = 0$. If $Output = 1$, the variable *done* is used to achieve silence when all BFS trees have been constructed. In that case, $x.done =$ TRUE for all x when DISJ halts.

Functions and Sets. The following functions can be computed by any given process x by examining its own and its neighbors' variables.

1. $Level(x) = \begin{cases} 0 \textbf{ if } Input(x) = 1 \\ \infty \textbf{ if } Input(x) = 0 \text{ and } N(x) \subseteq O_0 \\ 1 + \min\{y.level : y \in N(x) \cap O_1\} \textit{ otherwise} \end{cases}$
   When DISJ halts, $x.level = Level(x) = L(x)$ for all x.
2. $Chldrn(x) = \begin{cases} \{y \in N(x) \cap O_1 : y.parent = x \text{ and } y.level = 1 + x.level\} \textbf{ if } x \in O_1 \\ \emptyset \textbf{ if } x \in O_0 \end{cases}$
   the *children* of x in its local BFS tree.
3. $0\_Valid(x)$, Boolean, meaning that x is in a valid state with output bit 0, which is true if and only if all the following conditions hold.
   (a) $x \in O_0$
   (b) $x.level = Level(x)$
   (c) $x.parent = \perp$
4. $1\_Valid(x)$, Boolean, meaning that x is in a valid state with output bit 1, which is true if and only if all the following conditions hold.
   (a) $x \in O_1$
   (b) $x.level = Level(x)$
   (c) If $Input(x) = 0$ then $x.parent \in O_1$ and $x.level = 1 + x.parent.level$.
   (d) If $Input(x) = 1$ then $x.parent = \perp$ and $x.level = 0$.
   (e) If $y \in N(x)$ then $y.level + 1 \geq x.level$.
5. $Valid(x) \equiv 0\_Valid(x) \vee 1\_Valid(x)$, Boolean, meaning that x is *valid*. If $Valid(x) =$ FALSE, we say x is *invalid*.
   An invalid process x is enabled to execute the Reset action, A1, which causes x to become valid.

6. *Can_Recruit*(x), Boolean, meaning that there is a neighbor of x which can be recruited by x. This function is TRUE if and only 1_*Valid*(x) and there is some $y \in N(x) \cap O_0$ such that $y.level = x.level + 1$.
7. *Done*(x), Boolean, indicates that there should be no further recruitment of processes by x or any descendant of x in its local BFS tree.
   This function is TRUE if and only if 1_*Valid*(x), *Can_Recruit*(x) = FALSE, and $y.done$ for all $y \in Chldrn(x)$.

Actions of DISJ

We list the actions of DISJ, in Table 1. The first column of the table gives the name of the action, as well as its *priority*. The second column gives an informal name of the action.

The guard of each action is a Boolean function, which we express as a list of *clauses* in the third column. Each guard is the conjunction of the clauses. If the priority of an action is not 1, there is an additional unlisted clause, which states that no action of higher priority is enabled. For example, if the priority of an action is 3, it is not enabled if an action of priority 1 or 2 is enabled.

The fourth column of Table 1 lists the *statement* of each action. If a process is enabled and executes an action, then the statement, which consists of a list of assignments of values to the process' local variables, is executed.

We follow Table 1 by a detailed explanation of each of the actions.

Explanation of the Actions of DISJ**.** We now give a detailed explanation of each of the actions of DISJ.

Action A1 **(Reset):** If a process x is invalid, it executes A1, and then becomes 0-valid. An invalid process cannot change its parent or its level without first executing A1.

Action A2 **(Finish):** If $x \in O_1$, and if it appears to x, by looking at its own and its neighbors' variables, that construction of the local BFS trees is done, then $x.done$ is changed to TRUE. Alternatively, if $x \in O_1$ can determine that the local BFS trees are not finished, $x.done$ is changed to FALSE. Both changes are accomplished by the execution of Action A2.

When construction of local BFS trees is finished, all the *done* variables change to TRUE in a convergecast wave beginning at the leaves. When that wave reaches $x \in I_1$, then x can no longer execute Action A6, causing *color lock* to percolate down its tree. When that happens with every tree, all executions of Action A5 cease, and the configuration is final.

Action A3 **(Initialize):** If a process $x \in I_1$ is 0-valid, it initiates a local BFS tree with itself as the root, unless there is some neighbor y such that $y.parent = x$. The reason for this clause is that, otherwise, y could accidentally and erroneously link with the local BFS tree.

If such a y exists, then y is invalid, which implies that it will execute Action A1 during the next round, after which x is enabled to execute A3.

Action A4 **(Join):** If a process $x \in I_0$ is 0-valid and has a neighbor $y \in O_1$, and if $y.color = 1$ and $x.level = 1 + y.level$, then x can join y by executing

Table 1. Actions of DISJ

A1 priority 1	Reset	$\neg Valid(x)$	$\longrightarrow$	$x.out \leftarrow 0$ $x.level \leftarrow Level(x)$ $x.parent \leftarrow \perp$
A2 priority 2	Finish	$x.out = 1$ $x.done \neq Done(x)$	$\longrightarrow$	$x.done \leftarrow Done(x)$
A3 priority 3	Initialize	$Input(x) = 1$ $0\_Valid(x)$ $\forall y \in N(x) : y.parent \neq x$	$\longrightarrow$	$x.out \leftarrow 1$ $x.color \leftarrow 1$ $x.level \leftarrow 0$ $x.done \leftarrow$ FALSE
A4 priority 3	Join y	$y \in N(x)$ $Input(x) = 0$ $0\_Valid(x)$ $y.out = 1$ $y.level + 1 = x.level$ $y.color = 1$ $\forall z \in N(x) : z.parent \neq x$	$\longrightarrow$	$x.parent \leftarrow y$ $x.out \leftarrow 1$ $x.color \leftarrow 0$ $x.done \leftarrow$ FALSE
A5 priority 3	Reverse Color	$1\_Valid(x)$ $Input(x) = 0$ $\neg Can\_Recruit(x) \vee (x.color = 0)$ $x.parent.color = x.color$ $\forall y \in Chldrn(x) : y.color \neq x.color$	$\longrightarrow$	$x.color \leftarrow \neg x.color$
A6 priority 3	Absorb Color	$1\_Valid(x)$ $Input(x) = 1$ $\neg Can\_Recruit(x) \vee (x.color = 0)$ $\forall y \in Chldrn(x) : y.color \neq x.color$ $\neg x.done$	$\longrightarrow$	$x.color \leftarrow \neg x.color$

Action A4, unless there is some neighbor z such that $z.parent = x$. The reason for this clause is the same as the reason given for Action A3.

When x joins y, $x.color \leftarrow 0$. Thus, x starts a 0-color wave, which follows the 1-color wave that y belongs to.

Action A5 (Reverse Color): Color waves alternate in color, and no color wave can pass its preceding color wave. This rule is enforced by the guard of A5. In order for the next color wave to reach x, that wave must have already reached all children of x (if there are no children, then x initiates a new color wave by executing A5) and the current color wave of x must already have reached $x.parent$.

Action A6 (Absorb Color): Since color waves alternate colors and cannot pass each other, eventually every chain would have alternating colors, *i.e.,*

x and y would have different colors if $y = x.parent$. This situation is called *color lock*. A color locked chain can only recruit a process if its last process has color 1, and after it recruits that new process, which then has color 0, no further recruitment is possible. Thus, in order for the local BFS trees to grow, it is necessary for the root processes to *absorb* color waves. Action A6 by a process $x \in I_1$ consists of simply allowing the color wave that has reached its children to move up to x. This then destroys (absorbs) the process' current color wave.

If $x \in I_1$ and $x.done = \text{TRUE}$, the local tree is complete, and color locking is desired. In this case, x refuses to absorb its current color wave, the color waves "pile up" behind it, and color lock is achieved. When all local BFS trees reach color lock, the configuration of DISJ is final, and $x.out = 1$ for all x.

3.2 Legitimate Configurations

There are two kinds of legitimate configurations. We say that a process x is in a *legitimate state of type 0* if the following conditions hold.

1. $Input(x) = 0$.
2. $0\_Valid(x)$.
3. $N(x) \subseteq O_0$.

We say that a process x is in a *legitimate state of type 1* if the following conditions hold.

1. $1\_Valid(x)$.
2. $x.done$.
3. $y.done$ for all $y \in Chldrn(x)$.
4. If $Input(x) = 0$, then $x.parent.color \neq x.color$.

We say that a configuration is *legitimate*, of type 0 or 1, if all processes are in a legitimate state of type 0 or 1, respectively.

Properties of Legitimate Configurations. If the configuration is legitimate of type 0, all processes have the same state, where $level = \infty$. If the configuration is legitimate of type 1, the network is partitioned into clusters, each of which contains exactly one member of I_1. Each process belongs to the cluster containing the nearest member of I_1 (where ties are broken arbitrarily), and the *parent* pointers of the processes of each cluster form a BFS tree rooted at its member of I_1.

4 Self-stabilization and Silence

Our main result is Theorem 4.19 below, which follows immediately from the lemmas proved in this section.

4.1 Legitimacy and Silence

Remark 4.1. *Every legitimate configuration is final.*

Proof. Assume that the configuration is legitimate, and let x be a process.
Since $Valid(x)$, x cannot execute Action A1.

Suppose x is legitimate of type 0.
Since $x \in O_0$, x cannot execute Action A2.
Since $Input(x) = 0$, x cannot execute Action A3.
Since $N(x) \subseteq O_0$, x cannot execute Action A4.
Since $x \in O_0$, $1\_Valid(x) = \text{FALSE}$, and thus x cannot execute either Action A5 or A6.

Suppose x is legitimate of type 1.
For any $y \in N(x)$, since y is legitimate, $y \in O_1$ and $y.done = \text{TRUE}$. Thus, $Done(x) = \text{TRUE}$, and hence x cannot execute Action A2.
Since $x \in O_1$, x cannot execute either Action A3 or A4.
If $x \in I_0$, then $x.parent$ is legitimate of type 1. Then x cannot execute Action A5 since $x.parent.color \neq x.color$.
If $x. \in I_1$, then x cannot execute Action A6 since $x.done = \text{TRUE}$.
Thus, in either case, x is not enabled. and we are done.

We now prove the converse of Remark 4.1.

Lemma 4.2. *Every final configuration is legitimate.*

Proof. Assume that the current configuration of DISJ is final, but not legitimate. For any process x, we have $Valid(x) = \text{TRUE}$, and $x.done = Done(x)$ if $x \in O_1$, since otherwise x would be enabled to execute either Action A1 or A2.

Our proof is by contradiction. Assume that not all processes are in a legitimate state.

Case I: There is some $x \in O_0$ where x is not in a legitimate state, and $N(x) \subseteq O_0$. Then $0\_Valid(x)$. Since x is not legitimate of type 0, $Input(x) = 1$. Since all neighbors of x are valid, x is enabled to execute Action A3, contradiction.

Case II: There is some $x \in O_1$ such that $x.level > 0$ and $x.parent.color = x.color$. Without loss of generality, the level of x is maximum, *i.e.,* $y.parent.color \neq y.color$ for all $y \in O_1$ such that $y.level > x.level$.

If $Can\_Recruit(x) = \text{FALSE}$, then x is enabled to execute Action A5, since $y.color \neq x.color$ for all $y \in Chldrn(x)$. Suppose $Can\_Recruit(x) = \text{TRUE}$. Then there exists $y \in N(x) \cap O_0$ such that $y.level = 1 + x.level$, and $Level(y) = 1 + x.level$ since y is legitimate. Thus, y is enabled to execute Action A4. In either case, we have a contradiction.

Case III: There are processes $x \in O_0$ and $y \in N(x) \cap O_1$, and $z.color \neq z.parent.color$ for all $z \in O_1$ such that $z.level > 0$. Let r be the end of the chain

starting with y and following *parent* pointers. Then $r \in O_1$ and $r.level = 0$. Since $y.done = \text{FALSE}$, it follows by induction along the chain that $r.done = \text{FALSE}$. If $r \in B_1$ and $Can\_Recruit(r)$, then some neighbor of r can execute Action A4, contradiction. Otherwise, r is enabled to execute Action A6, contradiction.

4.2 Characteristics of a Legitimate Configuration

Lemma 4.3. *In a legitimate configuration, $x.level = L(x)$ for all x.*

Proof. By Remark 4.1, the configuration is final. If $Output = 0$, then $Level(x) = \infty$, since otherwise Action A1 would be enabled.

Suppose $Output = 1$, and $x.level \neq L(x)$ Without loss of generality, $L(x)$ is minimum subject to that condition. If $x \in I_1$, then x is enabled to execute Action A1, contradiction. Henceforth, assume $x \in I_0$, which implies that $x.level \neq L(x) =\geq 0$.

Case I: $x.level > L(x)$. Pick $r \in I_1$ such that $||r, x|| = L(x)$. Pick $y \in N(x)$ on the shortest path from x to r. Then $Level(x) \leq 1 + y.level = 1 + L(y) = L(x) < x.level$. Thus, x is enabled to execute Action A1, contradiction.

Case II: $x.level < L(x)$. For all $y \in N(x)$, $L(y) \geq L(x) - 1$, by the triangle inequality. Thus $Level(x) \geq L(x) > x.level$, which implies that x is enabled to execute Action A1, contradiction.

Corollary 4.4. *In a configuration which is legitimate of type 1, the network is partitioned into clusters, each containing one member of I_1. In each cluster, the parent pointers form a BFS tree rooted at its member of I_1.*

4.3 Energy

At any configuration of DISJ, and for any process x, let

$$Energy(x) = \begin{cases} 0 \text{ if } x.out = 0 \\ 1 \text{ if } (x.out = 1) \wedge (Chldrn(x) = \emptyset) \wedge (x.color = 0) \\ 2 \text{ if } (x.out = 1) \wedge (Chldrn(x) = \emptyset) \wedge (x.color = 1) \\ \max\Big(\{1 + Energy(y) : (y \in Chldrn(x)) \wedge (y.color \neq x.color)\} \cup \\ \quad \{2 + Energy(y) : (y \in Chldrn(x)) \wedge (y.color = x.color)\}\Big) \text{ otherwise} \end{cases}$$

We define Max_Energy to be the maximum energy of processes in the network.

Lemma 4.5. *Execution of Action A1, A2, A4, or A5 by any process does not increase Max_Energy.*

Proof. If a process x executes Action A1, there is no effect on Max_Energy if $x \in O_0$. If $x \in O_1$, then $Energy(x) \leftarrow 0$, and the effect on the energy of any process is non-positive. If x executes Action A2, there is no effect on color, and thus Max_Energy is unaffected.

Suppose x executes Action A4, attaching itself to $y \in N(x)$. Then $Energy(x) \leftarrow 1$, but $Energy(y) = 2$ before the step. Thus, Max_Energy does not decrease.

Suppose x executes Action A5. Let $y = x.parent$. Before the step, $Energy(y) \geq Energy(x) + 2$. Thus, the action could increase the energy of x by at most one.

Energy Plus Level. For any process x, define

$$Energy\_plus\_level(x) = \begin{cases} 0 \text{ if } x.level = \infty \\ Energy(x) + x.level \text{ otherwise} \end{cases}$$

Define $Max\_Energy\_plus\_level$ to be the maximum value of $Energy\_plus\_level(x)$ over all x.

Lemma 4.6. *The value of* $\max \left\{ \begin{matrix} Max\_Energy\_plus\_level \\ 2n \end{matrix} \right\}$ *cannot increase.*

Proof. If x is not a root, then $Energy\_plus\_level(x) \leq Energy\_plus\_level(x.parent)$ by the definition of $Energy$. Thus, the maximum value of $Energy\_plus\_level$, if greater than zero, is always achieved at a root, either a true root or a false root.

Let $\gamma \mapsto \gamma'$ be a step, and let M, M' be the values of $\max\{2n, Max\_Energy\_plus\_level\}$ at γ and γ', respectively. We use "prime" notation for the values of variables and functions at γ', and no 'prime' to indicate values at γ.

We need to prove $M' \leq M$. If $M' \leq 2n$, we are done. Therefore, we can assume that $M' > 2n$. Pick x such that $Energy\_plus\_level'(x) = M'$. If x is a true root at γ', then $Energy\_plus\_level'(x) = Energy'(x) \leq 2n$. Thus, x is a false root at γ'.

If x did not execute at the step, then $Energy'(x) \leq Energy(x)$ and thus $M' = Energy\_plus\_level'(x) \leq Energy\_plus\_level(x) \leq M$. If x executed at the step, then x could not have been a false root at γ, Let $y = x.parent$. Then $Energy'(x) < Energy(y)$ by the definition of $Energy$. Since $x.level = 1 + y.level$, we have $M' = Energy\_plus\_level'(x) \leq Energy\_plus\_level(y) \leq M$.

4.4 Silence

We define an infinite computation of an algorithm to be *repetitive* if every configuration that occurs in the computation occurs infinitely often, and if every transition between two configurations that occurs also occurs infinitely often.

Lemma 4.7. *If* DISJ *has an infinite computation, then* DISJ *has a repetitive infinite computation.*

Proof. Let Γ be an infinite computation of DISJ on a network $\mathcal{G}$. Let M be the value of $Max\_Energy\_plus\_level$ at the first configuration of Γ. By Lemma 4.6, $Max\_Energy\_plus\_level \leq M$ at all configuration of Γ. Thus, the number of possible values of $x.level$ for any given process $x \in \mathcal{G}$. is bounded by $M + 1$.

The number of possible values of $x.parent$ is bounded by the degree of the network, and the number of possible values of every other variable of DISJ is bounded as well. Thus, the number of distinct configurations in the computation Γ is finite.

Let $\mathcal{P}$ be the set of distinct consecutive pairs of configurations of DISJ which occur in the computation $\mathcal{G}$.

Let $\mathcal{F}$ be the set of members of $\mathcal{P}$ that occur only finitely many times in Γ. Since $\mathcal{F}$ is finite, there is some step γ of Γ after which no member of $\mathcal{F}$ occurs. Let Γ' be obtained by deleting all steps of Γ up to and including γ. In Γ', every pair of consecutive configurations is repeated infinitely many times, and thus Γ' is repetitive.

We now prove that DISJ is silent. Our proof is by contradiction – throughout the remainder of this subsection, we assume that Γ is an infinite computation of DISJ. Without loss of generality, by Lemma 4.7, Γ is repetitive.

Sets of Processes.

1. $S =$ the set of processes which never execute.
2. $A =$ the set of processes which execute.
3. EO_i, for $i = 0, 1$, is the set of processes which are in O_i forever.
4. $EB = EO_0 \cup EO_1$.
5. EO_1C_j, for $j = 0, 1$, is the set of processes in EO_1 whose color remains j forever.
6. $EO_1C = EO_1C_0 \cup EO_1C_1$.

Remark 4.8. *If $x \in EB$, then $x.level$ cannot change.*

Lemma 4.9. *If $Input(x) = 1$, then $x \in EB$.*

Proof. If $x \in S$, then $x \in EB$. If $y \in N(x) \cap S$, and $y.parent = x$. Then x cannot execute either Action A3 or A4, and hence $x \in EB$. Otherwise, suppose $x \notin EB$. Then x will eventually execute Action A3, and will never again execute Action A1, hence $x \in EO_1$, contradiction.

Define the function f on processes.

$$f(x) = \begin{cases} \infty \text{ if } x \in EO_0 \\ x.level \text{ if } x \in EO_1 \\ 1 + \min\{f(y) : y \in N(x)\} \text{ otherwise} \end{cases}$$

Lemma 4.10. *$x.level \geq f(x)$.*

Proof. By contradiction. Let $\Lambda = \min\{x.level : x.level < f(x)\}$. If $x.level = \Lambda$ and $x.level < f(x)$, then x will execute Action A1. When all such processes have executed, Λ will increase. Since Λ is bounded above by the diameter of the network, eventually $x.level \geq f(x)$.

Lemma 4.11. *All processes are in EB.*

Proof. By contradiction. Suppose $x \notin EB$. Let h be the minimum value of $x.level$, taken over all configurations of Γ. If $h = 0$, then $Input(x) = 1$, and hence $x \in EB$ by Lemma 4.9. Otherwise, there is some $y \in N(x)$ such that $y \in EO_1$ and $y.level = h - 1$. Thus, $Level(x) \leq h$ at every configuration of Γ. Since any neighbor of x whose level is less than h must be in EO_1, we have that $Level(x)$ cannot change, and hence must always be equal to h. Thus, x will remain valid and cannot execute Action A1. Hence $x \in EO_1$, contradiction.

Corollary 4.12. *For any process x, $x.parent$ never changes.*

Lemma 4.13. $EO_0 \subseteq S$.

Proof. If $x \in EO_0$, then the only action that x could execute is A1. By Remark 4.8 and Lemma 4.11, no valid process in EO_0 can become invalid, and thus $Level(x)$ cannot change. Thus, x can execute Action A1 at most once.

Lemma 4.14. *if $x \in EO_1$ and either $x.parent \in EO_1C$ or $y \in EO_1C$ for some $y \in Chldrn(x)$, then $x \in EO_1C$.*

Proof. By the guards of Actions A5 and A6, x cannot execute either of those actions more than once.

Lemma 4.15. $EO_1 \subseteq S$.

Proof. By contradiction. By Lemma 4.11 and Corollary 4.13, $A \subseteq EO_1$.

We first prove, by contradiction, that $x.done$ never changes for any $x \in A$. Let x be the process of greatest level such that $x.done$ changes. But $Done(x)$ cannot change, and so x can execute Action A2 at most once, contradiction.

Let $\mathcal{A}$ be the graph whose nodes are processes in A and whose edges are defined by the parent pointers. Each component of $\mathcal{A}$ is a tree rooted at its member of minimum level. Let $\mathcal{T}$ be one of those trees. If any member $x \in \mathcal{T}$ is a neighbor of any member of EO_0, then x can change color at most once. Thus, by Lemma 4.14 applied inductively, $\mathcal{T} \subseteq EO_1C$. If any member of $\mathcal{T}$ is linked, by a parent pointer, to any process not in A, then, also by Lemma 4.14 applied inductively, $\mathcal{T} \subseteq EO_1C$. Since no value of *done* in $\mathcal{T}$ can change, $\mathcal{T} \subseteq S$.

Now, suppose that no member of $\mathcal{T}$ is a neighbor of any member of EO_0 or is linked by a parent pointer to any process not in A. Then the root of $\mathcal{T}$ has level 0 and $x.done =$ TRUE for all $x \in \mathcal{T}$. The root of $\mathcal{T}$ cannot execute Action A6, and thus, by Lemma 4.14 applied inductively, $\mathcal{T} \subseteq EO_1C$, and thus $\mathcal{T} \subseteq S$.

4.5 Time Complexity of DISJ

Using the concept of energy, we can prove that, in the case that $Output = 0$, energy must decrease during every round, and thus must reach zero after at most $2n$ rounds. After one more round, a legitimate configuration of type 0 is achieved.

In the case that $Output = 1$, color waves actually slow down convergence. A simple flooding algorithm, which would work if we were guaranteed that $I_1 \neq \emptyset$, would take at most $Diam$ rounds, where $Diam$ is the diameter of $\mathcal{G}$. Unfortunately, the addition of color waves causes the case where $Output = 1$ to also take $O(n)$ rounds.

Time Complexity when $I_1 = \emptyset$

In this subsection, we analyze the time complexity of DISJ in the case that $I_1 = \emptyset$.

Lemma 4.16. *If $I_1 = \emptyset$ and $O_1 \neq \emptyset$, then Max_Energy decreases during the next round.*

Proof. During the first round, all the first processes of chains with energy Max_Energy will execute Action A1. The remaining chains will have smaller energy. Since $I_1 = \emptyset$, no process can execute Action A4. By Lemma 4.5, no other action can increase Max_Energy. Thus Max_Energy decreases.

Lemma 4.17. *If $I_1 = \emptyset$, then the configuration will be legitimate of type 0 within $2n + 1$ rounds of an arbitrary initialization.*

Proof. In the initial configuration, $Max\_Energy \leq 2n$, By Lemma 4.16, within $2n$ rounds, $Max\_Energy = 0$, and thus $O_1 = \emptyset$. Within one more round, every process which is not valid will execute Action A1, and the configuration will then be legitimate of type 0.

Lemma 4.18. *If $Output = 1$, then* DISJ *converges within $O(n)$ rounds.*

We only sketch the proof. The initial value of Max_Energy cannot exceed $2n$. We can show that within $O(n)$ rounds, $Max\_Energy = O(Diam)$, after which DISJ converges within $O(Diam)$ additional rounds.

From the above lemmas, we conclude our main result, below.

Theorem 4.19 DISJ *is self-stabilizing and silent, takes $O(n)$ rounds, and works under the unfair daemon.*

Proof. Let Γ be any computation of DISJ. By Lemmas 4.11 and 4.15, and Corollary 4.13, Γ is finite. By Lemma 4.2, the last configuration of Γ is legitimate, and hence DISJ is self-stabilizing and silent. By Lemmas 4.18 and 4.17, DISJ converges in $O(n)$ rounds from an arbitrary initial configuration.

References

1. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of Computing Machinery 17, 643–644 (1974)
2. Dolev, S.: Self-Stabilization. The MIT Press (2000)
3. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing leader election in optimal space under an arbitrary scheduler. Theoretical Computer Science 412(40), 5541–5561 (2011)

Uniform Consensus with Homonyms and Omission Failures*

Carole Delporte-Gallet, Hugues Fauconnier, and Hung Tran-The

LIAFA-Université Paris-Diderot, France
{cd,hf,hung}@liafa.univ-paris-diderot.fr

Abstract. In synchronous message passing models in which some processes may be homonyms, i.e. may share the same id, we consider the consensus problem. Many results have already been proved concerning Byzantine failures in models with homonyms [10], we complete here the picture with crash and omission failures.

Let n be the number of processes, t the number of processes that may be faulty ($t < n$) and l ($1 \leq l \leq n$) the number of identifiers. We prove that for crash failures and send-omission failures, uniform consensus is solvable even if $l = 1$, that is with fully anonymous processes for any number of faulty processes.

Concerning omission failures, when the processes are *numerate*, i.e. are able to count the number of copies of identical messages they received in each round, uniform consensus is solvable even for fully anonymous processes for $n > 2t$. If processes are not numerate, uniform consensus is solvable if and only if $l > 2t$.

All the proposed protocols are optimal both in the number of communication steps needed, and in the number of processes that can be faulty.

All these results show, (1) that identifiers are not useful for crash and send-omission failures or when processes are numerate, (2) for general omission or for Byzantine failures the number of different ids becomes significant.

1 Introduction

Generally distributed algorithms assume either that all processes have distinct identifiers and more rarely that they are anonymous. These two models are two extremes of the same model called homonyms in [10]: n processes use l different identifiers. Hence the case $l = 1$ corresponds to a fully anonymous model and the case $l = n$ to model in which all processes have different identifiers.

Anonymous models have a very restricted computational power and many impossibility results have been proved with anonymous models (e.g. leader election, Byzantine consensus[2,3,5,8,18,22] ...) and it is interesting to determine how identifiers are needed and useful. Beyond this theoretical interest, models with homonyms have also a practical interest. In large systems, it is not so easy

* Supported by ANR VERSO SHAMAN.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 161–175, 2013.

to ensure that all processes have unique and unforgeable identifiers. Moreover, in some cases, users of a system may wish to preserve their privacy and for this appear in the system not as an individual but rather as member of some group [9].

As agreement protocols are major tools for fault-tolerance (e.g. state machine approach [24]), to evaluate the power of identifiers we consider the classical consensus problem. In [10,12,13] many results have been proved concerning the Byzantine consensus problem in models with homonyms. In this paper we complete this picture considering less severe process failures.

With homonyms, l ($1 \leq l \leq n$) distinct identifiers are assigned to each process. Several processes may be homonyms and share the same identifier. When a process p receives a message from a process q with identifier i, then p knows that the message has been sent by some process with identifier i but does not know wether it was sent by q or another process having the same identifier i.

We restrict ourselves to synchronous message passing models. In the models we consider, computation proceeds in rounds. In each round each process sends messages to all other processes and then receives all messages that were sent to it during the round.

When processes with the same identifier send the same message, it can be assumed or not that the processes receiving this message know how many times this message was sent. Then, as in [10], we consider two variants of the model, in the first one processes are *innumerate*: each process receives only a *set* of messages without multiplicity, whereas in the second one processes are *numerate*: each process receives a *multiset* of messages.

Only benign process failures are considered here. More precisely we study: *process crashes* (a process stops its code and if the crash occurs in a round some messages from this process may be not received by some processes), *send omission* (a process may crash or omit to send messages to some processes), *receive omission* (a process may crash or omit to receive some messages) and *general omission* (a process may crash or commit send or receive omissions).

Results: Concerning innumerate processes we prove that uniform consensus[1] is solvable even with $l = 1$ (anonymous processes) for crash failures and send-omission failures. For general omission and innumerate processes, uniform consensus is solvable if and only if $l > 2t$. Hence there is no algorithm for fully anonymous processes, but the number of identifier needed depends only on the number of tolerated faulty processes. Moreover the solution we propose is in $t+1$ rounds for any t such that $t < n$, and is then optimal [1,16,21] concerning both the communication complexity in number of rounds and the resiliency. Hence, working with anonymous processes gets no penalty with crash or send-omission failures and in this sense identifiers are not useful.

Numerate processes are more powerful, even if $l = 1$ (anonymous processes) uniform consensus is solvable for general omission failures if and only if $n >$

[1] In the uniform version of the consensus, if a faulty process decides, the decided value has to satisfy the same properties as for correct processes.

Table 1. Necessary and sufficient conditions on the number identifiers for solving consensus in a system of n processes with at most t faulty processes

	Send-omission	General-omision	Byzantine	Restricted Byzantine
Innumerate processes	$l \geq 1$	$l > 2t$	$l > 3t$	$l > 3t$
Numerate processes	$l \geq 1$	$l \geq 1$	$l > 3t$	$n > 3t \wedge l > t$

$2t$ [23], that is with the same bound as for processes having different identifiers. Hence the ability of counting the number of messages may be a way to avoid identifiers.

Adding results from [10], Table 1 summarizes the necessary and sufficient conditions on the number of identifiers to solve the consensus. In the restricted Byzantine case, a Byzantine process is only able to send one message to each process in each round.

From this table, we may observe that (1) numerate processes enable to avoid the use of identifiers for omission failures, (2) for innumerate anonymous processes consensus is solvable only for send-omisison failures, and (3) identifiers are really needed for Byzantine failures.

2 Model and Definitions

We consider a synchronous message passing system of $n \geq 2$ processes. Each process has an identifier from the set $\mathcal{L} = \{1, \cdots l\}$. l is the number of different identifiers. We assume that $n \geq l$ and each identifier is assigned to at least one process. If $n > l$ some processes share the same identifiers and are homonyms, when $l = 1$ all processes have the same identifier and the processes are anonymous, when $l = n$ all processes have different identifiers. Processes with the same identifier execute the same code. For convenience, we sometimes refer to individual processes by names, but these names cannot be used by processes themselves in the algorithms. The total number of processes, n, is known by the processes.

Processes communicate by messages. Processes may only send messages to all processes with the same identifier. When a process with identifier id sends a message to the processes of identifier id', all processes with identifier id' receive m. When a process receives a message m it knows the identifier i of the sender of the message but it does not know which process with identifier i sent this message.

We consider a synchronous round based model for which in each round, each process sends the same set of messages to all other processes and then receives all messages that were sent to it during that round.

Process failures. We assume that communication is reliable in the sense that every message sent by correct processes is received by all correct processes in the round it was sent. We restrict ourselves to the following benign failures of processes:

- *Crash failure:* A faulty process stops its execution prematurely. After it has crashed, a process does nothing.
- *Send Omission failure:* A faulty process crashes or omits sending messages it was supposed to send to other processes.
- *General Omission failure:* A faulty process crashes or omits sending and/or receiving messages it was supposed to send to/from other processes.

A send (receive) omission failure actually models a failure of the output (input) buffer of a process. A buffer overflow is a typical example of such a failure. Note that when a process crashes in a round or commits a send omission, some processes may receive the message of this process for this round and some other not.

In the following t denotes an upper bound on the number of faulty processes.

Innumerate and numerate processes. As processes are anonymous, among the messages received in the round, a process may only distinguish between messages having different values, but it may be assumed or not that a process is able count the number of identical messages received in the round. [10]

More precisely, if more than one process sends the same message m in a round, the model ensures either that each process only knows that m has been sent or that each process knows not only that m has been sent but also how many time m has been sent in the round. In the first case, the processes are said *innumerate* whereas in the second case they are said *numerate*. When a process is innumerate, it cannot count the number of copies of identical messages it receives in the round and in this case each process p receives a *set* of messages. When a process is numerate, the messages it receives in round r is a *multiset* of messages.

The uniform consensus problem. The goal of a consensus algorithm is, for a set of processes proposing values, to decide on exactly one of these values. We consider the uniform consensus problem as defined in [21] by the following three properties:

1. *Termination*: Every correct process eventually decides.
2. *Uniform Validity*: If a (correct or not) process decides v, then v was proposed by some process.
3. *Uniform Agreement*: No two (correct or not) processes decide different values.

3 Consensus with Send-Omission Failures

In this section we prove that uniform consensus is solvable with send-omission failures for all t less than n even if processes are innumerate.

The crash-tolerant uniform consensus protocols in models in which processes have distinct identifiers described in [4,20,19] are based on a "flood set" strategy. Each process p maintains a local variable that contains its current estimates of

the decision value. Initially, the local variable is set to the input proposed by p. Then, during each round, each non-crashed process first broadcasts its current estimate, and then updates it to the smallest values among the estimates it has received. After $t+1$ rounds, as there is at least one round without any crash, all processes will have the same estimate. These algorithms do not use identifiers for processes and solve directly the uniform consensus problem when all processes are anonymous ($l = 1$) in presence of any number of crashes.

With omission failures, faulty processes may commit omissions in any round, and it is possible that there is failures in all rounds. To circumvent this difficulty, in [21,23,25], each process keeps track (explicitly or implicitly) of the set of processes it considers to be correct. A process does not accept messages from processes outside of this set. In [17], the current estimate of each process is updated to the current estimate of the leader selected in each round. All these ways use the fact that each process identifies the sender.

We present here a protocol that solves uniform consensus despite up to $t < n$ processes that commit send-omission failures even if all processes are fully anonymous. The underlying principles of this algorithm are inspired by [23,25]. Roughly speaking, the algorithm ensures that if some process changes its estimate in round r, then another process has changed its estimate in the previous round. After the first round, when a process changes its estimate to some value, this value may only come from a faulty process, if some process changes its estimate in round k, then at least $k-1$ processes are faulty.

The protocol for a process p is described in Figure 1. Each process p maintains local variables *new* and *old*: *old* is the estimate of the previous round and *new* the current estimate of the round. Initially, *new* is set to v, the initial value of p.

Note that after the first round, *new* is different from *old* if and only if the process has changed its estimate. Moreover a process changes its estimate only for a smaller value, then if $new < old$ that means that the process has changed its estimate. During each round r, each process first broadcasts its current value of variables *new* and *old* and then updates them as follows: the variable *old* is set to the value of variable *new* at round $r-1$. Variable *new* may change only if the process receives some pairs (v, o) with $v < o$. From the previous remark, a process changes its variable *new* in round r only if it sees that a process has changed its value of variable *new* in the previous round. If *new* is modified, it is updated by the min of the previous value of *new* and the min of all v received from processes having changed their estimate in the previous round.

Finally, at round $t+1$, each process decides on the maximum of values received in round $t+1$.

To prove the correctness of the algorithm, we use the following notation: we say that a process "changes its estimate to some value v_0 in round r", if at the end of this round, $new = v_0 \wedge v_0 < old$. We say that a process "keeps its estimate in round r", if at the end of this round, $new = old$.

Let p_0 be the process having the minimum input value v_{min}. Let $new_p(r)$ be the value of variable *new* of the process p at the end of round r, $old_p(r)$ be the

Code for process p

Variable:

1 $input = \{v\}$ /* v is the value proposed value */
2 $new = input$
3 $old = input$

Main code:

4 ROUND 1
5 **send** new to all processes
6 $old = new$
7 $new = Min\{v|p$ has received v in this round$\}$

8 ROUND r from 2 to t
9 **send** (new, old) to all processes
10 $old = new$
11 **let** $G_p[r] = \{v|$ p has received (v, o) in this round and $v < o\}$
12 **if** $G_p[r] \neq \emptyset$ and $Min\{v|v \in G_p[r]\} < new$ **then** $new = Min\{v|v \in G_p[r]\}$

13 ROUND $t + 1$
14 **send** new to all processes
15 **decide** $Max\{v|p$ has received v in this round$\}$

Fig. 1. Anonymous Consensus Protocol with at most t send omission processes

value of variable *old* of process p at the end of round r, $V[r]$ be the set of values of variable *new* of all processes (not crashed at the end of round r) at the end of the round r: $V[r] = \{new_p(r)|p$ is any process$\}$, and v_{max} be the largest value of set $V[t]$.

We begin with two simple facts:

Fact 1. *$old_p(r + 1) = new_p(r)$ for every round $1 \leq r < t$ and every process p not crashed at the end of round $r + 1$.*

Fact 2. *$new_p(r) \leq old_p(r)$ for every round $1 \leq r \leq t$ and every process p not crashed at the end of round r.*

Lemma 1. *$V[r + 1] \subseteq V[r]$ for every round $1 \leq r < t$.*

Proof. Consider variable $new_p(r + 1)$ for some process p not crashed at the end of round $r + 1$. There are two cases:

- p changes its estimate in round $r + 1$ to some value v_0 at Line 12. Then $new_p(r + 1) = v_0$ where v_0 is the value of variable *new* of some process q that sent the pair (v_0, old) to p with $v_0 = new_q(r)$.
- p keeps its estimate in this round. Hence, $new_p(r + 1) = old_p(r + 1) = new_p(r)$.

Thus, $V[r + 1] \subseteq V[r]$ for every round $1 \leq r < t$.

We directly get:

Lemma 2. *If p_0 is correct then $new_p(r) = v_{min}$ for every round $1 \leq r \leq t$ and every process p not crashed at the end of round r.*

Lemma 3. *If a correct process has input value v_0 or changes its estimate to some value v_0 in round $1 \leq r < t$ then $new_q(t) \leq v_0$ for every process q not crashed at the end of round t.*

Proof. If some correct process p has input v_0 then at the end of round 1, the variable *new* of all processes is less than or equal to v_0 (Line 7). By Lemma 1, we have $V[t] \subseteq V[1]$. Thus, $new_q(t) \leq v_0$ for every process q.

If p changes its estimate *new* to v_0 in round $1 \leq r < t$. Then $v_0 = new_p(r) < old_p(r)$. In the round $r+1 \leq t$, the correct process p sends the pair $(v_0, old_p(r))$ to all processes. At the end of the round $r+1$, all processes receive $(v_0, old_p(r))$. We consider any process q, after Line 12, $new'_q r + 1) \leq v_0$. By Lemma 1, we have $V[t] \subseteq V[r+1]$. Thus, $new_q(t) \leq v_0$ for every process q.

Lemma 4. *If $t > 1$ and $r \geq 2$, if some p changes its estimate to v_0 in round r then, there is a set of processes $\{q_1, \cdots, q_{r-1}\}$ such that for all i, $1 \leq i \leq r-1$, q_i changes its estimate to v_0 in the round i and $new_{q_i}(r-1) \leq v_0$.*

Proof. Since p changes its estimate to v_0 in round r, we must have $new_p(r) = Min\{v | v \in G_p[r]\}$ and there is at least one process q_{r-1} that sent (new, old) to p, with $new = v_0 < old$. Hence, at the end of round $r-1$, the process q_{r-1} must have $new_{q_r}(r-1) < old_{q_r}(r-1)$. That means that q_{r-1} has changed its estimate in round $r-1$.

By induction, we have a sequence of processes $(q_1, \cdots, q_{r-1})$ such that q_i changes its estimate in round i, $1 \leq i \leq r-1$.

Furthermore, if a process changes its estimate to v_0 in some round r_0 then after this round, its value of variable *new* is less than or equal to v_0 and no process can change its estimate to v_0 twice. Thus, all processes q_i are distinct and $new_{q_i}(r-1) \leq v_0$ for all i such that $1 \leq i \leq r-1$, proving the Lemma.

Lemma 5. *Either* (a) $new_q(t) = v_{max}$ *for every correct process q, or* (b) $V[t] = \{v_{min}, v_{max}\}$, $new_s(t) = v_{min}$ *for every faulty process s and some correct process changes its estimate to v_{min} in round t.*

Proof. If $t = 1$ then consider two cases:

- if p_0 is correct and then by Lemma 2, $new_q(t) = v_{min}$ for all correct processes q.
- if p_0 is faulty. At the end of round 1, either all correct processes have the same value of *estimate* or some correct process changes its estimate to v_{min}, proving the Lemma.

faux

If $t > 1$, we consider the set of values of variable *new* of correct processes at the end of round t. Since $n > t$, this set is not empty.

Assume that (a) is not satisfied then for some correct process p $new_p(t) < v_{max}$. Let v_0 be $new_p(t)$ ($v_0 < v_{max}$). Consider the following two cases:

- v_0 is the input value of p or p changes its estimate to v_0 in a round $1 \leq r < t$ then by Lemma 3, $new_q(t) \leq v_0$ for every process q, contradicting the hypothesis that $v_0 < v_{max}$.
- p changes its estimate to v_0 in round t. By Lemma 4, we have a set of processes $\{q_1, \cdots, q_{t-1}\}$ such that q_i changes its value of variable new to v_0 in the round i, $1 \leq i \leq t-1$. Moreover, no process among them is correct because if some process q_i is correct then the value of variable new at the end of the round t of every process is less than or equal to v_0, hence $v_{max} \leq v_0$, contradicting the hypothesis that $v_0 < v_{max}$. Moreover, p_0 is faulty because if p_0 were correct then by Lemma 2, $new_p(1) = v_{min}$ and p could not change its estimate in round t, contradicting the hypothesis. Therefore, we have a set of t faulty processes $\{q_1, ..., q_{t-1}\} \cup \{p_0\}$.

  Since all processes q_i that have ever changed their estimate to v_0 in some round $r \geq 1$, v_0 cannot be the input value of one of these processes. On the other hand, by Lemma 3, if v_0 is the input value of a correct process then $new_q(t) \leq v_0$ for every process q, by definition of v_{max}, $v_{max} = v_0$, contradicting the hypothesis that $v_0 < v_{max}$. Hence, v_0 may only be the input value of faulty process p_0. That means that $new_p(t) = v_0 = v_{min}$. Moreover, all faulty processes q_i that changed their estimate to v_{min} in some round $r \geq 1$ do not change their estimate more because v_{min} is minimum value, hence $new_s(t) = v_{min}$ for every faulty process $s \in \{q_1, ..., q_{t-1}\} \cup \{p_0\}$.

Proposition 1. ***Uniform agreement****: No two processes decide different values.*

Proof. By Lemma 5, we have either

- If $new_q(t) = v_{max}$ for every correct process q then, every process p not crashed at round $t+1$ receives v_{max} in round $t+1$ and decides v_{max} at Line 15.
- $V[t] = \{v_{min}, v_{max}\}$, $new_s(t) = v_{min}$ for every faulty process s. Hence, v_{max} must come from some correct process. At the beginning of round $t+1$, this process sends v_{max} to all and every process not crashed at round $t+1$ receives the value v_{max} and decides v_{max}.

The Termination and Uniform Validity are trivially satisfied, then with the previous proposition we deduce:

Theorem 1. *Uniform consensus is solvable in $t+1$ rounds with send-omission failures of any number of processes even if all processes are anonymous.*

This algorithm is optimal concerning the number of rounds $(t+1)$ and the number of tolerated faulty processes $(t < n)$.

4 Consensus with General-Omission Failures

In this section we give an algorithm solving uniform consensus with general-omission failures if processes are numerate (even if they are anonymous). In

a second subsection we prove there is no solution for uniform consensus with general-omission failures when processes are anonymous and innumerate. More precisely we prove that at least $l > 2t$ identifiers are needed. Recall that uniform consensus is solvable with processes having unique identifiers for general-omission failures only if there is a majority of correct processes, then we always assume in this section that $n > 2t$.

4.1 Numerate Processes

In this subsection, processes are anonymous and numerate.

An algorithm in Figure 2 solves the consensus in general omission model base on the same principles as Figure 1. Round $t + 1$ has to be adapted to general-omission failures for which it is not ensured that all correct processes have the same estimate in round $t + 1$.

Code for process p

Variable:

1 $input = \{v\}$ /* v is the proposed value */
2 $new = input$
3 $old = input$

Main code:

4 ROUND 1
5 **send** new to all processes
6 $old = new$
7 $new = Min\{v|\ v$ in the set of messages received $\}$

8 ROUND r from 2 to t
9 **send** (new, old) to all processes
10 $old = new$
11 **let** $G_p[r] = \{v|\ p$ has received (v, v_1) in this round and $v < v_1\}$
12 **if** $G_p[r] \neq \emptyset$ and $Min\{v|v \in G_p[r]\} < new$ **then** $new = Min\{v|v \in G_p[r]\}$

13 ROUND $t + 1$
14 **send** (new, old) to all processes
15 **if** for some v received $n - t$ pairs $(v, *)$ in this round
16 **then** decides v
17 **else if** received at least $n - t$ pairs
18 and one of them is (x, y) such that $x < y$
19 **then**
20 **let** $G_p = \{v|\ p$ has received $(v, *)$ in this round $\}$
21 **if** $\exists(x, y)$ such that $x < y$ and (x, y) received in this round
and $x = Min\{v|v \in G_p\}$
22 **then decide** x

Fig. 2. Anonymous Consensus Protocol with at most t general-omission processes

We now present the steps of the proof that the protocol of Figure 2 satisfies the specification of uniform consensus. We use the same notations $p_0, v_{min}, V, new_p(r)$ and $old_p(r)$ as in the proof of Theorem 1. Let $C[r]$ be the set of values of variable *new* of all correct processes at the end of the round r: $C[r] = \{new_p(r) | p$ is a correct process$\}$, and c_{max} be the largest value of set $C[t]$.

Although here processes may commit receive omission the following lemmata may be proved in a very similar way as in the proof of Theorem 1.

Lemma 6. *$V[r+1] \subseteq V[r]$ for every round $1 \leq r < t$.*

Lemma 7. *If p_0 is correct then $new_p(r) = v_{min}$ for every correct process p and every round $1 \leq r \leq t$.*

Lemma 8. *If a correct process changes its estimate to some value v_0 in round $1 \leq r < t$ or has input v_0 then $new_q(t) \leq v_0$ for every correct process q.*

Lemma 9. *If $t > 1$ and $r \geq 2$, if some process p changes its estimate to v_0 in round r then, there is a set of processes $\{q_1, ..., q_{r-1}\}$ such that for all i, $1 \leq i \leq r-1$, q_i changes its estimate to v_0 in the round i.*

Lemma 10. *Either* (a) *$new_q(t) = c_{max}$ for every correct process q, or* (b) *$V[t] = \{v_{min}, c_{max}\}$, $new_s(t) = v_{min}$ for every faulty process s, some correct process changes its estimate to v_{min} in round t, and t processes are faulty.*

Proof. If $t = 1$ then consider two cases:

- if p_0 is correct then by Lemma 7, $new_q(1) = v_{min}$ for all correct processes.
- if p_0 is faulty. At the end of round 1, either all correct processes have the same value of *estimate* or some correct process changes its estimate to v_{min}, proving the Lemma.

If $t > 1$, we consider the set of values of variable *new* of correct processes at the end of round t. Since $n > 2t$, this set is not empty.

Assume that (a) is not satisfied then for some correct process p $new_p(t) < v_{max}$. Let v_0 be $new_p(t)$ ($v_0 < c_{max}$). Consider the following two cases:

- v_0 is the input value of p or p changes its estimate to v_0 in a round $1 \leq r < t$ then by Lemma 8, $new_q(t) \leq v_0$ for every correct process q. By definition of c_{max}, we must have $c_{max} = v_0$, contradicting the hypothesis that $v_0 < c_{max}$.
- p changes its estimate to v_0 in round t. By Lemma 9, we have a set of processes $\{q_1, \cdots, q_{t-1}\}$ such that q_i changes its value of variable *new* to v_0 in the round i, $1 \leq i \leq t-1$. Moreover, no process among them is correct because if any process q_i is correct then the value of variable *new* at the end of the round t of every process is less than or equal to v_0, hence $c_{max} \leq v_0$, contradicting the hypothesis that $v_0 < c_{max}$. On the other hand, p_0 is faulty because if p_0 were correct then by Lemma 7, $new_p(1) = v_{min}$ and p could not change its estimate in round t, contradicting the hypothesis. Therefore, we have a set of t faulty processes $\{q_1, ..., q_{t-1}\} \cup \{p_0\}$.

As all q_i changed their estimate to v_0 in some round $r \geq 1$, v_0 is not any input value of these processes. On the other hand, by Lemma 8, if v_0 is input value of a correct process then $new_q(t) \leq v_0$ for every process q. By definition of c_{max}, we must have $c_{max} = v_0$, contradicting the hypothesis that $v_0 < c_{max}$. Then, v_0 may only be the input value of faulty process p_0. That means that $new_p(t) = v_0 = v_{min}$. Moreover, all faulty processes q_i that have changed their estimate to v_{min} in some round $r \geq 1$ do not change their estimate after because v_{min} is minimum value. Hence $new_s(t) = v_{min}$ for every faulty process $s \in \{q_1, ..., q_{t-1}\} \cup \{p_0\}$.

Lemma 11. *Suppose that $new_q(t) = c_{max}$ for every correct process q. Thus, if a faulty process changes its estimate to v_0 in round t then $v_0 \geq c_{max}$.*

Proof. If p_0 is correct then at the end of round 1, we have $new_q(1) = v_{min}$ for every correct process q. q never changes its estimate because v_{min} is the minimum value. Thus, $c_{max} = v_{min}$. If a faulty process changes its estimate to v_0 in round t then obviously, $v_0 \geq v_{min}$.

Now, consider the case where p_0 is a faulty process. Suppose that a faulty process q changes its estimate to v_0 in round t. By Lemma 9, we have a set of processes $\{q_1, \cdots, q_{t-1}\}$ such that q_i changes its value of variable *new* to v_0 in the round i, $1 \leq i \leq t-1$. Moreover, the faulty process p_0 that never changes its estimate must be different from all processes in the set $\{q_1, \cdots, q_{t-1}\}$. If all these processes were faulty with $p0$ and q we would have $t+1$ faulty processes, contradicting the hypothesis that there are at most t faulty processes. Therefore, at least one of processes of set $\{q_1, \cdots, q_{t-1}\}$ is correct, say p. p changes its estimate to v_0 in round $1 \leq r < t$: $new_p(r) = v_0$. But we have always $new_p(t) \leq new_p(r)$ for every $1 \leq r < t$. Thus, $v_0 = new_p(r) \geq new_p(t) = c_{max}$.

Lemma 12. *If two processes decide at Line 16 then they decide the same value.*

Proof. Suppose that process p decides v_0 at Line 16 and process q decides v_1 at Line 16. Thus, p receives at least $n-t$ pairs $(v_0, *)$ and q receives at least $n-t$ pairs $(v_1, *)$ in round $t+1$. That means that at least $n-t$ processes sent $(v_0, *)$ and at least $n-t$ processes sent $(v_1, *)$. Since $(n-t)+(n-t) > n$, $v_0 = v_1$.

Theorem 2. *If processes are numerate, uniform consensus is solvable in $t+1$ rounds if $n > 2t$ even if all processes are anonymous.*

Proof. Termination: By Lemma 10, at the end of round t, either:

- the value of variable *new* of every correct process is the same, then all correct processes decide at Line 16.
- at least one correct process p changes its estimate to v_{min} and the value of variable *new* of every faulty process is v_{min}. Thus, every correct process q receives at least $n-t$ pairs and one of them is (v_{min}, y) such that $v_{min} < y$ from p. q decides v_{min} at Line 22.

Uniform Validity: if a process decides it decides some value in set $V[t]$. By Lemma 6, we have $V[t] \subseteq V[1]$. Moreover, $V[1]$ is a subset of the inputs proposed by

processes. Therefore, if a process p decides v, then v is input value of some process.

Uniform agreement: By Lemma 10, at the end of round t either:

- the value of variable *new* of every correct process is the same c_{max} then all correct processes decide this value at Line 16. Suppose that a faulty process q want to decide. If it decides at Line 16 then by Lemma 12 it must decide the same c_{max}. If it decides at Line 22 it has received at least $n - t$ pairs and one of them is (x, y) such that $x < y$ and $x = Min\{v|v \in G_q\}$. As q receives at least $n - t$ pairs, at least one pair comes from a correct process. then $x = Min\{v|v \in G_q\} \leq c_{max}$. If this pair (x, y) comes from a correct process then $x = c_{max}$. If the pair come from a faulty process. By Lemma 11, we have $x \geq c_{max}$. Thus, in all cases, we have always $x = c_{max}$. That means that q decides the same value as correct processes.
- $V[t] = \{v_{min}, c_{max}\}$, $new_s(t) = v_{min}$ for every faulty process s, some correct process p changes its estimate to v_{min} in round t, and t processes are faulty. As p is correct, all correct processes receive its value. Moreover a correct process receives at most $n - t - 1$ value different from v_{min} and cannot decide at Line 16 then it decides v_{min} at Line 22. If a faulty process q decide, by hypothesis it has $new_s(t) = v_{min}$.

4.2 Innumerate Processes

Proposition 2. *Uniform consensus is not solvable with innumerate processes if $l \leq 2t$ with general-omission failures.*

Proof. The proof is based on a classical partitioning argument. By contradiction, assume that there is a protocol that solves the uniform binary consensus problem with $l \leq 2t$.

Let a partition of the set of identifiers $\mathcal{L}$ into two sets $I = \{1, \ldots, l/2\}$ and $J = \{l/2 + 1, \ldots l\}$, such that $|I| \leq t$ and $|J| \leq t$. Consider the two following repartitions of identifiers. In repartition R, all identifiers in I are identifier of only one process, identifiers in $J - \{l\}$ are identifiers of only one process too and identifier l is the identifier of the remaining $n - l + 1$ processes. $R(I)$ and $R(J)$ denote respectively the set of processes with identifiers in I and the set of processes with identifiers in J for repartition R Repartition S is identical to R except that only one process has identifier l and the other processes having identifier l for R have now identifier 1. $S(I)$ and $S(J)$ denote respectively the set of processes with identifiers in I and the set of processes with identifiers in J for repartition S. Note that $R(I)$ and $S(J)$ contain at most t processes. Note also that as processes are innumerate if all processes with the same identifier send the same messages in R and S and have the same initial state, execution in R or S are indistinguishable for any process.

Consider the following executions:

Execution α. The repartition is R, all processes have 0 as initial value. Processes in $R(I)$ are crashed from the beginning. By validity the decision value is 0.

Execution α'. The repartition is R, the initial values are 0 for processes in $R(J)$ and 1 for processes in $R(I)$. Processes in $R(I)$ commit send and receive omission failures: processes in $R(J)$ do not receive any message from processes in $R(I)$ and processes in $R(I)$ do not receive any message from processes in $R(J)$. α' is indistinguishable from α for processes in $R(J)$ and the decision value is 0.

Execution β. The repartition is S, all processes have 1 as initial value. Processes in $S(J)$ are crashed from the beginning. By validity the decision value is 1.

Execution β'. The repartition is S, processes in $S(I)$ have 1 as initial value and processes in $S(J)$ have 0 as initial value. Processes in $S(J)$ commit send and receive omission and processes in $S(I)$ don't receive any message from $S(J)$ and processes in $S(J)$ don't receive any message from $S(I)$. β' is indistinguishable from β for processes in $S(I)$ and the decision value is 1.

Now consider any process p with identifier in I both for R and S. As processes are innumerate p receives in β' and α' exactly the same messages from identifiers in I, and receives no messages from identifiers in J, both execution are then indistinguishable for p. In β' it decides 1 then in α' it decides 1 too. But the decision value for α' is 0.

In the other hand, the consensus is solvable when $l > 2t$. The protocol is similar to the one presented in Figure 2 only Lines 13 - 22 (round $t + 1$) are replaced by:

```
15 ROUND t + 1
16   send (new, old) to all processes
17     if for some v received (v, ∗) from at least l − t identifiers in this round
18     then  decides v
19       else if received at least messages from l − t identifiers
20                 and one of them is (x, y) such that x < y
21                 then
22                   let G_p = {v| p has received (v, ∗) in this round }
23                   if ∃(x, y) such that x < y and (x, y) received in this round
                           and x = Min{v|v ∈ G_p}
24                   then decide x
```

Then we get:

Theorem 3. *Uniform consensus is solvable with innumerate processes with general-omission failures if and only* $l > 2t$.

5 Conclusion and Perspectives

One natural extension of this work is to consider partially synchronous models [14,15]. Some results for consensus with anonymous processes for a specific partially synchronous model are given in [11].

Concerning numerate processes in models like [15] in which the communication between all correct processes is eventually synchronous we conjecture that

consensus is solvable with a majority of correct processes. Concerning only crash failures, it may be noticed that sending regularly "alive" messages an anonymous failure detector [6,7] giving eventually the exact number of correct processes may be implemented, and then with this failure detector and a majority of correct processes consensus may be implemented.

When processes are innumerate in the Byzantine failures case, it has been proved [10] by a partitioning argument the lower bound of $l > (n + 3t)/2$ for consensus. This proof may be adapted to crash and omission failures giving a lower bound of $l > (n + 2t)/2$. This bound indicates that in partially synchronous models, to solve the consensus the homonymy of processes must be very restricted.

One more technical issue is the ability to have early stopping algorithms. We guess that with innumerate processes early-stopping algorithms are not possible or with very poor bounds.

In some way, the results of this paper shows that two mechanisms help to solve the consensus: identifiers of processes and the ability of processes to count identical messages they receive. Anonymity of processes may be balanced by the ability to count the number of identical messages received.

References

1. Aguilera, M.K., Toueg, S.: A simple bivalency proof that t-resilient consensus requires $t + 1$ rounds. Inf. Process. Lett. 71(3-4), 155–158 (1999)
2. Angluin, D.: Local and global properties in networks of processors (extended abstract). In: STOC, pp. 82–93. ACM (1980)
3. Attiya, H., Gorbach, A., Moran, S.: Computing in totally anonymous asynchronous shared memory systems. Information and Computation 173(2), 162–183 (2002)
4. Attiya, H., Welch, J.: Distributed Computing: fundamentals, simulations and advanced topics, 2nd edn. Wiley (2004)
5. 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)
6. Bonnet, F., Raynal, M.: Anonymous Asynchronous Systems: The Case of Failure Detectors. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 206–220. Springer, Heidelberg (2010)
7. Bonnet, F., Raynal, M.: The price of anonymity: Optimal consensus despite asynchrony, crash, and anonymity. TAAS 6(4), 23 (2011)
8. Buhrman, H., Panconesi, A., Silvestri, R., Vitányi, P.M.B.: On the importance of having an identity or, is consensus really universal? Distributed Computing 18(3), 167–176 (2006)
9. Chaum, D., van Heyst, E.: Group Signatures. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 257–265. Springer, Heidelberg (1991)
10. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kermarrec, A.-M., Ruppert, E., Tran-The, H.: Byzantine agreement with homonyms. In: PODC, pp. 21–30. ACM (2011)
11. Delporte-Gallet, C., Fauconnier, H., Tielmann, A.: Fault-tolerant consensus in unknown and anonymous networks. In: ICDCS, pp. 368–375. IEEE Computer Society (2009)

12. Delporte-Gallet, C., Fauconnier, H., Tran-The, H.: Byzantine Agreement with Homonyms in Synchronous Systems. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 76–90. Springer, Heidelberg (2012)
13. Delporte-Gallet, C., Fauconnier, H., Tran-The, H.: Homonyms with Forgeable Identifiers. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 171–182. Springer, Heidelberg (2012)
14. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM 34(1), 77–97 (1987)
15. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM 35(2), 288–323 (1988)
16. Dwork, C., Moses, Y.: Knowledge and common knowledge in a Byzantine environment: Crash failures. Information and Computation 88(2), 156–186 (1990)
17. Guerraoui, R., Kouznetsov, P., Pochon, B.: A note on set agreement with omission failures. Electr. Notes Theor. Comput. Sci. 81, 48–58 (2003)
18. Guerraoui, R., Ruppert, E.: Anonymous and fault-tolerant shared-memory computing. Distributed Computing 20(3), 165–177 (2007)
19. Kumar, G.V.: Elements of Distributed Computing. Wiley-Interscience (2002)
20. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
21. Neiger, G., Toueg, S.: Automatically increasing the fault-tolerance of distributed algorithms. Journal of Algorithms 11(3), 374–419 (1990)
22. Okun, M., Barak, A.: Efficient algorithms for anonymous Byzantine agreement. Theory of Computing Systems 42(2), 222–238 (2008)
23. Perry, K.J., Toueg, S.: Distributed agreement in the presence of processor and communication faults. IEEE Trans. Software Eng. 12(3), 477–482 (1986)
24. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
25. Srikanth, T.K., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing 2(2), 80–94 (1987)

Democratic Elections in Faulty Distributed Systems

Himanshu Chauhan and Vijay K. Garg*

Parallel and Distributed Systems Lab,
Department of Electrical and Computer Engineering,
The University of Texas at Austin
himanshu@utexas.edu, garg@ece.utexas.edu

Abstract. In this paper, we show that for elections in distributed systems the conversion from non-binary choices to binary choices does not always provide optimal results when the preferences of nodes are not identical. With this observation, we study the problem of conducting democratic elections in distributed systems in the form of social choice and social welfare functions with three or more candidates. We present some impossibility and possibility results for distributed democratic elections in presence of Byzantine behavior. We also discuss some existing election schemes, and present a new approach that attempts to mitigate the effects of Byzantine votes. We analyze the performance of these schemes through simulations to compare their efficacy in producing the most desirable social welfare rankings.

1 Introduction

Many problems in distributed systems require *election* for processes to carry out globally consistent actions. For example, the problem of binary consensus can be viewed as an election between two possible choices. The value decided by the protocol can be considered the winner elected by the system. The *leader election* problem requires that all the processes in the system agree on a leader. The agreed upon leader may then perform certain privileged tasks on assuming this role. Most protocols for leader election select processes with the lowest or the highest identifier value as the leader. It can be argued that such a selection on the basis of identifiers does not constitute an 'election' in true sense as the results are not based on the choices of the involved nodes in the system, assuming the nodes can indicate their preferences. Given that one of the fundamental problems in the area of distributed systems, the Byzantine Agreement problem, assumes malicious intent as well as collusion, it seems natural that the problem of fair democratic elections be also studied in this context.

Democratic elections have been studied extensively in the fields of economics and game theory. A large set of interesting problems for elections with three or more candidates have already been explored [1,2]. Arrow's theorem, an important result on this topic, shows impossibility of elections under some specific requirements [3]. Yet, the confluence of democratic elections (with more than two candidates) and distributed protocols has not been explored to the best of our knowledge. Involvement of Byzantine

* Supported by NSF NSF CNS-1115808, CNS-0718990, and the Cullen Trust for Higher Education Endowed Professorship.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 176–191, 2013.

processes in the system presents some additional challenges for this task. The notion of *strategy-proofness* [4] does not readily apply to Byzantine processes as they can be considered unaffected by individual losses. In this paper, we introduce the notion of democratic elections in distributed settings by addressing the following sub-problems:

Sub-optimality of Standard Protocol: With the background setting of elections, the idea of deciding on a winner based on every node's most preferred choice seems appealing. With this approach the eventual winner would be the node receiving highest number of votes (ties broken arbitrarily). This scheme is called 'plurality' scheme in economics and game-theory literature. However this approach does not always lead to outcomes that maximize the overall gains for the system. The term 'gains' may be attributed to any property that is relevant to global observations of the system, such as the overall latency of a message broadcast or the average load on each node in the system in some distributed computing protocol. For example, let us consider a system with seven nodes $\{P_1, P_2, \ldots, P_7\}$ that run a distributed protocol in presence of a leader node that controls the distribution of work. Based on the resulting latency or load values of their individual communications with three possible candidate nodes, the seven nodes want to elect a leader. Let *a*,*b*, and *c* denote the three possible outcomes of such an election among three candidates. For one such instance of voting assume that Table 1 represents the votes of each node in the system. This tabular representation means that P_1 prefers the outcome *b* the most, and then prefers *a* over *c*; the preferences of all other processes can also be inferred in this manner. 'Plurality' method on this vote profile, with coin-toss based tie breaking, elects *b* or *c* with equal probability, and never elects *a* . However, it is easy to verify that *a* beats both *b* and *c* on individual pairwise comparisons. Additionally, if a positional vote counting mechanism[1], such as Borda-Count Method (see Section 5) [5] is applied, then *a*'s score is strictly higher than those of both *b* and *c*. Hence, even though election of *a* seems the most optimal outcome for the overall system, the standard approach never elects *a*, and by electing either *b* or *c* reduces the social welfare[2] of the system.

Table 1. Votes by Processes P_1 to P_7

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1st choice	b	b	b	c	c	c	a
2nd choice	a	a	a	a	a	a	b
3rd choice	c	c	c	b	b	b	c

Strategic Voting by Byzantine Processes: The Byzantine processes can exhibit any kind of malicious behavior. One such malicious act is to cast strategic votes so that the overall social welfare is not maximized. For example, in Table 1, if P_7 is Byzantine, it may broadcast its vote (to all the processes) with *c* as its first choice and *a* as its

[1] Considers the positions of each candidate in all the votes, assigning fixed points to each position and then computing aggregate points of every candidate.

[2] Standard term from economics literature; defined in Section 2. For detailed explanations see [6].

last. With the changed vote profile, even the pairwise comparison, or positional voting schemes would not elect a. Thus, this fault would result in decrease of overall social welfare of the system. For binary choices, [7] studies the similar problem when one of the two available choices is more desirable, and it is beneficial for the system to agree on that choice despite the efforts of Byzantine processes.

In short the contributions of this paper are following:

- We introduce the problem of democratic elections in distributed systems by studying: Social Choice and Social Welfare in distributed settings with Byzantine faults.
- We present the impossibility and possibility results for some specific requirements for the problems.
- We propose a social welfare function called Pruned-Kemeny, and by means of simulations show that our scheme significantly outperforms other popular schemes for Byzantine Social Welfare problem.

2 Preliminaries

In economics and game theory, elections have been studied primarily in two forms – social choice functions, and social welfare functions [6]. For both of these forms, the voters are required to cast their vote indicating their preferences over all the candidates. As the result of voting, social choice functions elect one candidate as the winner; whereas social welfare functions produce an overall ranking of the candidates. Formally, these functions are defined as follows:

Let $\mathcal{A}$ be a set of choices/candidates and $\{P_1, ..., P_n\}$ be the set of n voters. Let $\mathcal{L}$ denote the set of linear orders on $\mathcal{A}$ ($\mathcal{L}$ is isomorphic to the set of permutations on $\mathcal{A}$). The preferences of each voter P_i are given by $\prec_i \in \mathcal{L}$, where $a \prec_i b$ means that P_i prefers choice b to choice a. A social welfare function $\mathcal{W}$ is a function of the form $\mathcal{W} : \mathcal{L}^n \rightarrow \mathcal{L}$; and a social choice function C takes the form $C : \mathcal{L}^n \rightarrow \mathcal{A}$.

The preferences of a voter are *strict* if the voter is not indifferent between any two candidates. Throughout the paper, we limit our focus to *strict* preferences. Construction of a social choice function from a social welfare function, and vice-versa is trivial [8]. Given a social welfare function $\mathcal{W}$, one could construct a social choice function by simply declaring the top-most ranked candidate in the result obtained by $\mathcal{W}$ as the social choice. Conversely, to construct a welfare function $\mathcal{W}$ from a given choice function C one could apply C over k candidates and place the winning candidate on top of the result of $\mathcal{W}$, and repeat this processes $k - 1$ times (at each iteration, removing the candidates already placed in the result).

A *ranking* is a total order over a fixed set of candidates. A *vote* is an individual voter's preference ranking over the set of candidates. Based on the above notation, $\prec_i$ is the vote of voter P_i. A *ballot* is a collection of the votes. The size of the ballot is the number of votes the ballot contains. A *scheme* is a mechanism that takes a ballot as input and produces a ranking or a winner as output. Given a ballot, the ranking/winner produced by any scheme is called the result of the scheme on that particular ballot instance.

Condorcet Candidate: If a candidate is preferred by all the voters over each of the other candidates in a head-to-head comparisons, then such a candidate is called *Condorcet Candidate*. It is not necessary that such a candidate always exists.

3 Model

We assume a synchronous distributed system consisting of n processes. In our model any two nodes in the system can communicate privately with each other, thus the induced communication graph is complete. Of the n nodes in the system, at most f can be Byzantine. For the synchronous model of communication, [9] showed that agreement can only be guaranteed when $f < n/3$. Throughout this paper, we assume that this bound of $f < n/3$ holds. All non-faulty processes in the system are called *good* processes, and the faulty processes are called *bad* processes. The terms *voters*, *processes*, and *nodes* represent the same entities, and are used interchangeably. The set of choices, $\mathcal{A}$, is known to all the nodes in the system and each node votes with its *strict* preferences as a total order over the elements of $\mathcal{A}$.

Byzantine Social Choice Problem: Given a set of n processes of which at most f are faulty, and a set $\mathcal{A}$ of k choices, design a protocol that elects one candidate as the social choice (while providing the guarantees 1 to 3 listed below).

Byzantine Social Welfare Problem: Given a set of n processes of which at most f are faulty, and a set $\mathcal{A}$ of k choices, design a protocol that produces a ranking of the choices (while providing the guarantees 1 to 3 listed below).

Protocol Guarantees

1. *Agreement*: All good processes decide on the same choice/ranking.
2. *Termination*: The protocol terminates in a finite number of rounds.
3. *Validity:* This condition imposes requirement on the choice/ranking decided based upon the preferences provided by the good processes.

If V is the validity condition selected for the election, then $\boldsymbol{BSC}(k, V)$ denotes the Byzantine Social Choice problem over k choices that satisfies the validity condition V; similarly $\boldsymbol{BSW}(k, V)$ denotes the Byzantine Social Welfare problem that is defined with the constraints of V for the available k choices. Some examples of validity conditions are listed in Table 2 in the context of BSC problem.

In the standard Byzantine agreement problem [9], all the good processes must agree on a common value $v \in \mathcal{A}$. The only requirement on the decided value is that if all good processes propose the same value v, then the value decided must also be v. If all good processes do not propose the same value, then there is no requirement on the value that is decided. In Byzantine Social Choice/Welfare problem the value decided by the protocol is important, as some of the choices/rankings may be more desirable than others.

4 Byzantine Social Choice (BSC)

For the Byzantine Social Choice (BSC) problem, we always require agreement, and termination conditions but may want to impose different validity conditions. In the

Table 2. Various Validity Conditions for Byzantine Social Choice

Condition	Description
S	If v is the top choice of all good voters, then v must be the winner.
M	If v is top choice of majority of good voters, then v must be the winner.
S'	If v is the last choice of all good voters, then v must *not* be the winner.
M'	If v is last choice of majority of good voters, then v must *not* be the winner.
P	If v is not the top choice of any good process, then v must *not* be the winner.

standard Byzantine Agreement literature, the problem of deciding from more than two choices is considered equivalent to that of choosing from a set of two choices because a solution for either one of the problems can be used to solve the other [10]. However, as we show in this paper (Sec. 1), this is not the case for the BSC problem. $\boldsymbol{BSC}(k, V)$ denotes a BSC problem over k choices that satisfies the validity condition V. Thus, $BSC(2, S)$ is the standard binary Byzantine Agreement. Note that when k equals two, S, P and S' are equivalent to the standard validity requirements for binary Byzantine Agreement protocol [11]. Similarly, M and M' are equivalent when there are only two choices.

$BSC(3, M)$ is the Byzantine social choice problem on three choices with agreement, termination, and the majority validity condition. We show in Section 4.1 that this problem is impossible to solve in a distributed system. However, somewhat surprisingly $BSC(3, M')$ is possible to solve. As an example of $BSC(3, M')$ consider the problem of leader election in a distributed system with Byzantine processes. Suppose that processes need to choose a leader among three choices. It is known that one of the three choices may be Byzantine and the good processes would want to avoid its election. Although there is no initial agreement on which of these choices is Byzantine, it is a reasonable assumption that majority of good processes will identify the Byzantine choice correctly. In Section 4.2, we give a protocol for solving $BSC(3, M')$.

Observe that $BSC(k, S)$ is simply the standard Byzantine agreement problem in which every process proposes its first choice. Hence $BSC(k, S)$ is solvable for any k so long as $f < n/3$. It is also possible to solve $BSC(k, S \wedge S')$. We give such a protocol in Section 4.2.

4.1 Impossibilities

Arrow [3] showed that for elections with three or more alternatives, no voting system that provides two basic properties: *Pareto Efficiency* and *Independence of Irrelevant Alternatives (IIA)*, can guarantee non-dictatorial elections. In this section, we show impossibilities for elections in distributed systems with Byzantine faults. We focus on instances of $BSC(k, V)$ problems which are impossible to solve for specified values of k and V. Let us first consider the case when k equals two. For this case, the conditions S, P and S' are equivalent. Standard Byzantine agreement protocols can be used to solve $BSC(2, S)$.

Lemma 1. *There is no protocol to solve* BSC$(2, M)$ *when* $f \geq n/4$.

Proof: If $f \geq n/4$, then good processes are at most $3n/4$. Suppose that the set of choices is $\{a, b\}$. Assume that just the bare majority of good processes propose value a. Thus, the total number of processes proposing a is at most $3n/8 + 1$. The number of processes proposing b is at least $5n/8 - 1$. Then for $n \geq 4$, we have that more processes are proposing b. Since processes do not know which processes are good, this problem is indistinguishable from the instance where $5n/8 - 1$ good processes propose b and remaining $3n/8 + 1$ processes propose a. In the second instance, the protocol must choose b, and therefore it will also choose b in the first instance. ■

Lemma 2. *There is no protocol to solve* BSC$(2, M')$ *when* $f \geq n/4$.

Proof: For binary choices, $k = 2$, it can be easily observed that the problem $BSC(2, M')$ is equivalent to $BSC(2, M)$. Thus, based on the result of previous lemma, $BSC(2, M')$ is also unsolvable when $f \geq n/4$. ■

Lemma 3. *There is no protocol to solve* BSC(k, P) *for any* $k \geq n$ *when* $f \geq 1$.

Proof: Given that $k \geq n$, consider the case when each process proposes a different value. Since each value appears exactly once, there is no way to distinguish the value proposed by a bad process from that proposed by a good process. ■

Theorem 1. *There is no protocol to solve* BSC(k, M) *for any* $k \geq 2$ *when* $f \geq n/4$.

Proof: Suppose that there is a protocol that solves $BSC(k, M)$ for any $k \geq 3$. We will use this protocol to solve $BSC(2, M)$. Given an instance of $BSC(2, M)$ problem, all the processes construct an instance of $BSC(k, M)$ by first constructing $k-2$ *artificial* choices. However, none of the good processes use these choices as their first two choices. Now they run the protocol for $BSC(k, M)$ which must choose a value that has been proposed by the majority (ties broken in favor of lower value) of good processes as the first choice. All good processes return this as the decided value for the given BSC $(2, M)$ problem. But by Lemma 1, $BSC(2, M)$ is unsolvable. ■

4.2 Possibilities

As $BSC(k, S)$ is solvable by standard Byzantine agreement [10] and $BSC(k, M)$ is unsolvable, it is natural to seek some validity conditions that admit solution. Consider the following validity condition:

M_o *(Overwhelming Majority)*: If there is a choice that is the first choice of at least $3/4^{th}$ good processes, then all the good processes elect that choice.
It can be observed that any protocol that ensures M also ensures M_o. Similarly M_o is a stronger requirement than S, and thus any protocol providing guarantee on M_o also guarantees S.

Lemma 4. *Protocol* α *given by Algorithm 7, solves* BSC(k, M_o) *when for any* $k \geq 2$ *when* $f < n/3$.

Proof: Let v be the value proposed by at least $3/4^{th}$ fraction of processes. It is easy to see that $3/4 * (n - f) > 1/4 * (n - f) + f$ for all values of $f < n/3$. Hence, all processes decide on v. ■

Algorithm 1. Protocol α at P_i to ensure $BSC(k, M_o)$ and therefore also $BSC(k, S)$

```
T: array[1..n] (container to store all the votes)          /* Proposals */
vote: array[1..k] (ranking of k candidates)                /* My vote */
/* Every process proposes its first choice */
T[i] = vote[1]                                             /* index starts from 1 */
/* Step 1: Exchange first choice with all */
for j = 1 to n do
   send T[i] to P_j
   receive T[j] from P_j
   /* if no value received from P_j set T[j] = 0 */
end for
/* Step 2: Agree on T vector : the ballot of all votes */
for j = 1 to n do
   run Standard Byzantine Agreement on T[j];
end for
/* Step 3: Choose the value with the highest tally, breaking ties in favor of the smaller value */
return the least value from 1..k that has the highest frequency in T.
```

We showed in Section 4.1 that P is impossible to achieve when $k \geq n$. However, if choices are limited, then P can be guaranteed as follows.

Lemma 5. *Protocol α solves* BSC(k, P) *for* $2 \leq k < n$ *when* $f < min(n/k, n/3)$.

Proof: It is sufficient to show that the largest tally would be of a value proposed by a correct process. Suppose, if possible, the largest tally is for the value v which is not proposed by any good process. The tally for v can be at most f. There are $n - f$ proposals by good processes. None of these proposals is for v, and therefore all these proposals are for remaining $k - 1$ values. Since none of these values had tally more than v, we get that the total number of proposals possible are $(k - 1) * f$. From $f < n/k$, we obtain that $(k - 1) * f < n - f$ which is a contradiction because all correct processes make at least one proposal. ■

However, if we were to require $(M' \wedge P)$ and use the steps in the protocol α with suitable adjustments (not picking a social choice that would violate M') to handle the validity requirements – it would be evident that the modified protocol α would not satisfy $(M' \wedge P)$. It is not possible for a protocol to deterministically know which nodes are *good* and which are *bad* in all the instances. Thus to provide M' the only option any deterministic protocol would have to discard a choice that appears $\lfloor (n - f)/2 + 1 \rfloor$ or more times as the bottom choice in the ballot. Consider the example ballot presented in Table 3, with P_6 and P_7 as Byzantine voters. In this example, a simple majority over the first choices would result in choosing c as the winner which violates M'. The modified protocol α (that attempts to provide M') will elect a as the winner. However, an overwhelming

Table 3. A ballot with P_6 and P_7 as Byzantine Voters

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1st choice	b	b	b	c	c	c	c
2nd choice	a	a	a	b	a	a	a
3rd choice	c	c	c	a	b	b	b

majority of *good* processes, 4 out of 5, prefer b over a. Note that the choice a is not the first choice for any process, leave alone being the first choice of a *good* process. In this example, with $n = 7$ and $f = 2$, $\lfloor (n - f)/2 + 1 \rfloor$ is 3 and thus it is clear that any protocol that guarantees M' can only choose a as winner (because both b and c are last choices for at least three processes).

We now show the surprising result that $BSC(k, M' \wedge S)$ is solvable for $k \geq 3$ when $f < n/3$. Protocol β, shown in Algorithm 2, is based on the idea of processes proposing their last choice. Since Byzantine processes may send conflicting values to different processes, Protocol β first agrees on the vector T of last choices. Each process then discards the values that appear as the last choice at least $\lfloor (n - f)/2 + 1 \rfloor$ times. It should be noted that since $f < n/3$, the size of *discard* set in protocol β is at most two. Now all the processes run Byzantine Agreement with their top choice from the remaining set.

Algorithm 2. The $BSC(k, M' \wedge S)$ Protocol β at P_i

```
T: array[1..n] (container to store all the votes)          /* Proposals */
vote: array[1..k] (ranking of k candidates)                /* My vote */
/* Every process proposes its last choice */
T[i] = vote[k]
/* Step 1: Exchange last choice with all */
for j = 1 to n do
   send T[i] to P_j
   receive T[j] from P_j
   /* if no value received from P_j set T[j] = 0 */
end for
/* Step 2: Agree on T vector, ballot of last choice votes */
for j = 1 to n do
   run Standard Byzantine Agreement on T[j];
end for
/* Step 3: Eliminate unqualified choices */
discard = set of choices to discard; initially {φ}
for j = 1 to k do
   /* count returns the frequency of any value in T */
   if (count(vote[j]) >= ⌊(n − f)/2 + 1⌋) then
      add vote[j] to discard
   end if
end for
/* Step 4: Now use the remaining choices for selecting top choices of processes */
run Byzantine Agreement on top choice ∉ discard
```

Lemma 6. *Protocol β, given by Algorithm 2, solves* BSC$(k, M' \wedge S)$ *for* $k \geq 3$ *when* $f < n/3$.

Proof: We first note that if P_i is good then $T[i]$ at P_j will be same as the value proposed by P_i. This means that if there is any value v that is considered the last choice by a majority of good processes then it appears at least $\lfloor (n-f)/2 + 1 \rfloor$ times in T vector; all such values are discarded. Since $k \geq 3$ and $| \mathit{discard} | \leq 2$, there is at least one value which is not discarded by any good process. Hence, the agreement phase in step 4 leads to selection of a choice proposed by some good process.

It is also easy to verify that the protocol satisfies S. If all good processes have v as their first choice, then it cannot appear $\lfloor (n-f)/2 + 1 \rfloor$ times as the last choice. Hence no good process will discard this choice and will propose it in step 4. ■

Lemma 7. *Protocol β does not guarantee* M_o *condition.*

Proof: Consider the vote ballot presented in Table 4 in which 4 out of 5 *good* processes have b as their first choice. Since it can not differentiate between *good* and *bad* processes based on the ballot, protocol β would be forced into electing a as the social choice. ■

Table 4. A ballot with P_6 and P_7 as Byzantine Voters

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1st choice	b	b	b	b	a	c	c
2nd choice	a	a	a	a	c	a	a
3rd choice	c	c	c	c	b	b	b

5 Byzantine Social Welfare (BSW)

The problem of Byzantine Social Welfare can be seen as an extension to the BSC problem. In the Byzantine Social Welfare (BSW) problem, the goal is to produce a *ranking*, a total order over k candidates, of choices as the result of elections. Multiple such schemes exist in the literature of economics and game theory. We now discuss some of these as social welfare functions, and propose a new scheme called Pruned-Kemeny specially tailored towards handling Byzantine votes. We focus only on the schemes that require a single round of voting. After exchanging their votes with all the other processes in the system, the processes participate in $O(f)$ rounds of agreement to ensure that all the good processes agree on the same ballot.

From here on, for notational convenience we use a short form representation of rankings such that *abc* represents ranking $a \succ b \succ c$.

PlacePlurality: For each position in the result ranking, the scheme finds the candidate with most votes for that position in the ballot, and places this candidate at that position in the result. Only the candidates that are not already placed in the result ranking are considered. Plurality based schemes satisfies S and S' criteria. Revisiting the example ballot of Table 1 from Section 1, one can verify that the rankings *cab* and *bac* are the two possible outcomes of a social welfare function that applies PlacePlurality.

Pairwise Comparison: This scheme uses the Condorcet Criterion and compares the pairwise preferences over the ballot for all $\binom{k}{2}$ pairs. Detailed description of the scheme is presented in [12]. Using this scheme, the output for the welfare function on the ballot of Table 1 is *abc*.

Borda Count: This scheme applies the positional voting approach to calculate the points scored by each candidate. Each position in a vote is assigned unique points - top position given the highest, and the bottom position given the lowest points. The cumulative score of a candidate is the sum of all the points it accumulates over the complete ballot. The resulting social welfare ranking is the list of candidates sorted in non-increasing order of their overall scores (ties broken arbitrarily). For the ballot of Table 1, the social welfare result using Borda Count scheme is *abc*.

Young in [13] shows with convincing arguments that most of the simple schemes, including the three presented above, do not necessarily produce the outcomes that are optimal in terms of overall representation of the voter preferences. The discussion in [13] points out that schemes that use the *mean* as the representative outcome of the voting process tend to generate 'inferior' results as opposed to the schemes that try to compute an outcome that is close to the *median*. The following two schemes, use the median as the basis for the result computation, and we show by means of simulation that they do produce *better* social rankings.

Kemeny-Young Scheme: This approach, proposed by J. Kemeny and H. Young in [14,13], uses a metric to identify a ranking that is closest to the median of the ballot. The metric used in this scheme is the *distance* between rankings, where *distance* between any two rankings is defined as the number of pairs on which the rankings differ. For example, taking $r = abc$ and $r' = bac$ the *distance* between these two rankings is 1, whereas if $r' = cba$, then the *distance* between r and r' is 3.

Algorithm 3 presents the steps involved in the scheme. The scheme iterates over each of the possible $k!$ permutations of k candidates and considers each ranking (permutation). The goal is to identify a ranking that maximizes the agreement on pairwise comparisons with the overall ballot. For a detailed analysis of the scheme, we refer the reader to [13]. Applying this scheme on the ballot in Table 1 gives a result ranking of *abc*.

Pruned-Kemeny: We propose a scheme called Pruned-Kemeny that is aimed towards mitigating the damaging effects of *bad* voters. The key motivation for this scheme is that good voters, while indicating their individual preferred choices would in addition be also inclined towards the final outcomes that are beneficial to the overall system. Where as the *bad* voters would not only send conflicting information to the *good* voters, but also focus on manipulating their vote preferences in order to reduce the overall welfare of the system. The steps of the scheme are presented in Algorithm 4. Similar to the Kemeny-Young scheme, our approach also iterates through all the possible permutations of candidates but we restrict the ballot in consideration for each iterated permutation. The restriction on the ballot is attained in the following manner:

Let $\mathcal{P}$ denote the set of all permutations of k candidates, and $\mathcal{B}$ be the initial ballot of n voters. For each ranking $r \in \mathcal{P}$ compute a pruned ballot $\mathcal{B}'$ by setting $\mathcal{B}' = \mathcal{B} \setminus \mathcal{F}$, where $\mathcal{F}$ is the collection of f most *distant* rankings in $\mathcal{B}$ from r. Hence, size of the

Algorithm 3. Kemeny-Young Scheme

```
P = Set of all permutations of k candidates,
B = agreed upon ballot of n votes
maxScore= 0, maxRank = nil
for each ranking r in P do
   score = Kemeny-YoungScore(r, B)
   if score > maxScore then
      maxScore= score
      maxRank = r
   end if
end for
return maxRank

Kemeny-YoungScore(ranking, ballot):
  score = 0
  for each pair (a>b) in ranking do
     score = score + # of a>b in ballot
  end for
return score
```

Algorithm 4. Pruned-Kemeny Scheme

```
P = Set of all permutations of k candidates,
B = agreed upon ballot of n votes
maxScore= 0, maxRank = nil
for each ranking r in P do
   F = f most distant rankings from r in B
   B' = B \ F
   score = Kemeny-YoungScore(r,B')
   if score > maxScore then
      maxScore = score
      maxRank = r
   end if
end for
return maxRank
```

restricted ballot $\mathcal{B}'$ is $n - f$. The score for ranking r is its Kemeny-Young score on $\mathcal{B}'$. The result of the scheme is the ranking with highest score (ties broken arbitrarily). For instance, when applied to the ballot of Table 1, this scheme produces *bac* as the result.

We show shortly that the problem of finding a solution to the election problem using either Kemeny-Young or Pruned-Kemeny scheme is NP-Hard. In the context of distributed systems, the round and message complexities for agreement on the the ballots, performed before application of the schemes, are essentially the complexities of the protocols used reach agreement. We use the Gradecast based Byzantine agreement protocol presented in [15], mainly because this protocol provides the early termination property. Based upon this, the agreement requires $O(f)$ rounds, and has the message complexity of $O(fn^3)$. For proofs and detailed discussions on these bounds we refer the reader to [15].

Lemma 8. *The problem of finding the result of a ballot using Pruned-Kemeny scheme is NP-Hard.*

Proof: Consider any instance of the problem of finding optimal rankings with Kemeny-Young scheme. Each such instance can be converted to an instance of the problem of finding the result with Pruned-Kemeny with f set to zero. Hence, the Pruned-Kemeny based optimization problem is at least as hard as the Kemeny-Young based problem, which is already known to be NP-Hard [16]. ■

Theorem 2. *Pruned-Kemeny satisfies S and S' requirements.*

Proof: First, we prove that Pruned-Kemeny satisfies S. Let us assume that Pruned-Kemeny violates S, and thus its output is ranking r that does not put v on top when all the good processes put v as their top choice. Hence, there is at least one candidate u that

is immediately above v in r. Construct a new ranking r' by swapping the places of u and v in r. We now show that r' would have a higher Kemeny-Young score than r, which would be a contradiction. Since r' puts v above u, it agrees with all the good processes on at least one more pairwise comparison. It may disagree with the votes of all the bad processes. Also, in the worst case scenario r' discards f good votes during the protocol run. Thus in the worst case the overall Kemeny-Young score of r' increases by

$$(n - f) - f - f = n - 3f$$

in comparison to the Kemeny-Young score of r. The first term of $(n - f)$ is due to the increment (by at least one point) in score for each *good* vote, however if we assume that it is possible to discard f *good* votes in the worst case, the second term indicates that adjustment. Also, all the *bad* processes might provide exact opposite rankings in their votes, hence a further decrease of f (third term) in points is possible in the worst case. Since $n \geq 3f + 1$, the score of r' is strictly greater than that of r, which means r being selected as the final outcome of Pruned-Kemeny is a contradiction. S' can be shown similarly by placing v at the bottom of each good vote. ■

Similar to the Kemeny-Young scheme, Pruned-Kemeny also performs exponential computations by iterating over all the $k!$ permutations. However, for small values of k and large values of n, the performance of the scheme is acceptable.

Lemma 9. *Kemeny-Young scheme satisfies S and S′ requirements.*

Proof: As Kemeny-Young is a special case of Pruned-Kemeny scheme with f set to zero; the proof immediately follows from Theorem 2. ■

6 Simulation Results

It is possible to have scenarios in which the *bad* voters need not just send conflicting information, but may as well have much more malignant intentions. Consider the case when the *good* voters want to reach a consensus on a ranking that is beneficial to the system as a whole, and thus have similar if not exactly the same preferences. On the other hand, the *bad* voters may want to minimize the benefit that the system may attain by the resulting welfare ranking (that is the outcome of the election). Schemes that do not assume that a small section of voters might behave in this manner, may thus produce rankings which are prone to manipulation by the *bad* voters. Given the knowledge that at most f voters can be *bad*, our scheme Pruned-Kemeny tries to produce best possible social welfare outcomes in presence of such hostile voting by the *bad* voters.

We now list the details of our experimental setup and the simulations performed to evaluate the utility of Pruned-Kemeny in computing 'near-optimal' welfare rankings in comparison to the other discussed schemes. Let ω represent an *ideal* ranking for the BSW problem, such that selection of ω as the result of the election maximizes the social welfare of the system. Let us assume that ω is not completely known to any good process, however each good process tends to favor the *ideal* ranking. The voting preferences of good and bad processes in presence of an *ideal* ranking are defined as follows:

Let *goodProb* denote the probability of a good voter ranking two candidates a and b in the same order as that in the ideal ranking ω, and *badProb* denote the probability of a *bad* voter ranking the candidates in the reverse order to that in ω. Hence, if ω ranks two candidates a and b with $a \succ b$ then *goodProb* is the probability that any *good* voter decides to put $a \succ b$ in its vote, and *badProb* is the probability that any *bad* voter puts $a \prec b$ in its vote. For our experimental setup we fix the following values:

$$n = 100, f = 33, badProb = 0.9$$

By setting f to its highest possible value, and *badProb* to a considerably high value in the possible range, we try to realize the assumption that *bad* voters would want to disrupt the election of ideal ranking, and would vote in opposite polarity of the *good* voters. The value of the number of candidates k is varied in the range [3,8]. For each value of k, the value of *goodProb* is varied from 0.55 to 0.90 in step increments of 0.05. For each such resulting configuration of $\langle k, n, f, goodProb, badProb \rangle$, 50 ballots (of $n = 100$ voters) are generated by fixing an *ideal* ranking and applying the probabilistic model on individual votes based on *goodProb* and *badProb*. We then apply the discussed schemes, and find the *distance* (defined in previous section) of their result rankings from the *ideal* ranking. We then compute the average distance over the 50 ballots for each configuration.

Figure 1 shows the variation in the average distance values. As evident from the plots, Pruned-Kemeny produces results that are much closer to the *ideal* ranking even for comparatively low values of *goodProb*. In addition, the plots also indicate that as the number of candidates increases, the results of Pruned-Kemeny consistently match the *ideal* ranking. Another interesting observation is that the distance of results for PlacePlurality from the *ideal* ranking increases significantly with increase in the number

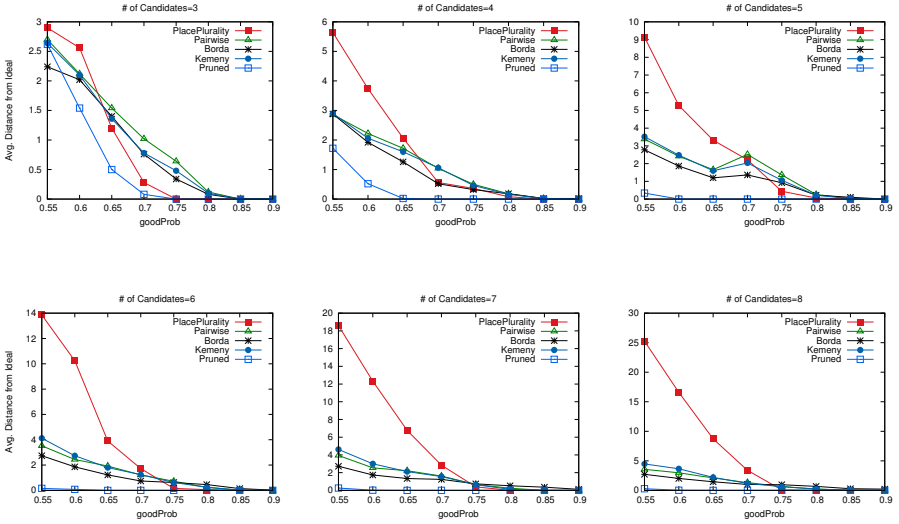


Fig. 1. Comparison of Average Distance of Results from Ideal

of candidates. This clearly indicates that PlacePlurality is not a good choice for a social welfare function. We argued in Section 1 that for more than three choices, the plurality based methods do not guarantee best results. The observations on the variation in result distances clearly validate our argument.

7 Discussion

Extensive literature is already present on the topic of leader election in distributed systems [11,17,18,19]. [11] presents various protocols and lower bounds for message complexity for the leader election problem in absence of Byzantine processes. Leader Election has also been studied in presence of Byzantine failures. [20] gives a randomized distributed protocol to elect a leader in the asynchronous full information model that tolerates $n/(6+\epsilon)$ cheaters with positive constant success probability in rounds that is polylogarithmic in n.

Our work studies the problem of democratic elections in a distributed system as social choice and social welfare ranking problems [6]. When number of choices is more than two, elections based on the top-preference-only model may not lead to optimal results, and hence we assume that processes in the system propose a ranking of candidates rather than a single leader. For agreement that is dependent only on the number of failures we use the deterministic early-stopping Byzantine agreement protocol from [15] to reach the agreement on every processes' vote within $min\{f+1, f_a+2\}$ rounds where f_a is the actual number of failures. We focus on the guarantees on the social choice or the social welfare ranking produced by the election, rather than on the message or bit complexity of election protocols.

Prisco et al. in [21] present some impossibility and possibility results for the *k-set* consensus problem in which each node starts with one value and the protocol must decide on a value so that at most total k values are decided by the correct processes. The *k-set* problem does not involve voting over multiple candidates. Under some specific boundary conditions there is a slight overlap between two impossibility results in [21] and those presented in this work.

In the standard Byzantine agreement [9] the protocols only need to guarantee agreement on some value that is proposed by a good process. With this objective, the protocols do not need to guard against the possibility of Byzantine voters affecting the eventual outcome by strategic reporting of their values. However, as we saw in Section 3 it is important to design voting mechanisms that do not allow this advantage to Byzantine voters.

8 Future Work

If all the good processes lean towards some fixed *ideal* ranking, even with weak inclinations, the simulation results indicate that our proposed approach Pruned-Kemeny provides desired results with much higher accuracy in comparison to other schemes. However, determining the provable guarantees for optimal results under some specific conditions is an important open challenge for this work.

Another interesting problem is to differentiate between the ideal results, and the results that comply with the Condorcet Criterion. It should be noted that for some given ballot, it is possible to have a clear Condorcet candidate/ranking yet the ideal winner/ranking might differ from it. However, in terms of computational complexity both Kemeny-Young and Pruned-Kemeny schemes are NP-Hard, where as a Condorcet candidate/ranking can be found in polynomial time. With this observation, it would be beneficial to design a social welfare scheme that can strike a balance between these two approaches. Depending on the constraints of the computing environment, this balanced scheme could have the flexibility to employ either the PrunedKemeny or the Condorcet scheme so that the difference between the social welfare resulting from the two outcomes is either relatively small or bounded in some acceptable form.

9 Conclusion

In this paper, we introduced the problem of democratic elections in distributed systems. We showed that the standard approach of reducing three or more choices to binary choices does not guarantee optimal outcomes, and hence the standard assumption of always having binary choices is weak. We presented impossibility results under some specific validity requirements, as well as showed some surprising possibilities that result from availability of more than two choices.

For producing results that are close to an ideal ranking when there exists one, we proposed a new scheme called Pruned-Kemeny that aims to counter the votes of Byzantine processes. The results of our simulations show that for the purpose of finding ideal order, Pruned-Kemeny provides significantly improved results over existing voting systems.

References

1. Arrow, K.J.: Social Choice and Individual Values. Yale University Press (1951)
2. Farquharson, R.: A Theory of Voting. Yale University Press (1969)
3. Arrow, K.J.: A difficulty in the concept of social welfare. Journal of Political Economy 58, 328–346 (1950)
4. Ishikawa, S., Nakamura, K.: The strategy-proof social choice functions. Journal of Mathematical Economics 6, 283–295 (1979)
5. Saari, D.G.: Mathematical structure of voting paradoxes: Positional voting. Economic Theory 15, 55–102 (2000)
6. Graaff, J.V.: Theoretical Welfare Economics. Cambridge University Press (1957)
7. Garg, V.K., Bridgman, J., Balasubramanian, B.: Accurate Byzantine Agreement with Feedback. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 465–480. Springer, Heidelberg (2011)
8. Buchanan, J.M.: Social choice, democracy, and free markets. Journal of Political Economy 62, 114–123 (1954)
9. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of ACM 27, 228–234 (1980)
10. Turpin, R., Coan, B.A.: Extending binary byzantine agreement to multivalued byzantine agreement. Inf. Process. Lett. 18, 73–76 (1984)

11. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
12. Young, H.P.: Condorcet's theory of voting. American Political Science Review 82(4), 1231–1244 (1988)
13. Young, H.P.: Optimal voting rules. Journal of Economic Perspectives 9(1), 51–64 (1995)
14. Kemeny, J.G.: Mathematics without numbers. Daedalus, Quantity and Quality 88(4), 577–591 (1959)
15. Ben-Or, M., Dolev, D., Hoch, E.N.: Simple gradecast based algorithms. CoRR, vol. abs/1007.1049 v3 (2010)
16. Bartholdi, J., Tovey, C.A., Trick, M.A.: Voting schemes for which it can be difficult to tell who won the election. Social Choice and Welfare 6(2), 157–165 (1989)
17. Ostrovsky, R., Rajagopalan, S., Vazirani, U.: Simple and efficient leader election in the full information model. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, STOC 1994, pp. 234–242 (1994)
18. Feige, U.: Noncryptographic selection protocols. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science (1999)
19. Russell, A., Zuckerman, D.: Perfect information leader election in log*n + o(1) rounds. In: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, FOCS 1998, pp. 576–583 (1998)
20. Kapron, B., Kempe, D., King, V., Saia, J., Sanwalani, V.: Fast asynchronous byzantine agreement and leader election with full information. In: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, pp. 1038–1047 (2008)
21. Prisco, R.D., Malkhi, D., Reiter, M.: On k-set consensus problems in asynchronous systems. In: Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC 1999), pp. 257–265 (1999)

Robust Deployment of Wireless Sensor Networks Using Gene Regulatory Networks

Azade Nazi[1], Mayank Raj[1], Mario Di Francesco[2], Preetam Ghosh[3], and Sajal K. Das[1]

[1] University of Texas at Arlington
[2] Aalto University
[3] Virginia Commonwealth University
{azade.nazi,mayank.raj}@mavs.uta.edu, mario.di.francesco@aalto.fi, pghosh@vcu.edu, das@uta.edu

Abstract. Sensor nodes in a Wireless Sensor Network (WSN) are responsible for sensing the environment and propagating the collected data in the network. The communication between sensor nodes may fail due to different factors, such as hardware failures, energy depletion, temporal variations of the wireless channel and interference. To maximize efficiency, the sensor network deployment must be robust and resilient to such failures. One effective solution to this problem has been inspired by Gene Regulatory Networks (GRNs). Owing to millions of years of evolution, GRNs display intrinsic properties of adaptation and robustness, thus making them suitable for dynamic network environments. In this paper, we exploit real biological gene structures to deploy wireless sensor networks, called bio-inspired WSNs. Exhaustive structural analysis of the network and experimental results demonstrate that the topology of bio-inspired WSNs is robust, energy-efficient, and resilient to node and link failures.

1 Introduction

Wireless Sensor Networks (WSNs) are used in a wide variety of applications, such as environment monitoring, tracking, pervasive security, smart healthcare, as well as in disaster management and recovery to deliver critical information. Sensor nodes are deployed over a monitoring area, perform in-network processing (e.g., aggregation, filtering) of data and forward the results to the sink via multi-hop paths. The communication between the sensors may fail due to different factors, such as, the distance between the nodes, temporal variations of the wireless channel and interference. Additionally, node failures can also occur due to hardware failures or sensor nodes running out of energy, thus disrupting the network. Hence, the deployment of WSNs must be robust and resilient to such failures.

Bio-inspired approaches have been extensively used for solving many problems in WSNs, for instance, the use of swarm intelligence to organize and route data in WSNs [19,7]. In this regard, Gene Regulatory Networks (GRNs) a graph-based

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 192–207, 2013.

representation of genes/proteins and the interactions between them, serve as a useful model for robust network design. Similar to sensor nodes, genes perform three major functions: sensing, actuating and signaling. In the sensing phase, genes sense the levels of specific proteins in the cells through signals mediated by other interacting genes and environmental variables, (e.g., temperature) to determine their own expression levels. Then, in the actuating phase, each gene produces enhancer or inhibitor proteins to regulate the expression level of other genes in the network. Lastly, in the signaling phase, genes interact with other genes to co-regulate targeted protein levels in the cell. This distributed behavior of the genes has evolved over many millennia into a robust network [18,15]. Thus, in order to design a robust WSN, the topology of GRNs can be used as a template for deploying sensor nodes.

In this paper, we use the topology of GRNs to deploy WSNs. Specifically, we extract a GRN sub-network and construct a WSN based on the extracted topology. Such WSN deployment requires to determine the locations of the sensor nodes so that the physical topology of the WSNs follows the topology of the extracted GRNs. We model this problem as a non-linear optimization by using the virtual force approach. We call the resulting WSNs as bio-inspired WSNs. We hypothesize that bio-inspired WSNs inherit the properties of GRNs, such as robustness [18,15] and small maximal diameter [9]. We analyze the topology of the bio-inspired WSNs with k-connected and randomly deployed WSNs using a graph-theoretic analysis. Furthermore, we carry out a comprehensive set of experiments to validate our claims. The performance of bio-inspired WSNs is compared with k-connected and random WSNs in terms of packet delivery, latency and energy consumption in presence of node and link failures.

The rest of the paper is organized as follows. Section 2 discusses several applications of GRNs and motivates their use for deploying WSNs. The deployment of bio-inspired WSNs is presented in Section 3. Graph-theoretic analysis of the bio-inspired WSN topology is presented in Section 4 followed by performance evaluation based on experiments. Finally, Section 5 concludes the paper with directions of future research.

2 Background and Motivation

In the literature many solutions exist for deployment of wireless sensor networks with different goals, including robustness [11], maximum coverage [23] and energy efficiency [22]. In [11] various algorithms for constructing the topology of WSNs with consideration of fault tolerance are discussed. The problem is modelled as finding a minimum-cost k-vertex connected topology which is NP-hard [11]. In [14], the authors show that k-connectivity is not a necessary and sufficient condition for robustness. They proved that robustness in WSNs can be further enhanced by optimally selecting the sinks. However, the optimal sink selection problem is also NP-Hard [14]. As a result, we took inspiration from nature to solve such complex problems.

Biological systems are governed by simple generic rules which produce efficient collaborative patterns for distributed resource management, task allocation and synchronization. Furthermore, millions of years of evolution has made biological systems adaptive, robust, resilient, efficient and exhibiting self-⋆ properties [18,15]. These features of biological systems inspire solutions to many problems in WSNs, such as maximizing network coverage, topology construction, routing, duty-cycling and clustering [19]. In this paper, we consider a robust and efficient network model for biological systems, called Gene Regulatory Network (GRN).

2.1 Applications of Gene Regulatory Networks to WSNs

With increasing level of detail and complexity, GRNs can be modelled as graphs, Boolean random networks and non-linear differential equations [12]. Boolean random networks and non-linear differential based models emulate the control process and the real time behavior of GRNs as well as its response to environmental changes. In [8], the Boolean network model is used to identify the attractors. The attractors represent the set of steady states in the GRNs at the end of evolution which are resilient to dynamic removal or functional problems of nodes in the network. This makes the GRNs functionally robust wherein cellular functions persist even when an element of the underlying genes are exposed to adverse conditions. Thus modeling WSNs using the Boolean network model can address the issue of robustness in them. The non-linear differential equation model of GRN has been used in [6] to identify the minimum number of sensors required for maximum coverage in WSNs. Furthermore, the authors in [6] show that the proposed algorithm performs similar to genetic-based optimization algorithms. In [17], the authors also use the non-linear differential equations based model to emulate the evolution process of genes in GRNs for self network configuration.

However, owing to the high complexity of models based on Boolean random networks and non-linear differential equations, the size of the networks that can be modeled using them is limited [21]. On the other hand, WSN deployments consist of a large number of sensor nodes with limited resources; hence, regulating WSN deployment behavior based on the above models may not always be feasible. In this paper, we use the evolved topology of GRNs, represented as an undirected graph, which provides the necessary properties to build robust WSNs. The graph models represent the genes and their interactions, implying much larger networks can be described using them. Each gene is represented as a node and an edge exists between two genes if one of the gene interacts with the other gene. The GRN topology posses many attractive properties, like robustness, resilience to failure and small diameter [9], which are desired in WSNs.

2.2 Motivation for Using GRN Topologies in WSNs

The properties of robustness and resilience to failures in GRNs are central to the proper functioning and adaptation of all organisms. These properties have been attributed to two of the features inherent to GRNs: 1) the existence of

motifs [18,15] which are specific sub-graphs that occur with higher frequencies in the underlying GRN topology; and 2) the degree distribution of gene nodes in the GRNs [4]. The robustness of a network is highly correlated to the abundance of motifs in the network which provide selective advantages in evolution [18,15]. Furthermore, the degree distribution of genes follows a power-law distribution [4]. In other words, gene nodes with high degree tend to have an early evolutionary history and, more importantly, the mechanism of attachment to them tends to be linear. Hence, a new node is twice as likely to link to a node that has twice as many connections. Thus, the ratio of the number of the nodes with high degree to those with low degree is very small, thereby making the GRN topology robust and resilient to failure because the probability that a node with high degree fails is low. We claim that a WSN built using the topology of GRNs will inherit the above-mentioned properties. Furthermore, the GRN topology exhibits a small diameter [9] which results in lower latency and energy consumption as packets are delivered to the sink using shorter paths in the WSNs.

In [13], the authors carried out a performance evaluation of WSNs built on top of GRNs. The paper considers a single source, flooding based routing protocol and static links, which makes the assumptions unrealistic. This motivates us to carry out an exhaustive investigation of the performance of WSNs built on top of GRNs and using the IEEE 802.15.4 standard specifications [3], thus paving the way for deployment of fault-tolerant and robust WSNs as well as efficient design of routing and other data dissemination algorithms.

3 Bio-inspired WSNs

Let us assume a set of sensors (V_w) needs to be deployed, such that the network is robust and resilient to failures. To construct a bio-inspired WSN topology we need to extract a sub-network from the GRN with the same number of nodes. In the sub-network, the interactions of genes represent the communications between the sensor nodes. The extracted sub-network of GRN is represented as $G'_g = (V'_g, E'_g)$ where V'_g is the set of genes and E'_g represents the interactions between the genes. Let $G_w = (V_w, E_w)$ be the corresponding bio-inspired WSN graph, such that V_w is the set of sensor nodes and E_w is the set of communication links in the WSN. To build the bio-inspired WSN (G_w) based on the extracted sub-network, we must follow the following rules: 1) every gene in the extracted sub-network is mapped to a sensor node in the WSN; and 2) an edge exists between two sensor nodes in G_w if an edge exists between the corresponding gene nodes in G'_g. An edge in the bio-inspired WSN implies that the two sensor nodes communicate with each other and thus are in the communication range of each other.

These rules imply that the gene sub-network and bio-inspired WSN are isomorphic to each other, i.e., $G_w \simeq G_g$. Ideally the WSN topology is dependent on the locations of the sensor nodes. However, in our case, we need to assign locations to the sensors to derive the given topology. Once the sensor locations are determined, we assume the sensors can be deployed at the given locations. We

address the issues of sub-network extraction and determining the sensor nodes location in the following subsections.

3.1 Extraction of GRN Sub-networks

The interactions of genes in the regulatory network of the Yeast has been extensively studied in the literature [16] and its network structure is commonly known. Yeast has 4,441 genes and 12,873 interactions and is referred as the GRN source network. Given the number of sensor nodes ($|V_w|$), a sub-network $G'_g = (V'_g, E'_g)$ needs to be extracted from the GRN source network $G_g = (V_g, E_g)$ such that $|V'_g| = |V_w|$, $V'_g \subseteq V_g$, and $E'_g \subseteq E_g$.

We used GeneNetWeaver [20] software to extract a sub-network which preserves the structural properties of the source network. The procedure to extract the GRN template follows a greedy algorithm which starts with a random selection of a seed node. The remaining nodes in the sub-network are iteratively chosen from the neighboring set of the selected nodes, such that the modularity of the sub-network is maximized. Recall that modularity is a measure of the structure of a network. The higher the modularity of the sub-network, the more it preserves the structural properties of the source network [16]. For example, let us consider the deployment of a bio-inspired WSN with 20 sensor nodes. As mentioned above, we first extract a sub-network from the Yeast GRN with the help of GeneNetWeaver software. The extracted sub-network is illustrated in Figure 1. Now in order to construct the bio-inspired WSN, we need to deploy the sensor nodes according to the topology of the extracted sub-network.

3.2 Determining Locations of Sensor Nodes in Bio-inspired WSNs

Let us assume the bio-inspired WSN with the given topology needs to be deployed in a grid size of $m \times m$ units. The node placement must satisfy the two constraints: 1) sensor nodes having an edge in G_w must be in the transmission range of each other; and 2) nodes not having an edge in G_w must not be in the transmission range of each other.

We formulate the problem of node placement using virtual forces. The use of virtual forces for determining the locations of the sensor nodes has been primarily used in scenarios wherein the physical topology of the WSN is built based on the goals of network deployment, like maximum coverage [23]. The nodes move in the network trying to balance the forces of attraction and repulsion with other nodes. The attractive and repulsive forces between the sensor nodes are determined by the above goals. However, in our approach we already have the desired physical topology of the WSNs and need to determine the locations of the sensor nodes based on it.

Assigning locations to the sensor nodes to satisfy the constraints is based on the principle that if two nodes have an edge, they attract each other. Whereas, if there is no edge between two nodes, they repulse each other. Let the total attractive and repulsive forces are represented as $\overrightarrow{F_a}$ and $\overrightarrow{F_r}$, respectively. An

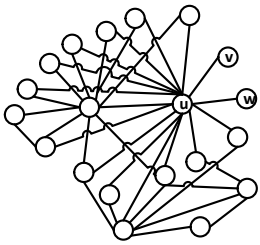


Fig. 1. Extracted Yeast Sub-network

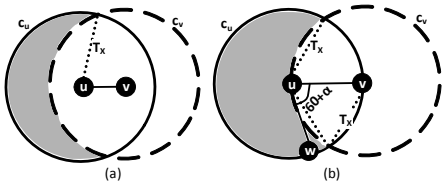


Fig. 2. Deployment of Sensor Nodes

optimal solution for assigning location to each node is achieved when the attractive and repulsive forces due to all the nodes cancel each other, i.e., $\overrightarrow{F_a} - \overrightarrow{F_r} = 0$. However, we claim that finding the optimal solution for the bio-inspired WSN graph is an intractable problem. In Figure 1, node u is the one with the highest degree and all of its neighbors do not share an edge with each other. Hence, the neighbors of u must be in the transmission range of u while they must not be in the transmission range of each other. In such scenarios, if the number of neighboring nodes is ≥ 5, a solution to the location assignment problem does not exist with the given constraints.

Theorem 1. *The maximum number of nodes that can be placed in transmission range of u but not in transmission range of each other is* 5.

Proof. We begin by placing node (u) on a 2-dimensional grid. Let the transmission range of the sensor nodes be T_x units, represented by the circle with the sensor node at its center. The transmission range of u is given by the circle C_u, as shown in Figure 2. The nodes in the transmission range of u should be placed within the circle C_u. Let v be the first node must be placed such that the overlap of its transmission range with C_u is minimum. This maximizes the non-overlapping area of C_u, increasing the probability of identifying the location of the remaining nodes, such that they are not in the transmission range of each other. As shown in Figure 2, minimum overlap of transmission range is achieved when v is placed on the circumference of the circle C_u. The next neighboring node (w) must be placed along the circumference in the non-overlapping area of C_u. So, the minimum distance between v and w must be $(T_x + \triangle T_x)$, i.e., the angular separation between the two nodes with u as the center must be $(60^o + \alpha)$. Hence, the maximum number of nodes that can be placed along the circumference without being in the transmission range of each other is given by $\lfloor \frac{360}{60+\alpha} \rfloor = 5$, where α is an infinitely small number. ■

In other words, if the number of neighboring nodes who are not in the transmission range of each other is ≥ 5, the attractive and repulsive forces do not cancel each other and the unbalanced force is given by $\overrightarrow{F_a} - \overrightarrow{F_r}$. The unbalanced force causes displacement of nodes from their optimal positions. Let us assume that the distance between the new position due to the displacement and the optimal

position is given by ϵ. Hence, the goal of the optimization is to minimize ϵ. However, the displacement of the sensor node may cause two nodes having an edge to move out of the transmission range of each other. This is not desired as it would change the extracted topology from the GRN. Furthermore, the displacement may also cause two nodes without an edge to move in the transmission range of each other. However, we can overcome this problem by not choosing to use these additional communication links. Thus, the extracted topology of the GRN is preserved in the bio-inspired WSN.

We formulate the sensor node placement problem as follows with the help of non-linear programming where the goal is to minimize ϵ, such that $\epsilon > 0$. Let there be $|V_w|$ nodes in the bio-inspired WSN. The location of each node u_i needs to be determined and represented as (x_i, y_i) where $u_i \in V_w$, $0 \leq x_i \leq m$ and $0 \leq y_i \leq m$. We assume two nodes in the network cannot be deployed in close vicinity of each other. Hence, we assume a lower bound (LB) on the distance between two nodes. The non-linear programming is formulated in Equation 1 with four inequalities. We realize that, due to the unbalanced force, the nodes are always in constant motion. However, given by the first and second constraints, if an edge exists between the nodes in G_w, they cannot move outside the transmission range (T_x) of each other and thus cannot move closer than LB distance of each other. Moreover, nodes without an edge between them would repulse each other, such that the distance between the nodes is maximum. The maximum distance between nodes in a $m \times m$ grid is its diameter and is given by $m\sqrt{2}$. However, due to the unbalanced force, the distance between the two nodes becomes $(m\sqrt{2} - \epsilon)$. As we minimize ϵ, the distance between two nodes without an edge approaches $m\sqrt{2}$, given by the third constraint in Equation 1. If node u has two neighbors v and w, with nodes v and w not sharing an edge between them. The unbalanced force generated from repulsive force between v and w and their attractive forces with u, causes v and w to move closer to each other. However, they should not move within the transmission range of each other. The fourth constraints ensures that they are not in the transmission range of each other, as $\epsilon > 0$.

$$
\begin{aligned}
&\text{minimize } \epsilon \\
&\text{subject to } \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - T_x \leq 0, \quad \forall (u_i, u_j) \in E_w \\
&\qquad LB - \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq 0 \quad \forall (u_i, u_j) \in E_w \\
&\qquad m\sqrt{2} - \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq \epsilon, \quad \forall (u_i, u_j) \notin E_w \\
&\text{and} \qquad \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - T_x \leq \epsilon, \quad \forall (u_i, u_j) \notin E_w
\end{aligned}
\tag{1}
$$

We use the Sequential Quadratic Programming (SQP) algorithm to estimate the location of the sensors using Equation 1. The sensors can now be deployed at their estimated location using the topology of the extracted sub-network.

4 Evaluation of Bio-inspired WSNs

In this section, we evaluate the robustness of bio-inspired WSNs and compare its performance with k-connected and randomly deployed WSNs. k-connectivity

in the network is a useful metric to provide robustness. Hence, we compare the robustness of the bio-inspired WSNs with k-connected WSNs. First, we carry out a graph-based analysis of the topological properties of the three networks and then substantiate our claims by performing exhaustive experiments.

In order to generate the topology of bio-inspired WSN, we used the Yeast as the GRN source network and extract the sub-networks by using the GeneNetWeaver software [20]. We generated a random geometric graph to represent a randomly deployed WSN. In random geometric graphs an edge exists between two nodes if their distance is less than a threshold. We assume the sensors are deployed in a grid of dimension 100m$\times$100m, with the transmission range of each sensor being 20m. For meaningful comparison between the randomly deployed WSN and bio-inspired WSN, both networks should have same number of nodes and edges. Hence, we generate a random geometric graph with the same number of nodes as the bio-inspired WSN. If the edges in the random WSN is less than the bio-inspired WSN, we iteratively generate another random WSN. When the random WSN has more edges than bio-inspired WSN we randomly remove edges from it to make the number of edges equal to the bio-inspired WSN. While removing edges we do not consider those edges whose removal will make the random WSN disconnected. For generating the k-connected WSN we use the k-UPVCS algorithm proposed in [10] on a random geometric graph with the same number of nodes as the bio-inspired WSN. The parameter k is selected uniformly at random as an input to the algorithm. The lower bound of k is chosen as the 1 and the upper bound is selected as the k-connectivity of the generated random geometric graph.

4.1 Structural Properties of Bio-inspired WSNs

Network performance, such as end to end delay and robustness is determined by the underlying topology represented as a graph. For example, end to end delay in the WSN is proportional to the diameter of the graph. On the other hand, robustness of a WSN can be defined as its capability to reliably deliver the packets from the source to a destination, on occurrence of node or link failures. The reliable delivery of packets depend on multiple factors:

- **Average Length of Shortest Path:** If the length of the average shortest path is high, it implies that the data is delivered to the sink using a higher number of hops. Since at each hop, the packet transmission may fail due to fading of wireless channel and interference, it reduces the likelihood of the packet being delivered to the sink.
- **Average Node Connectivity:** A graph is called k-node-connected or k-connected if there exists atleast k-node disjoint paths between each pairs of nodes. The k-connectivity of a graph represents the worst-case scenario wherein the graph is disconnected if at least k nodes fail. However, the failure of nodes may have minimal impact on connectivity of the network as the size of disconnected component may be small. Thus we use the average node connectivity as a metric to determine the global connectivity and

thr robustness of the network. Average connectivity is the average number of node disjoint paths between any pair of nodes.

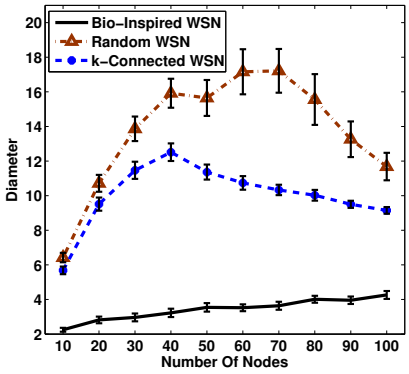
- **Average Edge Connectivity:** A graph is called k-edge-connected if there exist atleast k-edge disjoint paths between each pair of nodes. Thus, k-edge-connectivity can be used to evaluate the robustness of the network on occurrence of link failures. Similar to k-node connectivity, k-edge connectivity does not represent the global edge connectivity of the network. Therefore we consider average edge connectivity to measure the resilience of the network to link failures. It is defined as the average number of the edge disjoint paths between each pair of nodes.
- **Edge Persistence:** Edge persistence denoted as $\pi(G)$, is a metric to measure robustness in WSNs as proposed in [14]. This metric calculates the ratio of the number of failed edges to the number of nodes that become unreachable to the sink. A higher edge persistence ratio implies that a higher number of nodes are reachable on occurrence of link failures. For a graph $G(V, E)$ it is computed as shown in Equation 2, where A is the set of failed edges in G and $\lambda(A)$ is the number of nodes which cannot reach the sink due to the edge failures.

$$\pi(G) = min\left\{\frac{|A|}{\lambda(A)} : A \subseteq E, \lambda(A) > 0\right\} \tag{2}$$

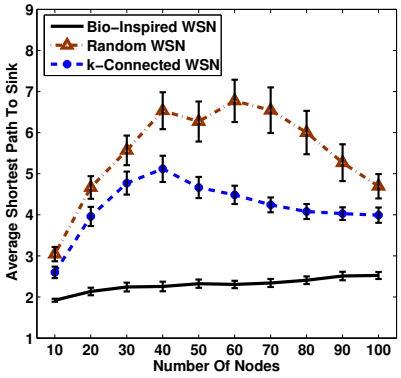
We evaluate the topologies of the three network based on the parameters discussed above. We vary the number of nodes from 10 to 100 in steps of 10. Each data point in the Figure 3 and Figure 4 represents the average data observed in 100 unique topologies of each bio-inspired, k-connected and random WSNs each and is shown with 95% confidence interval.

Figure 3a shows the diameter of the bio-inspired, k-connected and random WSNs. The results show that, since bio-inspired WSNs has lower average diameter than k-connected and random WSNs, they will exhibit lower end to end delay. Since the deployment area of the sensors is fixed, the density of the network increases with the number of sensor nodes. Thus as the density increases, the connectivity in the network improves and the average diameter of the network decreases. In order to compute the average shortest path, we consider the node with highest degree as the sink. In [13], the authors show that selecting nodes with highest degree as sinks leads to one of the best candidates for sink selection and, hence, this solution is used here. Figure 3b shows that the bio-inspired WSNs have lower average shortest path than k-connected and random WSNs, i.e., in bio-inspired WSNs, the probability of failure of the shortest-path is lower than others.

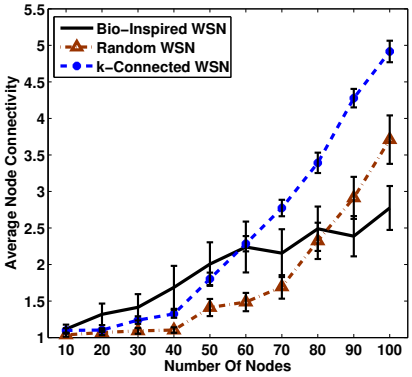
Figure 3c and Figure 3d show the average node connectivity and the average edge connectivity of the three networks, respectively. At a first glance, we may say that k-connected and random WSNs will perform better than bio-inspired WSNs in dense deployments as they have better edge and node connectivity. Node and link failures may create multiple components in the network. Although these components may exhibit high average node and edge connectivity,



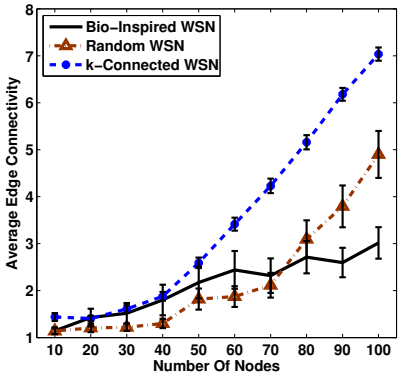
(a) Average Diameter



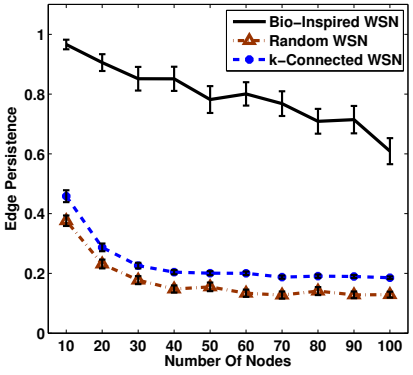
(b) Average Shortest Path to Sink Node



(c) Average Node Connectivity



(d) Average Edge Connectivity



(e) Average Edge Persistence

Fig. 3. Graph Based Analysis of Bio-inspired, k-connected and Random WSNs

the connectivity of the network depends only on the component containing the sink. Furthermore, edge-connectivity of the network is computed by permanently removing the edges from the network. However, in a real scenario, failures of links between sensor nodes may not necessarily be permanent. Temporal link failures may occur due to variations in wireless channel and interference. Hence, a further evaluation of the performance of the three WSNs is required.

Figure 3e shows the edge persistence of the three WSNs using the formulation shown in Equation 2 using the node with the highest degree as the sink. Owing to the power-law distribution of node degrees, random removal of edge creates smaller disconnected components to the sink than k-connected and random WSNs. Thus the bio-inspired WSNs is more robust than both of them.

These factors use different parameters to measure robustness. Even though bio-inspired WSNs perform better than k-connected and random WSNs when considering diameter, average shortest path and edge persistence, we cannot conclusively say that bio-inspired WSNs are more robust as they have lower average node and edge connectivity in dense deployment scenarios. Therefore, we conduct experiments to prove the robustness of bio-inspired WSNs in terms of data delivery ratio.

4.2 Simulation Scenario

We carried out the simulations using Castalia [1], a simulator for WSNs. The parameters for for the simulation for the wireless sensor network are selected based on the 802.15.4PHY and TI CC2420 transceiver specifications and are given in Table 1 [5,2]. The initial energy of each sensor node is selected as the capacity of two AA batteries. As discussed earlier, the bio-inspired WSNs are generated using the Yeast GRN by the GeneNetWeaver software [20], with varying number of nodes from 10 to 100 in steps of 10. We deploy the generated

Table 1. Simulation Scenario

RF Transceiver	Texas Instrument CC2420	Data Rate	250Kbps
Carrier Frequency	2.4 GHz	Total simulation time	10800s
Radio Propagation Model	Two Ray Path Loss Model	Data Generation Rate	Uniform (100ms,1s)
Modulation Scheme	PSK	Initial Energy	18720 Joules
Bandwidth	20MHz	Transmission Power Level	0dBm
Noise Floor	−100dBm	Power consumed in transmit mode	57.42mW
Receiver Sensitivity	−95.0dBm	Power consumed in receive mode	62mW
MAC Protocol	T-MAC	Power consumed in sleep mode	1.4mW

network topology on the grid using the formulation shown in Section 3.2. The k-connected WSNs and random WSNs are generated as described earlier. In all the WSNs, each sensor node generates data with the inter-arrival time of data being uniformly distributed between 100ms and 1s and forwards it to the sink. The sink is selected using the same rule as described in Section 4.1. Data from the source sensor node is forwarded to the sink along the shortest path.

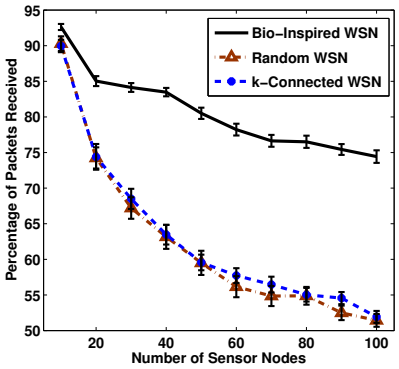
Each data point in the figures represents the average data observed in 100 unique topologies of Yeast based bio-inspired WSNs, k-connected WSNs and random WSNs each and is shown with 95% confidence interval. The simulation time is kept constant at 3 hrs.

4.3 Experimental Results

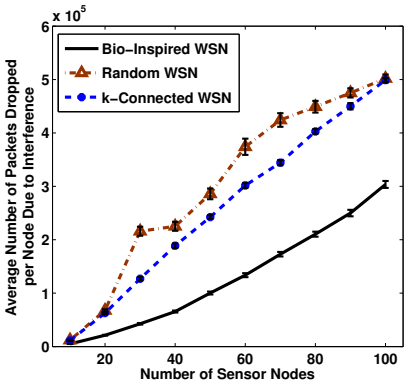
We compare the performance of the three generated WSN topologies using the percentage of packet delivered to the sink, latency and energy consumed by the sensor nodes as parameters without node failures. Furthermore, to evaluate the robustness of the given topologies we introduce node failures in the network to determine their capability to recover from permanent removal of communication links in the network.

Data Delivery. Figure 4a shows the average percentage of total packets delivered to the sink in bio-inspired, k-connected and random WSNs. As the density of sensor nodes in the network increases, more packets are dropped due to higher interference. Figure 4b shows the average number of control and data packets, dropped at the MAC layer due to interference for a given WSN. As the number of packets dropped due to interference increases with the number of sensor nodes, less packets are delivered to the sink. Although Figure 3c and Figure 3d show that, in high density deployments, k-connected and random WSNs should perform better than bio-inspired WSN, the bio-inspired WSN delivers more packets than k-connected and random WSN. In bio-inspired WSN the packets are delivered using fewer hops, hence, they are less susceptible to variations of wireless channel and interference. Thus, due to the topology of the bio-inspired WSN, it delivers more packets than k-connected and random WSNs. On the other hand, the percentage of packets delivered in k-connected WSN is similar to random WSN even though the k-connected WSN has a slightly lower number of packets dropped due to interference. It was observed that since the number of packets dropped due to interference includes a large number of control messages, the actual difference in the number of dropped data packets is low. Hence, they have similar percentage of data packets delivered to the sink in Figure 4a.

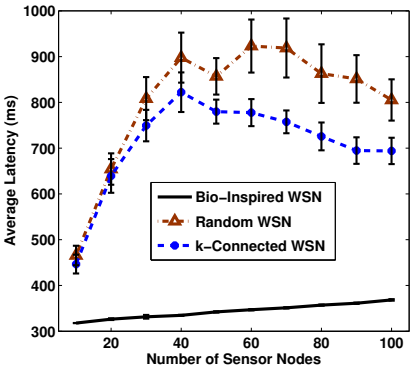
Latency. Figure 4c shows that the latency in bio-inspired WSN is lower than k-connected and random WSNs. As shown in Figure 3a and Figure 3b, the topology based on Yeast sub-network have a smaller diameter and average shortest path. Hence, the data is delivered to the sink using a fewer number of hops than k-connected and random WSNs, resulting in lower latency.



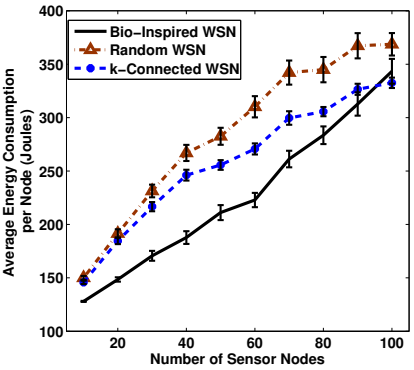
(a) Percentage of packets delivered



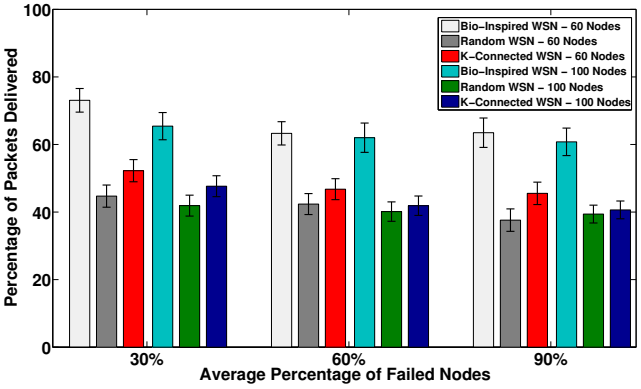
(b) Average number of dropped packets due to interference



(c) Average latency in data delivery



(d) Average energy consumption at a sensor node



(e) Percentage of Packets Delivered with Varying Node Failure Rate

Fig. 4. Simulation Results for Bio-inspired, k-connected and Random WSNs

Energy Consumption. Figure 4d shows that the average energy consumption of sensor nodes in the bio-inspired WSN is smaller than k-connected and random WSNs. Due to lower packet loss in the bio-inspired WSN, there are fewer number of packet re-transmissions at the MAC layer resulting in lower energy consumption. Furthermore in bio-inspired WSNs, due to lower average shortest path, fewer number of intermediate sensor nodes are used to deliver the data to the sink, thus resulting in lower energy consumption.

Data Delivery with Node Failure. We want to evaluate the performance of bio-inspired WSNs, k-connected WSNs and random WSNs on occurrence of node failure. We assume that the failure time between the nodes follows an exponential distribution. The performance of the three networks is compared under varying conditions of node failures. Specifically, we consider three scenarios, wherein the expected rate of node failure is selected such that the expected number of nodes failing during the simulation time is 30%, 60% and 90% of the total number of nodes. In the simulations, we assume a node stops generating data on failure. When a node fails, it informs its neighbors which in turn try to find an alternate path to the sink. We compare the performance of the three networks with 60 and 100 nodes.

Figure 4e shows the percentage of packets delivered to the sink with varying number of failed sensor nodes. For example, when on an average 30% of the 60 and 100 nodes fail during simulation, the bio-inspired WSN delivers more packets than k-connected and random WSNs. As the degree distribution of sensor nodes in bio-inspired WSNs follows power-law distribution, link-failures due to node failures prevents only a smaller subset of sensor nodes from reaching the sink when compared with k-connected and random WSNs. This causes a higher number of packets being delivered to the sink in bio-inspired WSNs even though the average node and edge connectivity of k-connected and random WSNs was higher as shown in Figure 3c and Figure 3d, respectively. Similar behavior is seen for 60% and 90% of node failures. However, in the simulation, since nodes in the disconnected components can also fail and hence stop generating packets, we cannot compare the results across different node failure rates. The results conclusively show that the bio-inspired WSNs are more robust, energy efficient and had lower latency than k-connected and random WSNs.

5 Conclusion and Future Work

In this paper, we investigated the use of the gene regulatory networks (WSNs) topology for deploying wireless sensor networks. We termed such a network as bio-inspired WSN and hypothesized that it is robust. To prove its robustness, we evaluated the structural properties of bio-inspired WSNs and compared it with k-connected and random WSNs. Based on the structural analysis we concluded that bio-inspired network would have lower latency and high edge persistence. Furthermore, through comprehensive experiments we show that the bio-inspired WSNs are robust, have lower latency and are energy efficient. The results in the

paper establish the feasibility of using GRNs based topology to deploy fault-tolerant and robust WSNs. In this paper, we deployed the WSNs using the topology of the Yeast sub-network. However, a bio-inspired WSNs can be deployed more flexibility, if we map the GRN topology on an already deployed WSNs. This would involve activating sensor nodes and links such that their interactions follow the GRN topology. We will investigate the feasibility of such a deployment and design more robust routing protocols based on it as a future work.

Acknowledgement. This research is partially supported by the NSF grants CNS-104965, IIS-1064460, IIP-1242521 and CNS-1150192.

References

1. Castalia, a simulator for wsns, http://castalia.npc.nicta.com.au/index.php
2. Cc2420 - single-chip 2.4 ghz ieee 802.15.4 compliant and zigbee ready rf transceiver, http://www.ti.com/product/cc2420
3. Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans), http://standards.ieee.org/about/get/802/802.15.html
4. Bonabeau, A.B.E.: Scale free networks. Scientific American, 60–69 (2003)
5. Bougard, B., Catthoor, F., Daly, D.C., Chandrakasan, A., Dehaene, W.: Energy efficiency of the ieee 802.15.4 standard in dense wireless microsensor networks: Modeling and improvement perspectives. In: Conf. on Design, Automation and Test in Europe, pp. 196–201 (2005)
6. Das, S., Koduru, P., Cai, X., Welch, S., Sarangan, V.: The gene regulatory network: an application to optimal coverage in sensor networks. In: 10th Annual Conf. on Genetic and Evolutionary Computation, pp. 1461–1468 (2008)
7. Dressler, F., Akan, O.: Bio-inspired networking: from theory to practice. IEEE Communications Magazine 48(11), 176–183 (2010)
8. Ghosh, P., Mayo, M., Chaitankar, V., Habib, T., Perkins, E., Das, S.: Principles of genomic robustness inspire fault-tolerant wsn topologies: A network science based case study. In: PERCOM 2011 Workshops, pp. 160–165 (2011)
9. Guelzim, N., Bottani, S., Bourgine, P., Kepes, F.: Topological and causal structure of the yeast transcriptional regulatory network. Nature Genetics 31(1), 60–63 (2002)
10. Hajiaghayi, M., Immorlica, N., Mirrokni, V.: Power optimization in fault-tolerant topology control algorithms for wireless multi-hop networks. IEEE/ACM Transactions on Networking 15(6), 1345–1358 (2007)
11. Hou, J.C., Li, N., Stojmenovi, I.: Topology Construction and Maintenance in Wireless Sensor Networks, pp. 311–341. John Wiley & Sons, Inc. (2005)
12. de Jong, H.: Modeling and simulation of genetic regulatory systems: a literature review. Journal of Computational Biology 9(1), 67–103 (2002)
13. Kamapantula, B.K., Abdelzaher, A., Ghosh, P., Mayo, M., Perkins, E., Das, S.K.: Performance of wireless sensor topologies inspired by e. coli genetic networks. In: IEEE Int'l Workshop on PerSeNS, pp. 308–313 (2012)
14. Laszka, A., Buttyn, L., Szeszlr, D.: Designing robust network topologies for wireless sensor networks in adversarial environments. Pervasive and Mobile Computing (2012)

15. Macneil, L.T., Walhout, A.J.: Gene regulatory networks and the role of robustness and stochasticity in the control of gene expression. Genome Research (March 2011)
16. Marbach, D.: Evolutionary reverse engineering of gene networks. Ph.D. thesis, Lausanne (2009), http://library.epfl.ch/theses/?nr=4503
17. Markham, A., Trigoni, N.: Discrete gene regulatory networks (dgrns): A novel approach to configuring sensor networks. In: INFOCOM, pp. 1–9 (2010)
18. Prill, R.J., Iglesias, P.A., Levchenko, A.: Dynamic properties of network motifs contribute to biological network organization. Public Library of Science (2005)
19. Ren, H., Meng, M.H.: Biologically inspired approaches for wireless sensor networks. In: Int'l Conf. on Mechatronics and Automation, pp. 762–768 (June 2006)
20. Schaffter, T., Marbach, D., Floreano, D.: Genenetweaver: In silico benchmark generation and performance profiling of network inference methods. Bioinformatics 27, 2263–2270 (2011)
21. Thomas, S., Alvis, B.: Current approaches to gene regulatory network modelling. BioMed Central (2007)
22. Wightman, P., Labrador, M.: A3: A topology construction algorithm for wireless sensor networks. In: IEEE GLOBECOM, pp. 1–6 (December 2008)
23. Yu, X., Huang, W., Lan, J., Qian, X.: A novel virtual force approach for node deployment in wireless sensor network, pp. 359–363 (2012)

Cellular Pulse Switching: An Architecture for Event Sensing and Localization in Sensor Networks

Qiong Huo, Bo Dong, and Subir Biswas

Michigan State University, East Lansing MI 48824, USA
{huoqiong,dongbo,sbiswas}@msu.edu

Abstract. This paper presents a novel energy-efficient pulse switching protocol for ultra-light-weight wireless cellular network applications. The key idea of pulse switching is to abstract a single pulse, as opposed to multi-bit packets, as the information exchange mechanism. Event monitoring with conventional packet transport can be prohibitively energy-inefficient due to the communication, processing, and buffering overheads of the large number of bits within a packet's data, header, and preambles. Pulse switching, on the other hand, is shown to be sufficient for event monitoring applications that require binary sensing. This paper presents a joint MAC and Routing architecture for pulse switching with a novel cellular event localization framework. Through analytical modeling and simulation experiments, it is shown that pulse switching can be an effective means for event based networking, which can potentially replace packet transport when the information is binary in nature.

Keywords: Ultra Wide Band Impulse Radio, Pulse Switching, Cellular Sensor Networks, Event Monitoring, Pulse Routing, Energy-efficiency.

1 Introduction

The objective of this paper is to develop a novel pulse switching framework[1] that uses a cellular network and localization structure to implement packet-less event communication. The key idea in this paper is to introduce a new abstraction of sensor event area cells and to use it for packet-less pulse switching for event monitoring. An example application is intrusion detection in which while surveying a building it is often sufficient for a sensor to generate an event to indicate an intrusion in its vicinity. Sending an event to a sink ideally requires a single bit information (i.e. binary) transport for which the traditional mode of packet communication can be highly energy inefficient. Such inefficiency stems from communication, processing, and buffering overheads of a large number of bits [1] in each packet. Another example application of the proposed mechanism for binary event sensing is structural health monitoring.

[1] This work was partially supported by a grant (NeTS 0915851) from National Science Foundation.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 208–224, 2013.

In the proposed paradigm, such an event can be coded as a single pulse, which is then transported multi-hop towards a sink while preserving the event's localization information in terms of its originating event area cell. The resulting operational lightness, leveraged via zero collision, zero buffering, no addressing, no packet processing, and ultra-low communication and energy budgets makes the protocol applicable for severely resource-constrained sensor devices.

The primary challenges for pulse switching in a cellular network are how to: 1) transport cellular localization information using a single pulse, and 2) route a pulse multi-hop. These problems are architecturally solved by integrating a pulse's (i.e. event's) cell of origin within the MAC-routing protocol syntaxes. Specifically, by observing the time of arrival of a pulse with respect to the MAC-routing frame, a sink can resolve the corresponding event location with a pre-set resolution. The problem of multi-hop pulse routing is solved by introducing a novel concept of cellular event area combined with synchronized pulse frames.

The contributions of this paper are: 1) development of a sensor-cell based event localization architecture, 2) integration of such localization with a pulse-switching protocol paradigm and its associated MAC and routing syntaxes for multi-hop operations, 3) development of response mechanism and pulse compression for added network resilience and energy efficiency, 4) analytical modeling for error handling measures, and finally 5) evaluation of the above components for event monitoring in a cellular sensor network.

Note that the proposed architecture is targeted mainly for small sensor networks with few tens of sensors distributed within a restricted geographical area. While not being scalable well for very large networks, the protocol can enable event monitoring for specific applications such as intrusion detection and structural health monitoring for aircraft wings and bridges [2].

2 Related Work

Very few efforts exist in the literature on packet-less networking using pulse communication. The paper in [3] reduces preamble and header overheads of packet communication by aggregating payloads from multiple short packets into a single large packet that is routed to a sink. While reducing the energy cost, aggregation still requires the inherent packet overheads. The paper in [4] proposes a binary sensing model in which each sensor returns only one-bit information regarding a target's presence or absence within its sensing range. Although this binary sensing saves energy to some degree, the approach in [4] too uses a packet abstraction. The objective of our work is to fully replace packets by routable pulses.

The authors in [5] develop models for energy and delay bounds for bit (i.e. packet based) and pulse communications in single hop networks. The main results in [5] are that the worst case energy performance of pulse communication can be substantially better than that of packet based communication, although with a possibly worse delay performance. A notable limitation is that the paper does not provide mechanisms for scaling these results for multi-hop networks. Also, no MAC and routing protocol details are provided. This limitation is

addressed in our work through the design of a MAC-routing framework that can implement multi-hop pulse switching with sensor-cell based event localization.

In our previous work in [6], the concept of multi-hop pulse switching was first introduced for binary event detection and tracking applications. The paper presented pulse switching in the presence of hop-angular event localization. Although it offered a fundamental conceptual framework for pulse switching, its hop-angular localization abstraction relied on an assumption of stationary nature of the wireless transmission and propagation properties. This assumption may not hold in certain applications due to various kinds of time-varying shadowing and fading phenomena. This paper adopts a novel sensor-cell based architecture that relies on a pre-set sensor cell structure for localization, which is not affected by non-stationary wireless transmission properties. In doing so, it achieves a network-wide uniform spatial resolution of event detection which was not possible in the approach used in [6]. Furthermore, a response mechanism is introduced in this paper for dealing with pulse losses and network faults.

Energy-efficiency in cellular networks has been a recent research interest in the context of green communication. The paper in [7] develops energy-efficiency metrics and mechanisms for dynamic operation of cellular base stations in order to provide energy savings. With real data traces, [7] derives a first-order approximation of the percentage of power savings by turning off base stations during low traffic periods while maintaining necessary coverage. This and most other cellular work in the literature, however, are targeted towards phone and data networks, thus rendering the approaches unusable for sensor networks and more distinctly for packet-less pulse switching. This paper merges the cellular abstraction with pulse switching for energy-efficient event sensing in sensor networks.

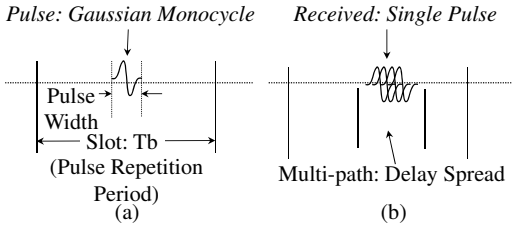


Fig. 1. Pulse switching with un-modulated UWB impulses

3 Pulse Realization Using UWB Impulse Radio

The ability to transmit and receive a single pulse without per-pulse synchronization overhead is a key requirement for pulse switching. Ultra Wide Band (UWB) [8] [9] Impulse Radio (IR) technology can be practically [10] used because of its support of single pulse transmission and reception.

UWB Slotting. Fig.1 depicts a UWB implementation of pulse structure. A typical UWB pulse width is $1ns$, and the pulse repetition period Tb (slot size) is $1000ns$ [8]. This large difference between pulse width and slot size minimizes the overlapping probability between pulses in adjacent slots in the presence of multi-path delay.

Modulation and Synchronization. Since each individual pulse carries information about an event by itself, unlike in packet based UWB-IR, no Time Hopping Sequence [9] based Pulse Position Modulation is needed in this architecture. Additionally, since the synchronization is achieved through a sink (see Section 5.1), no per-pulse synchronization preambles are needed. It helps avoiding huge synchronization overheads of up to 600 pulse repletion periods [11] as typically needed for packet preambles.

Energy Budget. The all-digital baseband operation of the UWB-IR enables it to be implemented in low-cost CMOS logic [10]. For example, with $0.18m$ CMOS based UWB, power consumptions of $4nJ$ for each pulse transmission (average $4mW$ with $1000ns$ pulse repetition period), $8nJ$ for each pulse reception (average $8mW$) and idling consumption of $8mW$ can be typical.

4 Sensor Cells and Event Localization

4.1 Pulse as Protocol Data Unit

Upon detection of an event, a sensor node generates a single (RF) pulse (e.g. UWB-IR [10]) which needs to be transported multi-hop to a sink. A pulse is able to represent: a) the very occurrence of the event, and b) its location of origin. An event can result in multiple pulses generated by all sensors detecting the event, and all such pulses need to be transported to the sink. With the localization information for each such pulse, several application level conclusions can be derived at the sink by correlating multiple event pulses.

4.2 Cellular Event Localization

As shown in Fig.2, a network may contain arbitrarily distributed sensors that detect events and send corresponding pulses to a sink. The figure also shows geographically overlaid sensor-cells (i.e. arbitrarily shaped and placed) that are pre-defined and individually represent event areas, and have unique *Cell-IDs*.

Each sensor is pre-programmed with the *Cell-ID* of its own cell and with a list of *Cell-IDs* of all the geographically neighboring sensor cells.

An event is localized at the spatial resolution of a sensor-cell. In a bridge monitoring scenario, for example, when one or multiple sensors

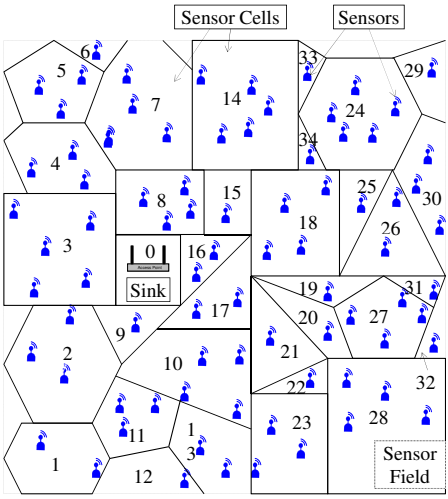


Fig. 2. Network model with arbitrarily defined sensor-cells

within a cell detect a structural fatigue or failure, the sink node will eventually be able to localize the event in terms of the cell that those detecting sensors belong to. If sensors from multiple cells detect such an event, the sink can resolve the event spanning multiple such cells. The sensors are not individually addressed, and therefore no per-sensor addressing is necessary at the MAC or routing layers. An event is always identified by the *Cell-ID* of its cell of origin. The energy non-constrained sink is assumed to make high-power transmissions with full network coverage for frame-synchronizing the sensors. Event localization resolution can be adjusted by changing the size and shape of sensor cells.

5 Cellular Pulse Switching

5.1 Joint MAC-Routing Frame Structure

Nodes in the proposed system are frame-by-frame time synchronized by the sink. They maintain MAC-Routing frames (see Fig.3), in which each slot is used for sending a single pulse. The slot includes a guard time to accommodate the cumulative clock-drift during a frame and the maximum cell-to-cell propagation delay. Due to UWB-IR's short frame size (s), the clock-drift can be very small.

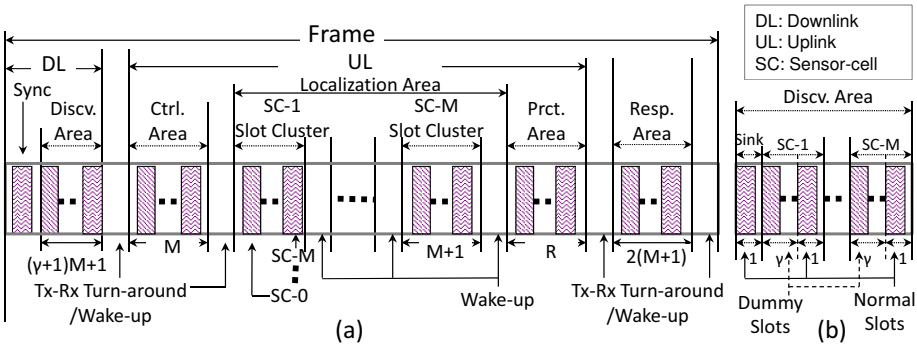


Fig. 3. MAC-Routing frame for multi-hop cellular pulse switching

As shown in Fig.3, each frame contains a downlink part, an uplink part, and a response area in which pulses can be sent either downlink or uplink. The downlink part of the frame includes a synchronization slot where the sink periodically transmits a full power pulse to frame-synchronize all nodes in the network. The following *DiscoveryArea* in the frame is designed for adaptive route discovery, as described in Section 5.2. In the uplink part, there are three components: a) *ControlArea* for energy management, b) *LocalizationArea* for representing an event's location of origin and the routing table information, and

c) *ProtectionArea* for false positive error protection. The *ResponseArea* is used for implementing pulse transmission reliability.

5.2 Route Discovery

This is a continuous background process that creates and maintains the routing table in each sensor in terms of the next-hop *Cell-IDs*. In order for routing to work, each sensor needs to be aware of its own *Cell-ID* and those of all its geographically neighboring cells. A sensor should also have wireless reachability to at least another sensor in one of its neighboring sensor cells. Note that the proposed architecture and the route discovery process do not assume any specific shape of a sensor cell and node's transmission coverage area.

The *Discovery Area* in the MAC-routing frame contains $((\gamma + 1)M + 1)$ slots, where M is the number of sensor-cells in the network and γ is the transceiver turnaround time (between transmission and reception modes) as a multiple of slot duration. As shown in Fig.3b, the first slot is allocated to the sink, and the rest of the slots are allocated to all M sensor-cells in contiguous slot clusters. Within a cluster, the first γ slots are designated as dummy slots during which transceiver turn-around happens, and the $(\gamma + 1)^{th}$ slot is designated as a normal slot, during which discovery related transmission happens as follows. Once in every discovery cycle, the sink initiates a routing discovery phase by sending a regular power (as opposed to a full power synchronization pulse) discovery pulse in the first slot of the *Discovery Area*. Upon receiving that pulse, all sensors within the sink's neighboring sensor cells register the reception time and the *Cell-ID* (retrieved from the temporal location of the pulse) of the sensor cell (which is the sink in this case) from which the discovery pulse was received. The sensors then forward the pulse to their neighbors in the normal slots of their corresponding clusters in the *Discovery Area* of the next frame. This time-stamp and forwarding process continues till the discovery pulse is flooded throughout the entire network. The pseudo-code for such a process is shown in Algorithm 1. At the end of each discovery phase, each node develops a routing table which is a list of time-stamps and the corresponding *Cell-IDs* of its neighboring cells, indicating which sensor cells forwarded the discovery pulse at what time. In the absence of queuing delay, the entry with the earliest time-stamp would indicate the shortest hop next-hop sensor cell to reach the sink. A node may have one or multiple shortest hop next-hop sensor cells depending on the shape, size, and relative locations of the cells with respect to the sink's location.

5.3 Pulse Forwarding in the Localization Area

The *Localization Area* of the uplink part of the frame is key to pulse forwarding. In a network with M sensor-cells, this area contains M slot-clusters, where each such cluster contains $(M+1)$ individual slots. Each slot-cluster corresponds to a specific *Cell-ID* which represents an event's cell of origin. In a given slot-cluster, each slot corresponds to a specific *Cell-ID* which represents the next-hop cell for a transmitted pulse within the slot cluster. The first slot in each cluster

Algorithm 1. Route Discovery

Discv(i): A discovery pulse forwarded by cell i, where a pulse in the normal slot of the i^{th} cluster of the Discovery Area of a frame ($1 \leq i \leq M$);
NB(i)=($nb_1, ..., nb_N$): Neighbor cells of cell i, $nb_j \neq i$,$1 \leq j \leq N$, N is the max cnt of cell i's neighbor cells;
L(i)=((t_1, id_1),...,(t_R, id_R)): A received discovery pulse list at a node in cell i; (t_j, id_j) represents a reception time-stamp and the corresponding Cell-ID of cell i's neighbor cell,$id_j \neq i, 1 \leq j \leq R, R \geq 0$;
TableSort(L(i)): Sort L(i) in terms of time-stamp;
RouteTable(i): Routing table of cell i destined to the sink;
while $t == k * F_{discv}, k \geq 0$, *$F_{discv}$ is discovery period* **do**
    sink sends a Discv(0)
end
if *a node in cell i receives a Discv(n) at frame t* **then**
    if $n \in NB(i)$ *//sender cell of Discv(n) is the neighbor of cell i* **then**
        inserts the time-stamp t and Cell-ID n to L(i); will forward Discv(i) at frame t+1;
    end
    else
        discards Discv(n);
    end
end
if $t == k * F_{discv} - 1, k \geq 0$*//the end of each discovery phase* **then**
    RouteTable(i)=TableSort(L(i)); Empty L(i);//establish or update the routing table
end

of *Localization Area* represents the sink. It is important to note that the slot cluster for a pulse remains unchanged during the entire forwarding of the pulse from the sensor of origin all the way to the sink node. This way, the localization information of the pulse (i.e. of the corresponding event) is preserved till the pulse arrives at the sink. What changes in a hop-by-hop basis is the specific slot within the slot-cluster depending on the specific next hop sensor-cell. This way, when the pulse arrives at the sink, it is still a part of the unchanged slot cluster, which indicates the pulse's cell of origin. Pulse forwarding decisions are made based on a sensor's routing table. The routing table maintains a sorted list of next-hop cells based on the hop-counts of the corresponding resulting routes (see Section 5.2). For routing with no diversity, a node chooses the best next-hop

Algorithm 2. Pulse Forwarding

F(i,δ): A node in cell i ($1 \leq i \leq M$) gets the top δ next-hop Cell-IDs ($k_1, ..., k_\delta$) from the routing table RouteTable(i), where δ is the route diversity, $k_j \neq i, 1 \leq k_j \leq M, 1 \leq j \leq \delta$;
Event(i,($k_1, ..., k_\delta$)): An event pulse originated from cell i and forwarded to cells of Cell-IDs ($k_1, ..., k_\delta$), where pulses respectively in the $((k_1 + 1)^{th}, ..., (k_\delta + 1)^{th})$ slots of the i^{th} cluster of the Localization Area of a frame, $k_j \neq i, 1 \leq j \leq \delta$;
if *a node in cell i senses an event at frame t* **then**
    sends an Event(i,F(i,δ)) at frame t; //generates a new event pulse
end
if *a node in cell i receives an Event(j,($k_1, ..., k_\delta$)) at frame t* **then**
    if $i \neq j, i \in (k_1, ..., k_\delta)$ *//cell i is one of next-hops for this received event pulse* **then**
        will forward the Event(j,F(i,δ)) at frame t+1;//the node in cell i updates ($k_1, ..., k_\delta$)
    end
    else
        discard the received event pulse;
    end
end

sensor cell from the routing table and forwards the pulse. With non-zero route diversity (parameterized as δ), a pulse is forwarded to top δ next hop sensor cells from the routing table. Details are provided in Algorithm 2.

Since the above pulse forwarding process is executed on a per-pulse manner, a large number of pulses can be simultaneously and independently routed in different parts of the *Localization Area* of a frame depending on their cells of origin and the respective next-hop cells.

5.4 Response Mechanism

A *Response Mechanism* is mainly for handling pulse loss errors. As shown in Fig.3, the *Response Area* of a frame contains $2(M+1)$ slots, where M is the number of sensor-cells in the network. The first $(M+1)$ slots in the *Response Area* are used for indicating the *Cell-ID* of a sensor cell which receives a pulse. The remaining $(M+1)$ slots are used for indicating the *Cell-ID* of the cell of origin of the event corresponding to the received pulse. The first slot in each $(M+1)$-slot set is allocated to the sink, and the rest M slots are allocated to sensor-cells. It is abstracted as a single-pulse acknowledgement for enhancing one-hop transmission reliability. After receiving a pulse in the *Localization Area* of a frame, a receiver sends a response pulse to the sender node during the *Response Area* at the end the frame (see Fig.3). If the sender does not receive a response pulse, it periodically retransmits the pulse in the following frames during a pre-specified time-out period. See the pseudo-code in Algorithm 3.

5.5 Protocol State Machine

Each sensor node maintains three separate state machines for *Route Discovery*, *Pulse Forwarding* and *Response* processes respectively. A state is defined in a per-frame manner and can be one of *Transmission* (T), *Listening* (L), or *Sleeping* (S). During *Route Discovery* and *Pulse Forwarding* processes, the initial state of each node is L and a state transition is triggered by pulse reception. During *Route Discovery* process, a node switches from L to T in the corresponding *Discovery Area* normal slot which is assigned to the node's *Cell-ID*. After forwarding the received discovery pulse in that slot the node switches back to L during the upcoming normal slots of the same frame. This way the node can receive discovery pulses from all its neighboring cells. The dummy slots in each slot-cluster of the *Discovery Area* of a frame are utilized for the transceiver turn around between T and L. During *Pulse Forwarding* process, when a node is transmitting a pulse, it keeps T in the *Localization Area* of the current frame. Once pulse forwarding is done, the node switches back to L in the *Localization Area* of the next frame. The state of a node in *Response Area* is complementary to that in *Localization Area* in a frame.The usage of state S for energy saving will be described in Section 6.1.

Algorithm 3. Response Mechanism

Resp(j,i): A response pulse forwarded by cell j for the event originated from cell i, where a pulse in the $(j+1)^{th}$ slot of the first cluster and another pulse in the $(i+1)^{th}$ slot of the second cluster of the Response Area of a frame, $i, j \neq 0, i \neq j, 1 \leq i, j \leq M$;
$L_{tx}(i)$: A transmitted event pulse list at a node in cell i, where each item includes a next-hop Cell-ID and the corresponding Cell-ID of the event origin's cell;
if *a node in cell m forwards an Event(i,*$(k_1, ..., k_\delta)$*) at frame t // δ is the route diversity* **then**
    inserts item (i,$(k_1, ..., k_\delta)$) into $L_{tx}(i)$;
end
if *a node in cell j receives an Event(i,*$(k_1, ..., k_\delta)$*) at frame t* **then**
    if $j \neq i, j \in (k_1, ..., k_\delta)$ *//cell j is one of next-hops for this received event pulse* **then**
        sends a Resp(j,i) at frame t;
    end
    else
        discard the received event pulse;
    end
end
if *a node in cell m receives the Resp(j,i) at frame t* **then**
    if $(j, i) \in L_{tx}(i)$ **then**
        transmission of the Event(i,$(k_1, ..., k_\delta)$) is successful;
    end
    else
        will retransmit the Event(i,$(k_1, ..., k_\delta)$) at the next frame t+1;
    end
end

6 Energy Saving Measures

6.1 Energy Saving via Intra-frame Interface Shut-Down

Protocol syntaxes are added for nodes to be able to selectively turn their RF interfaces off during appropriate parts of the frame. As for transmissions in Pulse Forwarding process, a node needs to be awake only during the slot clusters (of the *Localization Area*) at which it needs to transmit. During the other slot clusters, the node can simply keep the interface in sleep mode in order to save energy.

However, considering the asynchronous nature of pulse receptions, a node cannot sleep during all the non-transmission slot clusters in the *Localization Area*. To address this, a *Control Area* (see Fig.3) is added in the beginning of the uplink part of the frame. The slots in the *Control Area* of a frame are used for notifying about the impending receptions that are expected during the slot clusters of the *Localization Area* of the frame. When a node plans to send a pulse originally from the sensor-cell of *Cell-ID m* during the m^{th} slot cluster of the *Localization Area* of a frame, it also sends a pulse in the m^{th} slot in the *Control Area* of the same frame. All nodes remain awake during the *Control Area* for receiving the notification about impending transmissions in the m^{th} cluster of the *Localization Area* of the frame. Based on this information, the node can remain awake during the m^{th} slot cluster of the *Localization Area*.

Such intra-frame interface sleep can reduce the idling energy consumption. Additionally, the node's state in the *Control Area* of a frame is same as that in the m^{th} slot cluster of the *Localization Area* of the same frame.

6.2 Pulse Merging

Pulse merging may happen when multiple pulses are transmitted at the same exact slot by multiple nodes, and all such transmissions arrive at a node simultaneously. For example, in Fig.4, the *Cell-ID* of each node is marked at the bottom of the node. A pulse originates at node D in *cell 4* and gets forwarded to two neighboring *cells 2* and *3*. Nodes B and C in those cells independently forward the pulse to *cell 1*. As a result, node A in *cell 1* receives 2 overlapping pulses in the same slot (the 2^{nd} slot of the 4^{th} slot-cluster in the *Localization Area*). Note that instead of a functional collision, a single pulse with merged RF signal is detected by the receiver node. As long as the RF hardware can detect the presence of this overlapped pulse, the routing continues. Such merging could also take place when multiple pulses from the same sensor cell are originated in the same frame. In such scenarios, pulse merging provides inherent in-network aggregation for events originated from the same cells. Also note that pulse merging can occur at any stage of an end-to-end route, including at the sink.

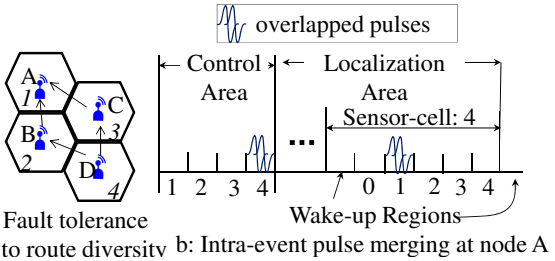


Fig. 4. Pulse merging due to overlapped pulse reception

Pulse merging can increase the effective transmission power, thus undesirably extending RF transmission range. Consequently, a pulse may reach further than it was originally intended to. Since the *Route Discovery*, *Pulse Forwarding* and *Response* processes in the proposed sensor-cell based event localization is completely insensitive to the wireless transmission range, the range extension due to pulse merging does not affect the protocol operation in any ways.

6.3 Spatial Pulse Compression

Although multiple sensors can send pulses for the same event within a sensor cell, ideally only one such pulse from the sensor-cell should be forwarded to the sink to inform about the event. Spatial pulse compression accomplishes this as follows. Upon detecting an event (OR receiving a pulse), a sensor *node-i* defers its pulse transmission for a back-off period that is randomly distributed between 0 to R_{rnd} frames. If another node within the same sensor cell sends a pulse before its deferring period is over, then *node-i* cancels its own transmission after hearing that pulse. Typically, this would lead to very few pulses (often a single pulse) transmission per event per sensor cell. Let k ($1 \leq k \leq N_i^{node}$) be the number of nodes in *cell i* that select the same smallest number and the rest ($N_i^{node} - k$) nodes choose greater numbers, where N_i^{node} is the number of nodes in *cell i*. Thus, k forwarding transmissions will occur within *cell i*. The rest

$(N_i^{node} - k)$ nodes receive those k pulses and stop their own back-off processes. The probability that k nodes select the same smallest number can be expressed as: $p(k, N_i^{node}, R_{rnd}) = \binom{N_i^{node}}{k} \sum_{m=1}^{R_{rnd}-1} (R_{rnd} - m)^{(N_i^{node}-k)} / (R_{rnd})^{N_i^{node}}$.

So the expected number of forwarding transmissions in *cell* i is:

$$N_i^{exp} = \sum_{k=1}^{N_i^{node}} kp(k, N_i^{node}, R_{rnd}) \tag{1}$$

7 Error Analysis

7.1 Protection from False Positive Pulses

If pulses are erroneously detected [12] by a node in the listening state (L) such that a false positive pulse in the *Control Area* corresponds to another false positive pulse in the *Localization Area* (see Fig.3), then a false positive event is produced. Once such a false positive event is generated, it is forwarded all the way to the sink as a regular event, leading to a false positive event reporting. In order to limit false positive events, we develop a novel Frame Protection Code (FPC) mechanism for the proposed cellular pulse switching. An FPC is an M-slot long protection code which is appended at the end of each frame (see Fig.3). For a transmitted event (i.e. a pulse in the i^{th} slot of the *Control Area* and a pulse in a slot of the i^{th} cluster of the *Localization Area*), the transmitter node sends an additional pulse in the i^{th} slot of the *Protection Area*. After the reception of such a protected pulse, the receiver node checks whether the three pulses respectively in the *Control Area*, the *Localization Area* and *Protection Area* match with each other or not. If yes, the pulse would be forwarded by the receiver in a next frame. Otherwise, the receiver simply declares an error.

Let False Positive Pulse Rate (FPPR) be the probability that a false positive pulse is detected due to faulty UWB detection in a given time-slot. False Positive Event Generation Rate (FPEGR) represents the probability of detecting at least one false positive event per frame per node at a given cell. A node in *cell* i is able to receive pulses forwarded by its neighboring *cell* n $(1 \leq i, n \leq M)$. As a result, the vulnerable area for false positive events in the *Control Area* are all slots except the i^{th} slot, and that in the *Localization Area* is the $(i+1)^{th}$ slot of any slot-cluster except the i^{th} cluster, and that in the *Protection Area* are all slots except the i^{th} slot if protection is enabled. In other words, there are (M-1) vulnerable matching sets of slots in the *Control Area*, *Localization Area* and *Protection Area* (with protection) respectively. The FPEGR at a node in *cell* i can be expressed as:

$$FPEGR = \sum_{j=1}^{M-1} (-1)^{j-1} \binom{M-1}{j} p^{\tau j} \tag{2}$$

where p is the FPPR, M is the number of sensor cells in the network, j $(1 \leq j \leq M-1)$ notifies the number of false positive events generated in *cell* i, and τ is the number of pulses in a frame (τ=2 without protection, τ=3 with

protection). Observe Eq.2, the quantity of FPEGR decreases with a higher τ. Based on FPEGR obtained from numerical computation (from Eq.2) and simulation in Section 8.3, it is shown that FPC offers good protection from false positive pulses by reducing the effective FPEGR. Note that FPEGR for any sensor cell is same for the given FPPR.

7.2 Immunity from Pulse Loss

Pulse losses can manifest in the form of un-reported events. Such effects, however, can be alleviated by exploiting the pulse transmission redundancy inherent to pulse routing, e.g. turning off the functionality of pulse compression or increasing the route diversity δ. More importantly, *Response Mechanism* can mitigate the effects of pulse loss errors in that a node would retransmit a pulse for as many times as possible until the successful pulse reception in a next-hop cell.

Let the Pulse Loss Rate (PLR) be the probability that a pulse is lost in a given time slot due to multi-path, channel noise, or various types of interferences. We analyze the case when the route diversity δ is 1. An event pulse in any cell is represented by one pulse in the *Control Area*, one corresponding pulse in the *Localization Area* and another corresponding pulse in the *Protection Area* (if protection is enabled). Loss of any of these τ(τ=2 OR τ=3) pulses in a frame will lead to the loss of the corresponding event. Therefore, the probability of losing an event on any transmission hop (termed as e) is the same as the probability of losing at least one of such τ pulses. This probability can be expressed as: e=1-(1-PLR)$^{\tau}$. The following model expresses the relation between Pulse Loss Rate (PLR) and the corresponding Event Loss Rate (ELR). Let n_i represent a node on the route of an event and p_i represent the probability that the node n_i fails to receive the event due to pulse losses along the route from the event-source to node n_i. Let $\Re_i$ represent the sub-set of *parent nodes* of n_i, such that: 1) each node in $\Re_i$ belongs to a neighbor cell which forwards pulses to n_i, and 2) when a node in $\Re_i$ forwards a pulse, node n_i is able to receive it. We assume the quantity of the times of retransmissions is θ_k which can be very large. So the probability that node n_i fails to receive an event from a parent node n_j for the θ_k times can be written as p_j+[(1-p_j)e]$^{\theta_k}$. Therefore, the probability that node n_i fails to receive the event can be written as:

$$p_i = \prod_{j \in \Re_i} \{p_j + [(1-p_j)(1-(1-PLR)^{\tau})]^{\theta_k}\} \tag{3}$$

For a given topology and PLR, p_i for a node n_i can be iteratively computed starting from a pulse's source node to the nodes in sensor-cells closer to the sink along its route. When n_i corresponds to the sink node, the quantity p_i represents the Event Loss Rate (ELR) for a given PLR. The results from simulation, as shown in Section 8.3, and the calculation according to Eq.3 prove that the response mechanism can indeed mitigate the effects of pulse loss errors on ELR.

8 Performance Evaluation

We developed an event-driven C++ simulator which implements MAC framing and pulse routing using the UWB IR model as presented in Sections 4 to 6. A network with terrain size of $10 \times 10\ m^2$ with evenly distributed sensor nodes and a sink node placed at the lower left corner of the terrain. Sensors are grouped into 115 regular hexagonal cells, each containing 5 nodes in average. This section presents simulation results for scenarios in which a static single event causes the sensor to generate a single event pulse, which is transported to the sink using the proposed routing and localization protocols. The route diversity is set to 1.

8.1 Pulse Transmission Count

Fig.5:a reports the number of pulse transmissions (i.e. with and without compression) in cells that are at specific cell-counts away from the sink. Protection is disabled. An event is generated at a cell which is 15 cell-count away from the sink. The event is then routed to the sink using the presented Cellular Pulse Switching (CPS) protocol. The resulting number of transmitted pulses within the cells along the route is plotted in Fig.5:a. The x-axis corresponds to distance of the cells along the route expressed as cell-count from the sink. Cell-count 1 represents the cell nearest to the sink and the highest cell-count represents the source cell. For comparison purpose, we also present pulse transmission count for the Hop-angular Pulse Switching (HPS) protocol reported in [6] applied to the same network. The x-axis for HPS corresponds to distance of the event areas along the route expressed as hop-counts from the sink. Hop-count 1 represents the hop distance nearest to the sink and the highest hop-count represents the source area for the HPS scenario. Although the event is generated at 10 hop-distances away, the shortest route for CPS is 15-sensor-cell long. It is because the size of a hop area in HPS is larger than that of a cell in CPS. For HPS, the resolution and the sector-constraint are set to 30° and 1 respectively. For CPS and HPS, the number of pulse transmissions in a sensor-cell (or hop area) is equal to the forwarding node count in the corresponding sensor-cell (or hop area) multiplied by the number of pulses in a frame.

As shown in Fig.5:a, the line corresponding to the CPS with no compression case is flat across different cell-counts except for the cell-counts 1 and 15. All five nodes in the sensor-cell corresponding to the cell-counts from 2 to 14 participate in forwarding pulses. So multiplying the node count 5 by 2, which is the number of pulses in a frame τ, results in 10 pulse transmissions. Pulse transmission count in the cell-count 15 is 2 due to the fact that only one node in the source-cell sends a pair of pulses in a frame. The sensor-cell in the cell-count 1 has 4 nodes. That is the reason why the pulse transmission count in the cell-count 1 is 8 which is slightly smaller than 10. With compression applied in CPS, the expected pulse transmission count is reduced to a very small value due to the back-off process (see Section 6.3). With such pulse compression, the pulse transmission count stays around 2 for a sufficiently large back-off counter R_{rnd}.

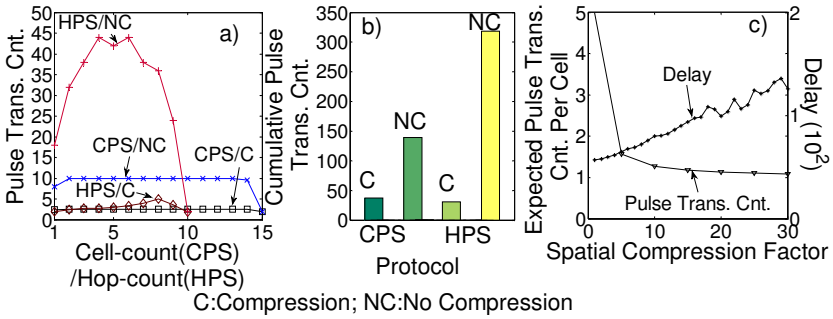


Fig. 5. Number of pulse transmissions and Impacts of spatial compression

Observe that the number of pulse transmissions in HPS maximizes at certain intermediate hop-counts in the case of no compression. In HPS, pulse routing is implemented in the form of constrained flooding with very few nodes participating in forwarding near the source and the sink nodes, and large number of nodes participating in the middle. This explains the pulse count maximization near the hop-count range from 4 to 6. As the compression case in CPS, the expected number of pulse transmissions in HPS is also decreased to a very small value due to the compression mechanism as outlined in Section 6.3. Because of this small spread in the forwarding transmission count, the line corresponding to the case of compression with HPS in Fig.5:a does not show an obvious maximum.

More importantly, the pulse transmission counts in HPS are much larger than that in CPS for up to 10 hop-counts without compression in that the hop area in HPS includes more nodes than a cell in CPS. With compression, the number of pulse transmissions in HPS is slightly more than that in CPS, because the pulse transmissions cannot be reduced much more when approaching to the minimum pulse transmission count (e.g. 2). In this case, the effectiveness of compression on reducing the pulse transmission count overrides that of CPS.

Fig.5:b reports the cumulative pulse transmission count along the entire route in both CPS and HPS for the cases of compression and no compression. The cumulative pulse transmission count of the single event in HPS is 2.23 times greater than that in CPS without compression. With compression applied, the cumulative number of pulse transmissions in HPS is only 1.07 times greater than that in CPS, due to the effectiveness of pulse compression.

These results indicate that CPS can achieve better energy efficiency than HPS by transporting events to the sink with lower number of pulse transmissions. Since HPS was proven to be more energy-efficient than packet switching [6], these results demonstrate that CPS can provide a more energy-efficient solution by replacing packet switching in applications involving binary event sensing. Additionally, since the number of nodes within a cell in CPS is generally smaller than that within a HPS hop-areas, CPS also offers better localization resolution.

8.2 Impacts of Spatial Pulse Compression

The spatial compression factor f_{comp}^{sp} is defined to be equal to the quantity R_{rnd}, which is the maximum value of a random back-off period. For a single event, the performance of pulse compression is impacted by f_{comp}^{sp} in terms of the expected pulse transmission count per cell and the reporting delay for the event. The delay is defined to be the time duration between the generation time of a single event and the arrival time at the sink of the first pulse representing the event. We set the source sensor cell at a distance corresponding to cell-count 15. Protection is disabled. Suppose that all nodes in the intermediate sensor-cells participate in forwarding pulses and the retransmission times θ_k is 1. According to Eq.1, the expected pulse transmission count per cell (containing 5 nodes on average) decreases with increasing f_{comp}^{sp}. This trend is validated in Fig.5:c. Observe that the pulse count saturates when f_{comp}^{sp} increases approximately beyond 10. The delay, however, almost monotonically increases with increasing f_{comp}^{sp}, indicating the effects of increased back-off time. A suitable range of f_{comp}^{sp} [5,10] can be chosen based on a desirable tradeoff between the pulse transmission count and the event reporting delay.

8.3 Pulse Error Analysis

This subsection presents results for effects of pulse error occurrences and the protection mechanism proposed in Section 7.

Impacts of False Positive Errors and Protection. The impacts of FPPR on FPEGR in any sensor-cell, both with and without protection, are shown in Fig.6:a. Observe that the simulation results do exactly match the FPEGR values that are numerically obtained from Eq.2. In Fig.6:a, for both with and without protection, FPEGR is extremely small within the practical range of FPPR (less than 10^{-4}) [13]. It indicates that for both with and without protection, the proposed pulse switching protocol is fairly immune to false positive errors. Also observe that for larger FPPR (i.e. larger than 10^{-4}), it shows less sensitivity to false positive errors with protection compared to that without protection.

Impacts of Pulse Loss Error. Figs.6:b, 6:c and 6:d depict impacts of PLR on ELR for a single event generated at a distance corresponding to cell-count 15. The route diversity δ is set to 1. Observe that the results derived from model (Eq.3) and simulations match quite well. For practical range of PLR (less than 10^{-4}) [13], the ELR for

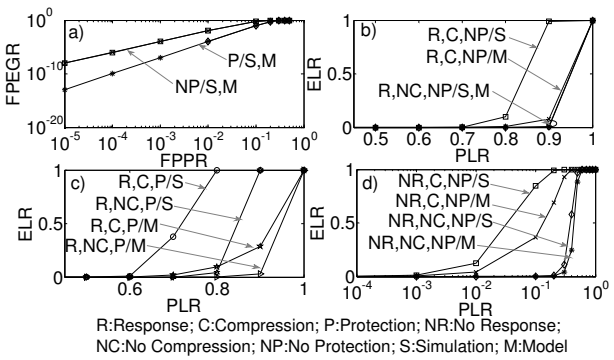


Fig. 6. Impacts of Errors

all cases remains vanishingly small and it is generally insensitive to the value of PLR. When response is disabled, ELR is higher with compression compared to that without compression. It is because lower transmission redundancy leads to a higher event loss probability. Although protection algorithm alleviates the effects of false positive pulse detection, it may sometime aggravate the impacts of UWB pulse losses due to that the value of e is greater with a higher τ (see Section 7). However, *Response Mechanism* offsets the side effects of *Pulse Compression* and *Protection* algorithms. In other words, *Response Mechanism* contributes a lot to decrease the ELR as low as zero when PLR is within the range $[10^{-3}, 0.8]$. So the proposed response mechanism in Section 5.4 strengthens the system reliability against high-rate pulse loss errors.

9 Conclusions

A novel cellular pulse switching protocol for ultra light-weight networking applications has been developed in this paper. A joint MAC-routing architecture for pulse switching with a cellular event localization strategy was presented. The key contribution of the presented architecture is to combine cellular event localization with a pulse switching protocol in a manner that allows a receiver to localize an event by observing the temporal position of a received pulse with respect to a synchronized frame structure. Through analytical models and simulation based experiments, it is shown that the proposed cellular pulse switching architecture can be an effective means for transporting information that is binary in nature with high energy efficiency and strong reliability against errors. Ongoing work on this topic includes the investigation of network-wide fault scenario, and development of implementations of an UWB impulse radio and an ultrasound based pulse communication link.

References

1. Yuanjin, Z., Rui, C., Yong, L.: A new synchronization algorithm for UWB impulse radio communication systems. In: Proceedings of ICCS (2004)
2. Farra, C., Park, G., Allen, D.W., Todd, M.D.: Sensor network paradigms for structural health monitoring (2006)
3. Jain, A., Gruteser, M., Neufeld, M., Grunwald, D.: Benefits of packet aggregation in Ad-Hoc wireless network. Technical Report Technical Report CU-CS-960-03, Department of Computer Science, Boulder, Colorado (2003)
4. Wang, Z., Bulut, E., Szymanski, B.: A distributed target tracking with binary sensor networks. In: IEEE ICC Workshops, pp. 306–310 (2008)
5. Fragouli, C., Orlitsky, A.: Silence is golden and time is money: Power-aware communication for sensor networks. In: Proceedings of Allerton Conference on Comm., Control and Computing (2005)
6. Huo, Q., Biswas, S., Plummer, A.: Ultra wide band impulse switching protocols for event and target tracking applications. In: IEEE SECON, pp. 197–205 (2011)
7. Oh, E., B., K., Liu, X., Niu, Z.: Toward dynamic energy-efficient operation of cellular network infrastructure. IEEE Communications Magazine 49, 56–61 (2011)

8. Haykin, S., Moher, M.: Modern Wireless Communications. Prentice-Hall, Inc., Upper Saddle River (2004)
9. Win, M.Z., Scholtz, R.A.: Impulse radio: How it works. IEEE Communications Letters (2) (1998)
10. Poucke, B., Gyselinckx, B.: Ultra-wideband communication for low-power wireless body area networks. Industrial Embedded Systems Resources Guide (2005)
11. Task group, I..: IEEE 802.15 working group for wireless personal area networks (WPANs) DS-UWB physical layer (2005)
12. Van Trees, H.L.: Detection, Estimation, and Modulation Theory - Part l. John Wiley & Sons
13. Andreyev, Y., Dmitriev, A., Efremova, E., Khilinsky, A., Kuzmin, L.: Qualitative theory of dynamical systems, chaos and contemporary wireless communications. International Journal of Bifurcation and Chaos 15 (2005)

Asynchrony from Synchrony

Yehuda Afek[1] and Eli Gafni[2]

[1] Blavatnik School of Computer Science, Tel Aviv University
[2] Computer Science Department, University of California, Los Angeles

Abstract. A synchronous message passing complete network with an adversary that may purge messages is used to precisely model tasks that are read-write wait-free computable.

In the past, adversaries that reduce the computational power of a system as they purge messages were studied in the context of their ability to foil consensus. This paper considers the other extreme. It characterizes the limits on the power of message-adversary so that it cannot foil the solution of tasks which are read-write wait-free solvable but can foil the solution of any task that is not read-write wait-free solvable. Put another way, we study the weakest message-adversary which allows for solving any task that is solvable wait-free in the read-write model.

A remarkable side-benefit of this characterization is a simple, as simple as can be, derivation of the Herlihy-Shavit condition that equates the wait-free read-write model with a subdivided-simplex. We show how each step in the computation inductively takes a subdivided-simplex and further subdivides it in the simplest way possible, making the characterization of read-write wait-free widely accessible.

Keywords: shared memory, distributed algorithms, wait-free, subdivided simplex, asynchronous computability.

1 Introduction

The seminal FLP result [15] shows that in a message-passing system with the possibility of a single processor failing by stopping (1-resilient system) consensus is not solvable. From a programmer's point of view this allows her to instruct processors to wait to receive messages from all processors but one (we assume the number of processors, n satisfies $n > 2$). That is, she may write her program to progress in rounds where in each round a processor sends a message to all and then waits to receive messages sent at this round from all but one processor. Thus she can view the model of computation as a synchronous system with a message-adversary that is allowed to purge at most one message incoming to each processor in a round. From FLP it follows that such a message-adversary can foil consensus. It is easy to see that if we change the message-adversary so it can purge any but at most $n - 2$ messages in a round then consensus could be reached.

Thus, we are led to consider a message-passing systems on n processors that progress in rounds. In each round all processors send to all. There is a synchronous indication of an end of a round. If p_i did not receive a message from p_j

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 225–239, 2013.

by the end of the round it has to be attributed to an adversary that purged the message. We consider adversaries as a constraint on the message pattern they may purge. This constraint applies uniformly to all rounds, and the patterns the adversary may purge in a round is independent of what it actually purged in previous rounds.

Clearly the weaker the message-adversary, the more tasks the programmer can solve, and vice versa, the more powerful adversary, the more it can purge, the less tasks the programmer can solve.

In this paper we introduce a general definition of message-adversaries and investigate the relations between the adversaries power and the tasks that may be computed in their presence. Specifically we identify the weakest message-adversary with which any task that is wait-free solvable in a read-write shared memory system is solvable in a synchronous network governed by that message-adversary. To recap, we consider a complete synchronous message passing directed network employing a full information algorithm by which each processor in each round sends all its history to all, and by the end of the round collects all the messages that have arrived on its links, i.e., those that were not purged by the adversary. In each round the adversary may remove a subset of the messages that have been sent. The subset removed in one round is independent of the subsets removed in previous rounds. Furthermore, the removal of a message might be asymmetric, a message removed on a link in one direction does not imply the removal of the message in the other direction.

To specify an adversary AD in a round we view a successful message from processor p_i to processor p_j as a directed edge from node i to node j. An adversary AD is a set of directed graphs such that in each round there is a digraph G, $G \in AD$ such that the message sent on each link in G, successfully reaches the other side. Notice, the adversary is restricted not to purge more than it is allowed, but can always leave more successful messages, i.e., the successful messages may induce a graph H such that $\exists G$, $G \in AD$ which is a subgraph of H.

For instance, the message adversary AD-1-*res(incoming)*, corresponding to 1-resilient systems is specified by all the directed graphs on n nodes in which the in-degree of each node is $n - 2$. Thus it takes $(n - 1)^n$ graphs to specify this adversary.

An adversary AD characterizes the asynchronous shared-memory model M, if all the tasks it allows to solve are exactly all the tasks solvable by M. In this paper we restrict our attention to message-adversaries that characterize the class of tasks are wait-free solvable in an asynchronous read-write shared-memory (RWWF) model. We show that the adversary that captures RWWF is the Traversal Path, TP adversary. The TP adversary can remove any set of messages in a round as long as the directed graph induced by the messages it leaves behind is a not necessarily simple, path of messages that goes through all the nodes (i.e., TP contains all possible paths that satisfy the above). Notice that TP satisfies the property that it leaves messages such that for any pair p_i,

p_j of processors there is a directed path of messages left that leads either from p_i to p_j or from p_j to p_i or both. This property will play a key role in the sequel.

There are many adversaries which characterize RWWF, of which TP is the strongest. We show this by presenting an equivalence between a few we are interested in. Two adversaries are equivalent if any task solvable by one is solvable by the other and vice versa. If we have two adversaries AD_1 and AD_2 that characterize RWWF then adversary $AD_1 \cup AD_2$ also characterizes RWWF. That follows because sets from at least one collection must be used infinitely often and that will allow for simulating a write. Thus there exists the most powerful adversary that still allows the system to characterize RWWF. The adversary TP is the most powerful. Any more powerful adversary can purge messages leaving no Traversal Path that goes through all the nodes. It is not hard to see that this means the existence of p_i and p_j where the adversary can prevent forever a chain of messages in any direction between them. Many read-write solvable tasks are not solvable without p_i knowing the input of p_j, or p_j knowing the input of p_i (e.g., snapshot).

A very weak adversary is TOUR (for TOURnament). The adversary TOUR may purge just a single message on each of two antiparallel links, in a round. It will play an instrumental role as we prove that the adversaries TOUR and TP are equivalent.

Adversaries are very simple creatures. They are independent from round to round. They do not involve any notion of "eventually." We can easily see that we do not necessarily need to describe an adversary by its behavior in a single round. In fact, if we take any fixed consecutive number of rounds k and describe what the adversary can do in these round, we get an adversary equivalent to a single round adversary. Indeed, we take adversary TOUR and "spread" it over $k = n(n-1)/2$ rounds where n is the size of the network, to get adversary PAIRS. The adversary PAIRS chooses a pair of nodes *pair* in a round, and each directed edge between the two is an a directed graph in PAIRS for this round. It may delete all messages not sent between processors in *pair*. Of the two messages between *pair* it has to leave unpurged at least one message. Over k rounds it "services" all possible pairs of processors. Notice, PAIRS is still independent between sets of k rounds, and is used as a tool show the equivalence of TOUR and a subdivided simplex.

Why is PAIRS interesting? We consider a geometric depiction of all the possible evolutions of processors' local states with adversary PAIRS. At the beginning each processor has its initial state and we depict this configuration as a simplex (generalization of triangle to higher dimensions) in an appropriate dimension. In round 1, when PAIRS chooses a pair, $\langle p_i, p_j \rangle$ the state of a processor not in $\langle p_i, p_j \rangle$ does not change, it is the initial view plus the information that one round passed. A round in which it received no message. The views (states) of p_i and p_j split - at least one or perhaps both receive a message from each other. To capture this, in the depiction of a copy of the original simplex we split the edge connecting $\langle p_i, p_j \rangle$ into three segments, by planting the views of each receiving a message from the other (see Figure 11.a). Thus the original edge p_i, p_j splits in

the new copy into three consecutive edges: (p_i', p_j''), (p_j'', p_i''), (p_i'', p_j'). Here a prime represents a state in which no message was received, while a double prime corresponds to a state in which a message was received. Notice that the fact that at least one message between $\langle p_i, p_j \rangle$ is not purged precludes the combination of views (p_i', p_j').

Thus, after one round we can depict the collection of possible views as three simplexes (see Figure 1.a): Each of the new segments together with the rest of the views p_l', $l \neq i, j$. A simplex now is a combination of views possible after one round. We have three combinations. If we do this depiction "in place" by planting the new nodes p_j'' and p_i'' on the former edge p_i, p_j and leaving all single primes nodes in their former place without a prime, what we did is we subdivided the simplex (triangulated a triangle). Continuing this process inductively we realize that all the computations of PAIRS after any number of rounds can be depicted as a subdivided simplex.

As we show that PAIRS is equivalent to TOUR which is equivalent to TP and hence allows to solve all the RWWF tasks, and only RWWF tasks. Thus a model that characterizes exactly RWWF corresponds to a subdivided simplex.

This simple result, that some full-information model that characterizes RWWF is a subdivided simplex, which can now be easily presented to undergraduates, has been one of the most important results in distributed computing [21]. Two decades after the discovery of the result it can be explained now in an elementary way. Another derivation in [11] uses a similar thrust of "in-place" subdivision, but the subdivision there is quite complex compared to the one at hand. It is surprising to us that subdivision of edges is "universal" in that it can replace the "colored" analog of the barycenteric subdivision [27].

Independent of the simple topological result our adversaries yield, this line of investigation also leads to a new classification, left to come, of network topologies. Given a network, and a message-adversary that allows for solving RWWF solvable tasks, are there tasks not solvable wait-free in the read write model which it necessarily can solve? Like ABD which solves more than RWWF tasks, characterizing RWWF in a non-complete network may restrict the adversary and consequently increase the power of the system to solve tasks not in RWWF.

For instance, to arrive at an adversary that characterizes exactly RWWF we use the complete network topology. It can be easily seen that if we consider an underlying network topology which is a single simple undirected path (each end point knows it is an end point), an adversary that characterizes RWWF necessarily solves 2-set consensus. Because, a processor can output one of the two end nodes of the path as a traversal path has to start with one of these two nodes and with at least one of them infinitely often. Thus, how does the power of the adversary that characterizes RWWF changes as we move along the spectrum, from complete network to a line network?

1.1 Related Work

The closest we can recall studies of computational power against synchronous message adversary is in the context of Byzantine agreement in the presence

of an adversary that corrupts messages rather than processors [14]. Similarly, many papers touch on the communication requirements to achieve consensus. Indeed, when a communication link is either up in both direction, or down in both directions, we either have connectivity and consequently consensus, or we have disconnected components with no coordination among them.

"Scenarios" of message faults in a round that can foil consensus in such a system were extensively studied in the past [13,29,30,32], and references there in, to name just few studies. By "scenarios" we mean some "special" predicate, e.g., how many send faults, how many receive faults, how many nodes are involved in the faults, etc.

This paper diverges from previous studies in two aspect: First, it studies a different computational model than consensus, and second, it studies a more general definition of adversary.

The current paper shows that TP can (nonblocking) emulate the read-write wait-free model, and vice versa. This result is in contrast to the celebrated ABD emulation [7] that shows how message-passing system can emulate shared-memory. Indeed, this emulation is more powerful than ours. The emulation of each read or write operation is independently wait-free. Our emulation of reads and writes is non-blocking. Some processor will progress. This is unavoidable in our context as the adversary is independent in what it does from round to round. Hence if it is allowed to make p_j unreachable from p_i by a path of messages in one round, it do so in all rounds thus making p_i forever invisible to p_j. Therefore, p_i cannot progress without knowing the output of p_j. Yet, we observe that non-blocking emulation suffices to solve tasks. Consequently with TP we are able to emulate RWWF and no more. In contrast the ABD emulation pays for the wait-freedom of the read write emulation - the ABD emulation that solves RWWF solves much more. In fact, the ABD emulation is powerful enough to solve $n/2$ set consensus. After hearing from $n/2$ processors a processor outputs the minimum value it has heard about.

In [22] Kuhn, Lynch, and Oshman study dynamic networks that are also governed by an adversary. In their case the adversary can erase edges (communication in both directions or none) and they restrict the adversary to various types of eventually connected network [3,4], called T-interval connectivity. While such a network can solve the consensus problem they investigated the complexity of key distributed computing problems such as determining the network size, and computing any function on the ids of the processors (e.g., leader election = consensus).

That the protocol complex of models that solve exactly RWWF task contains an image of a subdivided simplex was first established by Herlihy and Shavit [21]. In [11] this result was shown using the iterated model notion and immediate snapshot tasks. It assumed without proof that the immediate snapshot task is a subdivided simplex.

Paper organization: The paper is organized as follows. In the next section we discuss the model of synchronous dynamic network with message adversary in more detail. We then, as a warm-up, show "procedurally" how our TP adversary

implements Read-Write. In a subsequent section we rely on previous work that shows that the iterated snapshots (IS) model [11,18] solves exactly any task that is RWWF solvable. Then in a declarative manner of tasks implementing tasks, we show TP in complete networks is equivalent to IS. Subsequently, in Section 6, we introduce PAIRS which can be easily seen to be equivalent to TP. We then show in elementary inductive way how PAIRS gives rise to a protocol complex [21] which is precisely a subdivided simplex. In Section 6 we detail the construction for $n = 3$ and in the appendix the construction is detailed for an arbitrary n. Finally we close with conclusions.

2 Model

This paper deals with the relations between two basic models in distributed computing, the asynchronous read-write shared-memory, and the synchronous message passing with message delivery failures. In both there are n processors, $p_1, \ldots p_n$. In the shared memory case we consider the standard model [19] in which all communication between processors is via writing to and reading from shared single-write multi-reader atomic registers.

The network model we assume is synchronous complete network (unless stated otherwise) in which in each round each processor sends its entire history to all its neighbors i.e., all the other nodes in the network. In each round an adversary may purge a subset of the messages sent. All other messages are received by their destination by the end of the round. Depict a message successfully delivered from p_i to p_j in a round as a directed edge from p_i to p_j. The collection of directed edges in one round is called the round communication graph (RCG). The adversary is specified by a property that must be satisfied by any directed graph RCG it can create. The strongly connected (SC) adversary, is the one in which at each round the RCG must be a strongly connected directed graph spanning all the nodes. In the Traversal-Path (TP) adversary, the RCG in each round must contain a *directed* (not necessarily simple) path starting at one node and passing through all the nodes. A dynamic network ruled by an adversary Γ is called Γ-dynamic network.

We extensively use an adversary we call TOUR, defined only with respect to an underlying complete network. Its predicate is that RCG contains a tournament. In other words, of the two messages on a link sent in a round TOUR can purge at most one. For complete networks TOUR and TP are shown equivalent (with respect to the tasks each can solve, as defined in the Introduction).

We consider only adversaries whose behavior is oblivious to the processor ids, i.e., independent of the processors ids. Whether a graph is a valid RCG for the given adversary is invariant to renumbering of nodes.

A task [21] is a distributed computational problem involving n processors. Each participating process starts with a private input value, exchanges information with other participating processes and eventually outputs a value. The task is specified by a relation Δ that associates with every input vector

(one element per participating processor) a set of output vectors that are allowed given this input. See [21] for a more formal definition of a task.

We now describe the meaning of computation in a synchronous dynamic network. An input is an abstract "*my_item*." Computation evolves in synchronous rounds. In the first round each processor sends a message consisting of a pair (my_id, my_item) to all neighbors. Some messages get through some are deleted by the adversary. The messages that make it through have to comply with the predicate Γ that defines the RCGs that the adversary must maintain. In every subsequent round inductively a node sends a message with all its history to all its neighbors. A protocol to solve a task T with an adversary Γ is associated with a number k, and after k rounds, in which in the initial round my_item is instantiated to the senders input, each processor takes its history and maps it to an output of T. The protocol solves T if under the condition that in every round the communication graph RCG agrees with Γ then the outputs of all processors are valid combination through Δ with respect to the initial items which are the inputs.

Interchangeably, we will view a round as a task: The abstract "item" is the processor id, and the output of a node/processor is a set of ids (i.e., the first message above is now (my_id, my_id)). The task is then defined through a predicate on the combination of returned sets. Obviously, any variant of a model of "iterated shared memory" [16] can be captured by such a task, and consequently as an adversary in our network. Notice that the view of a task as being "invoked" by a processor or not is mute when we compute in synchronous networks. Processors do not "crash," they always are together in a round. It is just that some may not be observed by others in a round. Thus, say, t-resiliency is an adversary whose RCG has a strongly connected component (SCC) of size at least $n - t$, and this SCC is a source SCC in the graph of the RCG.

3 TOUR Network Solves RWWF - Direct Argument

In this section as an informal warm-up and for self containment we show a direct implementation of asynchronous read-write wait-free by a TOUR network. To show that TOUR solves every task solvable in RWWF, it is enough to show that TOUR emulates RWWF in a nonblocking way. After taking enough steps eventually there is a processor that generates an output. From there on, processors "ignore" this processor, i.e., continue the task emulation without ever waiting on the processor that has terminated. Therefore, some other processor progresses until producing an output in the non-blocking emulation. This process continues inductively until all processors have produced an output and terminate.

In the next section we achieve the same by reduction, hence the informality in this section. In the next section we also show that TOUR is equivalent to TP, thus TP solves RWWF as well.

How does TOUR simulate a single-writer multi-reader system? Suppose, in round 1 each processor p_i sends $\langle p_i, item_i \rangle$ to all neighbors. In round 2 each processor just forwards the items it received to all its neighbors. A new combinatorial result by Linial and the authors [5] shows that even though the TOUR's

RCG in round 2 is not the same as the RCG in round 1, nevertheless there exists at least one processor called *king* whose item has reached all. But how does a processor know its item has reached all? At the end of a round in which all the messages it receives contain its item, it knows that all processors have its item. Those processors from whom it receives a message have its item because they sent it to him. Those from whom it did not receive a message have it as at least at this round they must have received it, since if TOUR purged the message to it, then in this round it could not purge the message from it. Thus, in round 3, at least the *king* processor knows its item has been received by all.

It is now easy to see how to emulate the SWMR system: To write, processor p_i sends the value it writes as a new item. All processors forward in each round all the items they have ever received or that originated from them. Of course, the next new item a processor sends is tagged by its id and a sequence number, indicating what number of write it is for that processor. Thus each processor can order all the items it received that originated from the same processor. Processor p_i finishes its kth write when it learns as above that its kth item has been received by all.

To read, a processor takes all the highest index items it knows from each other processor (taking the initial value for processors it did not hear from), and treating this collection of n items as a single item, which it now writes, i.e. continues until it knows everybody saw all the items it is reading.

Obviously, a write of an *item* is linearized to the end of the first round in which its writing terminates or its read terminates. A read by a processor is linearized at the end of its read operation.

As argued above, progress is guaranteed, at least one process finishes writing and then reading. Thus eventually at least one processor has alternately read and wrote enough times so that it produces an output. It then writes this output. After its write we do not need to specify what it does. From now on it can be ignored as the rest of the processors continue as a smaller system with a TOUR adversary. Hence a new processor will do enough writes and reads to obtain an output, and the process continues inductively until all processors produce an output.

We now show that SWMR implements TOUR. Notice that the SWMR steps can be "stripped" into rounds by tagging each write with its round number, and keeping the entire sequence of writes of p_i in its SWMR register C_i. To simulate a send by p_i of item *item* in round j, processor p_i appends to its SWMR cell C_i the pair $\langle j,\ item\rangle$. After the send it simulates receiving messages in round j by reading one by one in some order all cells $C_l,\ l = 1, ..., n$. If cell C_k contains a pair $\langle j,\ item'\rangle$ it simulates receiving a message from p_k at round j containing $item'$. Since in shared memory if p_i and p_k both perform write and then read, either p_i observes p_k's write, or p_k sees p_i's write or both, in the simulation of round j. Thus we have that a stripped round of SWMR is an instance of TOUR: There is at least one message simulated between p_i and p_k at round j.

4 TOUR Network Solves RWWF - Via Reduction

Here we show that TOUR network is equivalent to the iterated atomic snapshot (IAS) model. Together with the result of [18], that shows that IAS is non-blocking equivalent to the asynchronous SWMR model, this reduction proves that TOUR solves RWWF and no more. The IAS model consists of a bank of atomic snapshot memories $M_1, M_2, M_3, ...$ To compute, a processor writes (submits) a value to memory M_i, starting with $i = 1$, and then obtains an atomic snapshot S_i of M_i, it then writes to memory M_{i+1}, and so forth until it is ready to output. In the full-information protocol a processor originally writes its input to M_1 and from there on submits its output from M_i as an input to M_{i+1}.

The snapshot task implements a round of TOUR network: The items submitted to the snapshot are the messages in a round. If p_i in the snapshot task returns the item of p_j (or p_j for short) we consider it that the message from p_j to p_i was successful. Since snapshot is an instance of shared memory pairs of processors do not miss each other, and RCG contains a tournament. Thus, the snapshot task implements one round of TOUR.

TOUR network implements a snapshot: To show that TOUR implements snapshot we borrow the algorithm from [18]. Although that algorithm was given in the context of AS it also works for TOUR network. We go through n rounds of TOUR. As before, in each round a processor p_i sends to all other nodes all the ids, S_i, it has obtained by now. Let $S_j[k]$ be the the set S_j at processor p_j the end of round k. Then, if at the end of round ℓ, $|S_j[\ell]| = \ell$, p_j returns $S_j[\ell]$.

Claim. The sets $S_i, i = 1, \ldots n$ thus returned in the above algorithm are snapshots. I.e., $\forall i, p_i \in S_i$ and $\forall (i \text{ and } j), S_i \subseteq S_j$ or $S_j \subseteq S_i$.

Proof. By induction. Assume that by the end of round $k-1$ at most $k-1$ processors returned and they returned snapshots, and all the processors which did not return have sets containing the largest (snapshot) set returned so far and all the sets are at least of size k at the beginning of round k.

Observe that the inductive assumption holds at the end of round $k = 1$. First we show that at round $k > 1$ only a single set can be returned. Since each of the pair p_i, p_j has a set of size k or more (i.e., $|S_i[k-1]| \geq k$ and $|S_j[k-1]| \geq k$), and one of them at least hears from the other, then if processor p_i returns then it saw only sets identical to his own ($S_i[k-1]$), so either p_j does not return if it has a different set, or it returns since it has the same set.

If p_i returns, then it received messages only from processors with a set identical to his ($S_i[k-1]$), thus $|S_i[k-1]| = k$) that p_i returns was sent to all processors who do not have an identical set, consequently they either heard already of $k+1$ items or they heard about k items and S_i adds at least one more since it is different. Thus the hypothesis that the set that continue to round $k+1$ contain S_i if S_i was returned is maintained. establishing containment.

Since processors heard about themselves and the maximum size returned by now is k then the number of processors returning in round k is at most k.

Obviously, as in the end of the previous section, task AS implements a round of $TOUR$. Thus we have established that TOUR-dynamic is equivalent to IAS which solves exactly the class RWWF.

5 TP is Equivalent to TOUR

TP implements TOUR We first show that $2n-1$ rounds of TP implements TOUR round. In each round a processor p_i sends to all its neighbors the set H_i of inductively all the ids it has heard so far, starting by setting $S_i = \{p_i\}$ in the beginning of first round, and sending S_i. At the end of each subsequent round it just sets S_i to the union of his set with all the sets it received in the round.

Claim. After $2n-1$ rounds for every p_i and p_j, either $p_i \in S_j$, or $p_j \in S_i$, or both.

Proof. Let H_i at the end of a round be the set of nodes p_k such that $p_i \in S_k$. If neither $p_i \in H_j$ nor $p_j \in H_i$, then, using Sperner Lemma [31] for dimension 1, in the next round of TP a message is successful from either a node in H_i to a node not in H_i or from a node in H_j to a node not in H_j, or both. Thus in each round the size of at least one of the sets H_i and H_j increases by one.

TOUR implements TP We show that a round of TOUR is a round of TP. Let G be the RCG of a round of TOUR. Thus is contains a tournament G'. Consider $SCC(G')$ - the graph of the strongly-connected-components of G'. Obviously $SCC(G')$ is an acyclic tournament and we can topologically sort it, where there is an edge leading from one component to the next in the topological order, establishing the requirement.

6 TP Network Colors a Subdivided Simplex

To show that the outputs of a multi round execution of TP network colors a subdivided simplex we show that PAIRS-dynamic implements TP, and show that the outputs of PAIRS color a subdivided complex. We show the equivalence of TP to PAIRS, by showing that PAIRS is equivalent to TOUR. That the latter is equivalent to TP has been shown in Section 5.

Recall the definition of TOUR: In the TOUR adversary in each round on each edge at least one message is delivered in one direction or both directions. In the PAIRS adversary we spread TOUR over $n(n-1)/2$ rounds where in each such round a message is sent in both directions of one unique edge and at most one of these two messages may be purged.

Clearly, TOUR implements PAIRS by going $n(n-1)/2$ rounds and at each round ignoring anything which is not associated with the particular edge of the round. To see that PAIRS implements TOUR, we emulate rounds of TOUR one after the other. Each round of TOUR is emulated by $n(n-1)/2$ rounds of PAIRS. In each such round of PAIRS, if a processor sends it sends what it sent

the first time it was scheduled in this round-of-TOUR emulation. At the end of the $n(n-1)/2$ rounds a processor just collects all the messages it has received.

6.1 The PAIRS Protocol-Complex at an End of a Round

Here we show that the protocol-complex of PAIRS is a subdivided complex. Keeping the exposition simple, we avoid unnecessary formalism and notation by restricting this section to $n = 3$. The $n = 3$ case generalizes to higher n's in a straightforward way. For completeness we repeat the construction given here, but for arbitrary n in Appendix A.

Consider all the possible local states of the 3 processors p_0, p_1, p_2 after round r and make a graph G_r out of it. The nodes of G_r are the pairs of processor-id and its possible local state at the end of round r. Two nodes are connected by an edge between them if there exists an execution E, that is an instantiation of the PAIRS adversary in rounds 1 to r, such that the corresponding processors are in the corresponding states. Assume inductively that G_r is a 3-colored triangulated triangle. Notice the colors here are the processor ids (not to be confused with other additional colorings of the complex used in some proofs, such as the impossibility of set-consensus, where the colors are the outputs of the processors in each final state).

The process starts with a triangle of the 3 processors in their initial state.

Let round $r+1$ be a round in which messages are sent (only) on link (p_i, p_j). How does the different behaviors of the adversary in round $r+1$ change the graph/simplex? First, for round $r+1$ subdivided simplex G_{r+1} start from a copy of the subdivided simplex G_r of round r. Each node that corresponds to a process p_k different than p_i or p_j remains the same in the new subdivided simplex except now it corresponds to the same state of process p_k as in the previous round with the additional knowledge that one more round has passed. In place of each (p_i, p_j) edge we get now a path of 3 edges $(p_i, p'_j), (p'_j, p'_i), (p'_i, p_j)$. The first node in the new path is a p_i node with a state in which p_i appends to its local state from round r that it did not receive a message from p_j in round $r+1$. Analogously, the node p_j at the other end of the path. The nodes in the middle, p'_j and p'_i, are nodes in which the corresponding processor appends to its state from round r the content of the message it received in round $r+1$. The nodes corresponding to the third processor p_k have only one new incarnation from G_r to G_{r+1} as p_k records that it did not receive any message in the round, and as said, that one more round, round $r+1$ has terminated. Obviously a node of type p_k that was connected to a (p_i, p_j) edge at the end of round r is now connected to the four nodes of the path that replaces the (p_i, p_j) edge in G_r. See Figure 1(a).

For example, consider this process of graph evolution by drawing G_{r+1} in the plane (Figure 1(a)). Initially the graph is the triangle of the initial states. Assume the first round of the PAIRS schedule is sending messages on edge $(1, 2)$. To construct the graph corresponding to this round take the $(p_1, p2)$ edge and replace it by a path by planting two middle nodes on the edge, denoting the new

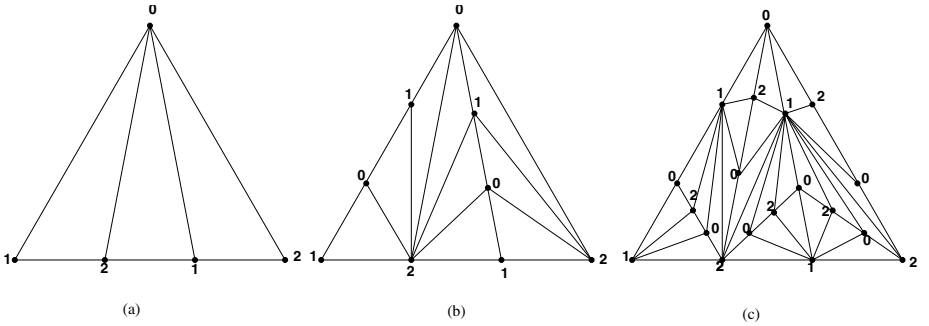


Fig. 1. A 3 processors, 3 rounds subdivision example. The order edges on which messages where exchanged is $(1,2),(0,1),(0,2)$. (a), (b) and (c) are the corresponding first, second and third split operations.

local states by the original (p_1, p_2) to get an alternating path of p_1, p_2, p_1, p_2. We now connect the middle nodes with node p_0, and we got G_1.

Inductively, we embed the initial triangle in the plane, it is 3-colored by the 3 ids. In any inductive step we embed node on an edge and connect them to the third node in the triangles the edge is in. Obviously we have an embedding and the corresponding triangilated triangle is 3-colored, and the original edges of the triangle we started with are now a face which is 2-colored by the colors p_i, p_j defining the original egde.

7 Conclusions

We have extended the study of message-adversary [13,29,30,32] in generalizing the notion of an adversary, and studying it at the other "extreme" than it was studied in the past: Instead of studying what limits must be put on the adversary so that every task is solvable (consensus [15,19]), we studied what limits must be put on the adversary so that the weakest tasks of interest are solvable - those that are solvable read-write wait-free.

We showed the benefit of such a "theoretical study" by showing that it yields the simplest-ever explanation of why read-write wait-free equates with subdivided simplex.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merrit, M., Shavit, N.: Atomic Snapshots of Shared Memory. In: Proc. 9th ACM Symposium on Principles of Distributed Computing (PODC 1990), pp. 1–13. ACM Press (1990)
2. Afek, Y., Attiya, H., Fouren, A., Stupp, G., Touitou, D.: Long-Lived Renaming Made Adaptive. In: PODC, pp. 91–103 (1999)

3. Awerbuch, B., Even, S.: Efficient and reliable broadcast is achievable in an eventually connected network (Extended Abstract). In: Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, PODC 1984, Vancouver, British Columbia, Canada, pp. 278–281 (1984)
4. Afek, Y., Gafni, E.: End-to-end communication in unreliable networks. In: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 1988, Toronto, Ontario, Canada, pp. 131–148 (1988)
5. Afek, Y., Gafni, E., Linial, N.: A King in two tournaments (submitted for publication)
6. Afek, Y., Stupp, G., Touitou, D.: Long-lived Adaptive Collect with Applications. In: FOCS, pp. 262–272 (1999)
7. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message-Passing Systems. J. ACM 42(1), 124–142 (1995)
8. Bar-Noy, A., Naor, J.: Sorting, Minimal Feedback Sets and Hamilton Paths in Tournaments. SIAM Journal on Discrete Mathematics 3(1), 7–20 (1990), doi:10.1137/0403002.
9. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on the Theory of Computing (STOC 1993), pp. 91–100. ACM Press (1993)
10. Borowsky, E., Gafni, E.: Immediate Atomic Snapshots and Fast Renaming (Extended Abstract). In: PODC, pp. 41–51 (1993)
11. Borowsky, E., Gafni, E.: A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). In: Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC 1997), pp. 189–198. ACM Press (August 1997)
12. Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: The BG Distributed Simulation Algorithm. Distributed Computing 14(3), 127–146 (2001)
13. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. Distributed Computing 22(1), 49–71 (2009)
14. Dolev, D., Reischuk, R.: Bounds on information exchange for Byzantine agreement. JACM 32(1), 191–204 (1985)
15. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
16. Gafni, E.: Round-by-Round Fault Detectors, Unifying Synchrony and Asynchrony (Extended Abstract). In: PODC, pp. 143–152 (1998)
17. Gafni, E., Koutsoupias, E.: Three-Processor Tasks Are Undecidable. SIAM J. Comput. 28(3), 970–983 (1999)
18. Gafni, E.: The 0–1-Exclusion Families of Tasks. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 246–258. Springer, Heidelberg (2008)
19. Herlihy, M.P.: Wait-Free Synchronization. ACM Transactions on programming Languages and Systems 11(1), 124–149 (1991)
20. Herlihy, M., Rajsbaum, S., Tuttle, M.R.: Unifying Synchronous and Asynchronous Message-Passing Models. In: PODC, pp. 133–142 (1998)
21. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
22. Kuhn, F., Lynch, N., Oshman, R.: Distributed Computation in Dynamic Graphs. In: 42nd ACM Symposium on Theory of Computing (STOC 2010) (2010)
23. Lamport, L.: On Interprocess Communication. Part II: Algorithms. Distributed Computing 1(2), 86–101 (1986)

24. Landau, H.: On dominance relations and the structure of animal societies, III: The condition for score structure. Bulletin of Mathematical Biophysics 15(2), 143–148 (1953)
25. Moran, S., Wolfstahl, Y.: Extended impossibility results for asynchronous complete networks. Inf. Process. Lett. 26(3), 145–151 (1987)
26. Moses, Y., Rajsbaum, S.: The Unified Structure of Consensus: A Layered Analysis Approach. In: PODC, pp. 123–132 (1998)
27. Munkres, J.R.: Elements of algebraic topology. Addison-Wesley (1984)
28. Saks, M., Zaharoglou, F.: Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)
29. Nicola, S., Peter, W.: Time is Not a Healer. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 304–313. Springer, Heidelberg (1989)
30. Nicola, S., Peter, W.: Agreement in synchronous networks with ubiquitous faults. Theor. Comput. Sci. 384(2-3), 232–249 (2007)
31. Sperner, E.: Fifty years of further development of a combinatorial lemma. In: Numerical solution of highly nonlinear problems (Sympos. Fixed Point Algorithms and Complementarity Problems, Univ. Southampton, Southampton), Part A, pp. 183–197, Part B, pp. 199–214 (1979)
32. Schmid, U., Weiss, B., Keidar, I.: Impossibility results and lower bounds for consensus under link failures. SIAM Journal on Computing 38(5), 1912–1951 (2009)

A The PAIRS Subdivided Complex

For the sake of completeness we follow here the arguments given in Section 6 and show somewhat more formally that the PAIRS model when executed for $k = f(j)$ rounds implies a subdivision of the input complex. The argument here is for arbitrary n. Each elementary simplex in the subdivision corresponds to a particular k rounds execution of PAIRS, i.e., a particular instantiation of the adversary. The number of iterations, $k = f(j)$ is taken as the number of rounds required by the PAIRS in the worst case to emulate a complete wait-free execution of a task under consideration (see Section 3).

W.l.o.g, instead of subdividing the input complex we create a subdivision $\mathcal{T}$ of an n-vertex simplex $\mathcal{P}$, corresponding to an input less initial configuration simplex. Every vertex v in $\mathcal{T}$ is associated with a processor's name $\chi(v)$. Distinct vertices at the same simplex of $\mathcal{T}$ are associated with distinct processors, so that χ is a proper coloring of the graph which is $\mathcal{T}$'s one-dimensional skeleton. Clearly, initially χ is a proper vertex coloring of $\mathcal{P}$. In addition, the construction of $\mathcal{T}$ is such that χ has the *Sperner property* i.e., if vertex $y \in V(\mathcal{T})$ is on $\mathcal{P}$'s boundary and if σ is the lowest-dimensional face of $\mathcal{P}$ that contains y, then $\chi(y) = \chi(x)$ for some vertex x of σ.

Lemma 1. *Given the PAIRS model on n processors, and k the number of rounds executed by the model, then there is a subdivision $\mathcal{T}$ of the n-vertex simplex $\mathcal{P}$ with proper coloring χ of $V(\mathcal{T})$, the vertices of $\mathcal{T}$. Such that each simplex in the subdivision represents the final states of the n processors at the end of the execution. The coloring $\chi : V(\mathcal{T}) \to \{0, \ldots, n-1\}$ satisfies Sperner's condition.*

Proof. We go through a sequence of subdivision refinements, $\mathcal{T}_i$, $i = 0, \ldots, k$, where, $\mathcal{P} = \mathcal{T}_0$, $\mathcal{T}_{i+1}$ is a subdivision refinement of $\mathcal{T}_i$ that corresponds to all possible behaviors in the $i'th$ round of the PAIRS adversary, and $\mathcal{T}_k = \mathcal{T}$. Thus we go repeatedly over the sequence of edges in the order in which PAIRS goes over them, for k steps. In each step we consider the three possible behaviors of the adversary on the corresponding edge (one message delivered in one direction, in the other direction or in both). The resulting $\mathcal{T}$ is finite, with 3^k simplices.

$\mathcal{T}_0 = \mathcal{P}$ is properly colored by $\chi : V(\mathcal{P}) \to \{0, \ldots, n-1\}$, i.e., each vertex is associated with the processor id which is its color, and with its initial state. Every refinement step consists of several *xy-split* operations, that split each simplex of $\mathcal{T}_i$ into three simplices in $\mathcal{T}_{i+1}$. In an *xy-split*, where x, y is a pair of adjacent vertices of $\mathcal{T}$ we partition the edge xy into three segments $x = z_0, z_1, z_2, z_3 = y$ in this order. All old vertices maintain their χ values, while $\chi(z_1) := \chi(y)$ and $\chi(z_2) := \chi(x)$. Furthermore, every simplex σ of $\mathcal{T}$ with $x, y \in V(\sigma)$, say $\sigma = \text{conv}(\{x, y\} \dot{\cup} S)$ (i.e., $S \cup \{x, y\}$ is the set of vertices of σ) is split accordingly to three simplices $\sigma_0 \cup \sigma_1 \cup \sigma_2$ where $\sigma_i = \text{conv}(\{z_i, z_{i+1}\} \dot{\cup} S)$. We note, following Section 2 that an *xy-split* corresponds to the three possible behaviors of the adversary between processors $\chi(x)$ and $\chi(y)$. Vertex x corresponds to the state of processor $\chi(x)$ in $\mathcal{T}_i$ concatenated with it sending the message in round $i+1$ but not receiving any message, vertex z_1 corresponds to processor $\chi(y)$ in $\mathcal{T}_i$ concatenated with it sending its state (message) in round $i+1$ and receiving the message from processor $\chi(x)$, and so forth. To construct $\mathcal{T}_{i+1}$ from $\mathcal{T}_i$ we apply the *xy-split* operation to all pairs of vertices, x, y in $\mathcal{T}_i$, such that $(\chi(x), \chi(y))$ is the unordered pair of processors associated with round $i+1$ in the PAIRS schedule. See Figure 1 for an example.

It remains to show that the coloring $\chi : V(\mathcal{T}) \to \{0, \ldots, n-1\}$ satisfies Sperner's condition. Clearly $\mathcal{P}$ satisfies the condition. Consider by contradiction the first time that the condition fails, in an xy-split operation o in which the edge x_o–y_o is split, creating the two new vertices z_{o1}, z_{o2}, with colors $\chi(x_o)$ and $\chi(y_o)$. The only way the condition can fail is if either of the z's is not in the union of the carriers of x_o and y_o. But, by simple algebra, the carrier of any point along the x_o–y_o edge (line) belongs to the union of the carriers of x_o and y_o, thus $\chi(z_{o1})$ and $\chi(z_{o2})$ are colors of vertices in their carrier, concluding the proof.

Maximal Antichain Lattice Algorithms for Distributed Computations

Vijay K. Garg*

Parallel and Distributed Systems Lab,
Department of Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, TX 78712
garg@ece.utexas.edu
http://www.ece.utexas.edu/~garg

Abstract. The lattice of maximal antichains of a distributed computation is generally much smaller than its lattice of consistent global states. We show that a useful class of predicates can be detected on the lattice of maximal antichains instead of the lattice of consistent cuts obtaining significant (exponential for many cases) savings. We then propose new online and offline algorithms to construct and enumerate the lattice of maximal antichains. Previously known algorithm by Nourine and Raynoud [NR99, NR02] to construct the lattice takes $O(n^2m)$ time where n is the number of events in the computation, and m is the size of the lattice of maximal antichains. The algorithm by Jourdan, Rampon and Jard [JRJ94] takes $O((n + w^2)wm)$ time where w is the width of the computation. All these algorithms assume as input the lattice of maximal antichains prior to the arrival of a new event. We present a new online incremental algorithm, OLMA, that computes the newly added elements to the lattice without requiring the prior lattice. Since the lattice may be exponential in the size of the computation, we get a significant reduction in the space complexity. The OLMA algorithm takes $O(mw^2 \log w_L)$ time and $O(w_L w \log n)$ space where w_L is the width of the lattice of maximal antichains. The lower space complexity makes our algorithm applicable for online global predicate detection in a distributed system. For the purposes of analyzing offline traces, we also propose new enumeration algorithms to traverse the lattice.

1 Introduction

A distributed computation can be modeled as a partially ordered set (poset) of events based on the happened-before relation [Lam78]. Given any poset, there are three important distinct lattices associated with it: the lattice of consistent cuts (or ideals), the lattice of normal cuts, and the lattice of maximal antichains. The lattice of consistent cuts captures the notion of consistent global states in

* Supported in part by the NSF Grants CNS-1115808, CNS-0718990, CNS-0509024, and Cullen Trust for Higher Education Endowed Professorship.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 240–254, 2013.

a distributed computation and has been discussed extensively in the distributed computing literature [Mat89, CM91, GM01]. The other lattices have not received as much attention. For a poset P, its completion by normal cuts is the smallest lattice that has P as its suborder [DP90]. Its applications to distributed computing are discussed in [Gar12]. In this paper, we discuss the lattice of maximal antichains with applications to global predicate detection.

For the set of events in a distributed computation ordered with happened-before relation, a subset of events forms an *antichain* if all events in the subset are pairwise concurrent. Informally, an antichain captures a possible set of events that could have occurred concurrently, and the events do not have any causal or happened-before relationship with each other. The lattice of all antichains is isomorphic to the lattice of all consistent cuts. An antichain A is *maximal* if there does not exist any event that can be added to the set without violating the antichain property. The *lattice of maximal antichains*, denoted by $L_{MA}(P)$ is the set of all maximal antichains under the order consistent with the order on the lattice of consistent cuts.

The lattice of maximal antichains captures all maximal sets of concurrent events and has applications in detection of global predicates because it is usually much smaller than the lattice of consistent cuts. In the extreme case, the lattice of consistent cuts may be exponentially bigger in size than the lattice of maximal antichains. We show in this paper that some global predicates can be detected on the lattice of maximal antichains instead of consistent cuts, thereby providing an exponential reduction in the complexity of detecting them. Fig. 1(i) shows a distributed computation with six events. For example, process P_1 executes a send event a, and then receives a message at event d. The message sent by P_1 is received by P_3 as event e. Fig. 1(ii) shows the computation with vector clocks. Fig. 2(i) shows the poset corresponding to the computation. Its lattices of consistent cuts and maximal antichains are shown in Fig. 2(ii), and (iii) respectively.

In this paper, we also discuss algorithms for computing L_{MA} for a distributed computation (or a trace of events) given as a finite poset P with implicit representation using vector clocks. Incremental algorithms assume that we are given a poset P and its lattice of maximal antichains L and we are required to construct L_{MA} of the poset P' corresponding to P extended with an element x. The algorithms by Jourdan, Rampon and Jard [JRJ94], and Lourine and Raynaud

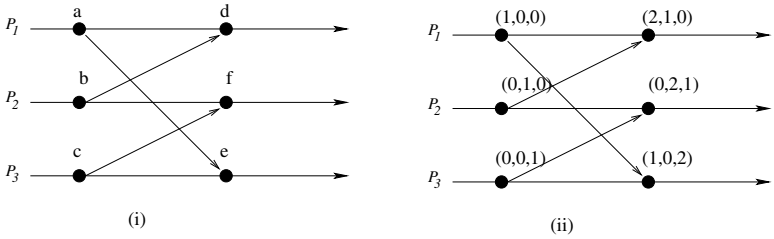


Fig. 1. (i) A computation (ii) Its equivalent representation in vector clocks

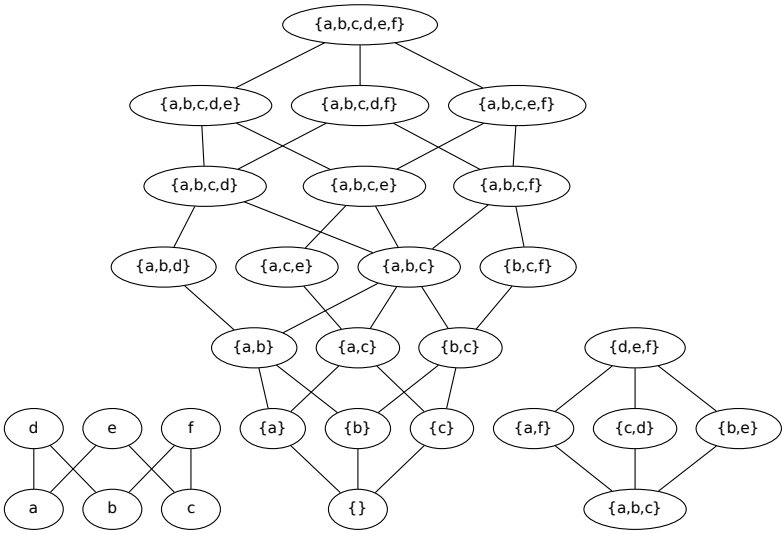


Fig. 2. (i) The original poset. (ii) Its lattice of ideals (consistent cuts) (iii) Its lattice of maximal antichains.

[NR99, NR02] fall in this class. These algorithms store the entire lattice L_{MA} and have space complexity of $O(mn \log n)$ where n is the size of the poset P, and m is the size of $L_{MA}(P)$. The algorithm by Jourdan, Rampon and Jard [JRJ94] has $O(w^3 m)$ time complexity and the algorithm by Lourine and Raynaud [NR99, NR02] has $O(mn)$ time complexity.

Our first algorithm called ILMA is a simple modification of the algorithm by Nourine and Raynoud [NR99, NR02] based on vector clocks. The algorithm requires $O(wm \log m)$ time and $O(wm \log n)$ space where w is the width of the poset P. The second algorithm called OLMA does not require lattice $L_{MA}(P)$ to compute elements of the new lattice. Let m_x be the size of the set of maximal antichains that include the element x. Then, the algorithm OLMA requires $O(w^2 m_x \log w_L)$ time and $O(w_L w \log n)$ space, where w_L is the width of the lattice of maximal antichains. Since w_L is much smaller than m, there is a significant reduction in the space complexity. The algorithm OLMA answers the open problem posed in [NR99]: "One open question is the enumeration of the family generated by a basis without computing the tree or the lattice with the same complexity."

Even though the OLMA algorithm has lower space complexity than ILMA, in the worst case the size of w_L can be exponential in the number of processes. If the goal is to not *construct*, but simply *enumerate* (or check for some global predicate) all the elements of L_{MA}, then we propose BFS and DFS based

enumeration of lattice of maximal antichains. Earlier algorithms for enumeration of closed sets by Ganter [Gan84] use lexical enumeration. The BFS and DFS algorithms are more meaningful in the context of distributed systems. For example, when searching for a maximal antichain satisfying a given predicate the programmer may be interested in the maximal antichain that appears first in the BFS enumeration. It is important to note that algorithms for BFS and DFS enumeration of lattices are different from the standard graph-based BFS and DFS enumeration because our algorithms cannot store the explicit graph corresponding to the lattice. Hence, the usual technique of marking the visited nodes is not applicable.

Table 1 summarizes the time and space complexity for construction and enumeration algorithms for the lattice of maximal antichains with the notation in Table 2.

Table 1. Lattice Construction of Maximal Antichains

Incremental Algorithms	Time Complexity	Space Complexity
Jourdan et al. [JRJ94]	$O(w^3m)$	$O(mn \log n)$
Nourine and Raynaud [NR99, NR02]	$O(mn)$	$O(mn \log n)$
Algorithm ILMA [this paper]	$O(wm \log m)$	$O(mw \log n)$
Algorithm OLMA [this paper]	$O(m_x w^2 \log w_L))$	$O(w_L w \log n)$
Offline Algorithms		
Jourdan et al. [JRJ94]	$O((n+w^2)wm)$	$O(mn \log n)$
Nourine and Raynaud [NR99, NR02]	$O(mn^2)$	$O(mn \log n)$
Algorithm ILMA [this paper]	$O(nwm \log m)$	$O(mw \log n)$
BFS-MA [this paper]	$O(mw^2 \log m)$	$O(w_L w \log n)$
DFS-MA [this paper]	$O(mw^4)$	$O(nw \log n)$
Lexical by Ganter [Gan84]	$O(mn^3)$	$O(n \log n)$

Table 2. The notation used in the paper

Symbol	Definition	Symbol	Definition
n	size of the poset P	m	size of the maximal antichains lattice L
w	width of the poset P	m_x	number of strict ideals $\geq D(x)$ (Section 2)
w_L	width of the lattice L		

The paper is organized as follows. Section 2 gives the background definitions. Section 3 discusses the lattice of maximal antichains and some other lattices that are useful for incremental lattice construction. Section 4 discusses incremental and online construction of the lattice of maximal antichains. Section 5 discusses enumeration of the lattice of maximal antichains. We discuss distributed computing applications in Section 6.

2 Background: Posets with Implicit Representation

We assume that the reader is familiar with the basic concepts of posets and lattices [DP90]. A partially ordered set (or *poset*) is a pair $P = (X, \leq)$ where X is a set and $\leq$ is a reflexive, antisymmetric, and transitive binary relation on X. We write $x \leq y$ when $(x, y) \in P$. If either $x \leq y$ or $y \leq x$, we say that x and y are *comparable*; otherwise, we say x and y are *incomparable* or *concurrent*. A subset $Y \subseteq X$ is called an *antichain (chain)*, if every distinct pair of points from Y is incomparable (comparable) in P. The *width* (*height*) of a poset is defined to be the size of a largest antichain (chain) in the poset.

Given a subset $Y \subseteq X$, the *meet* of Y, if it exists, is the greatest lower bound of Y and the *join* of Y is the least upper bound. In Fig. 2(i), the meet of the set $\{d, e\}$ is a. The meet of the set $\{b, c\}$ does not exist. An element is join-irreducible (meet-irreducible) if it cannot be expressed as join (meet) of other elements. In Fig. 2(i), the elements $\{a, b, c\}$ are join-irreducible but $\{d, e, f\}$ are not. A poset $P = (X, \leq)$ is a *lattice* if joins and meets exist for all finite subsets of X. The largest element of a lattice is called the *top* element.

Let P be a poset with a given chain partition of width w. In a distributed computation, P would be the set of events executed under the happened-before partial order. Each chain would correspond to a total order of events executed on a single process. In such a poset, every element e can be identified with a tuple (i, k) which represents the k^{th} event in the i^{th} process. In this paper, we keep the order relation of the poset implicit using vector clocks [Mat89, Fid89] as explained next. For $e \in P$, let $D[e]$, the *down-set* of e, be the elements of P that are less than or equal to P. The set $D[e]$ can equivalently be captured using a vector $e.V$ such that $e.V[i] = j$ iff there are exactly j elements on chain i that are less than or equal to e. It is easy to verify that $e \leq f$ iff $e.V \leq f.V$. The above discussion shows 1-1 correspondence between each element e of a poset and its vector $e.V$.

In this paper, we also use the set $D(e)$, the strict down-set of e, which contains all elements in the poset P that are strictly less than e. The reader should note the difference in the notation $D[e]$ and $D(e)$; the former is a down-set and the latter a strict down-set. The notation $D(e)$ can be extended to apply for sets as follows: $D(Y) = \cup_{e \in Y} D(e)$. We also use the dual notation for *up-sets*, $U(Y)$ and $U[Y]$.

A subset Q is an ideal (order ideal, or a consistent cut) of P if it satisfies the constraint that if f is in Q and e is less than or equal to f, then e is also in Q. For any element $e \in P$, $D[e]$ is always an ideal. In distributed computing, when a distributed computation is modeled as a poset of event, the order ideals are called *consistent cuts*, or *consistent global states* [CL85].

Any ideal Q of P can be represented using a vector $Q.V$ with the interpretation that $Q.V[i] = j$ iff exactly j elements of chain i are in Q. We use the set and the vector notation for an ideal interchangably. Given two ideals Q and R, their intersection (union) is simply the component-wise minimum (maximum) of the vectors for Q and R. The set of order ideals is closed under both union and intersection and therefore forms a lattice under the set containment order.

3 Maximal Antichain Lattice

We first define three different but isomorphic lattices: the lattice of maximal antichain ideals, the lattice of maximal antichains and the lattice of strict ideals. Besides giving an insight in the structure of the lattice of maximal antichains, these lattices have different closure properties making them useful in different contexts. The lattice of strict ideals is closed under union and is used in our ILMA and OLMA algorithms. The lattice of maximal ideals is closed under intersection and is used in our DFS-MA algorithm.

Definition 1 (Maximal Antichain). *An antichain A is maximal in a poset $P = (X, \leq)$ if every element in $X - A$ is comparable to some element in A.*

In Fig. 3(i), the set $\{d, e\}$ is an antichain but not a maximal antichain because f is incomparable to both d and e. The set $\{d, e, f\}$ is a maximal antichain. It is easy to see that A is a maximal antichain iff $D(A) \cup U[A] = X$.

Definition 2 (Maximal Ideal). *An ideal Q of a poset $P = (X, \leq)$ is a* maximal antichain ideal *(or, maximal ideal) if the set of its maximal elements, denoted by $maximal(Q)$, is a maximal antichain.*

The set of maximal ideals is closed under intersection but not union. In Fig.3(ii) the ideals $\{a, b, c, d\}$ and $\{a, b, c, e\}$ are maximal ideals, but their union $\{a, b, c, d, e\}$ is not a maximal ideal.

Definition 3 (Lattice of Maximal Ideals of a Poset). *For a given poset $P = (X, \leq)$, its lattice of maximal ideals is the poset formed with the set of all the maximal ideals of P under the set inclusion. Formally,*

$$L_{MA}(P) = (\{A \subseteq X : A \text{ is a maximal ideal of } P\}, \subseteq).$$

For the poset in Figure 3(i), the set of all maximal ideals is:

$$\{\{a, b, c\}, \{a, b, c, d\}, \{a, b, c, e\}, \{a, b, c, f\}, \{a, b, c, d, e, f\}\}.$$

The poset formed by these sets under the $\subseteq$ relation is shown in Figure 3(iii). This poset is a lattice with the meet as the intersection.

A lattice isomorphic to the lattice of maximal ideals is that of the maximal antichains.

Definition 4 (Lattice of Maximal Antichains of a Poset). *For a given poset $P = (X, \leq)$, its lattice of maximal antichains is the poset formed with the set of all the maximal antichains of P with the order $A \preceq B$ iff $D[A] \subseteq D[B]$.*

In Section 4 we discuss incremental algorithms for lattice construction. In these algorithms, we have the lattice $L_{MA}(P)$ for a poset P and our goal is to construct $L_{MA}(P \cup \{x\})$ where x is a new event that is not less than any event in P. It would be desirable if all the elements of $L_{MA}(P)$ continue to be elements of $L_{MA}(P')$. However, this is not the case for maximal antichains. An antichain that is maximal in P may not be maximal in $P \cup \{x\}$. For example, in Fig. 3(i), suppose that f arrives last. The set $\{d, e\}$ is a maximal antichain before the arrival of f,

but not after. The algorithm by Jourdan, Rampon and Jard explicitly determines the maximal antichains that get changed when a new event arrives. In this paper, and also in [NR99], the problem is circumvented by building the lattice of strict ideals instead of the lattice of maximal antichains. If S is a strict ideal of P then it continues to be one on arrival of x so long as x is a maximal element of $P \cup \{x\}$. The lattice of strict ideals is isomorphic to the lattice of maximal antichains, but easier to implement via an incremental algorithm.

Definition 5 (Strict Ideal). *A set Y is a strict ideal of a poset $P = (X, \leq)$, if there exists an antichain $A \subseteq X$ such that $D(A) = Y$.*

Definition 6 (Lattice of Strict Ideals of a Poset). *For a given poset $P = (X, \leq)$, its lattice of strict ideals is the poset formed with the set of all the strict ideals of P with the $\subseteq$ order.*

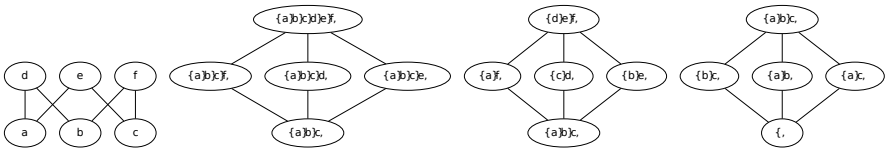


Fig. 3. (i) The original poset. (ii) Its lattice of maximal ideals (iii) Its lattice of maximal antichains (iv) Its lattice of strict ideals.

Fig. 3 shows a poset with the three isomorphic lattices: the lattice of maximal ideals, the lattice of maximal antichains and the lattice of strict ideals. To go from the lattice of maximal ideals to the lattice of maximal antichains, a maximal ideal Q is mapped to the antichain $maximal(Q)$. Conversely, a maximal antichain A is mapped to the maximal ideal $D[A]$. To go from the lattice of antichains to the lattice of strict ideals, a maximal antichain A is mapped to the set $D(A)$. Conversely, a strict ideal Z is mapped to an antichain as the minimal elements in Z^c, the complement of Z. For example, when Z equals $\{b, c\}$, its complement is $\{a, d, e, f\}$. The set of the minimal elements of the set $\{a, d, e, f\}$ is $\{a, f\}$ which is the antichain corresponding to $\{b, c\}$. The correctness of this mapping is shown in the proof of correctness of ILMA algorithm (Theorem 1).

4 Incremental Algorithms to Construct Lattice of Maximal Antichains

In this section, we give two incremental algorithms, ILMA and OLMA. The ILMA algorithm makes it easier to understand the difference in the technique of previous algorithms and the OLMA algorithm.

4.1 The ILMA Algorithm

The ILMA algorithm is a modification of the algorithm given by Nourine and Raynaud [NR99], based on computing the lattice of strict ideals. There are two main differences. First, our algorithm does not use the lexicographic tree used in their algorithm. Another difference is that we have used the implicit representation of the poset and the lattice based on vector clocks making the algorithm more suitable for distributed systems.

The ILMA algorithm, shown in Fig. 4, is based on computing closure under union of strict ideals. It takes as input the poset P, an element x, and the lattice of maximal antichains of P. The poset and the lattice are assumed to be represented using vector clocks. It outputs the lattice of maximal antichains of $P' = P \cup \{x\}$. At step 1, we compute the vector clock corresponding to the set $D(x)$. The vector clock for x, V corresponds to $D[x]$. By removing x from the set $D[x]$, we get $D(x)$. The removal of x is accomplished by decrementing $S[i]$ in step 1. At step 2, we add S to L and make L closed under union.

Input: P: a finite poset as a list of vector clocks
L: lattice of maximal antichains as a balanced binary tree of vector clocks
x: new element
Output: $L' :=$ Lattice of maximal antichains of $P \cup \{x\}$ initially L

// **Step 1**: Compute the set $D(x)$
Let V be the vector clock for x on process P_i;
$S := V$; $S[i] := S[i] - 1$;
// **Step 2**:
if $S \notin L$ then
    $L' := L' \cup \{S\}$;
    forall vectors $W \in L$:
        if $max(W, S) \notin L$ then $L' := L' \cup max(W, S)$;

Fig. 4. The Algorithm ILMA for Construction of Lattice of Maximal Antichains

The above algorithm can also be used to compute the lattice of maximal antichains of any poset in an offline manner by repeatedly invoking the algorithm with elements of the poset in any total order consistent with the poset. To start the algorithm, the initial poset P would be a minimal element of P and the corresponding lattice L would be a singleton element corresponding to the empty strict downset.

We now show the correctness of the algorithm.

Theorem 1. *The lattice L' constructed by ILMA algorithm is isomorphic to the lattice of maximal antichains of P'.*

Proof. It is easy to verify that every vector in L' is a strict ideal I of the poset $P' = P \cup \{x\}$. By induction, we can assume that L contains all the strict ideals of P. Step (1) adds the strict ideal for $D(x)$ and step (2) takes its union with all existing strict ideals. Since *max* is an idempotent operation, it is sufficient to iterate over L once.

The bijection from L' to the set of maximal antichains is as follows. Let I be a strict ideal in L', i.e., there exists an antichain A such that $D(A) = I$. Let I^c denote the complement of I, and let B equal to the set of the minimal elements of I^c. Thus, $B = minimal(I^c)$. It can be shown that every strict ideal I gives a unique antichain B by this construction. To show that B is a maximal antichain it is sufficient to show that $D(B) \cup U[B] = X$. This claim follows from the facts that $A \subseteq minimal(I^c)$, $D(A) = I$ and $U[minimal(I^c)] = I^c$. ■

The time complexity of ILMA is dominated by Step 2. Checking if the vector is in L requires $O(w \log m)$ time if L is kept as a balanced binary search tree of vector clocks. Thus, the time complexity of Step 2 is $O(wm \log m)$. By repeatedly invoking ILMA algorithm for a maximal element, we can construct the lattice of maximal antichains of a poset with n elements in $O(nwm \log m)$ time. The algorithm by Jourdan, Rampon and Jard takes $O((n + w^2)wm)$ time, and the algorithm by Nourine and Raynaud takes $O(mn^2)$ time. The space complexity is dominated by storage requirements for L. With implicit representation, we have to store m elements where each element is stored as a vector of dimension w of coordinates each of size $O(\log n)$. Hence, the overall space complexity is $O(mw \log n)$.

4.2 The OLMA Algorithm

In the ILMA algorithm, we traverse the lattice L for every element x. It requires us to maintain the lattice L which may be exponentially bigger than poset P, making the algorithm impractical for distributed computations. We now show an online algorithm OLMA which does not require the lattice L but only uses the poset P. Let M be the set of new elements (strict ideals) generated due to x. The time complexity of OLMA is dependent on the size of M independent of the size of the lattice L.

The incremental online algorithm OLMA is shown in Fig. 5. At lines (1) and (2), we compute the vector S for the set $D(x)$. At line (3), we check if S is already in $L_{MA}(P)$. Note that we do not store the lattice $L_{MA}(P)$. The check at line (3) is done by checking if S is a strict ideal of P. If this is the case, we are done and M is an empty set. Otherwise, we need to enumerate all strict ideals that are reachable from S in the lattice $L_{MA}(P')$. We do so in lines (5)-(15) by traversing the strict ideals greater than or equal to S in the BFS order. The set $\mathcal{T}$ consists of strict ideals that have not been explored yet. At line (7), we remove the smallest strict ideal H and enumerate it at line (8). For global predicate detection applications, we would evaluate the global predicate on H at this step. To find the set of strict ideals that are reachable by taking union with one additional event e, we explore the next event e after the ideal H along every process. There are two cases to consider.

```
   Input: a finite poset P, x maximal element in P' = P ∪ {x}
   Output: enumerate M such that L_MA(P') = L_MA(P) ∪ M

(1)    S := the vector clock for x on process P_i;
(2)    S[i] := S[i] − 1;
(3)    if S is not a strict ideal of P then
(4)    // BFS(S): Do Breadth-First-Search traversal of M
(5)        T := set of vectors initially {S};
(6)        while T is nonempty do
(7)            H := delete the smallest vector from T in the levelCompare order;
(8)            enumerate H;
(9)            foreach process k with next event e do
(10)               if (D(e) ⊆ H) then
(11)                   if (succ(e) exists then T := T ∪ {max(H, D(succ(e)))};
(12)               else
(13)                   T := T ∪ {max(H, D(e))};
(14)           endfor;
(15)       endwhile;
(16)   endif;

       int levelCompare(VectorClock a, VectorClock b)
(1)        if (a.sum() > b.sum()) return 1;
(2)        else if (a.sum() < b.sum()) return -1;
(3)        for (int i = 0; i < a.size(); i++)
(4)            if (a[i] > b.[i]) return 1;
(5)            if (a[i] < b[i]) return -1;
(6)        return 0;
```

Fig. 5. The Algorithm OLMA for Construction of Lattice of Strict Ideals

If $(D(e) \subseteq H)$, then the smallest event on process k that will generate new strict ideal by taking union with H is the successor of e on process k, $succ(e)$, if it exists. Since $D(succ(e))$ contains e which is not in H, we are guaranteed that $max(H, D(succ(e)))$ is strictly greater than H. It is also a strict ideal because it corresponds to union of two strict ideals H and $D(succ(e))$.

If $(D(e) \not\subseteq H)$, then the smallest event on process k that will generate new strict ideal by taking union with H is e. We add to $\mathcal{T}$ the strict ideal $max(H, D(e))$.

This method of BFS traversal is guaranteed to explore all strict ideals greater than or equal to S as shown by the next theorem.

Theorem 2. *The Algorithm OLMA enumerates all $H \in M$ such that $L_{MA}(P')$ $= L_{MA}(P) \cup M$.*

Proof. We first show that M contains only strict ideals greater than or equal to $D(x)$. Since we enumerate (at line 8) only the vectors deleted from $\mathcal{T}$, it is sufficient to show the claim for $\mathcal{T}$. The claim is initially true because $\mathcal{T}$ initially

contains S which is a strict ideal equal to $D(x)$. For induction, we assume that H deleted at line 7 is also such a strict ideal. We add vectors only at lines 11 and 13. Since the set of strict ideals is closed under union, and both $D(e)$ and $D(succ(e))$ are strict ideals, we get that the vector added to $\mathcal{T}$ at lines 11 and 13 are also strict ideals. Both the ideals contain H due to the max operation, and hence they are greater than or equal to $D(x)$.

We now show that any strict ideal greater than or equal to $D(x)$ is added to $\mathcal{T}$ at some point in the algorithm. Let T be any strict ideal of P' greater than or equal to $D(x)$. We use induction on r, the size of $T - D(x)$. When r is zero, T equals $D(x)$, and is part of $\mathcal{T}$ due to line (5). Assume by induction that any strict ideal, T' with $|T' - D(x)| < r$ is added to $\mathcal{T}$. Let A be a minimal set such that $D(A) = T$. Since $r > 0$, there exists $y \in A$ that is not in $D(x)$. Let $T' = D(A - \{y\})$. T' must be a proper subset of T; otherwise, A is not a minimal set such that $D(A) = T$. From induction hypothesis T' is added to $\mathcal{T}$ in the algorithm. Let z be the predecessor of y on the process that contains y. We have the following cases.
Case 1: $z \in T'$. Since $T' \in \mathcal{T}$ and $z \in T'$, y would be considered at line (9) when T' is explored. Since $T' = D(A - \{y\})$ and T' is a proper subset of T, we know that $D(y) \not\subseteq T'$. By line (13), T is added to $\mathcal{T}$.
Case 2: $z \notin T'$. There are two cases to consider.
Case 2.1: $D(z) \subseteq T'$. Since $T' \in \mathcal{T}$ and $D(z) \subseteq T'$, z would be considered at line (9) when T' is explored. Then by line (11), we add $max(T', D(y))$. Therefore, T is added to $\mathcal{T}$.
Case 2.2: $D(z) \not\subseteq T'$. When T' is explored, we add to $\mathcal{T}$, $T'' := max(T', D(z))$ due to line (13). Clearly, $D(z) \subseteq T''$. Since $T'' \in \mathcal{T}$ and $D(z) \subseteq T''$, when we explore T'', we add $max(T'', D(y))$ due to line (11). Therefore, we get that T is added to $\mathcal{T}$.

Finally, we show that no vector is added to $\mathcal{T}$ again once it has been deleted. The function *levelCompare* provides a total order on all vectors. The vector H deleted is the smallest in $\mathcal{T}$. Any vector that we add due to H is strictly greater than H (either at line (11) or line (13)). Hence, once a vector has been deleted from $\mathcal{T}$ it can never be added back again. ■

We now analyze the time and space complexity of the algorithm OLMA. Lines (7) to (15) are executed for every strict ideal in M. Suppose that the number of strict ideals greater than or equal to $D(x)$ is m_x. The foreach loop at line (9) is executed w times. Computing max of two vectors at lines (11) and (13) take $O(w)$ time. Adding it to the set $\mathcal{T}$ takes $O(w \log w_L)$ time if $\mathcal{T}$ is maintained as a balanced binary tree of the vectors, where w_L is the maximum size of $\mathcal{T}$. Since $\mathcal{T}$ corresponds to BFS enumeration, it can also be viewed as the width of the lattice L in the worst case. Hence, the total time complexity for enumerating M is $O(m_x w^2 \log w_L)$. Recall that the ILMA algorithm traversed over the entire lattice when adding a new element resulting in $O(wm \log m)$ complexity for incremental construction.

We now compute the complexity of the OLMA algorithm to build the lattice for the entire poset. For simplicity, we bound m_x by m. Since the OLMA

algorithm would be called n times, the time complexity is $O(nmw^2 \log w_L)$. The space complexity of the OLMA algorithm is $O(w_L w \log n)$ bits to store the set $\mathcal{T}$ where w_L is the maximum size that $\mathcal{T}$ will take during BFS enumeration.

5 Traversal Based Algorithms for Enumerating Lattice of Maximal Antichains

In some applications (such as global predicate detection discussed in Section 6), we may not be interested in storing L_{MA} but simply enumerating all its elements (or storing only those elements that satisfy a given property). In this section, we consider the problem of enumerating all the maximal antichains of a computation in an offline manner. In the OLMA algorithm, we enumerate all strict ideals greater than or equal to $D(x)$, when x arrives. We can use the OLMA algorithm in an offline manner as well. We simply use $BFS(\{\})$ instead of $BFS(D(x))$ which enumerates all the ideals. The time complexity is $O(mw^2 \log w_L)$ and the space complexity is $O(w_L w \log n)$. We call this algorithm BFS-MA.

We now show that the space complexity can be further reduced by using DFS (depth first search) enumeration of L_{MA}. The depth first search enumeration requires storage proportional to the height of L_{MA} which is at most n.

In previous section, we had used the lattice of strict ideals instead of lattice of maximal ideals. In this section, we use the lattice of maximal ideals because the lattice of maximal ideals is closed under intersection which allows us to find the smallest maximal ideal that contains a given set (at line (3) in Fig. 6).

One of the main difficulties is to ensure that we do not visit the same maximal antichain ideal twice because we do not store all the nodes of the lattice explicitly and hence cannot use the standard technique of marking a node visited during traversal. The solution we use is similar to that used for the lattice of ideals [AV01] and the lattice of normal cuts [Gar12]. Let $pred(H)$ be the set of all maximal ideals that are covered by H in the lattice. We use the total order *levelOrder* on the set $pred(H)$. We make a recursive call on H from the maximal ideal G iff G is the biggest maximal ideal in $pred(K)$ in the total order given by *levelOrder*. To find $pred(H)$, we first note that every maximal antichain of a poset P is also a maximal antichain of its dual P^d. Hence the lattice of maximal antichains of P is isomorphic to the lattice of maximal antichains of P^d. Traversing $L_{MA}(P)$ in the upward direction (in the Hasse diagram) is equivalent to traversing $L_{MA}(P^d)$ in the backward direction.

The algorithm for DFS enumeration is shown in Fig. 6. From any maximal ideal G, we explore all enabled events to find maximal ideals with at least one additional event. There are at most w enabled events and for each event it takes $O(w^2)$ time to compute the smallest maximal ideal K at line (3). At line (4) we check if K covers G using the characterization provided by Reuter [Reu91] as follows. A maximal ideal K covers the maximal ideal G in the lattice of maximal ideals iff $(K - G) \cup (U[Maximal(G)] - U[Maximal(K)])$ induces a complete height-one subposet of P with $(K - G)$ as the maximal elements and $(U[Maximal(G)] - U[Maximal(K)])$ as minimal element. This check can be performed in $O(w^2)$ time.

```
    Algorithm DFS-MaximalIdeals(G)
    Input: a finite poset P, starting state G
    Output: DFS Enumeration of all maximal ideals of P
(1)         output(G);
(2)         for each event e enabled in G do
(3)             K := smallest maximal ideal containing Q := G ∪ {e};
(4)             if K does not cover G then go to the next event;
(5)             M := get-Max-predecessor(K) ;
(6)             if M = G then
(7)                 DFS-MaximalIdeals(K);

    function VectorClock get-Max-predecessor(K) {
    //takes K as input vector and returns the maximal ideal that is biggest in the
levelCompare order

(1)         H = maximal ideal in P^d that has the same maximal antichain
as K
(2)         // find the maximal predecessor using maximal ideals in the dual poset
(3)         for each event e enabled in the cut H in P^d do
(4)             temp := advance along event e in P^d from cut H;
(5)             // get the set of maximal ideals reachable in P^d
(6)             pred := smallest Maximal ideal containing temp that covers H
(7)         return the maximal ideal that corresponds to maxPred in P;
```

Fig. 6. Algorithm DFS-MA for DFS Enumeration of Maximal Ideals

In line (5), we traverse K using recursive call only if M equals G. Since there can be w predecessors for K and it takes $O(w^2)$ time to compute each predecessor; the total time complexity to determine whether K can be inserted is $O(w^3)$. Hence the overall time complexity of the algorithm is $O(mw^4)$.

The main space requirement of the DFS algorithm is the stack used for recursion. Every time the recursion level increases, the size of the maximal ideal increases by at least 1. Hence, the maximum depth of the recursion is n, and the space requirement is $O(nw \log n)$ bits because we only need to store vectors of dimension w at each recursion level.

6 Application of Lattice of Maximal Antichains

Global predicate detection problem has applications in distributed debugging, testing, and software fault-tolerance. The problem can be stated as follows. Given a distributed computation (either in an online fashion, or an offline fashion), and a global predicate B (a boolean function on the lattice of consistent global states), determine if there exists a consistent global state that satisfies B. The global predicate detection problem is NP-complete [CG98], even for the restricted case when the predicate B is a singular 2CNF formula of local predicates [MG01]. The key problem is that the lattice of consistent global states may

be exponential in the size of the poset. Given the importance of the problem in software testing and monitoring of distributed systems, there is strong motivation to find classes of predicates for which the underlying space of consistent global states can be traversed efficiently. The class of linear predicates [CG98] and relational predicates [TG93, IG06] are two such classes. We now describe a class called *antichain* predicates which satisfies the property that they hold on the lattice L_{CGS} iff they hold on the lattice L_{MA}. We give examples of predicates that occur in practice which belong to this class.

A global predicate B is an *antichain-consistent* predicate if its evaluation depends only on maximal events of a consistent global state and if it is true on a subset of processes, then presence of additional processes does not falsify the predicate. Formally,

Definition 7 (Antichain-Consistent Predicate). *A global predicate B defined on L_{CGS} is an* antichain-consistent predicate *if for all consistent global states G and H: $(maximal(G) \subseteq maximal(H)) \wedge B(G) \Rightarrow B(H)$.*

We now give examples of antichain-consistent predicate.

- *Violation of mutual exclusion*: Consider the predicate, B, "there is more than one process in the critical section." The relevant critical section events for this predicate are entry to the critical section and exit from the critical section. B is true in a global state G iff $maximal(G)$ has more than one critical section event. Clearly, if B is true in G and $maximal(G)$ is contained in $maximal(H)$, then it is also true in H.
- *Violation of resource usage*: The predicate, B, "there are more than k concurrent activation of certain service," a slight generalization of the previous example, is also antichain-consistent.
- *Global Control Point*: The predicate, B, "Process P_1 is at line 35 and P_2 is at line 23 concurrently," is also antichain-consistent.

We can now show the following result.

Theorem 3. *There exists a consistent global state that satisfies an antichain-consistent predicate B iff there exists a maximal ideal that satisfies B.*

Proof. Let G be a consistent global state that satisfies B. If G is a maximal ideal, we are done. Otherwise, consider G^c, the complement of G. Since G is not a maximal ideal, there exists $y \in minimal(G^c)$ such that y is incomparable to all elements in $maximal(G)$. It is easy to see that $G_1 = G \cup \{y\}$ is also a consistent global state. Furthermore, $maximal(G) \subseteq maximal(G_1)$. Since B is antichain-consistent, it is also true in G_1. If G_1 is a maximal ideal, we are done. Otherwise, by repeating this procedure, we obtain H such that $maximal(G) \subseteq maximal(H)$, and H is a maximal ideal. From the definition of antichain-consistent, we get that $B(H)$.

The converse is obvious because every maximal ideal is also an ideal. ∎

Hence, instead of constructing the lattice of ideals, we can use algorithms in Section 4 and Section 5 to detect an antichain-consistent global predicate resulting in significant reduction in time complexity.

References

[AV01] Alagar, S., Venkatesan, S.: Techniques to tackle state explosion in global predicate detection. IEEE Transactions on Software Engineering 27(8), 704–714 (2001)

[CG98] Chase, C.M., Garg, V.K.: Detection of global predicates: Techniques and their limitations. Distributed Computing 11(4), 191–201 (1998)

[CL85] Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)

[CM91] Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, pp. 163–173 (May 1991)

[DP90] Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)

[Fid89] Fidge, C.J.: Partial orders for parallel debugging. In: Proc. of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, vol. 24(1), pp. 183–194 (1989)

[Gan84] Ganter, B.: Two basic algorithms in concept analysis. Technical Report 831, Techniche Hochschule, Darmstadt (1984)

[Gar12] Garg, V.K.: Lattice completion algorithms for distributed computations. In: Proc. of Principles of Distributed Systems - 16th International Conference, OPODIS 2012 (December 2012)

[GM01] Garg, V.K., Mittal, N.: On slicing a distributed computation. In: 21st Intnatl. Conf. on Distributed Computing Systems (ICDCS 20 01), Washington, Brussels, Tokyo, pp. 322–329. IEEE (2001)

[IG06] Ikiz, S., Garg, V.K.: Efficient incremental optimal chain partition of distributed program traces. In: ICDCS, p. 18. IEEE Computer Society (2006)

[JRJ94] Jourdan, G.-V., Rampon, J.-X., Jard, C.: Computing on-line the lattice of maximal antichains of posets. Order 11, 197–210 (1994)

[Lam78] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. of the ACM 21(7), 558–565 (1978)

[Mat89] Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms: Proc. of the Intnatl. Workshop on Parallel and Distributed Algorithms, pp. 215–226. Elsevier Science Publishers B.V., North-Holland (1989)

[MG01] Mittal, N., Garg, V.K.: On detecting global predicates in distributed computations. In: 21st Intnatl. Conf. on Distributed Computing Systems (ICDCS 2001), Washington, Brussels, Tokyo, pp. 3–10. IEEE (2001)

[NR99] Nourine, L., Raynaud, O.: A fast algorithm for building lattices. Inf. Process. Lett. 71(5-6), 199–204 (1999)

[NR02] Nourine, L., Raynaud, O.: A fast incremental algorithm for building lattices. J. Exp. Theor. Artif. Intell. 14(2-3), 217–227 (2002)

[Reu91] Reuter, K.: The jump number and the lattice of maximal antichains. Discrete Mathematics 88(23), 289–307 (1991)

[TG93] Tomlinson, A.I., Garg, V.K.: Detecting relational global predicates in distributed systems. In: Proc. of the Workshop on Parallel and Distributed Debugging, San Diego, CA, pp. 21–31 (May 1993)

On the Analysis of a Label Propagation Algorithm for Community Detection*

Kishore Kothapalli[1], Sriram V. Pemmaraju[2], and Vivek Sardeshmukh[2]

[1] International Institute of Information Technology, Hyderabad, India 500 032
kkishore@iiit.ac.in
[2] Department of Computer Science, The University of Iowa, Iowa City, IA 52242-1419, USA
firstname-lastname@uiowa.edu

Abstract. This paper initiates formal analysis of a simple, distributed algorithm for community detection on networks. We analyze an algorithm that we call Max-LPA, both in terms of its convergence time and in terms of the "quality" of the communities detected. Max-LPA is an instance of a class of community detection algorithms called *label propagation* algorithms. As far as we know, most analysis of label propagation algorithms thus far has been empirical in nature and in this paper we seek a theoretical understanding of label propagation algorithms. In our main result, we define a clustered version of Erdös-Rényi random graphs with clusters $V_1, V_2, \ldots, V_k$ where the probability p, of an edge connecting nodes within a cluster V_i is higher than p', the probability of an edge connecting nodes in distinct clusters. We show that even with fairly general restrictions on p and p' ($p = \Omega\left(\frac{1}{n^{1/4-\epsilon}}\right)$ for any $\epsilon > 0$, $p' = O(p^2)$, where n is the number of nodes), Max-LPA detects the clusters $V_1, V_2, \ldots, V_n$ in just two rounds. Based on this and on empirical results, we conjecture that Max-LPA can correctly and quickly identify communities on clustered Erdös-Rényi graphs even when the clusters are much sparser, i.e., with $p = \frac{c \log n}{n}$ for some $c > 1$.

1 Introduction

The problem of efficiently analyzing large social networks spans several areas in computer science. One of the key properties of social networks is their *community structure*. A *community* in a network is a group of nodes that are "similar" to each other and "dissimilar" from the rest of the network. There has been a lot of work recently on defining, detecting, and identifying communities in real-world networks [9, 7, 25]. It is usually, but not always, the tendency for vertices to be gathered into distinct groups, or communities, such that edges between vertices in the same community are dense but inter-community edges are sparse [21, 9].

* This work was done when the first author (KK) was visiting The University of Iowa on an Indo-US Science and Technology Forum Fellowship. The work of the second author (SP) was partially supported by National Science Foundation grant CCF 0915543.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 255–269, 2013.

A *community detection* algorithm takes as input a network and outputs a partition of the vertex set into "communities". Detecting communities can allow us to understand attributes of vertices from the network topology alone.

There are many metrics to measure the "quality" of the communities detected by a community detection algorithm. A popular and widely adopted metric is *graph modularity* defined by Newman [22]. This measure is obtained by summing up, over all communities of a given partition, the difference between the observed fraction of links inside the community and the expected value of this quantity for a null model, that is, a random network having the same size and same degree sequence. Other popular measures include *graph conductance* [12] and *edge betweenness* [9].

The community detection problem has connections to the *graph partitioning* problem which has been well studied since 1970s [5, 13, 14, 26]. Graph partitioning problems are usually modeled as combinatorial optimization problems and this approach requires a precise sense of the objective function being optimized. Sometimes additional criteria such as the number of parts or the sizes of parts also need to be specified. In contrast, the notion of communities is relatively "fuzzy" [8] and changes from application to application. Furthermore, researchers in social network analysis are reluctant to over-specify properties of communities and would rather let algorithms "discover" communities in the given network. For a survey of the different approaches that have been proposed to find community structure in networks, see Fortunato's work [8].

The focus of this paper is a class of seemingly simple community detection algorithms called *label propagation* algorithms (LPA). Raghavan et al. [25] seem to be the first to study label propagation algorithms for detecting network communities. The advantage of a LPA is, in addition to its simplicity, the fact that it can be easily parallelized or distributed. The generic LPA works as follows: initially each node in the network is assigned a unique label. In each iteration every node updates its label to the label which is the most frequent in its neighborhood; ties are broken randomly. One obtains variants of LPA by varying how the initial label assignment is made, how ties are broken, and whether a node includes itself in computing the most frequent label in its neighborhood. In this paper, we analyze a specific instance of LPA called Max-LPA in which nodes are assigned initial labels uniformly at random from some large enough space. Also, if there is a tie, it is broken in favor of the larger label. Finally, a node includes its own label in determining the most frequent label in its neighborhood.

At any point during the execution of a LPA, a community is simply all nodes with the same label. The intuition behind using a LPA for community detection is that a single label (the maximum label in the case of Max-LPA) can quickly become the most frequent label in neighborhoods within a dense cluster whereas labels have trouble "traveling" across a sparse set of edges that might connect two dense clusters. A LPA is said to have *converged* if it starts cycling through a collection of states. Ideally, we would like LPA to converge to a cycle of period one, i.e., to a state in which any further execution of LPA yields the same state. However, this is not always possible. In fact, part of the difficulty of analyzing

LPA stems from the randomized tie-breaking rule. This way of breaking ties makes it difficult to estimate the period of the cycle that the algorithm eventually converges to. The version of LPA that we analyze, namely Max-LPA, does not suffer from this problem because Poljak and Sůra [24] have shown in a different context that Max-LPA converges to a cycle of period 1 or 2.

Despite the simplicity of LPA, there has been very little formal analysis of either the convergence time of LPA or the quality of communities produced by it. There have been papers [25, 16, 4] that provide some empirical results about LPAs. For example, the number of iterations of label updates required for the correct convergence of LPA is around 5 [25], but it is hard to derive any fundamental conclusions about LPA's behavior, even on specific families of networks, from these empirical results. One reason for this state of affairs is that despite its simplicity, even on simple networks, LPA can have complicated behavior, not unlike epidemic processes that model the spread of disease in a networked population [20]. Our goal in this paper is to initiate a systematic analysis of the behavior of Max-LPA, both in terms of its convergence time and in terms of the "quality" of communities produced.

Watts and Strogatz [28] have pointed out that the classical Erdös-Rényi model of random graphs differs from real-world social, technological, and biological networks in several critical ways. Following this, a variety of other random graph models have been considered as models of real-world networks. These include the *configuration* model [18, 2], the *Watts-Strogatz* model [28], *preferential attachment* models [1], etc. (for definitions and more examples, see [19]). There is no empirical study or formal analysis of LPAs on these classes of networks. As our first step towards developing analysis techniques for LPAs we define a clustered version of Erdös-Rényi random graphs and present a formal proof of the running times of LPAs on these networks. We realize that Erdös-Rényi networks and even clustered Erdös-Rényi networks are inadequate models of real world networks, but believe that our analysis techniques could be useful in general.

The variants of LPA can naturally be viewed as *distributed* algorithms, meaning each node only has *local* knowledge, i.e., knowledge of its label and the labels of its neighbors obtained by means of message passing along edges of the networks. Distributed algorithms are generally classified as *synchronous* or *asynchronous* algorithms. (The reader is referred to standard books (e.g., [23]) for a full exposition of these terms). Here we analyze a synchronous version of Max-LPA. The algorithm proceeds in rounds and in each round each node sends its label to all neighbors and then updates its label based on the labels received from neighbors and its own label.

1.1 Preliminaries

We use $G = (V, E)$ to denote an undirected connected graph (network) of size $n = |V|$. For $v \in V$, we denote by $N(v) = \{u : u \in V, (u, v) \in E\}$ the neighborhood of v in graph G, by $deg(v) = |N(v)|$ the degree of v, and by $\Delta(G) = \max_{v \in V} deg(v)$ the maximum degree over all the vertices in G. A *k-hop neighborhood* ($k \geqslant 1$) of v is defined as $N_k(v) = \{w : \text{dist}_G(w, v) \leq k\} \setminus \{v\}$.

Algorithm 1. MAX-LPA on a node v

$i = 0$
$l_v[i] \leftarrow$ random(0,1)
while true **do**
  $i++$;
  send $l_v[i-1]$ to $\forall u \in N(v)$
  receive $l_u[i-1]$ from $\forall u \in N(v)$
  $l_v[i] \leftarrow \max\left\{\ell \mid \sum_{u \in N'(v)}[\ell_u[i-1] == \ell] \geq \sum_{u \in N'(v)}[\ell_u[i-1] == \ell'] \text{ for all } \ell'\right\}$
end while

We denote the *closed neighborhood* (respectively, *closed k-hop neighborhood*) of v as $N'(v) = N(v) \cup \{v\}$ (respectively, $N'_k(v) = N_k(v) \cup \{v\}$).

Denote by $\ell_u(t)$ the label of node u just before round t. When the round number is clear from the context, we use ℓ_u to denote the current label of u. Since the number of labels in the network is finite, LPA will behave periodically starting in some round t^*, i.e., for some $p \geq 1$, $0 \leq i < p$, and $j = 0, 1, 2, \ldots$,

$$\ell_u(t^* + i) = \ell_u(t^* + i + j \cdot p)$$

for all $u \in V$. Then we say that MAX-LPA has *converged* in t^* rounds.

We now describe MAX-LPA precisely (see **Algorithm 1**). Every node $v \in V$ is assigned a unique label uniformly and independently at random. For concreteness, we assume that these labels come from the range $[0, 1]$. At the start of a round, each node sends its label to all neighboring nodes. After receiving labels from all neighbors, a node v updates its label as:

$$l_v \leftarrow \max\left\{\ell \mid \sum_{u \in N'(v)}[\ell_u == \ell] \geq \sum_{u \in N'(v)}[\ell_u == \ell'] \text{ for all } \ell'\right\}, \tag{1}$$

where $[\ell_u == \ell]$ evaluates to 1 if $\ell_u = \ell$, otherwise evaluates to 0. Note that there is no randomness in the algorithm after the initial assignments of labels.

By "w.h.p." (with high probability) we mean with probability at least $1 - \frac{1}{n^c}$ for some constant $c \geqslant 1$. In this paper we repeatedly use the following versions of a tail bound on the probability distribution of a random variable, due to Chernoff and Hoeffding [3, 11]. Let $X_1, X_2, \ldots, X_m$ be independent and identically distributed binary random variables. Let $X = \sum_{i=1}^{m} X_i$. Then, for any $0 \leq \epsilon \leq 1$ and $c \geqslant 1$,

$$\Pr\left[X > (1+\epsilon) \cdot E[X]\right] \leq \exp\left(-\frac{\epsilon^2 E[X]}{3}\right) \tag{2}$$

$$\Pr\left[X < (1-\epsilon) \cdot E[X]\right] \leq \exp\left(-\frac{\epsilon^2 E[X]}{2}\right) \tag{3}$$

$$\Pr\left[|X - E[X]| > \sqrt{3c \cdot E[X] \cdot \log n}\right] \leq \frac{1}{n^c} \tag{4}$$

1.2 Results

As mentioned earlier, the purpose of this paper is to counterbalance the predominantly empirical line of research on LPA and initiate a systematic analysis of Max-LPA. Our main results can be summarized as follows:

- As a "warm-up" we prove (Section 2) that when executed on an n-node path Max-LPA converges to a cycle of period one in $\Theta(\log n)$ rounds w.h.p. Moreover, we show that w.h.p. the state that Max-LPA converges to has $\Omega(n)$ communities.
- In our main result (Section 3), we define a class of random graphs that we call *clustered Erdös-Rényi graphs*. A clustered Erdös-Rényi graph $G = (V, E)$ comes with a node partition $\Pi = (V_1, V_2, \ldots, V_k)$ and pairs of nodes in each V_i are connected with probability p_i and pairs of nodes in distinct parts in Π are connected with probability $p' < \min_i\{p_i\}$. Since p' is small relative to any of the p_i's, one might view a clustered Erdös-Rényi graph as having a natural community structure given by Π. We prove that even with fairly general restrictions on the p_i's and p' and on the sizes of the V_i's, Max-LPA converges to a period-1 cycle in just 2 rounds, w.h.p. *and* "correctly" identifies Π as the community structure of G.
- Roughly speaking, the above result requires each p_i to be $\Omega\left(\left(\frac{\log n}{n}\right)^{1/4}\right)$. We believe that Max-LPA would correctly and quickly identify Π as the community structure of a given clustered Erdös-Rényi graph even when the p_i's are much smaller, e.g. even when $p_i = \frac{c \log n}{n}$ for $c > 1$. However, at this point our analysis techniques do not seem adequate for situations with smaller p_i values and so we provide empirical evidence (Section 4) for our conjecture that Max-LPA correctly converges to Π in $O(\text{polylog}(n))$ rounds even when $p_i = \frac{c \log n}{n}$ for some $c > 1$ and p' is just a logarithmic factor smaller than p_i.

1.3 Related Work

There are several variants of LPA presented in the literature [4, 10, 27, 17]. Most of these are concerned about "quality" of the output and present empirical studies of output produced by LPA.

Raghavan et al. [25], based on the experiments, claimed that at least 95% of the nodes are classified correctly by the end of 5 rounds of label updates. But the experiments that they carried out were on the small networks.

Cordasco and Gargano [4] proposed a semi-synchronous approach which is guaranteed to converge without oscillations and can be parallelized. They provided a formal proof of convergence but did bound the running time of the algorithm. Lui and Murata [17] presented a variation of LPA for bipartite networks which converges but no formal proof is provided, neither for the convergence nor for the running time.

Leung et al. [16] presented empirical analysis of quality of output produced by LPA on larger data sets. From experimental results on a special structured network they claimed that running time of LPA is $O(\log n)$.

2 Analysis of Max-LPA on a Path

Consider a path $\mathcal{P}_n$ consisting of vertices $V = [n]$ and edge set $E = \{(i, i+1) \mid 1 \leq i < n\}$. In this section, we analyze the execution of Max-LPA on a path network $\mathcal{P}_n$ and prove that in $O(\log n)$ rounds Max-LPA converges to a state from which no further label updates occur and furthermore in such a state the number of communities is $\Omega(n)$ w.h.p..

Lemma 1. *When* Max-LPA *is executed on path network $\mathcal{P}_n$, independent of the initial label assignment, it will converge to a state from which no further label updates occur.*

Proof. First we show that at any point in the execution of Max-LPA, the subgraph of $\mathcal{P}_n$ induced by all nodes with the same label, is a single connected component. This is true before the first round since the initial label assignment assigns distinct labels to the nodes. Suppose the claim is true just before round t. Let $S = (i, i+1, \ldots, j)$ be the subgraph of $\mathcal{P}_n$ consisting of nodes with label ℓ, just before round t of Max-LPA.

- If S contains two or more nodes then none of the nodes in S will ever change their label. Moreover, the only other nodes that can acquire label ℓ in round t are nodes $i-1$ and $j+1$. Hence, after round t, the set of nodes with label ℓ still induces a single connected component.
- If S contains a single node, say i, then the only way in which label ℓ might induce multiple connected components after round t would be if in round t: (i) node $i-1$ acquires label ℓ, (ii) node $i+1$ acquires label ℓ, and (iii) node i changes its label to some $\ell' \neq \ell$. (i) and (ii) above can only happen if ℓ is larger than the labels of nodes $i-1$ and $i+1$ just before round t. But, if this were true, then node i would not change its label in round t.

Hence, in either case the nodes with label ℓ would induce a connected component.

According to Poljak and Sůra [24], Max-LPA has a period of 1 or 2 on any network with any initial label assignment. To obtain a contradiction we suppose that Max-LPA has a period of 2 when executed on $\mathcal{P}_n$ for some n and some initial label assignment. Therefore for some $v \in V$ and some time t, $\ell_v(t+2i) = \ell$ and $\ell_v(t+2i+1) = \ell'$ for $\ell \neq \ell'$ and all $i = 0, 1, 2, \ldots$. For v to change its label from ℓ to ℓ' in a round it must be the case that $\ell < \ell'$. This is because v cannot have two neighbors with label ℓ' since ℓ' can only induce one connected component. Hence, v acquires the new label ℓ' by tie breaking. By a symmetric argument, for v to change its label from ℓ' to ℓ in the next round, it must be the case that $\ell' < \ell$. Both conditions cannot be met and we have a contradiction. □

Definition 1. *A node v is said to be k-hop maxima if its label ℓ_v is (strictly) greater than the labels of all nodes in its k-neighborhood. As a short form, we will use local maxima to refer to any node that is a 1-hop maxima.*

Let $M = \{i_1, i_2, \ldots, i_r\}$, $i_1 < i_2 \cdots < i_r$ be the set of nodes which are 2-hop maxima in $\mathcal{P}_n$ for the given initial label assignment. For any $1 \leq j < r$, nodes i_j and i_{j+1} are said to be *consecutive* 2-hop maxima.

Lemma 2. *When* Max-LPA *converges, the number of communities it identifies is bounded below by the number of 2-hop maxima in the initial label assignment.*

Proof. Since all initial node labels are assumed to be distinct, in the first round of Max-LPA every node $u \in V$ acquires a label by breaking ties. Since ties are broken in favor of larger labels, all neighbors of each $i_j \in M$ will acquire the corresponding 2-hop maxima label ℓ_{i_j}. Thus after one round of Max-LPA, for each $i_j \in M$, there are three consecutive nodes in $\mathcal{P}_n$ with label ℓ_{i_j}. None of these nodes will change their label in future rounds and hence there will be a community induced by label ℓ_{i_j} when Max-LPA converges. □

Lemma 3. *Let D be the maximum distance in $\mathcal{P}_n$ between a pair of consecutive nodes in M. Then the number of rounds that* Max-LPA *takes to converge is bounded above by $D + 2$.*

Proof. Call a node v *isolated* if its label is distinct from the labels of its neighbors. After the first round of Max-LPA each node $i_j \in M$ and its neighbors acquire label ℓ_{i_j}. Therefore, after the first round, every connected component of the graph induced by isolated nodes has size bounded above by D. We now show that in each subsequent round, the size of every connected component of size two or more will reduce by at least one. Let S be a component in the graph induced by isolated nodes, just before round t. Let i be the node with maximum label in S. Since S contains at least two nodes, without loss of generality suppose that $i+1$ is also in S. In round t, node i could acquire the label of a node outside S. If this happens i would cease to be isolated after round t. Similarly, in round t, node $i+1$ could acquire the label of a node outside S and would therefore cease to be isolated after round t. If neither of these happens in round t, then node $i+1$ will acquire the label of node i in round t and node i will not change its label. In this case, both i and $i+1$ will cease to be isolated nodes after round t. In any case, we see that the size of the component S has shrunk by at least one in round t. Thus in $D+1$ rounds $\mathcal{P}_n$ we will reach a state in which all components in the graph induced by isolated nodes have size one. Isolated nodes whose labels are larger than the labels of neighbors will make no further label updates. The remaining isolated nodes will disappear in one more round. □

Theorem 1. *When* Max-LPA *is executed on a path $\mathcal{P}_n$, it converges to a state from which no further label updates occur in $O(\log n)$ rounds w.h.p. Furthermore, in such a state there are $\Omega(n)$ communities.*

Proof. Partition $\mathcal{P}_n$ into "segments" of 5 nodes each. Let S denote the set of center nodes of these segments. The probability that a node in $\mathcal{P}_n$ is a 2-hop maxima is $\frac{1}{5}$. Therefore the expected number of nodes in S that end up being 2-hop maxima is $n/25$. Now note that for any two nodes $i, j \in S$, node i being a 2-hop maxima is independent of node j being a 2-hop maxima due to the fact that there are at least 4 nodes between i and j. Therefore, we can apply the lower tail Chernoff bound (3) to conclude that w.h.p. at least $n/50$ nodes in $\mathcal{P}_n$ are 2-hop maxima. Combining this with Lemma 2 tell us that when Max-LPA

converges, it does so to a state in which there are at least $n/50$ communities with high probability.

Now consider a contiguous sequence of k 5-node segments. The probability that none of the centers of the k segments are 2-hop maxima is $(4/5)^k$. Note that here we use the independence of different segment centers becoming 2-hop maxima. Hence, for a large enough constant c, the probability that none of the centers of $k = c \log n$ consecutive segments are 2-hop maxima is at most $1/n^2$. Using the union bound and observing that there at most n consecutive segment sequences of length k, we see that the probability that there is a sequence of $k = c \log n$ consecutive segments, none of whose centers are 2-hop maxima, is at most $1/n$. Therefore, with probability at least $1 - 1/n$ every sequence of $k = c \log n$ consecutive segments contains a segment whose center is a 2-hop maxima. It follows that the distance between consecutive 2-hop maxima is at most $5c \log n$ with probability at least $1 - 1/n$. The result follows by combining this with Lemma 3. □

The argument given here establishing a linear lower bound on the number of communities can be easily extended to graphs with maximum degree bounded by a constant. The argument bounding the convergence time depended crucially on two properties of the underlying graph: (i) degrees being bounded and (ii) number of paths of length $O(\log n)$ being polynomial in number. Thus the convergence bound can be extended to other graph classes satisfying these two properties (e.g., trees with bounded degree).

3 Analysis of Max-LPA on Clustered Erdös-Rényi Graphs

We start this section by introducing a family of "clustered" random graphs that come equipped with a simple and natural notion of a community structure. We then show that on these graphs Max-LPA detects this natural community structure in only 2 rounds, w.h.p. provided certain fairly general sparsity conditions are satisfied.

3.1 Clustered Erdös-Rényi Graphs

Recall that for an integer $n \geq 1$ and $0 \leq p \leq 1$, the Erdös-Rényi graph $G(n, p)$ is the random graph obtained by starting with vertex set $V = \{1, 2, \ldots, n\}$ and connecting each pair of vertices $u, v \in V$, independently with probability p. Let Π denote a partition $(V_1, V_2, \ldots, V_k)$ of V, let π denote the real number sequence $(p_1, p_2, \ldots, p_k)$, where $0 \leq p_i \leq 1$ for all i and let $0 \leq p' < \min_i\{p_i\}$. The *clustered Erdös-Rényi* graph $G(\Pi, \pi, p')$ has vertex set V and edges obtained by independently connecting each pair of vertices $u, v \in V$ with probability p_i if $u, v \in V_i$ for some i and with probability p', otherwise (see Figure 1). Thus each induced subgraph $G[V_i]$ is the standard Erdös-Rényi graph $G(n_i, p_i)$, where $n_i = |V_i|$.

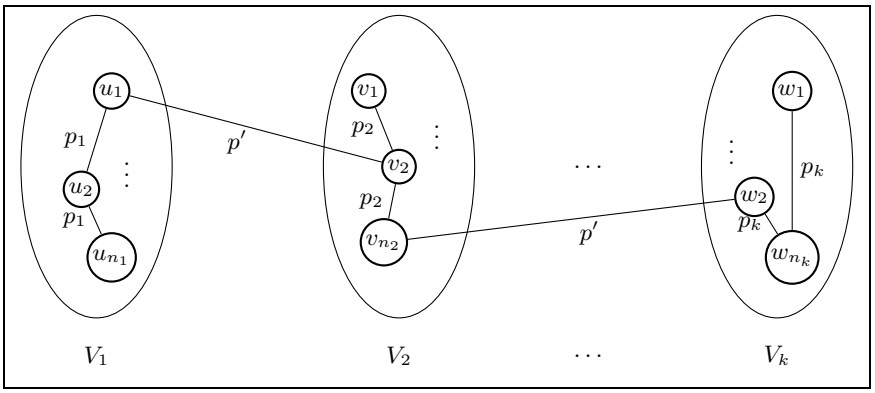


Fig. 1. The clustered Erdös-Rényi graph. We connect two nodes in the i-th ellipse (i.e., V_i) with probability p_i and nodes from different ellipses are connected with probability $p' < \min_i\{p_i\}$.

Given that $p' < p_i$ for all i, one might view $G(\Pi, \pi, p')$ as having a natural community structure given by the vertex partition Π. Specifically, when p' is much smaller than $\min_i\{p_i\}$, the inter-community edge density is much less than the intra-community edge density and it may be easier to detect the community structure Π. On the other hand as the intra-community probabilities p_i get closer to p', it may be hard for an algorithm such as Max-LPA to identify Π as the community structure. Similarly, if an intra-community probability p_i becomes very small, then the subgraph $G[V_i]$ can itself be quite sparse and independent of how small p' is relative to p_i, any community detection algorithm may end up viewing each V_i as being composed of several communities.

In the rest of the section, we explore values of the p_i's and p' for which Max-LPA "correctly" and quickly identifies Π as the community structure of $G(\Pi, \pi, p')$.

3.2 Analysis

In the following theorem we establish fairly general conditions on the probabilities $\{p_i\}$ and p' and on the node subset sizes $\{n_i\}$ and n under which Max-LPA converges correctly, i.e., to the node partition Π, w.h.p. Furthermore, we show that under these circumstances just 2 rounds suffice for Max-LPA to reach convergence!

Lemma 4. *Let $G(\Pi, \pi, p')$ be a clustered Erdös-Rényi graph such that $p' < \min_i\{\frac{n_i}{n}\}$. Let ℓ_i be the maximum label of a node in V_i. Then for any node $v \in V_i$ the probability that v is not adjacent to a node outside V_i with label higher than ℓ_i is at least $1/2e$.*

Proof. Let v' be a node in $V \setminus V_i$. Given that $|V_i| = n_i$ and ℓ_i is the maximum label among these n_i nodes, the probability that the label assigned uniformly at

random to v' is larger than ℓ_i is $1/(n_i+1)$. The probability that v has an edge to v' *and* v' has a higher label than ℓ_i is $p'/(n_i+1)$. Therefore the probability that v' has no edge to a node outside V_i with label larger than ℓ_i is

$$\left(1-\frac{p'}{n_i+1}\right)^{n-n_i}.$$

We bound this expression below as follows:

$$\left(1-\frac{p'}{n_i+1}\right)^{n-n_i} > \left(1-\frac{p'}{n_i}\right)^{n} > \left(1-\frac{1}{n}\right)^{n} > \frac{1}{2e}.$$

□

Theorem 2. *Let $G(\Pi,\pi,p')$ be a clustered Erdös-Rényi graph. Suppose that the probabilities $\{p_i\}$ and p' and the node subset sizes $\{n_i\}$ and n satisfy the inequalities:*

$$\textit{(i) } n_i p_i^2 > 8np' \qquad \textit{and} \qquad \textit{(ii) } n_i p_i^4 > 1800c\log n,$$

for some constant c. Then, given input $G(\Pi,\pi,p')$, Max-LPA *converges correctly to node partition Π in two rounds w.h.p. (Note that condition (ii) implies for each i, $p_i > \frac{\log n_i}{n_i}$ and hence each $G[V_i]$ is connected.)*

Proof. Let $V_i = \{u_1, u_2, \ldots, u_{n_i}\}$ and without loss of generality assume that $\ell_{u_1} > \ell_{u_2} > \cdots > \ell_{u_{n_i}}$. Since all initial node labels are assumed to be distinct, in the first round of Max-LPA every node $u \in V$ acquires a label by breaking ties. Since ties are broken in favor of larger labels, all neighbors of u_1 in V_i that have no neighbor outside V_i with a label larger than ℓ_{u_1} will acquire the label ℓ_{u_1}. Consider a node $v \in V_i$. Let β denote the probability that v has no neighbor outside V_i with label larger than ℓ_{u_1}. Note that inequality (i) in the theorem statement implies the hypothesis of Lemma 4 and therefore $\beta > 1/2e$. The probability that v is a neighbor of u_1 and does not have a neighbor outside V_i is $\beta \cdot p_i$. Hence, after the first round of Max-LPA, in expectation, $n_i \cdot \beta \cdot p_i$ nodes in V_i would have acquired the label ℓ_{u_1}. In the rest of the proof we will use

$$X := n_i \cdot \beta \cdot p_i.$$

Now consider node u_j for $j > 1$. For a node $v \in V_i$ to acquire the label ℓ_{u_j} it must be the case that v is adjacent to u_j, not adjacent to any node in $\{u_1, u_2, \ldots, u_{j-1}\}$, and not adjacent to any node outside V_i with a label higher than ℓ_{u_j}. Since ℓ_{u_j} is smaller than ℓ_{u_1}, the probability that v is not adjacent to a node outside V_i with label higher than ℓ_{u_j} is less than β. Thus the probability that a node in V_i acquires the label ℓ_{u_j} is at most $p_i(1-p_i)^{j-1} \cdot \beta < p_i(1-p_i) \cdot \beta$. Furthermore, the probability that a node outside V_i will acquire the label ℓ_{u_j} at the end of the first round is at most p'. Therefore, the expected number of nodes in V that acquire the label u_j, at the end of the first round, is in expectation at

most $n_i \cdot p_i(1-p_i) \cdot \beta + (n-n_i)p'$. We now use inequality (i) and the fact that $2\beta e > 1$ to upper bound this expression as follows:

$$n_i \cdot p_i(1-p_i) \cdot \beta + (n-n_i)p' < n_i \cdot p_i(1-p_i) \cdot \beta + \frac{2\beta e \cdot n_i p_i^2}{8} < n_i \cdot p_i \left(1 - \frac{3p_i}{4}\right) \cdot \beta.$$

Therefore, the expected number of nodes in V that acquire the label u_j, at the end of the first round, is in expectation at most

$$Y := n_i \cdot p_i \left(1 - \frac{3p_i}{4}\right) \cdot \beta.$$

It is worth mentioning at this point that $X - Y = n_i p_i^2 \beta/4$.

Note that all the random variables we have utilized thus far, e.g., the number of nodes adjacent to u_1 and not adjacent to any node outside V_i with label higher than ℓ_{u_1}, can be expressed as sums of independent, identically distributed indicator random variables. Hence we can bound the deviation of such random variables using the tail bound in (4). In particular, let Y' denote $Y + \sqrt{3cY \log n}$ and X' denote $X - \sqrt{3cX \log n}$. With high probability, at the end of the first round of Max-LPA, the number of nodes in V_i that acquire the label u_1 is at least X' and the number of nodes in V that acquire the label ℓ_{u_j}, $j > 1$, is at most Y'. Next we bound the "gap" between X' and Y' as follows:

$$\begin{aligned} X' - Y' &= X - Y - \sqrt{3cX \log n} - \sqrt{3cY \log n} \\ &> \frac{3n_i p_i^2 \beta}{4} - 2\sqrt{3cX \log n} \\ &> \frac{3n_i p_i^2 \beta}{4} - 2\sqrt{3cn_i p_i \beta \log n} \\ &> \frac{3n_i p_i^2 \beta}{4} - \frac{3n_i p_i^2 \beta}{5} \\ &= \frac{3n_i p_i^2 \beta}{20} \end{aligned}$$

The second inequality follows from $X - Y = 3n_i p_i^2 \beta/4$ and $Y < X$, the third from the fact that $X = n_i p_i \beta$, and the fourth by using inequality (ii) from the theorem statement.

We now condition the execution of the second round of Max-LPA on the occurrence of the two high probability events: (i) number of nodes in V_i that acquire the label u_1 is at least X' and (ii) the number of nodes in V that acquire the label ℓ_{u_j}, $j > 1$, is at most Y'. Consider a node $v \in V_i$ just before the execution of the second round of Max-LPA. Node v has in expectation at least $p_i X'$ neighbors labeled ℓ_{u_1} in V_i. Also, node v has in expectation at most $p_i Y'$ neighbors labeled ℓ_{u_j}, for each $j > 1$, in V. Let us now use X'' to denote the quantity $p_i X' - \sqrt{3cp_i X' \log n}$ and Y'' to denote the quantity $p_i Y' + \sqrt{3cp_i Y' \log n}$. By using (4) again, we know that w.h.p. v has at least X'' neighbors with label ℓ_{u_1} and at most Y'' neighbors with a label ℓ_{u_j}, $j > 1$. We will now show that $X'' > Y''$ and this will guarantee that in the second round

of Max-LPA v will acquire the label ℓ_{u_1}, with high probability. Since v is an arbitrary node in V_i, this implies that all nodes in V_i will acquire the label ℓ_{u_1} in the second round of Max-LPA w.h.p.

$$\begin{aligned} X'' - Y'' &= p_i(X' - Y') - \sqrt{3cp_iX' \log n} - \sqrt{3cp_iY' \log n} \\ &> \frac{3n_ip_i^3}{20} - 2\sqrt{3cp_iX' \log n} \\ &> \frac{3n_ip_i^3}{20} - 2\sqrt{3cn_ip_i^2\beta \log n} \\ &> \frac{3n_ip_i^3}{20} - \frac{n_ip_i^3\beta}{10} \\ &= \frac{3n_ip_i^2}{20} \\ &> 0 \end{aligned}$$

The second inequality follows from the bound on $X' - Y'$ derived earlier and $Y' < X'$, the third from the fact that $X' < n_ip_i\beta$, and the fourth by using inequality (ii) from the theorem statement.

Thus at the end of the second round of Max-LPA, w.h.p., every node in V_i has label ℓ_{u_1}. This is of course true, w.h.p., for all of the V_i's. Now note that every node $v \in V_i$ has, in expectation n_ip_i neighbors in V_i and fewer than np' neighbors outside V_i. Inequality (i) implies that $np' < n_ip_i/8$ and inequality (ii) implies that $n_ip_i = \Omega(\log n)$. Pick a constant $\epsilon > 0$ such that $n_ip_i(1+\epsilon)/8 < n_ip_i(1-\epsilon)$. By applying tail bound (2), we see that w.h.p. v has more than $n_ip_i(1-\epsilon)$ neighbors in V_i and fewer than $n_ip_i(1+\epsilon)/8$ neighbors outside V_i. Hence, w.h.p. v has no reason to change its label. Since v is an arbitrary node in an arbitrary V_i, w.h.p. there are no further changes to the labels assigned by Max-LPA. □

To understand the implications of Theorem 2 consider the following example. Suppose that the clustered Erdös-Rényi graph has $O(1)$ clusters and each cluster had size $\Theta(n)$. In such a setting, inequality (ii) from the theorem simplifies to requiring that each $p_i = \Omega((\log n/n)^{1/4})$ and inequality (ii) simplifies to $p' < p_i^2/c$ for all i. This tells us, for instance, that Max-LPA converges in just two rounds on a clustered Erdös-Rényi graph in which each cluster has $\Theta(n)$ vertices and an intra-community probability of $\Theta(1/n^{1/3})$ and the inter-community probability is $\Theta(1/n^{2/3})$.

This example raises several questions. If we were willing to allow more time for Max-LPA to converge, say $O(\log n)$ rounds, could we significantly weaken the requirements on the p_i's and p'. Specifically, could we permit an intra-community probability p_i to become as small as $c \log n/n$ for some constant $c > 1$? Similarly, could we permit the inter-community probability p' to come much closer to the smallest p_i, say within a constant factor.

We believe that it may be possible to obtain such results, but only via substantively different analysis techniques.

4 Empirical Results on Sparse Erdös-Rényi Graphs

In the previous section we proved that if the clusters (each V_i) in a clustered Erdös-Rényi graphs were dense enough and the inter-cluster edge density (fraction of edges between nodes in different V_i) was relatively low, then Max-LPA would correctly converge in just 2 rounds. Specifically, our result requires each cluster to be Erdös-Rényi random graph $G(n,p)$ with $p = O\left(\left(\frac{\log n}{n}\right)^{1/4}\right)$. In this section we ask: how does Max-LPA behave if individual clusters are much sparser? For example, how does Max-LPA behave on $G(n,p)$ with much smaller p, say $p = \frac{c \cdot \log n}{n}$ for some $c > 1$. The proof technique used in the previous section does not extend to such small values of p. However, we believe that Max-LPA converges quickly and correctly even on clustered Erdös-Rényi graphs whose clusters are of the type $G(n,p)$ for $p = \frac{c \cdot \log n}{n}$ for $c > 1$. In this section, we ask (and empirically answer) two questions:

1. Can one expect there to be a constant c such that Max-LPA, when run on $G(n,p)$ with $p \geq \frac{c \log n}{n}$ will, with high probability, terminate with one community. If the answer to Question 1 is "yes" what might the running time of Max-LPA, as a function of n be for appropriate values of p.
2. Consider a clustered Erdös-Rényi graph with two parts V_1 and V_2 of equal size, and each $p_i = \frac{c \log n}{n}$ for some $c > 1$. Let $p' = \frac{c'}{n}$ for some c'. Are there constants c, c' for which Max-LPA will quickly converge and correctly identify (V_1, V_2) as the community structure?

We are interested in values of p of the form $\frac{c \cdot \log n}{n}$ because $\frac{\log n}{n}$ is the threshold for connectivity in Erdös-Rényi graphs [6]. For the details about the simulation setup and results reader is referred to the full version of this paper [15].

We observed in our simulations that Max-LPA when executed on Erdös-Rényi graphs with $p = \frac{c \log n}{n}$ and $c > 1$, with high probability, terminate with one community. It also seems to be the case that as c increases, we are getting more single community runs. We also observed that the running time seems to grow in a linear fashion with logarithm of graph size. Also as c increases the running time decreases, which implies that as the graph becomes more dense Max-LPA converges more quickly to a single community. Our results lead us to conjecture that when Max-LPA is executed on Erdös-Rényi graphs $G(n,p)$ with $p = O(\frac{\log n}{n})$ it will, with high probability, terminate with a single community in $O(\log n)$ rounds. To answer the second question mentioned above, we executed Max-LPA on $G(\Pi, \pi, p')$ with $\Pi = (V_1, V_2)$, $|V_1| = |V_2| = n/2$, $\pi = (p, p)$, $p' = 0.6/n$ for various values of n and p. We observed that as p increases, Max-LPA converges and correctly identifies (V_1, V_2) as the community structure.

5 Future Work

We believe that with some refinements, the analysis technique used to show $O(\log n)$-rounds convergence of Max-LPA on paths, can be used to show

poly-logarithmic convergence on sparse graphs in general, e.g., those with degree bounded by a constant. This is one direction we would like to take our work in.

At this point the techniques used in Section 3 do not seem applicable to more sparse clustered Erdös-Rényi graphs. But if we were willing to allow more time for Max-LPA to converge, say $O(\log n)$ rounds, could we significantly weaken the requirements on the p_i's and p'? Specifically, could we permit an intra-community probability p_i to become as small as $c \log n/n$ for some constant $c > 1$? Similarly, could we permit the inter-community probability p' to come much closer to the smallest p_i, say within a constant factor? This is another direction for our research.

Acknowledgments. We would like to thank James Hegeman for helpful discussions and for some insightful comments.

References

[1] Albert, R., Barabási, A.-L.: Statistical mechanics of complex networks. Rev. Mod. Phys. 74, 47–97 (2002)
[2] Bender, E.A., Rodney Canfield, E.: The asymptotic number of labeled graphs with given degree sequences. Journal of Combinatorial Theory, Series A 24(3), 296–307 (1978)
[3] Chernoff, H.: A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations. The Annals of Mathematical Statistics 23(4), 493–507 (1952)
[4] Cordasco, G., Gargano, L.: Community detection via semi-synchronous label propagation algorithms. In: 2010 IEEE International Workshop on Business Applications of Social Network Analysis (BASNA), pp. 1–8. IEEE (2010)
[5] Elsner, U.: Graph Partitioning - A Survey (1997)
[6] Erdős, P., Rényi, A.: On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci. 5(17) (1960)
[7] Flake, G.W., Lawrence, S., Giles, C.L.: Efficient identification of web communities. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 150–160. ACM (2000)
[8] Fortunato, S.: Community detection in graphs. Physics Reports 486(3-5), 75–174 (2010)
[9] Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proceedings of the National Academy of Sciences 99(12), 7821 (2002)
[10] Gregory, S.: Finding overlapping communities using disjoint community detection algorithms. Complex Networks, 47–61 (2009)
[11] Hoeffding, W.: Probability inequalities for sums of bounded random variables. J. Amer. Statist. Assoc. 58, 13–30 (1963)
[12] Kannan, R., Vempala, S., Vetta, A.: On clusterings: Good, bad and spectral. J. ACM 51(3), 497–515 (2004)
[13] Karger, D.R.: Minimum cuts in near-linear time. J. ACM 47(1), 46–76 (2000)
[14] Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell System Technical Journal 49(2), 291–307 (1970)
[15] Kothapalli, K., Pemmaraju, S.V., Sardeshmukh, V.: On the analysis of a label propagation algorithm for community detection. arXiv preprint (2012)

[16] Leung, I.X.Y., Hui, P., Lio, P., Crowcroft, J.: Towards real-time community detection in large networks. Arxiv preprint arXiv:0808.2633 (2008)
[17] Liu, X., Murata, T.: How does label propagation algorithm work in bipartite networks? In: Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, vol. 03, pp. 5–8. IEEE Computer Society (2009)
[18] Molloy, M., Reed, B.: A critical point for random graphs with a given degree sequence. Random Structures & Algorithms 6(2-3), 161–180 (1995)
[19] Newman, M.E.J.: Networks: An Introduction. OUP Oxford (2010)
[20] Newman, M.E.J.: The spread of epidemic disease on networks. Physical Review Letters 66, 16128 (2002)
[21] Newman, M.E.J.: The Structure and Function of Complex Networks. SIAM Review 45(2), 167–256 (2003)
[22] Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Physical Review E 69(2), 26113 (2004)
[23] Peleg, D.: Distributed computing: a locality-sensitive approach, vol. 5. Society for Industrial Mathematics (2000)
[24] Poljak, S., Sura, M.: On periodical behaviour in societies with symmetric influences. Combinatorica 3(1), 119–121 (1983)
[25] Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Physical Review E 76(3), 36106 (2007)
[26] Suaris, P.R., Kedem, G.: An algorithm for quadrisection and its application to standard cell placement. IEEE Transactions on Circuits and Systems 35(3), 294–303 (1988)
[27] Šubelj, L., Bajec, M.: Unfolding network communities by combining defensive and offensive label propagation. In: In Proceedings of the ECML PKDD Workshop on the Analysis of Complex Networks 2010 (ACNE 2010), pp. 87–104 (March 2011)
[28] Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature 393, 440–442 (1998)

How to Survive and Thrive in a Private BitTorrent Community

Adele Lu Jia[1], Xiaowei Chen[2], Xiaowen Chu[2],
Johan A. Pouwelse[1], and Dick H.J. Epema[1]

[1] Department of Computer Science,
Delft University of Technology, The Netherlands
[2] Department of Computer Science,
Hong Kong Baptist University, Hong Kong

Abstract. Many private BitTorrent communities employ Sharing Ratio Enforcement (SRE) schemes to incentivize users to contribute. It has been demonstrated that communities that adopt SRE are greatly oversupplied, i.e., they have much higher seeder-to-leecher ratios than communities in which SRE is not employed. Most previous studies focus on showing the positive effect of SRE in achieving high downloading speed. However, in this paper we show through measurements that SRE also induces severe side-effects. Under SRE, users are forced to seed for excessively long times to maintain adequate sharing ratios to be able to start new downloads, though most of the time their seedings are not very productive (in terms of low upload speed). We also observe that many users who seed for very long times still have low sharing ratios. We find that this is due to the counter-intuitive phenomenon that long seeding times do not necessarily lead to large upload amounts. Based on these observations, we discuss possible strategies for users to gain sharing ratios efficiently, which help them to survive and thrive in private communities.

1 Introduction

BitTorrent is a popular Peer-to-Peer (P2P) protocol for file distribution. A key of its success lies in its Tit-For-Tat (TFT) incentive policy, which works reasonably well in fostering cooperation among downloading peers [1] (also known as *leechers*). However, TFT does not provide any incentive for peers to remain in the system after the download is complete, in order to *seed* the entire file to others. Therefore, peers are free to engage in "Hit and Run" behavior, the scenario under which a peer leaves immediately upon completing a download.

To provide incentives for seeding, in recent years there has been a proliferation of so-called *private* BitTorrent communities. While most previous works [1–4] focus on demonstrating the positive effects of the high seeder-to-leecher ratios,

[1] In this paper, we use *user* and *peer* alternatively to refer to the individuals in a private community.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 270–284, 2013.

i.e., oversupply, achieved in these communities, in this paper we show through measurements that oversupply also induces severe adverse effects.

Most private communities employ a private-tracker-based method that maintains centralized accounts and records the *sharing ratio* of each peer, i.e., the ratio between its total amount of upload and download. Community administrators specify some *threshold* above which all members are required to maintain their sharing ratios. This mechanism is known as *Sharing Ratio Enforcement* (SRE). Community members whose sharing ratios drop below the threshold are warned and then banned from downloading, or even expelled from the community. In this way, it is guaranteed that each peer provides a certain level of contribution to the community.

The main motivation for implementing SRE is to close the gap between bandwidth demand and supply as observed in public communities, where there is significantly more demand than supply [3]. Thus, the basic design goal of SRE is to achieve higher system-wide downloading speed by increasing the bandwidth supply. Several measurement studies have shown that SRE is very effective in increasing the bandwidth supply [1–5]. For instance, [3] reports seeder-to-leecher ratios that are at least 9 times higher in private communities than in public ones, while downloading speeds are found to be 3-5 times higher. However, is high downloading speed the unique and ultimate design goal of private communities? Do all the users enjoy it without experiencing any adverse effects?

To answer these questions, in this paper we analyze SRE's performance through measurements based on an extended range of metrics including the seeding time, the upload speed, and the evolution of sharing ratio. These metrics are highly related to user experience, but have never been considered before in previous measurement work. Our previous work [6,7] has dealt with analyzing the pros and cons of SRE schemes based on theoretical models and simulations. This paper complements these works by presenting observations from real world communities. The main contributions of this paper are as follows:

- We perform a measurement study of three private communities that provide user-level information including the upload amount, download amount, seeding time, leeching time, and sharing ratio of *each* user. Among the tens of existing communities we have examined, they are the only ones that provide such detailed information.
- We use one of the three communities as an example, and we show that while users achieve very high downloading speeds, to maintain adequate sharing ratios they have to seed for excessively long times (compared to their downloading times), though most of the time their seedings are not very productive due to the oversupply induced by SRE.
- For users who intend to increase their sharing ratios to survive and thrive in a private community, our analysis shows that seeding for longer durations is not as effective as increasing the upload speed. In order to do so, besides upgrading the internet access, users can also try to avoid the oversupply by joining swarms in their early stages.

– Given the possible user strategies against SRE, we further analyze the existing strategies adopted by administrators in private communities, which are initially designed to further incentivize contribution. We show that some strategies have limited or even negative effects on the performance and we propose our remedies.

2 Methodology

In order to obtain a better understanding of private BitTorrent communities it is critical to be able to collect data on their operation. Over the years it has been proven to be a challenge to obtain detailed traces of user behavior, due to a combination of technical constraints and privacy concerns. For instance, prior work was never able to capture both detailed user profiles, content availability, and precise information on every user download.

To support our analysis, we have examined 38 elite private communities, out of which three communities (CHDBits [8], ChinaHDTV [9], and HDStar [10]) were selected for detailed regular deep crawling of HTML pages. These specific three communities were selected as they are the only ones that provide information detailed enough for our analysis. We have obtained the following three datasets for each community:

1. **Community-level user profile:** in this dataset, we crawl the profile page of *each* community user and obtain the information of its upload and download amount, its seeding and leeching time, its sharing ratio at the time of snapshot, and the time it joined the community.
   It should be noted that the seeding time of each user recorded by the tracker is *swarm-based*, i.e., simultaneously seeding in multiple swarms counts separately. For instance, after a user has seeded in two swarms for 10 hours, $2 \times 10 = 20$ hours will be added to its seeding time. Similarly, the leeching time recorded by the tracker is also swarm-based. In later sections, when we calculate the average upload speed of a user, we calculate its per-swarm average upload speed, i.e., the total upload amount divided by the swarm-based seeding time. In this way, we get a rough estimation of a user's seeding time and upload speed. Though more accurate calculation of the seeding time and upload speed would be better, to the best of our knowledge, until now no private communities provide this information and it is also impossible to deploy a client and contact every user individually to get this information.
2. **Community-level torrent profile:** in this dataset, we crawl the community trackers and collect information of *each* torrent, including the number of seeders and leechers, the number of finished downloads at the time of the snapshot, and the time the torrent was published.
3. **Torrent-level user activity:** the tracker records a user's torrent-level action times, such as the time of joining the swarm, the time of starting seeding, etc. The precision of the recorded action time decreases with time. For example, if a user started to seed 10 hours ago, its action time will be "10

hours ago". However, if a user started to seed one month 23 days and 10 hours ago, its action time will only be "one month and 23 days ago".

In order to obtain the action times with precision in hours, we examine all the torrents released within 24 hours and choose the one with the largest number of participants. We follow this torrent for 7 days and record the activity of *each* user who has participated or is participating in it. The collected information includes each user's upload amount, download amount, seeding time, and leeching time in this swarm, as well as the time it joins and leaves the swarm.

We analyze the three communities we measured in detail. Due to the limit of space, we only demonstrate the results of CHDBits. We collected the data in May, 2011. At the time of our measurement, CHDBits had 31,547 registered users, 40,040 torrents, and a total download amount of 24.3 PB. For dataset 1 and 2, information of all the users and torrents are collected. For dataset 3, information of 3,776 users attracted to the chosen torrent is collected. Results about the other two communities can be found in our technical report [11]. The datasets of the other two communities demonstrate similar performance as that of CHDBits.

3 A General View: The Rich Are Rich and the Poor Are Poor

Fig. 1 shows the CDF of the user's sharing ratio in CHDBits (dataset 1). We see that around 15% users have sharing ratios less than 1 (considered as the *poor*), while around 18% users have sharing ratios larger than 5 (considered as the *rich*). The behavior of accumulating a large sharing ratio may be triggered by various motivations, such as altruism, a desire to be part of the rich elite of the community, or a habit of saving sharing ratio for the future. The rich peers have little worry about staying in the community, since their sharing ratios are far beyond the SRE threshold, which for CHDBits is 0.7. On the other hand, poor peers are at the risk of being expelled from the community. As a consequence, they need to be concerned a lot about their decisions: they may download new contents they really desire, but this might reduce their sharing ratios to a more risky level.

One may argue that the poor peers are free-riders, who intend to keep low and risky sharing ratios that are just enough to stay in the community. However, the highly restricted membership in private communities, especially in CHDBits and many other private communities where new members can only join by a limited number of invitations, makes it very difficult to get a new membership. Hence, we conjecture that not all poor peers are strategic and psychologically strong enough to face being expelled from the community due to insufficient sharing ratios. Interestingly, as we will show in the following sections, the poverty is partially induced by the fact that the poor peers are not strategic enough.

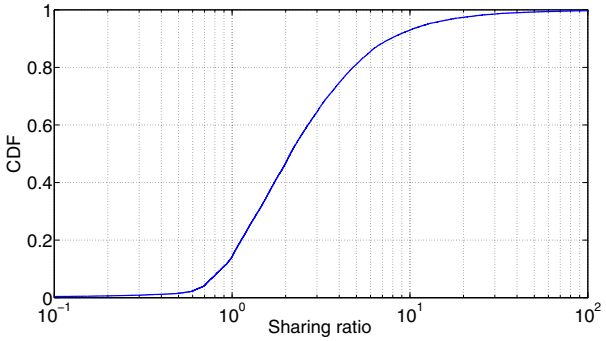


Fig. 1. The CDF of user's sharing ratio

4 Long Seeding Time: The Expense of High Downloading Speed

Many previous studies have shown that under SRE, users seed for long durations [1–4]. They consider this as a positive effect of SRE since long seeding durations lead to high downloading speeds. However, in this section we argue that the long seeding durations can also be seen as a negative effect, especially for poor peers.

4.1 Long Seeding Times, Even for Poor Peers

Fig. 2 shows the CDF of user's seeding time and leeching time in CHDBits (dataset 1). Consistent with the theoretical results of our previous work [7], in general the seeding time is much longer than the leeching time: the median leeching time is 70 days while the median seeding time is 1,100 days. Remember that the seeding and leeching time of users are swarm-based, leading to very high values.

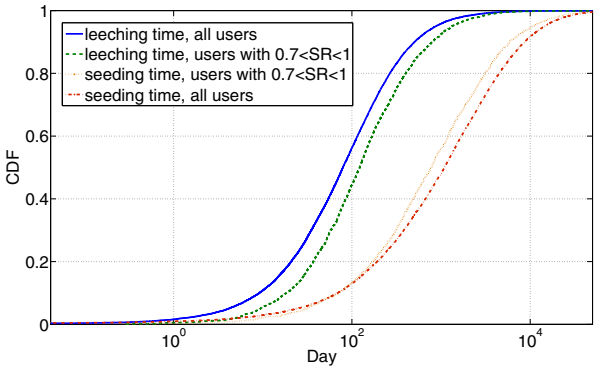


Fig. 2. The CDF of user's seeding and leeching time

Intuitively, longer seeding times than leeching times for rich peers (in terms of high sharing ratios) are to be expected, since the rich peers are saving sharing ratios by seeding. However, we observe from Fig. 2 that, even though the poor peers in general seed slightly shorter than rich peers, they still seed much longer than they leech. While intuitively poor peers should be the ones that are not "hard-working" enough, why do some of them seed for long durations but still have low sharing ratios? In the following section, we explore the possible reasons.

4.2 Possible Reasons?

One may argue that the long seeding times of poor peers are due to the fact that even though they contribute more, they also consume more. Hence, they seed for long durations but they still have low sharing ratios. This argument is partially true. Andrade *et al.* [12] have shown and we also observe from our measurement (Fig. 3, dataset 1) that the individual upload amount (contribution) increases with the corresponding download amount (consumption), with the Spearman's rank correlation coefficient equal to 0.8110. Spearman's rank correlation coefficient assesses how well the relationship between two variables can be described using a monotonic function [13]. However, this doesn't necessarily mean that heavy contributions induce long seeding times, nor does it mean that long seeding times lead to heavy contributions.

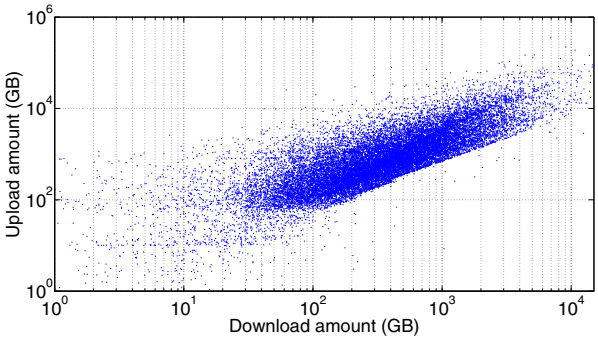
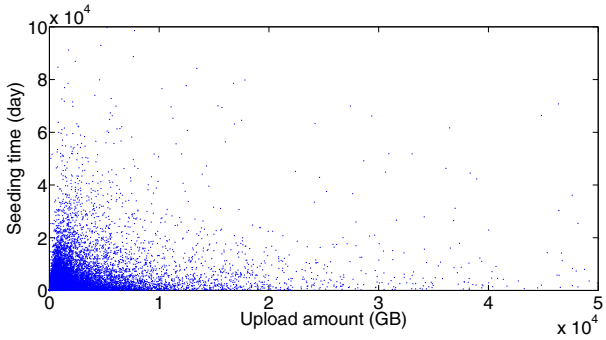
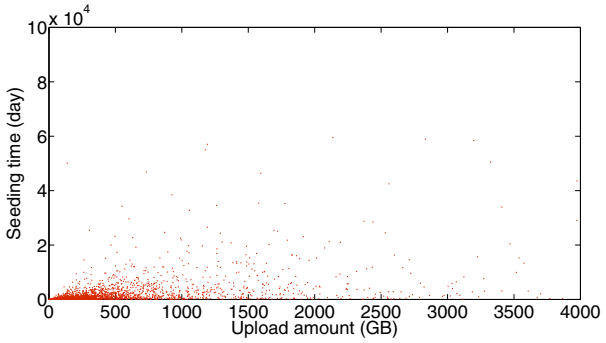


Fig. 3. Upload amount vs. download amount (with Spearman's rank correlation coefficient equal to 0.8110)

Quite counter-intuitively, as shown in Fig. 4(a), a peer's upload amount has little relation to its seeding time: many peers seed for long durations but only have uploaded relatively small amounts of data, while other peers seed for relatively short durations but have successfully achieved large upload amounts. The same argument is also applicable to poor peers (Fig. 4(b)). This interesting phenomenon implies that for poor peers who intend to increase their upload amount to become rich, seeding for longer durations may not be an effective method, even if intuitively it seems so.



(a) for all users



(b) for users with $0.7 \leq SR \leq 1$

Fig. 4. Seeding time vs. upload amount

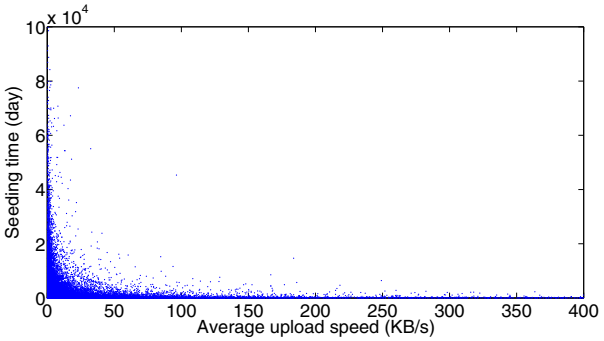
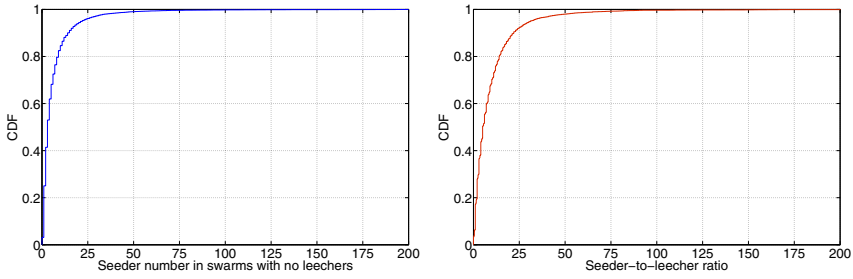


Fig. 5. Seeding time vs. upload speed (with Spearman's rank correlation coefficient equal to -0.6318)

Though there is no strict relationship between a peer's seeding time and its upload amount, we do observe that a peer's seeding time is related to its average upload speed, regardless of its upload amount. As shown in Fig. 5, most of the long seeding durations happen to the peers with relatively small upload speeds, and for peers who have high upload speeds, the seeding times are normally short.

The most intuitive reason for a low upload speed is a limited internet access. However, we argue that this is not the only reason. From dataset 2, at the time when we crawled the site, CHDBits had 33,041 active swarms (with at least one leecher or one seeder), among which 26,402 swarms (79.9%) had no leechers at all. As shown in Fig. 6(a), 40% of the swarms with no leechers still have at least 5 seeders, and 5% of these swarms even have more than 20 seeders. For swarms with at least 1 leecher, the seeder-to-leecher ratio (SLR) is quite high: as shown in Fig. 6(b), 50% of these swarms have as SLR larger than 6, and 5% of these swarms even have as SLR larger than 30. We see clearly that a majority of the swarms in CHDBits are heavily oversupplied. In such swarms, seeders are not able to perform any actual uploads due to the insufficient demand and unsatisfied supply. We term this situation *unproductive seeding*. As a consequence, users have to seed for excessively long durations to achieve the sharing ratio required by SRE.



(a) The CDF of the number of seeders in swarms with no leechers

(b) The CDF of seeder-to-leecher ratio in swarms with at least one leecher

Fig. 6. Oversupply in CHDBits swarms

While a low upload speed mainly leads to a long seeding time, in the next section we show its influence on a user's status. We analyze the reasons for the poor being poor and discusses strategies for users to become rich efficiently.

5 Why the Poor Are Poor and How to Become Rich?

As the sharing ratio is defined as the ratio between a peer's upload and download amount, two possible reasons for a peer being poor are that it has downloaded too much or has uploaded not enough. The download amount depends on a user's interests in contents. We do not suggest users to download less so as to

become rich, since the fundamental user experience that should be guaranteed by communities is that users should not need to limit their download needs. Following this argument, in this section we focus on the user upload activity and analyze why some users have uploaded not enough (hence, are poor) and how they can improve it (to become rich).

5.1 Community level

In Section 4.2 we have shown that the seeding time has little influence on the upload amount but the upload speed does. The upload speed further influences whether a user is rich or poor. As shown in Fig. 7 (dataset 1), in general rich peers ($SR \geq 5$) have much higher upload speeds than poor peers ($SR \leq 1$). For example, 80% of the poor peers upload at a speed less than 20 KB/s, while at least 40% rich peers can upload at a speed larger than 50 KB/s. Together with the result in Section 4.2, we conclude that instead of seeding for longer durations, peers who intend to become rich should seed with higher upload speeds. And to seed with a higher upload speed, a user could upgrade its internet access or choose a swarm that is less oversupplied.

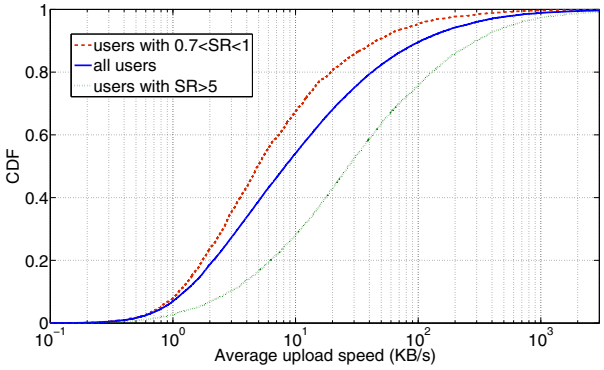


Fig. 7. The CDF of user's average upload speed in CHDBits

One may argue that the above analysis is based on community-level activities, which only provide a macroscopic view that is not enough to show the underlying details. To explore this, in the following subsection we focus on a single swarm and demonstrate the torrent-level user performance, and we discuss possible strategies for users to become rich.

5.2 Torrent Level

Different Individual Upload Amount in One Swarm: Fig. 8 shows the CDF of the user upload amount in a single swarm (dataset 3), from which we

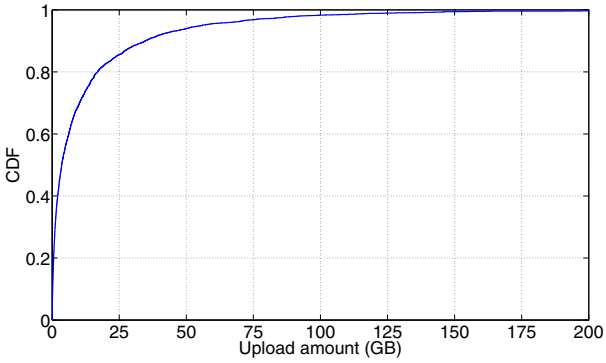


Fig. 8. The CDF of user's upload amount in one swarm

observe that a small fraction of users have uploaded considerably more than the others. For example, 60% of the users have uploaded less than 10 GB, which is less than the amount they have downloaded (11.7 GB). On the other hand, 5% of the users have uploaded more than 50 GB. Of course, the users who managed to upload more will become richer. While these users have participated in the very same swarm, why did some manage to gain a lot while others didn't?

Possible Reasons and How to Gain More: One intuitive reason for a small upload amount is a short seeding time. However, similar to the analysis in Section 4.2, again we find the counter-intuitive result that in one swarm a peer's upload amount is not related to its seeding time (Fig. 9). On the other hand, it is related to its upload speed. As shown in Fig. 10, most of the small upload amounts happen to the peers with relatively low upload speeds, and peers with high upload speeds normally have uploaded a large amount.

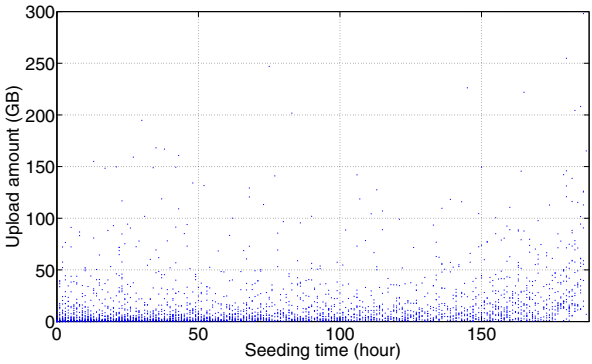


Fig. 9. Upload amount vs. seeding time in one swarm

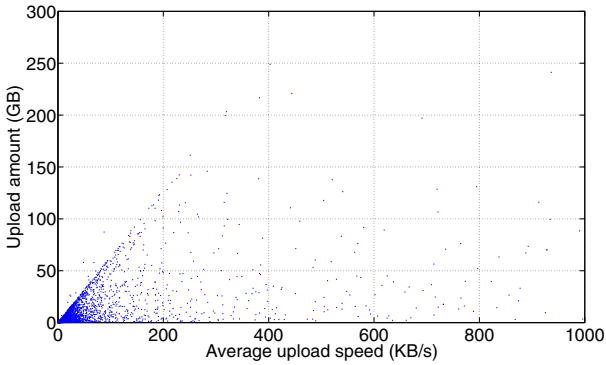


Fig. 10. Upload amount vs. upload speed in one swarm (with Spearman's rank correlation coefficient equal to 0.7876)

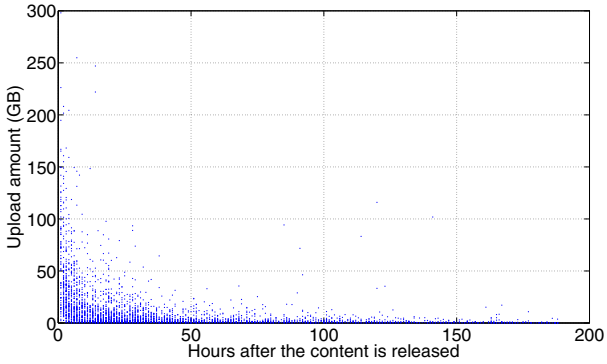


Fig. 11. Upload amount vs. time of starting seeding in one swarm (with Spearman's rank correlation coefficient equal to −0.6491)

When we organize the peers according to the time they start to seed, we find another interesting phenomenon: peers that start to seed earlier normally have uploaded more (Fig. 11). The same phenomenon has also been observed by Kash *et al.* in [14]. One may argue that the peers who start to seed earlier can seed for longer durations, hence they upload more. However, in Fig. 9 we already show that the upload amount is not related to the seeding time. Then why do peers that start to seed earlier upload more?

As shown in Fig. 12(a), after the burst at the first two hours since the file was published, the peer arrival rate decreases dramatically. On the other hand, the number of seeders increases quickly at the first 60 hours, then decreases with a much smaller rate (Fig. 12(b)). In general, the number of leechers is negligible compared to the number of seeders. As a consequence, peers who join late have to compete with a large number of seeders for uploading, which leads to a low

upload speed, and hence a small upload amount. Therefore, peers who intend to become richer should join the swarm in it's early stage, when it is still not extremely oversupplied.

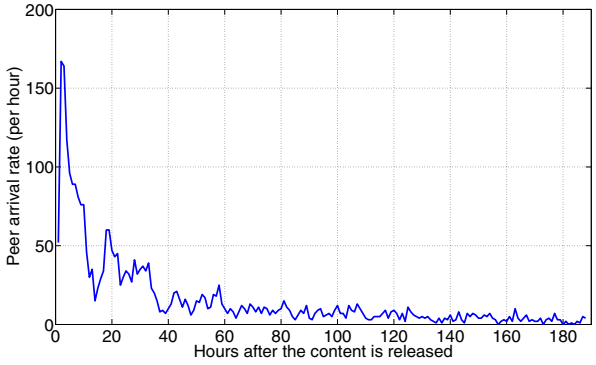
6 Discussion

Though altruistic users always exist, we conjecture that most users in private communities are *selfish*. Their initial goal in a community is to download all the contents they are interested in. To achieve this, while not limiting their download needs, they always try to increase their sharing ratios when it is possible. The strategies they apply mainly optimize their own benefit, without considering the *social welfare*, i.e., the performance of other users. For example, users may seed all the files they have downloaded to increase the opportunity of performing some actual uploading during seeding. However, this directly increases the bandwidth supply and makes the upload competition even more severe. As we discussed in Section 5, joining swarms earlier helps users gain sharing ratios more efficiently. However, if a majority of users strategically join the swarm immediately after a new content is published, then 1) many users will download something they don't want, only for gaining sharing ratios; 2) the downloading speed in the early stage of a swarm will be very low, because a large number of strategic users joining simultaneously makes the swarm heavily *flash-crowded*[2]; and 3) it will be more difficult to perform any actual uploads after the early stage, since only a few non-strategic users will join the swarm during that period.

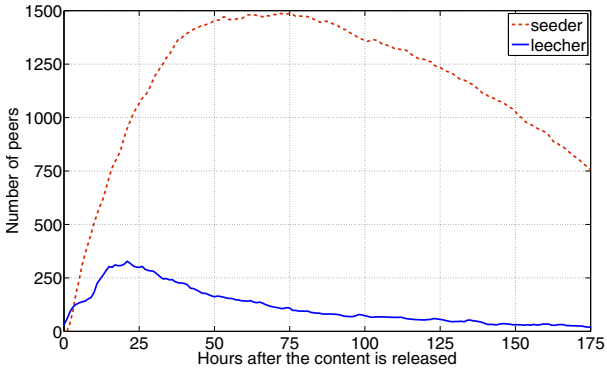
Private community administrators that intend to adopt strategies, or remedies, to alleviate the side-effects of SRE, should take the potential strategic user behaviors into account. For example, some private communities try to further incentive contribution beyond SRE by giving rich peers the priority to access the newly published contents [15]. However, as discussed previously, joining early in a new swarm will help the users, especially the poor users, gain sharing ratios more efficiently. By giving the priority to the rich peers, administrators are basically taking the opportunities away from the poor peers for gaining sharing ratios. Unless the administrators intend to let the rich be richer and the poor be poorer (which will lead to a more intense competition and a potential deterioration of performance as discussed previously), we suggest administrators to remove these restrictions.

Another example would be *free-leech* and *seeding-bonus*. Some communities [8–10,16] temporarily adopt free-leech and/or seeding-bonus for certain swarms, which means that a user can download the file for free and/or get extra bonus for seeding. Under free-leech periods, users are attracted to those swarms because of the low price for downloading. In this way, the bandwidth demand is increased and the oversupply in the system is alleviated. Meanwhile, when free-leech is applied to an relatively old swarm, the benefit of joining early is also reduced. The same argument is also applicable to seeding-bonus. Under seeding-bonus

[2] We refer a swarm to be flash-crowded when it has a sudden increase in the number of leechers.



(a) Peer arrival rate (per hour)



(b) Evolution of leechers and seeders

Fig. 12. Performance in one swarm

periods, peers are attracted to the swarms to seed. Hence, when seeding-bonus is applied to old swarms, the file availability is improved. However, administrators should be careful and not adopt free-leech or seeding-bonus for a long time, otherwise strategic users might wait and not download anything until the files are for free, or only seed in swarms with seeding bonus.

In our previous work [7], we propose a self-organizing strategy named *SRE with supply-based price* that prevents this potential manipulation of strategic users. Instead of manually adopting free-leech (i.e., zero price), this strategy inversely relates the price for downloading one unit of data to the seeder-to-leecher ratio in the swarm. With a larger seeder-to-leecher ratio, i.e., an increasing supply, SRE with supply-based price automatically decreases the price. Once the supply goes tight again, it will automatically increase the price. In this way, the demand and supply are automatically balanced and reasonable downloading and seeding times are achieved.

7 Related Work

To date, only few works have analyzed private communities. Zhang *et al.* [4] investigate hundreds of private trackers and depict a broad and clear picture of the private community landscape. Chen *et al.* [1] compare system behaviors among 13 private trackers and 2 public trackers, and they show their differences regarding user viscosity, single torrent evolution, user behaviors, and content distribution. Liu *et al.* [2] also perform measurement studies and further develop a model to show that SRE indeed provides effective incentives, but is vulnerable to collusion.

While these studies all focus on demonstrating the high seeding level achieved by private communities, there have been a few preliminary works that show the adverse effects. Andrade *et al.* [12] focus on the dynamics of resource demand and supply, and they show that users typically try to increase their contribution levels by seeding for longer and not by providing more bandwidth to the system. However, our paper shows that providing limited bandwidth is not the will of users, but it is a consequence of the oversupply in private communities. Chen *et al.* [17] also notice the oversupply problem and provide a model to identify the optimal stable SLR range. However, they didn't analyze the reason or propose strategies to solve the problem of oversupply. Kash *et al.* [14] demonstrate that there are significant disparities in the cost of new and old files in a private community named DIME, and users compensate for the high cost of older files by downloading more copies of newer files or by preferentially consuming older files during free-leech periods. Particularly, they have shown that after a period of free-leech, there are more download activities in the community. This is consistent with our result that during free-leech, there is more demand and the oversupply is alleviated. Besides analyzing positive and negative effects of SRE, our work further discusses the performance of well-adopted community strategies, their effects against strategic user behavior and the remedies we proposed.

8 Conclusion

While previous works only focus on showing the effectiveness of SRE in increasing the bandwidth supply, in this paper we provide a better understanding of private communities by demonstrating both the positive and negative effects of SRE. We show that swarms in private communities are greatly oversupplied. Users achieve very high downloading speeds, but at significant expense including excessively long seeding times and very low upload speeds. As a consequence, users with small sharing ratios are forced to limit their downloading needs so as to keep adequate sharing ratios to stay in the community. For users who intend to increase their sharing ratios, we show that seeding for longer durations is not as effective as increasing the upload speed. If it is not realistic for the users to upgrade their internet access, we suggest them to join swarms early or to join undersupplied swarms.

Acknowledgements This work was supported by the Future and Emerging Technologies programme FP7-COSI-ICT of the European Commission through project QLectives (grant no.: 231200) and the EU FP7 project P2PNEXT (grant no.: 216217).

References

1. Chen, X., Chu, X.: Measurements, analysis and modeling of private trackers. In: Proceeding of the 10th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2010 (2010)
2. Liu, Z., Dhungel, P., Wu, D., Zhang, C., Ross, K.: Understanding and improving incentives in private p2p communities. In: Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS 2010) (2010)
3. Meulpolder, M., D'Acunto, L., Capota, M., Wojciechowski, M., Pouwelse, J., Epema, D., Sips, H.: Public and private bittorrent communities: A measurement study. In: Proceedings of 9th International Workshop on Peer-to-Peer Systems (IPTPS 2010) (2010)
4. Zhang, C., Dhungel, P., Di Wu, Z.L., Ross, K.: Bittorrent darknets. In: Proceedings of the 30th IEEE International Conference on Computer Communications (INFOCOM 2010) (2010)
5. Jia, A., D'Acunto, L., Meulpolder, M., Pouwelse, J.: Modeling and analysis of sharing ratio enforcement in private bittorrent networks. In: Proceedings of the IEEE International Communications Conference (ICC 2011) (2011)
6. Rahman, R., Hales, D., Vinko, T., Pouwelse, J., Sips, H.J.: No more crash or crunch: Sustainable credit dynamics in a p2p community. In: Proceeding of the International Conference on High Performance Computing and Simulation, HPCS 2010 (2010)
7. Jia, A., Rahman, R., Vinko, T., Pouwelse, J., Epema, D.: Fast download but eternal seeding: the reward and punishment of sharing ratio enforcement. In: Proceeding of the 11th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2011 (2011)
8. CHDBits, http://chdbits.org/
9. ChinaHDTV, http://www.chinahdtv.org
10. HDStar, http://www.hdstar.org/
11. Jia, A.L., Chen, X., Chu, X., Pouwelse, J.: From user experience to strategies: how to survive in a private community. Technical Report PDS-2011-004, Delft University of Technology (September 2011)
12. Andrade, N., Santos-Neto, E., Brasileiro, F., Ripeanu, M.: Resource demand and supply in bittorrent content-sharing communities. Computer Networks 53 (2008)
13. Spearman, C.: The proof and measurement of association between two things. American Journal of Psychology 15 (1904)
14. Kash, I., Lai, J., Zhang, H., Zohar, A.: Economics of bittorrent communities. In: Proceeding of the 6th Workshop on the Economics of Networks, Systems, and Computation, NetEcon 2011 (2011)
15. BitSoup, http://bitsoup.org
16. PolishTracker, http://polishtracker.net/
17. Chen, X., Chu, X., Li, Z.: Improving sustainability of private p2p communities. In: Proceeding of the 20th International Conference on Computer Communications and Networks (ICCCN 2011) (2011)

Optimal Migration Contracts in Virtual Networks: Pay-as-You-Come vs Pay-as-You-Go Pricing

Xinhui Hu[1], Stefan Schmid[2], Andrea Richa[1], and Anja Feldmann[2]

[1] SCIDSE, Arizona State University, Tempe, AZ 85287, USA
{xinhui.hu,aricha}@asu.edu
[2] TU Berlin & T-Labs, Germany
{stefan,anja}@net.t-labs.tu-berlin.de

Abstract. Network virtualization realizes the vision of an Internet where resources offered by different stakeholders are used and shared by multiple coexisting virtual networks. The abstraction introduced by network virtualization opens new business opportunities. We expect that in the near future, infrastructure providers (or resource brokers and resellers) will offer flexibly specifiable and on-demand virtual networks over the Internet, similarly to the elastic resources in today's clouds.

This paper initiates the discussion on the optimal resource allocations in such an economic environment. We attend to a scenario where a flexible service (such as a web service or an SAP database) is implemented over a virtual network. This service can be seamlessly migrated closer to the current locations of the (mobile) users. We assume that a virtual network provider offers different contracts to the service provider, and we distinguish between two fundamentally different pricing models: (1) a *Pay-as-You-Come* model where the service provider needs to decide in advance which time-based contracts to buy in order to implement the service, and a (2) *Pay-as-You-Go-model* where the service provider is charged only when the service terminates and only for the amount of resources actually used. In both cases, the virtual network provider may offer a *discount* if more resources are bought, e.g., buying a resource contract of double duration or of twice as much bandwidth only costs fifty percent more than a simple contract. We describe two optimal migration algorithms PAYC (for the Pay-as-You-Come model) and PAYG (for the Pay-as-You-Go model), provide a quantitative comparison of the two pricing models, and discuss their implications. Finally, extensions to online algorithms are discussed.

1 Introduction

The Internet becomes more and more virtualized and programmable (or "software-defined"), and we witness a trend towards extending the cloud paradigm to the *network*. Researchers in the field of *network virtualization* develop prototype architectures that herald flexibly specifiable, fully *virtual networks* (VNets) (also known as *CloudNets*): virtual networks that can be requested at short notice (and even be migrated arbitrarily within the specification constraints), while providing isolation guarantees (e.g., in terms of QoS or security). This paradigm has the potential to open a network infrastructure

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 285–299, 2013.

for a wide range of new and innovative services, and it is believed that new economical entities will emerge that lease (or re-lease) infrastructure parts to service providers.

We expect that in the near future, such virtual networks connecting arbitrary locations (and spanning multiple autonomous systems and providers) in the Internet can be leased similarly to the resource leasing models of today's clouds. This paper attends to a use case for such dynamic VNets where a service provider offers a flexible and latency-critical service (for instance a web service, an SAP server or a game server) to its mobile customers whose demand and locations changes over time (e.g., due to time-zone effects or commuting). We assume that the service provider itself uses the resource services of a substrate infrastructure provider (e.g., a physical infrastructure provider or a virtual network provider) in order to offer a low-latency access to a server which can be migrated seamlessly in the VNet (i.e., without reconfiguration or changes of routable network addresses). The service provider is faced with the challenge that while moving the server closer to the customers improves QoS (and/or reduces roaming costs), frequent migrations come at service interruption and bulk data transfer costs. We initiate the study of optimal offline and online migration strategies for the service provider under two different pricing models.

Our Contribution. This paper initiates the study of the virtual server migration problem from an economical perspective. We compare the two most basic pricing policies *Pay-as-You-Come* and *Pay-as-You-Go* (see, e.g., [14]), in which a service provider has to pay in advance for time-based contracts respectively in retrospect for the resources actually used. The service provider receives a discount when buying larger contracts, e.g., a contract of twice the resource volume only costs 50% more. As a first step, we, in this paper, design offline migration algorithms for different settings and discount functions. We find that optimal offline solutions can indeed by computed in polynomial time by using non-trivial dynamic programs. We theb use these algorithms to quantify and compare the two pricing models by simulation. We discuss the implications of these models and find that, as expected, Pay-as-You-Come pricing yields higher costs on the service provider side than Pay-as-You-Go pricing, especially for moderate discounts. Interestingly, the distribution and structure of the costs and the used contracts differ significantly for the two pricing schemes, and it turns out that the QoS guarantees under the Pay-as-You-Go model are much better due to the efficient resource investments.

Note that offline algorithms are particularly interesting if demand patterns can be predicted well (e.g., if it depends on time-zone effects or commuter behavior). However, offline algorithms can also serve as a yardstick to evaluate the performance (i.e., the so-called *competitive ratio*) of online algorithms in simulations. This paper also initiates exploring online migration strategies.

2 Economical Service Migration

A virtual network topology can be modeled as a graph $G = (V, E)$ where $V(G)$ denotes the set of nodes and $E(G)$ the set of links. We assume that a service provider can place its service (i.e., the server) on any location in the virtual network. Requests can originate from different access points in $V(G)$, and the access cost is given by the shortest path (depending on some given metric D) to the server location in $V(G)$. In order to reduce

the access cost, the virtual server can be migrated along the links in $E(G)$. To do so, the service provider needs to purchase bandwidth along the migration path.

We attend to a scenario where a virtual network provider offers the service provider a choice of contracts of different durations in which dedicated resources can be leased in the virtual network (e.g., for migration), i.e., $\mathfrak{D} = \{d_1, d_2, \ldots, d_k\}$ (we assume $d_1 < d_2 < \ldots < d_k$). In addition to the contract durations, the service provider can choose between different bandwidths along the links, i.e., it can choose among the following set of bandwidths for each link: $\mathfrak{B} = \{b_1, b_2, \ldots, b_q\}$ (we also assume that $b_1 < b_2 < \ldots < b_q$).

We consider two different pricing models. Under *Pay-as-You-Go* pricing, a customer only needs to pay for the used resources after the actual consumption (or at regular time intervals T), and the best contract is determined according to the usage pattern *a posteriori*. Pay-as-You-Go pricing is often used in the context of cloud resource leasing. In contrast, in the *Pay-as-You-Come* model, a customer needs to decide *in advance* which kind of *time-based* contracts she is interested in, and needs to buy them before the actual resource usage. Examples for this model can be found, e.g., in the context of private Internet access where users often pay in advance and independently of the actual usage pattern.

In this paper, in order to focus on the main tradeoffs, we initiate the study of these pricing models in a simplified scenario where the virtual network consists of two locations only (e.g., one in the U.S. and one in Asia); we will refer to these locations by L (left) and R (right) respectively and normalize their distance to one unit. The server can be migrated arbitrarily between the two locations if a corresponding resource contract is present for the bulk-data transfers. Concretely, a contract in the *Pay-as-You-Come* model consists of a duration d_i and the bandwidth b_j to lease the virtual link between the two sites for d_i units (e.g., days) and at a bandwidth of b_j (e.g., Mbit/s). The price of the contract is given by a function $f(d_i, b_j)$, where $f(\cdot, \cdot)$ describes a monotonic increasing discount over the contract duration and over the amount of reserved resources. For example, a twice as long contract may cost only 50% more, and doubling the reserved bandwidth may cost only 30% more. In the *Pay-as-You-Go* model, the customer only needs to pay when the service is finished or after a given duration, i.e., every T time units (e.g., a month), and only for the resources (and bulk data transfers) that are actually used. Concretely, if μ_i migrations are performed during the time period T at a bandwidth of $b_j \in \mathfrak{B}$, the overall costs amount to $f(\mu_i, b_j)$.

The main objective is to minimize the migration and contracting costs (denoted by MigCost and ConCost) while providing good Quality-of-Service (QoS) guarantees (minimize access cost AccCost). Hence, we seek to minimize the following cost function:

$$\text{Cost} = \text{AccCost} + \text{MigCost} + \text{ConCost}$$

We assume there are n requests total, denoted by a set $< r_1, r_2, \cdots, r_n >$ at respective times $< t_1, t_2, \cdots, t_n >$. The access cost is given by the latency of the requests $r_i \in V(G)$ to the location of the server $s_i \in V(G)$, i.e., AccCost $= \sum_i D[r_i, s_i]$ where r_i and s_i denote the ith request node and the server location at time t_i. The migration cost MigCost is given by the service interruption time (see also [5]), i.e., the time to transfer

the server which is determined by the bandwidth of the weakest link along the migration path. (In a system supporting live migration, this cost can be negligible and set to zero.) Concretely, the migration cost is computed as MigCost $= \sum_i S \cdot D[s_{i-1}, s_i]/b_i$, where S is the server size, $D[s_{i-1}, s_i]$ denotes whether the locations $s_{i-1} \in \{L, R\}$ and $s_i \in \{L, R\}$ differ (recall that $D[s_{i-1}, s_i]$ is 1 if $s_{i-1} \neq s_i$, and 0 otherwise), and $b_i \in \mathfrak{B}$ is the (minimal) bandwidth along the migration path. Finally, the contract cost is computed as described above, i.e., ConCost $= \sum_i f(d_i, b_i)$ for the Pay-as-You-Come model and as ConCost $= f(\mu, b_i)$ for Pay-as-You-Go model, where $d_i \in \mathfrak{D},\ b_i \in \mathfrak{B}$ and μ is the total number of migrations.

3 Migration Strategies

This section presents optimal algorithms to compute the best set of contracts and optimal migration strategies for the two presented pricing models. We will first present an algorithm PAYC for the Pay-as-You-Come model and prove its optimality, and then extend this algorithm to a PAYG algorithm which solves the Pay-as-You-Go model. Both our algorithms PAYC and PAYG are based on dynamic programming, and fill out matrices such that optimal substructures are reused.

3.1 Pay-as-You-Come

Let us now turn our attention to the first, time-based pricing model. Our PAYC algorithm stores intermediate *minimum total cost* results (access, migration and contract costs) in a 3-dimensional matrix $C_{n\times n\times 4}$ where n is the total number of requests. $C[i, j, k]$ denotes an entry of the matrix, where $i, j \in [1, n]$ and $k \in \{(s, s')|s, s' \in \{L, R\}\}$. $C[i, j, (s, s')]$ denotes the minimum total cost for satisfying all requests from r_i to r_j for a scenario where at the beginning of the ith request the server is at node s and at the end of request j the server is at node s'. We also need a matrix $(AM_m)_{n\times n\times 4}$ for each bandwidth $b_m \in \mathfrak{B}$. For a fixed bandwidth b_m during the entire interval $[t_i, t_j]$, entry $AM_m[i, j(s, s')]$ stores the combined access and migration costs for the best migration strategy that satisfies the sequence of requests from r_i to r_j, assuming that the server is located at node s at the start of request r_i and at node s' at the end of request r_j. The contract costs, given by the function f, are not included in the entries of AM_m.

Given these data structures, we can describe algorithm PAYC (Algorithm 1) for the Pay-as-You-Come model. PAYC exploits that the optimal contract from request time t_i to request time t_j can either be decomposed into two consecutive subperiods with no overlapping contracts, or be obtained by buying a contract of long duration d_v and bandwidth b_m if $d_{v-1} < t_j - t_i + 1 \leq d_v$, where $d_v, d_{v-1} \in \mathfrak{D}$.

PAYC starts by initializing the optimal costs if we were to serve only one request r_i, for all possible combinations of starting server location s and ending server location s' at time t_i. According to our model, the access cost is equal to the distance between the current requesting node r_i and the server location s' at the end of time t_i, denoted by $D[r_i, s']$. If the request at time t_i comes from the server location s', then no access cost is needed since $D[r_i, s']$ is 0; otherwise the access cost is positive. Recall that the migration cost for request r_i is computed as $S \cdot D[s, s']/b_m$, where $b_m \in \mathfrak{B}$ is the selected bandwidth and S is the migrated server size. We store the respective optimal cost

Algorithm 1. Algorithm PAYC

Input: Requests $<r_1, r_2, ..., r_n>$ at respective times $<t_1, t_2, ..., t_n>$.
Output: Minimum cost.

1: **for** $i = 1$ **to** n **do**
2: **for** all pairs $(s, s') \in \{L, R\}^2$ **do**
3: **for** $m = 1$ **to** q **do**
4: $AM_m[i, i, (s, s')] \leftarrow D[s', r_i] + S \cdot D[s, s']/b_m$
5: $C[i, i, (s, s')] \leftarrow \min_{1 \le m \le q}\{AM_m[i, i, (s, s')] + f(d_1 * D[s, s'], b_m)\}$
6: **for** $l = 2$ **to** n **do**
7: **for** $i = 1$ **to** $n - l + 1$ and **pairs** $(s, s') \in \{L, R\}^2$ **do**
8: $j \leftarrow i + l - 1$
9: $C[i, j, (s, s')] \leftarrow \min_{i \le u < j; s'' \in \{L,R\}}\{C[i, u, (s, s'')] + C[u + 1, j, (s'', s')]\}$
10: **if** $d_{v-1} < t_j - t_i + 1 \le d_v$, for some $v = \{1, \cdots, k\}$ **then**
11: **for** $m = 1$ **to** q **do**
12: $AM_m[i, j, (s, s')] \leftarrow \min_{s'' \in \{L,R\}}\{AM_m[i, i, (s, s'')] + AM_m[i + 1, j, (s'', s')]\}$
13: **if** $C[i, j, (s, s')] > \min_{1 \le m \le q}\{AM_m[i, j, (s, s')] + f(d_v, b_m)\}$ **then**
14: $C[i, j, (s, s')] \leftarrow \min_{1 \le m \le q}\{AM_m[i, j, (s, s')] + f(d_v, b_m)\}$
15: **return** $\min_{s_{\text{final}} \in \{L,R\}} C[1, n, (s_{\text{init}}, s_{\text{final}})]$

of satisfying request r_i (which may or may not incur a non-zero access cost $D[r_i, s']$, depending on whether $r_i \neq s'$ or not) using bandwidth b_m, with starting and ending positions of the server s and s' respectively, in $AM_m[i, i, (s, s')]$. We choose a bandwidth $b_m \in \mathfrak{B}$ such that the total cost, including the contract cost $f(d_1, b_m)$ if a migration accurs, is minimized, and store the optimal total cost in $C[i, i, (s, s')]$.

Next, we consider the total costs for sequences of more than one request. Note that there are l requests occurring between time t_i and t_j, where $i < j$ are defined in Lines 7 and 8 of the algorithm and $l\ (= j - i + 1) > 1$. We have two alternative options: (i) we can split the interval $[t_i, t_j]$ at the time t_u of request r_u, where $i \le u < j$, and buy contracts for the intervals $[t_i, t_u]$ and $[t_{u+1}, t_j]$ independently for the two possible locations s'' of the server at time t_u; or (ii) we can buy a long contract of duration $d_v \in \mathfrak{D}$ and some bandwidth $b_m \in \mathfrak{B}$ to cover all the l requests if the period $t_j - t_i + 1$ is between d_{v-1} and d_v. The smaller cost of these two cases gives the optimal cost for the interval $[t_i, t_j]$.

We also update $AM_m[i, j, (s, s')]$, for all possible bandwidths b_m. Basically we extend the intervals already considered by one request (r_i), and we store in $AM_m[i, j, (s, s')]$ the migration strategy that minimizes the total access and migration costs for satisfying requests r_i through r_j using bandwidth b_m for starting and ending positions of the server s and s' respectively. Note that by taking into account all possible positions of the server at the end of request r_i, we consider all the possibilities of adding r_i to all the best possible strategies already computed for the subsequence $r_{i+1}, \ldots, r_j$ (ending at node s').

We process the previous steps in increasing order of l until l spans all the requests. Thus, the optimal cost is given by $\min_{s_{\text{final}} \in \{L,R\}} C[1, n, (s_{\text{init}}, s_{\text{final}})]$, where s_{init} is the initial server location.

Theorem 1. PAYC *(see Algorithm 1) computes the optimal contracts for Pay-as-You-Come model. The time complexity of* PAYC *is* $O(n^2(n+kq))$*, where* n *is the number of requests,* k *is the number of contract durations and* q *is the number of different bandwidth contracts.*

Proof. The correctness follows by induction over the number of request l and by the optimal substructure property. Due to space constraints, we only sketch the proof. The claim is trivially true for sequences of one request (Lines 1–5). Consider the time interval from t_i to t_j with l requests, where $1 \leq i \leq j \leq n$ and $2 \leq l(=j-i+1) \leq n$. This interval is split into two subintervals (Case I), or a long contract is bought that covers the entire interval (Case II). In *Case I*, we split the cost at time t_u with the server located at s'' such that the total cost $C[i,u,(s,s'')] + C[u+1,j,(s'',s')]$ is minimized, where $i \leq u \leq j$ and $s'' \in \{L,R\}$. Since the number of requests in the two subintervals, $u-i+1$ and $j-u$, are shorter than l, by the induction hypothesis, $C[i,u,(s,s'')]$ and $C[u+1,j,(s'',s')]$ already store the optimal costs for these two intervals respectively. In *Case II*, we buy a long contract to cover the whole interval. Given a certain server location s'' at the start of the time t_{i+1}, $AM_m[i+1,j,(s'',s')]$ already stores the optimal access and migration strategy cost for bandwidth b_m for interval $[t_{i+1},t_j]$. Therefore, an optimal migration strategy for interval $[t_i,t_j]$ using bandwidth b_m can be obtained by adding r_i to the optimal strategies selected for the interval $[t_i+1,t_j]$ and optimizing over the choice on whether to migrate the server to serve r_i or not (resulting in the two possible choices for s'', the position of the server right after satisfying request r_i).

Now we consider the time complexity of the PAYC algorithm. Clearly, the first phase of the algorithm requires time $O(nq)$. The second phase consists of three nested loop and has a complexity of $O(n^2 \cdot (n+kq))$. □

3.2 Pay-as-You-Go

Optimal solutions can also be computed for the Pay-as-You-Go model, and the algorithm PAYG is similar to the algorithm PAYC. As discussed above, in the Pay-as-You-Come model we need to decide when to migrate, which contracts to buy, and how much bandwidth to use. In the Pay-as-You-Go model, we still need to make a decision on when to migrate and how much bandwidth should be reserved, but we do not have to explicitly decide on a time contract. However, unlike the Pay-as-you-Come model, in the Pay-as-you-Go model, a bandwith b_m has to be chosen and fixed for satisfying the entire sequence of requests $r_i,\ldots,r_j$. Also, the contract cost in this model is directly dependent on the number of migrations of the server, and hence we explicitly have to keep track of this number.

Algorithm PAYG is listed in Algorithm 2. PAYG uses a new matrix $(A_m)_{n\times n\times 4}$ to store the access cost under a certain bandwidth b_m, $1 \leq m \leq q$, and another matrix $(N_m)_{n\times n\times 4}$ is used to store the migration number for bandwidth b_m. A matrix $(C_m)_{n\times n\times 4}$ stores the total cost for bandwith b_m. In the entries of the new matrices, the elements $A_m[i,j,(s,s')]$ and $N_m[i,j,(s,s')]$ store the access cost and the number of migrations, respectively, for the optimal solution between time t_i and t_j with an initial server location s and a final server location s', where $s,\ s' \in \{L,R\}$. The entry $C_m[i,j,(s,s')]$ stores the total optimal cost within this time period for bandwith b_m.

Algorithm 2. Algorithm PAYG

Input: Requests $<r_1, r_2, ..., r_n>$ at respective times $<t_1, t_2, ..., t_n>$.
Output: Minimum Cost.

1: **for** $i = 1$ **to** n **do**
2: **for** all pairs $(s, s') \in \{L, R\}^2$ and $1 \leq m \leq q$ **do**
3: $A_m[i, i, (s, s')] \leftarrow D[s', r_i]$
4: $N_m[i, i, (s, s')] \leftarrow D[s, s']$
5: $C_m[i, i, (s, s')] \leftarrow A_m[i, i, (s, s')] + S \cdot N_m[i, i, (s, s')]/b_m + f(D[s, s'], b_m)$
6: **for** $l = 2$ **to** n **do**
7: **for** $i = 1$ **to** $n - l + 1$ **do**
8: $j \leftarrow i + l - 1$
9: **for** all pairs$(s, s') \in \{L, R\}^2$ and $1 \leq m \leq q$ **do**
10: $C_m[i, j, (s, s')] \leftarrow \min_{i \leq u < j; s'' \in \{L,R\}} \{A_m[i, u, (s, s'')] + A_m[u + 1, j, (s'', s')] + S \cdot (N_m[i, u, (s, s'')] + N_m[u + 1, j, (s'', s')])/b_m + f((N_m[i, u, (s, s'')] + N_m[u + 1, j, (s'', s')]), b_m)\}$
11: Let (u, s'') be the parameter and location of request r_u at t_u that minimized Line 10.
12: $A_m[i, j, (s, s')] \leftarrow A_m[i, u, (s, s'')] + A_m[u + 1, j, (s'', s')]$
13: $N_m[i, j, (s, s')] \leftarrow N_m[i, u, (s, s'')] + N_m[u + 1, j, (s'', s')]$
14: **return** $\min_{s_{\text{final}} \in \{L,R\},\ 1 \leq m \leq q} C_m[1, n, (s_{\text{init}}, s_{\text{final}})]$

The basic idea behind PAYG is to compute the optimal solution for a scenario where all the requests require the same bandwidth, and then choose the smallest cost among all the bandwidth options. PAYG starts off by computing the optimal costs for satisfying one request (Lines 1-5). Given the request r_i and the starting and ending server locations s, s', the access cost is given by $D[s', r_i]$ which is 0 if the final server location s' and the request location r_i coincide, and 1 otherwise. Meanwhile $D[s, s']$ will indicate that the server migrates to the other location to serve the current request if $D[s, s']$ is 1. Otherwise, there is no migration, and the starting and ending server locations s, s' describe the same node. We store the optimal solution in $C_m[i, i, (s, s')]$ for each bandwidth b_m, where $C_m[i, i, (s, s')] = D[s', r_i] + S \cdot D[s, s']/b_m + f(D[s, s'], b_m)$.

Now PAYG iterates over the number of requests l (Line 6). For each value of l, we compute all the possible cases, as in Lines 7-13. First, we select from $[1, n - l - 1]$ the value i denoting the index of the first of these l requests. Obviously, the index of the last of the l requests (denoted by j) would be $i + l - 1$, as in Line 8. Assume that the server is located at node s at the time when the ith request occurs, and located at node s' at the end of the jth request, where $s, s' \in \{L, R\}$. We look for a way to split the duration such that the total cost $C_m[i, j, (s, s')]$ is minimized, as shown in Line 10. We use u, m, and s'' to denote the index of the request occurring at the chosen split point, the chosen bandwidth, and the location of the server (Line 11). Therefore, the total cost consists of the summation of the access costs of two subintervals, the summation of the migration costs of two subintervals, and a long contract cost covering the whole period. Here, the access cost is computed as $A_m[i, u, (s, s'')] + A_m[u + 1, j, (s'', s')]$, the migration cost is computed as $(N_m[i, u, (s, s'')] + N_m[u+1, j, (s'', s')])/b_m$ and the contract cost

is computed as $f(N_m[i,u,(s,s'')]+N_m[u+1,j,(s'',s')],b_m)$, for a certain bandwidth b_m. We store the access cost in $A_m[i,j,(s,s')]$ (Line 12) and the number of migrations in $N_m[i,j,(s,s')]$ (Line 13) for the current duration.

For each bandwidth b_m, we store the optimal solution to serve all the requests in C_m matrix. Thus the optimal cost is hence obtained by computing $\min_{s_{\text{final}}\in(L,R),1\leq m\leq q} C_m[1,n,(s_{\text{init}},s_{\text{final}})]$ (Line 14).

The following claim follows by simple induction over the number of requests.

Theorem 2. PAYG *(see Algorithm 2) computes the optimal contracts for the Pay-as-You-Go model. The time complexity is $O(qn^3)$, where n is the number of requests and q is the number of different contract bandwidths.*

Proof. We argue by induction on the number of the requests l considered. In base case, when there is just one request ($l = 1$), lines 1-5 will give the optimal solutions under each bandwidth. As for the inductive step, we follow a similar strategy as for PAYC. We split at time t_u with the server located at s'' such that the access cost and the migration cost of two sub intervals will minimize the total amount for the current duration. Since we consider all possible splits at all times within the whole interval as well as all the server migrations (Line 10), we choose the best option for the longer interval.

Regarding the time complexity, Lines 3, 4, and 5 each take $O(1)$ time, respectively. Since Lines 3-5 are executed $4nq$ times, the total running time of Lines 1-5 is $O(nq)$. Considering that Lines 10-13 are executed $O(n^2q)$ times and $l \leq n$, we know that the running time of Lines 6-13 is $O(qn^3)$. Therefore, the time complexity of Algorithm 2 is $O(qn^3)$. □

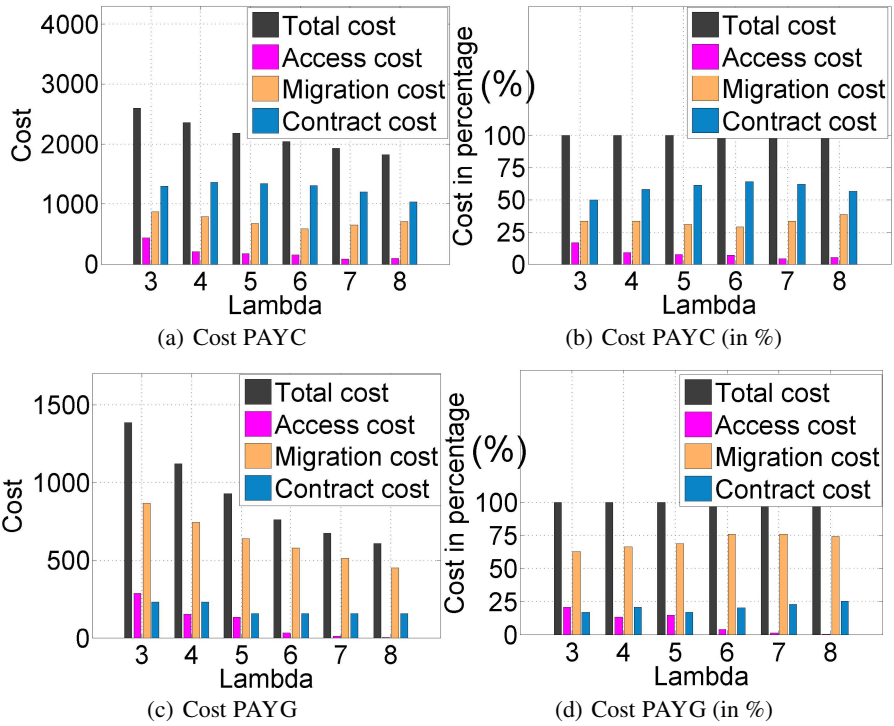
4 Quantitative Comparison

The presented economical migration algorithms allow us to shed light on the properties of the two pricing models. We study three different discount functions $f_{\text{lin}}, f_{\text{sqrt}}, f_{\text{log}}$ which offer cheaper contracts if longer (in terms of days) or larger (in terms of leased bandwidth) contracts are bought: f_{lin} is linear ("get twice as much for a 50% higher price"), f_{sqrt} grows according to a square root function and hence describes a steeper discount, and f_{log} even gives an even steeper logarithmic discount. For all three discount functions, the cost of a one-day contract with 50 Mbit/s bandwidth is the same, namely $f_i(1,50) = 6$ for $i \in \{\text{lin, sqrt, log}\}$. Concretely, we use $f_{\text{lin}}(d_i,b_j) = 1.5 \cdot f_{\text{lin}}(d_i/2,b_j) = 1.5 \cdot f_{\text{lin}}(d_i,b_j/2)$=$1.5^{(\lfloor \log d_i \rfloor + b_j/50-1)} \cdot f_{\text{lin}}(1,50)$, $f_{\text{sqrt}}(d_i,b_j) = \sqrt{d_i b_j/50} \cdot f_{\text{sqrt}}(1,50)$, and $f_{\text{log}}(d_i,b_j) = \log(d_i b_j/50) \cdot f_{\text{log}}(1,50)$. We assume a server of size $S = 250$ MB, and we assume that the access cost for one remote request is five units (a request originating at the node where the service is located is free). We study a scenario where the provider offers two different bandwidth capacities, namely 50 Mbit/s and 100 Mbit/s, and four types of contract durations, namely 1, 30, 60 and 100 days (i.e., $\mathfrak{B} = \{50,100\}$ and $\mathfrak{D} = \{1,30,60,100\}$).

We study a simple request pattern where requests originate from L and R in turn, e.g., requests originating in Asia alternate with requests originating in the U.S..

Simplified Demand Scenario: We assume that requests alternate infinitely between the two sites L and R in the following manner: requests originate from one site (one per round) for a time interval duration which is chosen according to an exponential distribution with parameter λ, before requests originate from the opposite side again (according to the same distribution).

We simulate $n = 1500$ requests, and present the average over five runs for each experiment.



(a) Cost PAYC (b) Cost PAYC (in %)

(c) Cost PAYG (d) Cost PAYG (in %)

Fig. 1. Cost distribution for PAYC and PAYG

We discuss the following simulations in more detail.

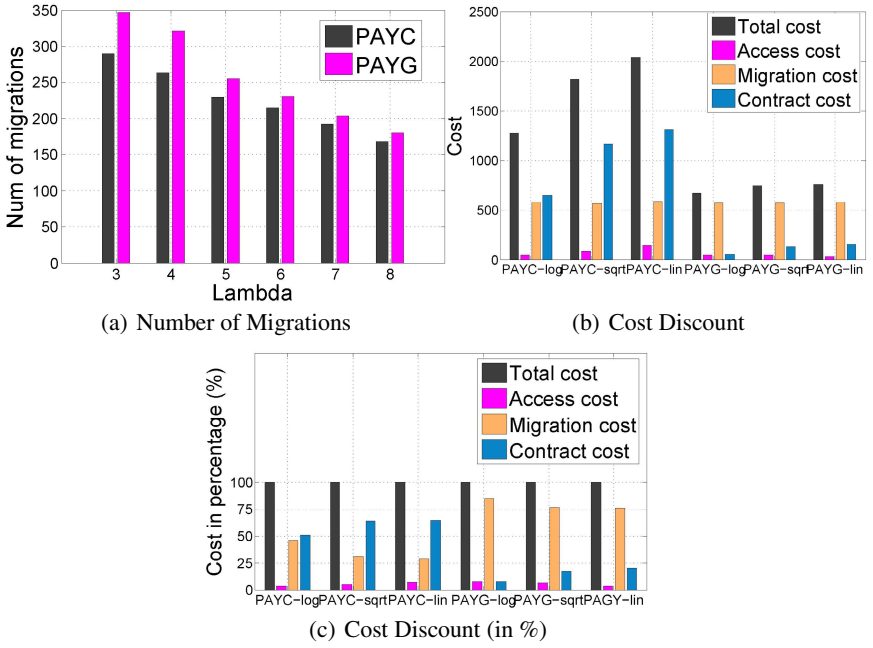
Cost Distribution and Number of Migrations. We analyze how the cost distributes among the access cost, the migration cost, and the contract cost for the two algorithms PAYC and PAYG. All experiments discussed here are conducted under the natural f_{lin} discount function. Figure 1(a) shows the absolute costs of PAYC as a function of λ. We observe that the total cost and the access cost decrease for larger λ while the migration and contract stay much more stable. This is clear as requests originating from one site for longer time periods render it worthwhile to migrate and buy longer contracts.

The contract increases firstly and then decrease after some point, since the total migration numbers decrease and hence the contract cost is reduced. As the number of migrations decrease, the average number of migrations within a contract is also decreased. Therefore, PAYC will buy smaller bandwidth for such contract, which will result in larger migration costs(also shown in Table 2). Figure 1(b) presents the relative shares of the three costs. While the access costs approach zero for larger λ since the server is often at the right location, the contract costs and the migration costs stay stable since PAYC migrates a lot even for larger λ. The same results for PAYG are shown in Figures 1(c) and 1(d), respectively. As a first takeaway, we see that the cost distribution of Figure 1(c) defers from Figure 1(a) in that the total costs are lower, i.e., Pay-as-You-Go is always the cheaper option than Pay-as-You-Come pricing for the customer. Also note that in contrast to the Pay-as-You-Come model, the migrations constitute a larger share of the overall costs, since the contract cost is given by the number of migrations and the amount of leased bandwidth under the discount function; hence the contract cost is lower than the one of PAYC for the same number of migrations. Moreover, there are relatively more frequent migrations under the PAYG model, see Figure 2(a), which also explains the lower access costs (i.e., this improves QoS experienced by the users). Regarding the relative cost shares (Figure 1(d)), we can see that the percentage for the access cost is decreasing while the percentages for the migration cost and the contract cost are increasing slowly. Again, when λ is large enough and the requests become more local, since migrations only occur at the beginning of each interval, the number of migrations (as well as all three cost components) eventually decreases.

Contract Distribution. Different pricing models and scenarios result in different types and combinations of contracts, and it is interesting to study the frequency (or popularity) distribution of the contracts. Table 1 reports on the average number of the contracts as a function of λ, for different contract durations and bandwidths, under the PAYC algorithm and for f_{lin}. We see that when λ is small and migrations are dense, longer duration contracts occur frequently since the server migrates often. However, as λ increases, all lengths of contracts decrease. As λ increases, the average number of migrations in a contract decreases and hence the smaller bandwidth will benefit more than the larger one. Therefore, it turns out to buy more contracts with smaller bandwidth. This can also be seen in Table 2 which records the average number of migrations in different contracts accordingly (average over five runs).

Table 1. Distribution of purchased contracts (discount function f_{lin})

λ / Dur-Bw	3	4	5	6	7	8
1-50	11.2	8	15.4	13.8	18.4	39.2
60-50	0	0	0	0	2.4	0.8
60-100	1.4	2	1.4	2.8	1	0.4
100-50	0	0	0	0.6	2	5.4
100-100	11	11	11.2	10	7.6	3.4



(a) Number of Migrations (b) Cost Discount

(c) Cost Discount (in %)

Fig. 2. Number of migrations and effect of discount function

Table 2. Number of migrations for each contract (discount function f_{lin})

Dur-Bw \ λ	3	4	5	6	7	8
1-50	1	1	1	1	1	1
60-50	0	0	0	0	8,5	0
60-100	17.67	14	13.5	11.5	0	0
100-50	0	0	0	13	13	12.57
100-100	27.33	23.58	19.45	17.33	15	14.5

Impact of Discount Function. Finally, let us compare the different discount functions in more detail. Figure 2(b) and Figure 2(c) explore the absolute and relative (in %) cost distributions for PAYC and PAYG under different discount functions. Clearly, the higher the discount, the smaller the total cost. Moreover, not surprisingly the performance of PAYG is always better than that of PAYC since the total cost is less for PAYG compared to that for PAYC. However, the difference of the costs for the two models is smaller for higher discounts, i.e., the difference for the logarithmic discount function is smaller than for a discount function which follows a square root.

5 A First Look at Online Migration

Although the main focus of this paper is on predictable demand scenarios and offline algorithms, in this section, we want to initiate the discussion of online algorithms. The online discussion builds upon our offline results in two respects: First, some algorithmic techniques from the offline variant may be used also for the online variants. For example, an online algorithm may try to predict the future from the past, and apply an optimal offline algorithm on a sequence of recent past requests in order to make decisions on how to deal with upcoming requests. Second, offline algorithms are often needed to evaluate the performance of an online algorithm. The ratio of the cost of an online algorithm divided by the cost of an optimal offline algorithm is also known as the *competitive ratio* [3].

Both online algorithms presented in the following are inspired by the (optimal) offline variants and seek to amortize costs over time. To simplify the presentation, we assume a constant bandwidth scenario.

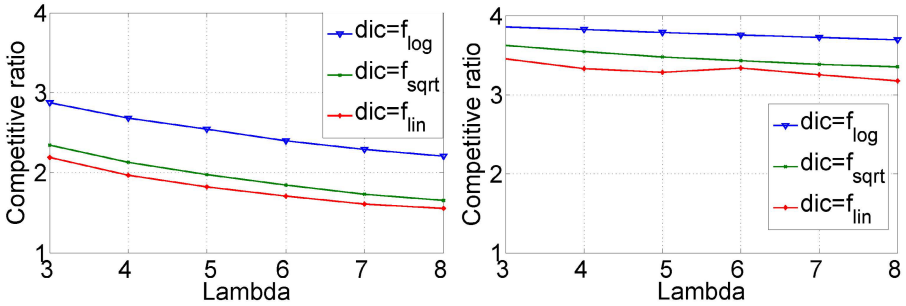
ONC: The online Pay-as-You-Come algorithm ONC tracks the access costs it incurs at the current location using a counter C. Once the counter exceeds the migration cost (given by the server size divided by the bandwidth), ONC migrates the server and resets C. If there is currently no contract available for migration, ONC checks whether a contract longer than the most recently used contract would have been better *for the past requests*. Concretely, ONC checks longer contracts one by one (in increasing order of length) and compares their costs in the corresponding intervals (starting from the last migration) to the cost ONC incurred during that time period. As soon as a better contract is found, it is chosen. Otherwise, ONC checks whether a contract shorter than the most recent contract should be chosen. The following heuristic is applied: ONC checks whether during the last contract, the number of migrations was larger in the first half or the second half of the contract time interval. In case of the first half, ONC will buy the shorter contract; otherwise, ONC chooses the same contract as last time.

Now let us discuss a simple online algorithm ONG for the Pay-as-You-Go model. Since the customers only need to pay for the resources actually consumed, ONG just needs to decide when to migrate.

ONG: Let the counter C_1 record the number of the migrations performed so far and let the counter C_2 denote the total access costs. If the access cost C_2 reaches the migration cost plus marginal migration contract costs (i.e., $f(C_1+1,b)-f(C_1,b)$, for bandwidth b), ONG migrates the server, increments counter C_1, and resets counter C_2.

Given our optimal offline algorithms, it is interesting to study the *competitive ratio* of ONC and ONG. We conduct simulations with the same three discount functions $f_{\log}$, f_{sqrt} and f_{lin}, the same contract set and the same access cost as in Section 4. The bandwidth used in our experiments is 50 Mbit/s.

The competitive ratios for ONC and ONG are presented in Figure 3. We observe that the ratios for both algorithms are relatively small (between 1.5 and 4) and decrease for larger λ (lower dynamics). This can be explained by the fact that with higher λ, requests



(a) Competitive ratio for ONC. (b) Competitive ratio for ONG.

Fig. 3. Effect of discount function on competitive ratio. We simulate 1500 requests and present the average over five runs.

remain more local and migration patterns more obvious. A second takeaway is that the competitive ratio for the lowest discount function f_{lin} is best, while higher discounts like f_{log} are handled worse by our online algorithms. Especially in the Pay-as-You-Come model, our online algorithm has more difficulties to deal with high discounts, as it tends to buy too many short contracts (ONC migrates more often than the offline algorithm). Also under Pay-as-You-Go pricing, the offline algorithm can exploit discounts relatively better, although to a lesser extent. (The offline algorithm migrates relatively more frequently for higher discounts.)

6 Related Work

Our work is motivated by the advent of first network virtualization prototype architectures such as GENI. For a good overview of the network virtualization field, see [7]. Theoretical research on network virtualization often focuses on the problem of how to *embed* VNets, e.g., [6,19,15] (and especially the survey [4]), while benefitting from specification flexibilities [13]. Naturally, there are also many papers and results on migration (e.g., [1,3,11,18]): the possibility to migrate is one of the key advantages of the virtualization abstraction; it is due to the decoupling of services from the physical infrastructure. Indeed, it has been shown that it can make sense to migration a Samba front-end server closer to the clients even for bulk-data applications [12]. Our work builds upon the formal migration model studied in [3] and ports it to an economical setting.

Economical aspects of network virtualization are much less well-understood, but there exist strong ties with related problems in, e.g., cloud computing. For example, Armbrust et al. [2] made an effort to understand cloud computing economical models for long-term hosting a service in the cloud. Dash et al. [9] proposed an economic model for self-tuned cloud caching targeting the service of scientific data. Recently, Pal and Hui [14] devised and analyzed three inter-organizational economic models relevant to cloud networks, and formulated non-cooperative price and QoS games between

multiple cloud providers existing in a cloud market. In the context of network virtualization, Schaffrath et al. [16] identified stakeholders and economical roles in a network virtualization environment. The authors distinguish between a physical infrastructure provider, a virtual network provider (i.e., resource reseller), a virtual network operator and a service provider. In terms of pricing, Even et al. [10] presented an online algorithm which decides which VNets to accept and embed such that the overall provider *benefit* is maximized. The benefit threshold of when to accept a VNet can be seen as a simple form of pricing. Migration is not considered in [10].

Finally, a description of our own network virtualization prototype (currently using VLANs) which is developed at Telekom Innovation Laboratories and NTT DoCoMo Eurolabs and which motivates our work can be found in [16]. Currently, migration is seamless (i.e., without the need for reconfigurations) but not live. See [8] for a migration demo.

7 Conclusion

There is a large body of literature on economical aspects of cloud computing, but much less is known about efficient (virtual) network pricing. Interestingly, while cloud (or node) resources are often priced according to a flexible *per-use* or pay-as-you-go policy, networking services such as MPLS connectivity are often charged according to usage-independent, time-based policies. [17] This is particularly surprising as network demand is likely to exhibit a higher variance over time than, e.g., storage resources. For instance, distributed SAP systems may be fully synchronized only sporadically (but then lead to high network loads), whereas the resource requirements of, e.g., a mail service normally grows monotonically over time.

We understand this paper as a first step to study the effect of virtual network pricing policies on *service migration*. We focused on the offline setting where demand patterns are given (e.g., describe regular time-of-day or commuter effects). Such online algorithms can also be useful to evaluate the competitive ratio of online algorithms in simulations. We presented two optimal algorithms for efficient service migration in different economic settings. We believe that the used algorithmic techniques are relatively general and can be extended to more complex scenarios, e.g., to networks supporting live migration or more complex virtual network topologies.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments.

References

1. Agarwal, S., Dunagan, J., Jain, N., Saroiu, S., Wolman, A., Bhogan, H.: Volley: automated data placement for geo-distributed cloud services. In: Pro. 7th USENIX NSDI (2010)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53(4), 50–58 (2010)

3. Arora, D., Bienkowski, M., Feldmann, A., Schaffrath, G., Schmid, S.: Online strategies for intra and inter provider service migration in virtual networks. In: Proc. IPTComm (2011)
4. Belbekkouche, A., Hasan, M., Karmouch, A.: Resource discovery and allocation in network virtualization. IEEE Communications Surveys Tutorials (99), 1–15 (2012)
5. Bienkowski, M., Feldmann, A., Jurca, D., Kellerer, W., Schaffrath, G., Schmid, S., Widmer, J.: Competitive analysis for service migration in VNets. In: Proc. ACM VISA, pp. 17–24 (2010)
6. Chowdhury, K., Rahman, M.R., Boutaba, R.: Virtual network embedding with coordinated node and link mapping. In: Proc. IEEE INFOCOM (2009)
7. Chowdhury, N.M.K., Boutaba, R.: A survey of network virtualization. Computer Networks 54, 862–876 (2010)
8. CloudNet. Migration Demo (2012), http://www.youtube.com/watch?v=llJce0F1zHQ
9. Dash, D., Kantere, V., Ailamaki, A.: An economic model for self-tuned cloud caching. In: Proc. IEEE International Conference on Data Engineering, pp. 1687–1693 (2009)
10. Even, G., Medina, M., Schaffrath, G., Schmid, S.: Competitive and Deterministic Embeddings of Virtual Networks. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 106–121. Springer, Heidelberg (2012)
11. Hajjat, M., Sun, X., Sung, Y.-W.E., Maltz, D., Sripanidkulchai, S.R.K., Tawarmalani, M.: Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In: Proc. ACM SIGCOMM (2011)
12. Hao, F., Lakshman, T.V., Mukherjee, S., Song, H.: Enhancing dynamic cloud-based services using network virtualization. SIGCOMM Comput. Commun. Rev. 40(1), 67–74 (2010)
13. Ludwig, A., Schmid, S., Feldmann, A.: The price of specificity in the age of network virtualization (short paper). In: Proc. 5th IEEE/ACM UCC (2012)
14. Pal, R., Hui, P.: Economic Models for Cloud Service Markets. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 382–396. Springer, Heidelberg (2012)
15. Schaffrath, G., Schmid, S., Feldmann, A.: Optimizing long-lived cloudnets with migrations. In: Proc. 5th IEEE/ACM UCC (2012)
16. Schaffrath, G., Werle, C., Papadimitriou, P., Feldmann, A., Bless, R., Greenhalgh, A., Wundsam, A., Kind, M., Maennel, O., Mathy, L.: Network virtualization architecture: proposal and initial prototype. In: Proc. ACM VISA (2009)
17. Schönherr, M.: T-labs berlin. Personal Communication (2012)
18. Wood, T., Shenoy, P., Ramakrishnan, K., der Merwe, J.V.: Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines. In: Proc. ACM VEE (2011)
19. Zhang, S., Qian, Z., Wu, J., Lu, S.: An opportunistic resource sharing and topology-aware mapping framework for virtual networks. In: Proc. IEEE INFOCOM (2012)

Parallel Scalar Multiplication on Elliptic Curves in Wireless Sensor Networks

Yanbo Shou[1], Herve Guyennet[1], and Mohamed Lehsaini[2]

[1] University of Franche-Comte, France
{yshou,herve.guyennet}@femto-st.fr
[2] University of Tlemcen, Algeria
m_lehsaini@mail.univ-tlemcen.dz

Abstract. In event-driven sensor networks, when a critical event occurs, sensors should transmit quickly and securely messages back to base station. We choose Elliptic Curve Cryptography to secure the network since it offers faster computation and good security using shorter keys than RSA. In order to minimize the computation time, we propose to distribute the computation of scalar multiplications on elliptic curves by involving neighbor nodes in this operation. The results of performance tests show that parallel computing certainly consumes much more resources, however it reduces considerably the computation time of scalar multiplications. This method is applicable in event-driven applications when execution time is the most critical factor.

Keywords: Wireless sensor networks, Elliptic curves, Scalar multiplication, Parallel computing.

1 Introduction

A wireless sensor node is a small electronic device which consists of sensing, data processing and communicating components [1]. Such sensor nodes can be programmed to collect environmental data and to communicate with other nodes. A sensor node is often equipped of a low-cost low-power microcontroller which is not capable of doing complicated calculations. As the ability of a single node is very limited, a wireless sensor network is usually constituted of a large number of nodes which are interconnected to each other to form a large network. In most of cases a sensor node has to cooperate with other nodes to achieve a common goal.

Sensors are often deployed in hostile and inaccessible areas for human being. Today, we can find a wide spectrum of sensor networks applications such as environmental monitoring [2], industrial sensing [3], home automation [4], medical care [5] or military surveillance [6].

However sensors are vulnerable and subject of various attacks due to their lack of resources and the unreliability of wireless connection, in [7] we can find the presentation of almost all possible attacks in wireless sensor networks. In order to secure wireless sensor networks, one of the most efficient solutions is to use

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 300–314, 2013.

cryptographic mechanisms [7,8]. Symmetric cryptographic algorithms are usually light weighted and can be efficiently implemented in hardware and software, but as we use the same key for data encryption and decryption, the key management becomes a challenging problem in wireless sensor networks, since the key can be exposed if a node is compromised. An other choice is asymmetric cryptography which is computationally more expensive but it's easier to manage the keys, a compromised node cannot provide clue to the private keys of non-compromised nodes [8]. Of course the security of sensor networks can still be threatened by physical attacks, but the protection against such attack is beyond the scope of this paper.

In this paper we choose the Elliptic Curve Cryptography (ECC) which is one of the most famous asymmetric cryptographic schemes. It has attracted considerable attention recently because of its shorter key length requirement comparing with the other widely used asymmetric cryptographic algorithm RSA. An elliptic curve cryptosystem using a 160-bits key can provide the same security level with a 1024-bit RSA key [9]. The security of elliptic curve cryptography mainly relies on the difficulty of discrete logarithm problem. All the points on a curve form a abelian group whose group law is the point addition which combines 2 points on the curve to get a third one. Based on the point addition, we may then perform point multiplication, also called scalar multiplication, for example $Q = kP$ where Q and P are 2 points on the curve and k is a positive integer. It's extremely difficult to compute the value of k given P and Q if k is big enough. Scalar multiplication is the most expensive operation on elliptic curves and it exists various solutions in the literature to optimize its performance [10], especially for embedded platforms which have very limited CPU power.

Parallelism is an other choice to improve the performance of scalar multiplication. Instead of performing calculations on a single sensor node, we try to split the computing task into smaller pieces which are then distributed to neighbor nodes and carried out simultaneously. In this paper we propose to use parallel computing to accelerate scalar multiplication which is the most complicated and time-consuming operation on elliptic curves. To the best of our knowledge, we are the first who have applied this technique in sensor networks. We find that the parallelization of scalar multiplication is efficient in execution time. The parallel overhead is relatively low comparing with the computation time, since the computing power of embedded microcontrollers are very weak.

We have also studied the memory usage et the energy consumption of parallel computing in wireless sensor networks. As more sensor nodes are involved in the computation, more resources will be consumed. However the objective of our solution is to accelerate the computation on elliptic curves in cases where execution time becomes the most critical factor.

The rest of the paper is organized as follows. Section 2 gives a presentation of basic concepts of elliptic curves, and in section 3 we present the related work of parallelization of scalar multiplication. Section 4 describes our parallel computing scheme for scalar multiplication in wireless sensor networks, and the results of performance test are given in section 5. Section 6 concludes the paper.

2 Basic Concepts of Elliptic Curve

Elliptic curve cryptography were proposed independently by Miller [11] and Koblitz [12] in the 80's. It has attracted researchers' attention in recent years due to its shorter key length requirement comparing with RSA, especially in the domain of embedded systems where devices have limited computing power.

In cryptography we work with the elliptic curves which are defined over a finite field F_q where $q = p^m$ and p is a prime number called the characteristic of F. If $m = 1$, then F is called a prime field, if $q = 2^m$, then F is a binary field.

In this paper we only work with the first case where $m = 1$ and $p \neq 2$ or 3, only for reason of easier explanation. Then a curve can be represented using the simplified Weierstrass equation (see formula 1).

$$y^2 = x^3 + ax + b \tag{1}$$

where the discriminant $\triangle$ of the curve

$$-16(4a^3 + 27b^2) \neq 0$$

All points on the curve, including the point at infinity, form an abelian group whose group law is the point addition. Suppose that $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ are 2 points on the curve and $P_3(x_3, y_3)$ is the sum of $P_1 + P_2$ which can be calculated using the formula 2.

$$\begin{cases} x_3 = s^2 - x_1 - x_2 \\ y_3 = s(x_1 - x_3) - y_1 \end{cases} \tag{2}$$

where

$$s = \begin{cases} (y_2 - y_1)/(x_2 - x_1) & \text{if } P_1 \neq P_2, \text{ addition} \\ (3x_1^2 + a)/2y_1 & \text{if } P_1 = P_2, \text{ doubling} \end{cases} \tag{3}$$

Geometrically P_3 is the reflection about x-axis of the intersection point of the curve with the line through P_1 and P_2.

We may also perform scalar multiplication $Q = kP$ which can be considered as a sequence of consecutive additions.

Q and P are 2 points on the curve and k is a positive integer. The point multiplication can be performed more efficiently than repeating point addition. The most basic method is using the Double and Add algorithm (see algorithm 1).

Suppose that integer k is represented in binary form $k = \sum_{i=0}^{l-1} k_i 2^i$ where l is the length of k. We keep reading k_i from the least significant bit to the most significant one. Every time when we read a bit, point P is doubled, and if k_i is a nonzero bit, $Q = Q + P$. The performance of the algorithm 1 mainly depends on the length of k and number of nonzero bits in its binary representation, the average number of operations needed is l point doublings and $\frac{l}{2}$ point additions.

The performance of point multiplication can be significantly improved by choosing appropriate representation of k, coordinate system and size of finite field.

Algorithm 1. Double and Add algorithm for point multiplication

Data: $k = (k_{l-1}, \ldots, k_1, k_0)_2$, $P \in E$
Result: $Q = kP$
$Q \leftarrow \infty$;
for i *from* 0 *to* $l-1$ **do**
  if $k_i = 1$ **then**
    $Q \leftarrow Q + P$;
  end
  $P \leftarrow 2P$;
end
return Q;

If $P = (x, y)$ is a point on an elliptic curve defined over a prime field F_p in which $-P = (x, -y)$, We can see that the subtraction of points on an elliptic curve is as efficient as addition [13], since $P_1 - P_2 = P_1 + (-P_2)$.

The first optimization possible is to use Non-Adjacent Form (NAF) method which uses a signed digit representation of k (see formula 4), and the average number of nonzero bits is reduced from $\frac{l}{2}$ to $\frac{l}{3}$ where l is the length of k.

$$k = \sum_{i=0}^{l-1} k_i 2^i \text{ where } k_i \in \{0, \pm 1\} \tag{4}$$

We read every digit k_i, if $k_i = 1$, then $Q = Q + P$, but if $k_i = -1$, $Q = Q - P = Q + (-P)$. If the length of k is l, the average number of operations is reduced to l point doubling and $\frac{l}{3}$ point addition (subtraction).

In formula 3 we see that both point addition and doubling need a field inversion which is more computationally expensive than field multiplication. An other optimization is to avoid the computation of field inversion by using projective coordinates.

In Jacobian coordinates, a projective point $(X, Y, Z), Z \neq 0$ is equivalent to affine point $(X/Z^2, Y/Z^3)$, and an elliptic curve is represented by the new equation

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{5}$$

We can then obtain the formula for point doubling in Jacobian coordinates by substituting x by $\frac{X}{Z^2}$ and y by $\frac{Y}{Z^3}$ in formula 2,

$$\begin{cases} X_3 = (3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2 \\ Y_3 = (3X_1^2 + aZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_3 = 2Y_1Z_1 \end{cases} \tag{6}$$

and the formula for point addition in Jacobian coordinates can be derived in the same manner.

There are obviously more multiplications using Jacobian coordinates, but during a point multiplication we only need to perform field inversion once to convert

Table 1. Operation counts for point doubling and addition. A=Affine, J=Jacobian, M=Multiplication, S=Squaring, I=Inversion.

Doubling		Addition	
$2A \to A$	1I,2M,2S	$A + A \to A$	1I, 2M, 1S
$2J \to J$	4M, 4S	$J + J \to J$	12M, 4S

the final result back to affine coordinates. The comparison of operations counts for both coordinate systems are given in table 1[13].

We may also improve the performance of point multiplication by choosing specific finite field. For example the modular reduction in prime field can be very fast if we use recommended NIST primes, like $p_{192} = 2^{192} - 2^{64} - 1$[14].

3 Parallelization of Scalar Multiplication

As previously presented, parallel computing can be used to accelerate computation and balancing workload. A task is divided into smaller ones which are then carried out simultaneously by different processors. The parallel computing of scalar multiplication is a hot research topic in cryptography and various solutions have been proposed in literature, but a lot of them are hardware implementations using multi-core [15] or FPGA [16] architecture.

The paper [17] presents a fast exponentiation method using precomputed table. For example we want to calculate g^n where g is a element of Z/pZ and n is a positive integer of length l. We can represent n in base h as follows

$$n = \sum_{i=O}^{l-1} a_i x_i, \quad 0 \leq a_i \leq h-1 \text{ and } 0 \leq i < l \tag{7}$$

We precompute and store $g^{x_0}, g^{x_1}, \ldots, g^{x_{l-1}}$ in a table, and then g^n can be computed easily using formula 8.

$$g^n = \prod_{d=1}^{h-1} c_d^d, \quad c_d = \prod_{i:a_i=d} g^{x_i} \tag{8}$$

The computation of g^n is consisted of three main steps:

1. Determine the representation of $n = \sum_{i=O}^{l-1} a_i x_i$ in base h.
2. Compute $c_d = \prod_{i:a_i=d} g^{x_i}$.
3. Compute $g^n = \prod_{d=1}^{h-1} c_d^d$.

This method is based on the precomputation, and it can also be applied to point multiplication by precomputing points $2^1P, 2^2P \ldots 2^{l-1}P$.

Most time is spent in the second and third steps, and both of them can be parallelized. For example, if we have $h-1$ processors, then we can parallelize the second step and each processor calculates its c_d separately. In step 3, each processor can calculate a c_d^d for one d.

An other method based on point precomputation is proposed in [18]. Suppose that we want to calculate $Q = kP$ where Q and P are 2 points represented in Jacobian coordinates and k is a positive integer of 160 bits.

We prepare a precomputed table which consists of 62 points.

$$\begin{aligned} A[s] &= \textstyle\sum_{j=0}^{4} a_{s,j} 2^{32j} G \\ B[s] &= \textstyle\sum_{j=0}^{4} a_{s,j} 2^{16+32j} G \end{aligned} \tag{9}$$

where $1 \le s \le 31$ and $a_{s,0}, \ldots, a_{s,4}$ is a binary representation of $s = \sum_{j=0}^{4} a_{s,j} 2^j$. Then the algorithm to compute kP is as follows:

Algorithm 2. Elliptic curve exponentiation based on precomputation

Data: $k = \sum_{i=0}^{l-1} k_i 2^i$, P
Result: kP
for $0 \le j \le 15$ **do**
  $u_j = \sum_{i=0}^{4} k_{32i+j} 2^i$;
  $v_j = \sum_{i=0}^{4} k_{32i+16+j} 2^i$;
end
$A[0] = \infty$;
$B[0] = \infty$;
$T = \infty$;
for i *from* 15 *to* 0 **do**
  $T \leftarrow 2T$;
  $T \leftarrow T + A[u_i] + B[v_i]$;
end
return T;

As this method is also based on precomputation, once the precomputed table is prepared, the exponentiation loop can be performed separately by different processors.

In [19] an other method is proposed for performing parallel scalar multiplication $Q = kP$ with 2 processors using a shared memory.

The first processor initially reads P and then keeps scanning k_i and computing point doubling. It writes $2^i P$ into the buffer whenever a non-zero k_i is detected. The second processor reads $2^i P$ from the buffer and performs additions. The computation is terminated when there is no more $2^i P$ in the buffer.

An other method for performing parallel computing of fast exponentiation is presented in [20]. Suppose that we want to calculate g^R where R is a positive integer of length n. We divide R into h blocks R_i of length $a = \lceil \frac{n}{h} \rceil$, then each R_i is still divided into v smaller blocks $R_{i,j}$ of length $b = \lceil \frac{a}{v} \rceil$ (formula 10).

$$\begin{aligned} R &= R_{h-1} \dots R_1 R_0 &&= \textstyle\sum_{i=0}^{h-1} R_i 2^{ia} \\ R_i &= R_{i,v-1} \dots R_{i,1} R_{i,0} &&= \textstyle\sum_{j=0}^{v-1} R_{i,j} 2^{jb} \end{aligned} \tag{10}$$

Let $g_0 = g$ and define $g_i = g_{i-1}^{2^a} = g^{2^{ia}}$ for $0 < i < h$. Using formula 10, we can express g^R as

$$g^R = \prod_{i=0}^{h-1} g_i^{R_i} = \prod_{j=0}^{v-1} \prod_{i=0}^{h-1} (g_i^{2^{jb}})^{R_{i,j}} \tag{11}$$

If the binary representation of R_i is $R_i = e_{i,a-1} e_{i,a-2} \dots e_{i,0}$ and $R_{i,j} = e_{i,jb+b-1} e_{i,jb+b-2} \dots e_{i,jb+1} e_{i,jb}$, then the formulas 11 can be rewritten as follows

$$g^R = \prod_{k=0}^{b-1} (\prod_{j=0}^{v-1} \prod_{i=0}^{h-1} g_i^{2^{jb} e_{i,jb+k}})^{2^k} \tag{12}$$

If we precompute and store the following values for all $1 \le i < 2^h$ and $0 \le j < v$.

$$\begin{aligned} G[0][i] &= g_{h-1}^{e_{h-1}} g_{h-2}^{e_{h-2}} \dots g_0^{e_0} \\ G[j][i] &= (G[j-1][i])^{2^b} = (G[0][i])^{2^{jb}} \end{aligned} \tag{13}$$

The formula 12 can be still rewritten as

$$g^R = \prod_{k=0}^{b-1} (\prod_{j=0}^{v-1} G[j][I_{j,k}])^{2^k} \tag{14}$$

where $I_{j,k} = e_{h-1,bj+k} \dots e_{1,bj+k} e_{0,bj+k} (0 \le j < b)$. Then the computation of g^R can be parallelized using precomputed $G[j][i]$.

We can see that all those parallelization schemes are based on the point precomputation and scalar decomposition. The configuration of elliptic curves, like field type, curve form and coordinate system, doesn't have any impact on the result of calculation. Thus the optimization methods presented in section 2 can be used with the parallelization schemes to improve their performance.

4 Parallel Scalar Multiplication in Wireless Sensor Networks

In this section we present our method of parallel computing for accelerating scalar multiplication on elliptic curves in wireless sensor networks. A sensor network is composed of a great number of low-cost and low-power sensor nodes which are always deployed in hostile territories and then work in unattended mode. These sensors don't have enough power to perform complicated calculations, but in some disaster monitoring applications, sensors might have only a few seconds to send message back to base station before being destructed.

Our method is designed for event-driven applications in which sensors remain idle during most of the time to preserve energy, but when a sensor detects a critical event, the message should be sent back to base station as fast as possible, even at the expense of consuming more energy. We suppose that in a cluster-based sensor network, sensors use a symmetric cryptographic primitive, like Trivium [21,22], to secure internal cluster communications. For inter-cluster communication, we use the elliptic curve cryptography since it provides increased security and digital signature.

Our method is mainly based on the idea of [20] since it offers a efficient scalar decomposition and it doesn't require shared memory [19]. The goal is to reduce the execution time by asking neighbour nodes to perform computations together. We suppose that the elliptic curve is preloaded before node deployment, and the generator point G doesn't change during the entire lifetime of the network. When a node wants to perform a multiplication $Q = kG$, the node looks for available neighbours in the same cluster and asks them to participate in computation. The node which leads the computation is the *master node* and the other ones are called *slave nodes*.

At first the master node splits the integer k into n blocks B_i of $b = \lceil \frac{l}{n} \rceil$ bits according to the number of available neighbours.

$$B_i = \sum_{j=ib}^{ib+b-1} a_j 2^j \tag{15}$$

As the generator point G is chosen a priori and it doesn't change, the precomputation of points $G_i = 2^{ib}G$ is then possible, and calculation of kG

$$Q = kG = \sum_{i=0}^{l-1} a_i 2^i G \tag{16}$$

can be divided into n independent parts

$$\begin{aligned} Q_0 &= B_0 G \\ Q_1 &= B_1 2^b G \\ &\dots \\ Q_{n-1} &= B_{n-1} 2^{b(n-1)} G \end{aligned} \tag{17}$$

Then $Q = Q_0 + Q_1 + \ldots + Q_{n-1}$ and each Q_i can be computed independently using the basic Double and Add algorithm.

Before task distribution, the master node copies one of the n blocks into its local memory, for example the block B_0, and then places the remaining blocks B_i where $0 < i < n$ into a task distribution message, and task assignment informations are also sent with B_i.

For example, k is a positive integer of 160 bits and 4 nodes participate in computation of kG. Thus $n = 4$, $b = 40$ and all nodes have G_i for $0 \leq i \leq 3$ precomputed and stored in theirs memories, like in figure 1. The master splits k

into 4 blocks of 40 bits. It keeps the block B_0 in its local memory and places the remaining 3 blocks into a task distribution message. Then it attaches task assignment informations to the message, encrypts it using the cluster's cryptographic primitive and broadcasts it to slave nodes.

$G[0]$	$G[1]$	$G[2]$	$G[3]$
G	$2^{40}G$	$2^{80}G$	$2^{120}G$

Fig. 1. Array containing precomputed points

An example of task assignment is given in figure 2. ID_i are the IDs of slave nodes. When the slave nodes receive the message, they decrypt it and calculate respectively $B_1G[1]$, $B_2G[2]$ and $B_3G[3]$.

0	1	2	3	4	5
B_1	B_2	B_3	ID_1	ID_2	ID_3

Fig. 2. Structure of the task distribution message

The entire procedure is driven by a simple protocol (see diagram 3). When a critical event takes place, like forest fire, volcanic eruption or earthquake. There's a chance that several nodes detect this event at almost the same time. All nodes which detect the event should turn on the radio and get ready for parallel computing. The protocol is described by the following steps :

1. A node detects a critical event and it turns on its radio. Then it waits for call of parallel computing from other nodes for a random period.
2. If it doesn't receive any call, then it becomes the Master node, and it broadcasts a call for parallel computing.
3. If it receives a call from an other node, then it becomes a Slave node and it should reply to show its availability.
4. After received all replies from slave nodes, the master node broadcasts the task distribution message and then waits for results.
5. Slaves receive the task distribution message and perform theirs computations.
6. Slaves send theirs results back to their master and get back to initial state.
7. The master node combines the received results to get the final result and sends it back to base station.

As this method is based on precomputation too, the maximum number of points to precompute and store depends on the requirement of the application and the memory size of sensor nodes. More points are precomputed, more nodes can participate in parallel computing, but more energy will be consumed.

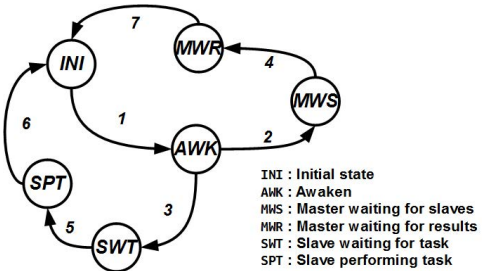


Fig. 3. Protocol of proposed parallel computing scheme

5 Experimental Studies

To test the performance of our method, we have implemented it in nesC on Crossbow's Telosb motes [23] which are then deployed randomly in a zone of $10m \times 10m$. We ask them to keep repeating scalar multiplications $Q = kG$ on a preloaded elliptic curve defined over $NIST_{192}$ prime field [14] for both affine and Jacobian coordinates, G is the generator point of the curve and the scalar k is an integer of 160 bits using NAF representation.

The execution times in milliseconds with and without parallel computing are given in table 2. It's hard to compare the absolute values of the results to other implementations due to the variety of techniques and test scenarios. Here we only interest in the performance gain produced by parallel computing.

Table 2. Execution time (ms) and gain of proposed parallel computing scheme

Nb of nodes	Affine	Gain	Jacobian	Gain
1	2307.27	–	1003.55	–
2	1189.96	48.43%	549.71	45.22%
3	861.48	62.66%	424.60	57.69%
4	665.68	71.15%	342.51	65.87%
5	583.29	74.72%	309.93	69.12%
6	581.32	74.80%	311.87	68.92%

We can see that the gain increases when more nodes participate in the calculation. Figure 4 shows that the execution time decreases gradually with increase in the number of nodes. Suppose that the execution time using p nodes is T_p, we evaluate the performance of our method by calculating its speedup $S_p = \frac{T_1}{T_p}$ and the results are given in table 3 and represented graphically in figure 5.

There is a considerable drop in speedup when we use more than 5 nodes, since the network needs more time to schedule radio communications and the master node requires more time to combine the received points and get the final result.

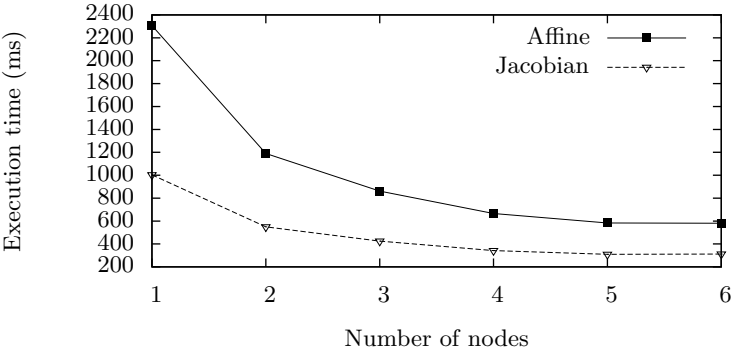


Fig. 4. Execution times using the proposed parallel computing scheme

Table 3. Speedup of the proposed parallel computing scheme

Nb of nodes	1	2	3	4	5	6
Affine	1.00	1.94	2.68	3.47	3.96	3.97
Jacobian	1.00	1.83	2.36	2.93	3.24	3.22

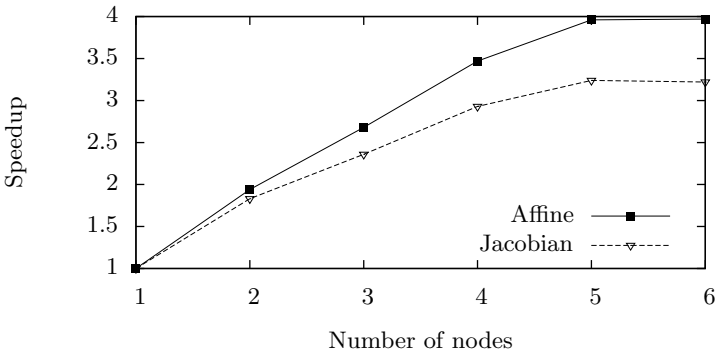


Fig. 5. Speedup $S_p = \frac{T_1}{T_p}$ of the proposed parallel computing scheme

The parallel overhead is shown in table 4 and in figure 6. We can see that when we use more than 5 nodes, there is a significant increase in overhead. Thus according to the result of our experiment, the number of nodes should be limited to less than 5.

Table 4. Overhead (ms) of the proposed parallel computing scheme

Nb of nodes	1	2	3	4	5	6
Average overhead	0.00	36.33	92.39	88.86	121.84	196.78

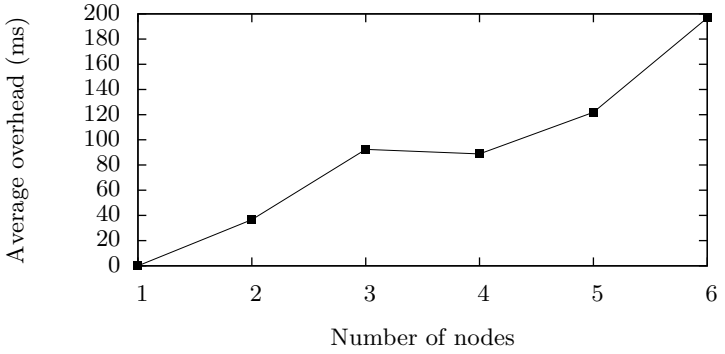


Fig. 6. Parallel overhead (ms) of the proposed method

Suppose that the elliptic curve is defined in a field of 192 bits [14], so a point in affine coordinate (x, y) can be represented by 2 integers of 192 bits, then it can also be converted to Jacobian coordinates $(x, y) \rightarrow (x, y, 1)$. The memory size needed to store precomputed points are given in table 5.

Table 5. Memory needed (Byte) to store precomputed points

Nb of nodes	1	2	3	4	5	6
Memory needed	0	48	96	144	192	240

Telosb mote has 48KB of Flash memory [23] and Micaz has 128KB [24], which is obviously sufficient to store those precomputed points.

As previously presented, during the parallel computing more nodes are involved in the computation, so more energy is consumed since they need to communicate between them. In order to estimate the energy consumption of our method, we have run simulations with Avrora [25]. It gives only theoretical simulated results, but they're precise enough to compare the energy consumption according to the number of nodes participating in the computation (see table 6).

In figure 7, we can see that when computation is done on a single node, it consumes very little energy. However when parallel computing is used, as slave nodes have to receive tasks from the master node and return the results back to it, they consume much more energy.

Table 6. Energy consumption (Joule) of proposed method. TEC: Total energy consumption.

Nb of nodes	1	2	3	4	5	6
TEC	0.889	2.125	3.156	4.192	5.225	6.256

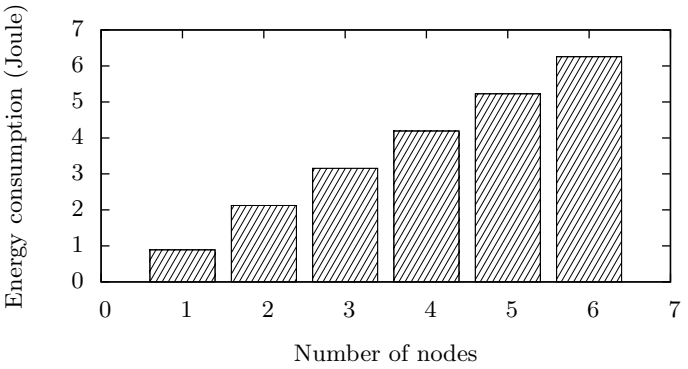


Fig. 7. Total energy consumption (Joule)

When we parallelize the calculation between 6 nodes, our method provides a gain of around 70.0% in execution time. We have also tried to encrypt and decrypt the task distribution message using the symmetric cryptographic primitive Trivium, the result shows that the calculation can be done in less than 1 millisecond, thus its impact on the result of the performance test can be safely neglected.

However as previously presented, in sensor networks sensor nodes work in unattended manner and communicate with each other using wireless connection which might be unstable due to radio interferences and low battery level. In such cases the master node should be able to detect the faults. For example we may divide the computation into 2 tasks which are then carried out by 3 nodes. The master node compare the results returned by 2 slaves, if they're different, it means that 1 of the 2 results is wrong. We may also apply trust and reputation assessment techniques in our system [26]. If a node always returns erroneous results, it will be excluded from the parallel computing. There are still other strategies, but the presentation of fault tolerance techniques is not the objective of this paper.

6 Conclusion

In this paper we use parallel computing technologies to accelerate the scalar multiplication on elliptic curves. We have tested our method using up to 6 Telosb motes, and the results show that we obtain a gain of about 70.0% in execution

time. However we propose that the number of nodes should be limited to less than 5 due to parallel overhead. As the method is based on precomputation, nodes need to store precomputed points locally.

The only drawback is the energy consumption since nodes have to communicate with each other for task distribution and result retrieval. Thus parallel computing should not be used as the default computation mode in wireless sensor networks, it can only be used in cases where execution time is the most critical factor, like in disaster monitoring and military applications.

In our future work, we will try to reduce the energy consumption by minimizing radio transmissions. As nodes communicate with each other using unreliable wireless communication, fault tolerance will also be taken into account.

References

1. Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks 38(4), 393–422 (2002)
2. Werner-Allen, G., Lorincz, K., Ruiz, M., Marcillo, O., Johnson, J., Lees, J., Welsh, M.: Deploying a wireless sensor network on an active volcano. IEEE Internet Computing 10(2), 18–25 (2006)
3. Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., Turon, M.: Wireless sensor networks for structural health monitoring. In: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, pp. 427–428. ACM (2006)
4. Baker, C., Armijo, K., Belka, S., Benhabib, M., Bhargava, V., Burkhart, N., Der Minassians, A., Dervisoglu, G., Gutnik, L., Haick, M., et al.: Wireless sensor networks for home health care. In: 21st International Conference on Advanced Information Networking and Applications Workshops, AINAW 2007, vol. 2, pp. 832–837. IEEE (2007)
5. Welsh, M., Moulton, S., Fulford-Jones, T., Malan, D.: Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In: International Workshop on Wearable and Implantable Body Sensor Networks, London, UK (April 2004)
6. Gosnell, T., Hall, J., Jam, C., Knapp, D., Koenig, Z., Luke, S., Pohl, B., Schach von Wittenau, A., Wolford, J.: Gamma-ray identification of nuclear weapon materials. Technical report, Lawrence Livermore National Lab, Livermore, CA, US (1997)
7. Walters, J., Liang, Z., Shi, W., Chaudhary, V.: Wireless sensor network security: A survey. Security in Distributed, Grid, Mobile, and Pervasive Computing 1, 367 (2007)
8. Zhou, Y., Fang, Y., Zhang, Y.: Securing wireless sensor networks: a survey. IEEE Communications Surveys & Tutorials 10(3), 6–28 (2008)
9. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.: Comparing elliptic curve cryptography and rsa on 8-bit cpus. In: Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2004, Cambridge, MA, USA, August 11-13, vol. 6, p. 119. Springer-Verlag New York Inc. (2004)
10. Gordon, D.: A survey of fast exponentiation methods. Journal of Algorithms 27(1), 129–146 (1998)
11. Miller, V.S.: Use of Elliptic Curves in Cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
12. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48(177), 203–209 (1987)

13. Hankerson, D., Vanstone, S., Menezes, A.: Guide to elliptic curve cryptography. Springer-Verlag New York Inc. (2004)
14. Brown, M., Hankerson, D., López, J., Menezes, A.: Software Implementation of the NIST Elliptic Curves Over Prime Fields. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 250–265. Springer, Heidelberg (2001)
15. Panda, B., Khilar, P.: Fpga based implementation of parallel ecc processor. In: Proceedings of the 2011 International Conference on Communication, Computing & Security, pp. 453–456. ACM (2011)
16. Purnaprajna, M., Puttmann, C., Porrmann, M.: Power aware reconfigurable multiprocessor for elliptic curve cryptography. In: Design, Automation and Test in Europe, DATE 2008, pp. 1462–1467. IEEE (2008)
17. Brickell, E.F., Gordon, D.M., McCurley, K.S., Wilson, D.B.: Fast Exponentiation with Precomputation: Algorithms and Lower Bounds. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 200–207. Springer, Heidelberg (1993)
18. Miyaji, A., Ono, T., Cohen, H.: Efficient Elliptic Curve Exponentiation. In: Han, Y., Quing, S. (eds.) ICICS 1997. LNCS, vol. 1334, pp. 282–290. Springer, Heidelberg (1997)
19. Ansari, B., Wu, H.: Parallel scalar multiplication for elliptic curve cryptosystems. In: Proceedings of the 2005 International Conference on Communications, Circuits and Systems, vol. 1, pp. 71–73. IEEE (2005)
20. Lim, C.H., Lee, P.J.: More Flexible Exponentiation with Precomputation. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 95–107. Springer, Heidelberg (1994)
21. De Canniere, C., Preneel, B.: Trivium specifications. estream, ECRYPT Stream Cipher Project, Report 30 (2005)
22. Raddum, H.: Cryptanalytic results on trivium. estream, ECRYPT Stream Cipher Project, Report 39 (2006)
23. MEMSIC: Telosb mote platform datasheet (November 2011), http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html
24. MEMSIC: Mica2/micaz mote platform datasheet (June 2011), http://www.memsic.com/products/wireless-sensor-networks/wireless-modules.html
25. Titzer, B., Lee, D., Palsberg, J.: Avrora: Scalable sensor network simulation with precise timing. In: Fourth International Symposium on Information Processing in Sensor Networks, IPSN 2005, pp. 477–482. IEEE (2005)
26. Chen, H., Wu, H., Zhou, X., Gao, C.: Reputation-based trust in wireless sensor networks. In: International Conference on Multimedia and Ubiquitous Engineering, MUE 2007, pp. 603–607. IEEE (2007)

PeerVault: A Distributed Peer-to-Peer Platform for Reliable Data Backup

Adnan Khan[1], Mehrab Shahriar[1], Sk Kajal Arefin Imon[1], Mario Di Francesco[2,1], and Sajal K. Das[1]

[1] Center for Research in Wireless Mobility and Networking (CReWMaN)
University of Texas at Arlington
{firstname.lastname}@mavs.uta.edu, das@uta.edu
[2] Department of Computer Science and Engineering
Aalto University School of Science
mario.di.francesco@aalto.fi

Abstract. Large scale peer-to-peer (P2P) systems are envisioned as a way to provide online storage service. For a reliable storage service, the participating peers are required to maintain strict commitments for their online duration. On the other hand, recent results show that users participating in volunteer computing collectively exhibit certain patterns in terms of their long-term availability, a metric that denotes periodic online durations for a considerably long time interval. In this article we introduce PeerVault, a P2P platform that leverages the long-term availability of the computer users to form a distributed reliable storage service, targeted to backup of personal data. We further present a distributed monitoring scheme that assists PeerVault to detect peer churns and ensure the reliability of the proposed backup service. To the best of our knowledge, this is the first effort to describe the architecture of a reliable P2P backup service exploiting the long-term availability and idle resources of computing devices. We conduct experiments based on the availability traces of hundreds of thousands of hosts from the SETI@home computing project. The obtained results show that the proposed approach is effective in terms of availability as well as reliability of the offered backup service.

Keywords: Peer-to-peer backup, distributed storage, reliability, monitoring.

1 Introduction

Traditional approaches for reliable storage of data involve maintaining redundant *backup* copies on a variety of storage devices which evolved over time from tapes and optical media to external hard drives. More recently, the availability of storage space in data centers and ubiquitous access to the Internet have made remote storage solutions increasingly appealing. In this context, cloud-based services such as Amazon Cloud Drive, Microsoft SkyDrive and DropBox have become very popular. Such systems are easy to use and provide seamless service

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 315–329, 2013.

as they exploit the reliability inherent to the cloud infrastructure. Most of these services provide a limited storage space without any charge to attract new users. Subsequently, users pay for additional storage space on a monthly or yearly basis.

Although computer users keep on purchasing additional storage for data backup, a significant portion of their own local storage remains unutilized. According to [1], almost half of the users consistently utilize less than 40% of their available disk space. Therefore, a different approach to design a backup system is to exploit the unutilized disk space and bandwidth in a peer-to-peer (P2P) architecture. Such *P2P backup systems* have also received attention by the research community [2] and even resulted in actual commercial products [3, 4] in the last few years. However, most of the existing systems require the users to be online for a considerably long period of time, e.g., more than 80% time of a day [4]. Moreover, these systems require the users to contribute their own storage space to receive the backup service [3] and thus excludes the users who are unwilling to contribute their own resources.

In this article, we introduce *PeerVault*, a platform which exploits the long-term availability of online computer users, as well as their idle resources, in order to realize a distributed online backup service based on a P2P infrastructure. In this approach, participating users advertise their unused storage and network resources based on which PeerVault decides how to store the data by ensuring their long-term availability. To receive the backup service, users are not explicitly required to contribute any resources. Even though the backup service can be supported by an appropriate revenue model [5], in this work we focus on the architectural aspects of the system. The major contributions of this article are as follows.

- We design a novel distributed storage system based on erasure coding which realizes a seamless online backup service on top of idle peers connected over the Internet.
- We propose the concept of *peer path* to derive an efficient solution for distributing data to the peers. Peer paths encapsulate individual peer availabilities to offer a seamless backup service over a given time interval.
- We devise a distributed monitoring scheme to detect peer churn. The proposed algorithm is shown to monitor all the involved peers with high probability, while incurring a nominal bandwidth.
- Through extensive simulations based on the traces of the SETI@home project [6], we show that the proposed approach is effective in terms of long-term service availability.

The remainder of the article is organized as follows. Section 2 details the proposed PeerVault architecture with focus on the feasibility of the offered service. Section 3 introduces a randomized scheme to monitor peer churn in our system. Section 4 presents the details of the simulation setup and the obtained results. Section 5 summarizes the related work and, finally, Sect. 6 concludes the article with directions for future research.

2 PeerVault Architecture

The proposed PeerVault architecture is based on the three basic components illustrated in Fig. 1. The *source peers* are the end-users of the system and are willing to store data (namely, *files*) in exchange for a high reliability. On the other hand, the *storage peers* provide their bandwidth and storage resources to realize the distributed backup service. Finally, the *tracker* supervises the resources offered by the storage peers as well as the mapping between files and peers. Source peers can request a certain amount of remote storage space for a particular period of time, with a minimum bandwidth desired for uploading or retrieving the data. Similarly, a storage peer can choose the amount of space it is willing to share, the minimum upload and download bandwidth, and its availability periods.

Throughout our discussion, the *availability* of a storage peer will refer to its compliance with the advertised resources. We will refer to *service availability* of PeerVault at a given time instant as the accessibility of the stored files at that particular instant. Moreover, we will refer to *service reliability* as the long-term availability (i.e., in a sufficiently large time period) of the offered service. Since PeerVault is based on a P2P infrastructure, intermittent deviation from the advertised resources and also permanent departure of the storage peers are possible. The service availability of PeerVault relies on the group availability of the storage peers instead of the individual availabilities. Thus, service availability can be ensured, even when storage peers have some deviation from their advertised resources. Moreover, in Sect. 3 we explicitly provide a mechanism to detect and adjust with the deviations to ensure service reliability.

2.1 Distributed Storage Scheme

In PeerVault, a file is distributed by a source peer to a set of storage peers in the form of chunks. We exploit erasure coding to create these chunks from a given file. The basic idea behind this approach is to encode data by adding some redundancy. As a result, the original data can be obtained from the encoded data even when part of them is not available. Erasure coding operates on individual *chunks* of a file, where each chunk is of fixed size λ. In the following, we will assume that the source data (i.e., a file) is split into k chunks, and then encoded into $n = \eta k$ chunks, where η is the *replication factor* (see Fig. 1). Erasure coding guarantees that the original file can be reconstructed from any k distinct encoded chunks among the n encoded ones.

A suitable value of η is obtained through a preliminary negotiation phase between the source peer and the tracker, based on the resources available in the system. After that, the source peer applies erasure coding on the given file to produce n different chunks. The tracker derives a mapping between an encoded chunk and a set of storage peers, known as a *peer path*. The storage peers in the mapping are selected based on their advertised resources. Thus, for the entire file (i.e., the n encoded chunks), the tracker finds n peer paths and provides the related mapping to the source peer. The tracker also ensures that no storage

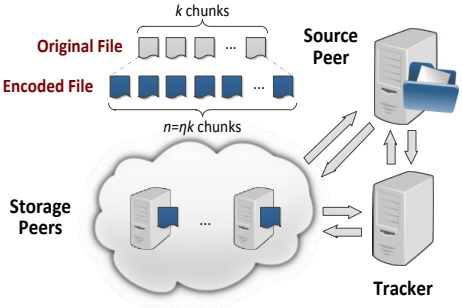


Fig. 1. System Architecture of PeerVault

peer receives more than $k-1$ chunks of a given file. As a consequence, no storage peer can reconstruct or access the given file.

2.2 Characterization of Storage Peers

Different storage peers provide their resources during different time intervals. On the other hand, a source peer may need to store or retrieve a file at any time instant. In the following, we will build our storage scheme based on the availability of the storage peers so that the requirements of the source peers are successfully satisfied.

First, we denote the i-th storage peer by means of its unique identifier, p_i. We assume that the availability of storage peers is periodic over a time frame, defined as *service time frame*. Specifically, the availability of storage peers is characterized in terms of the considered service time frame. For instance, a given peer could be available from Monday to Friday between 12 AM to 8 PM when the service time frame is equal to one week. Within a service time frame, a peer can be available during multiple contiguous time intervals, referred to as the *availability periods*. We represent the j-th availability period of p_i as p_{ij}. In detail, we define as *arrival time* and *departure time* the instants corresponding to the beginning and the end of a single (contiguous) availability period, respectively. For a given availability period p_{ij}, we denote the corresponding arrival time as $a(p_{ij})$ and the departure time as $d(p_{ij})$. Each availability period p_{ij} is associated with its *offered bandwidth* $b(p_{ij})$, which is the minimum between the upload and download bandwidths of the storage peer during the availability period. Moreover, each availability period has an associated cost per unit storage, represented by $c(p_{ij})$. The *duration* of an availability period is denoted by $A(p_{ij}) = (a(p_{ij}), d(p_{ij}))$. The *overlapping time* between two availability periods p_{ij} and p_{kl} is finally defined as $T(p_{ij}, p_{kl}) = \min\{d(p_{ij}), d(p_{kl})\} - \max\{a(p_{ij}), a(p_{kl})\}$ if $d(p_{kl}) > a(p_{ij})$ and $d(p_{ij}) > a(p_{kl})$, otherwise $T(p_{ij}, p_{kl}) = 0$.

Let us consider the example scenario represented in Fig. 2a. For clarity, we assume that each storage peer has a single availability period, denoted by a single subscript corresponding to the peer identifier (i.e., p_i represents the only

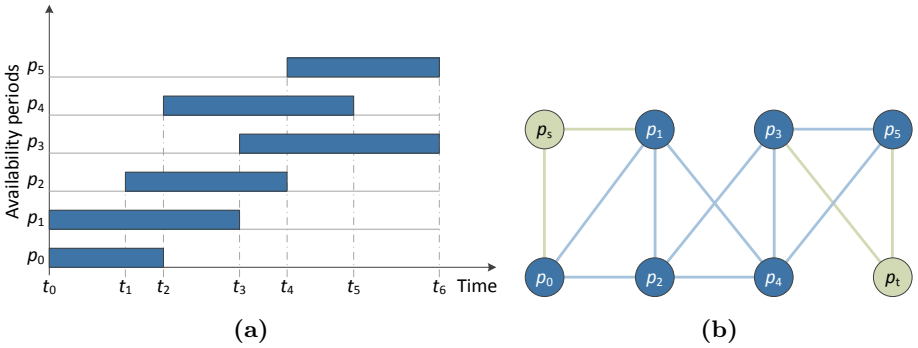


Fig. 2. (a) Availability periods of different storage peers as a function of time. (b) Interval graph corresponding to (a) with the addition of dummy nodes p_s and p_t.

availability period of the i-th peer). The durations of the availability periods p_1 and p_2 are $A(p_1) = (t_0, t_3)$ and $A(p_2) = (t_1, t_4)$, respectively. Note here that the availability period p_0 overlaps with both p_1 and p_2. Specifically, the overlapping time between p_0 and p_1 is $T(p_0, p_1) = t_2 - t_0$, while that between p_0 and p_2 is $T(p_0, p_2) = t_2 - t_1$.

2.3 Managing Storage Requests

The backup service is requested by a source peer (for an individual file) in terms of the following parameters: the *target availability interval* (δ_s, δ_e); the desired minimum download bandwidth μ; and the requested storage space ρ. We assume that the chunk size for the given file is λ and that the target availability interval requested by the source peer is equal to the service time frame.

We use interval graphs [7] to model the considered scenario. An undirected graph $G = (V, E)$ is called an *interval graph* if a one-to-one mapping between the vertices V and a set of intervals I can be established, such that two vertices are connected by an edge in G if and only if there is an intersection between the corresponding intervals. In our case, $V = \{p_{ij}\}$ and $I = \{I_{ij}\} = A(p_{ij}) = \{(a(p_{ij}), d(p_{ij}))\}$ for $0 \leq i < m$ and $0 \leq j < n_i$, where m is the number of storage peers and n_i is the number of availability periods of p_i.

We construct a constrained interval graph, G_c, for the given storage request according to the availability periods of the storage peers. Let us assume, for an availability period p_{ij}, the offered bandwidth and the cost are denoted by $b(p_{ij})$ and $c(p_{ij})$, respectively. Now we restrict the nodes in the graph G_c to those with offered bandwidth higher than $\frac{\mu}{k}$. Furthermore, we restrict the edges between any two nodes p_{ij} and p_{kl} so that their overlapping time is longer than the minimum overlapping time τ, where $\tau = (\min\{b(p_{ij}), b(p_{kl})\})^{-1} \cdot \lambda$. Note that a chunk stored in a peer can be transferred to the next peer along the associated peer path in the minimum overlapping time. Finally, we define the weight of an edge between nodes p_{ij} and p_{kl} as $w(p_{ij}, p_{kl}) = \frac{c(p_{ij}) + c(p_{kl})}{2}$.

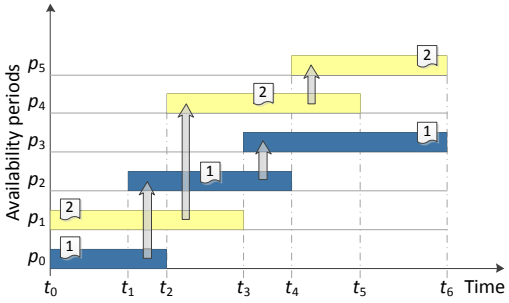


Fig. 3. Dissemination of file chunks

On the basis of the target availability interval (δ_s, δ_e), we add two dummy availability periods p_s and p_t, so that a storage request can be mapped to a path between a single source and a single destination in G_c. The duration of the availability periods associated to the dummy nodes are set to $A(p_s) = (\delta_s, \delta_s + \tau)$ and $A(p_t) = (\delta_e - \tau, \delta_e + \tau)$, respectively. We also set $b(p_s) = b(p_t) = \frac{\mu}{k}$ and $c(p_s) = c(p_t) = 0$. As a consequence, a peer path can be referred by a path between p_s and p_t in G_c. Formally, a *peer path* associated with the interval (δ_s, δ_e) is the set of m availability periods $\mathcal{P}(\delta_s, \delta_e) = \{p_{i_1 j_1}, p_{i_2 j_2}, \ldots, p_{i_m j_m}\}$ such that $T\left(p_{i_x j_y}, p_{i_{x+1} j_{y+1}}\right) > \tau$, $\forall i \in [1, m]$, $a(p_{i_1 j_1}) \leq \delta_s$ and $d(p_{i_m j_m}) \geq \delta_e$. For instance, $\mathcal{P}(t_0, t_6) = \{p_s, p_0, p_2, p_3, p_t\}$ is a peer path in Fig. 2b. Note that the parameters assigned to the dummy nodes ensure the inclusion of the peers with the required amount of overlapping time in a peer path.

For a storage request with n encoded chunks, our system associates a distinct peer path $\mathcal{P}_r(\delta_s, \delta_e)$, with $0 \leq r < n$, to each of the chunks in the source file. For a storage request, we intend to assign at most one chunk to a single availability period so that any interruption during this availability period has minimal impact. Moreover, a storage peer is not allowed to receive more than $k - 1$ chunks of the file, even though it may have multiple availability periods. Otherwise, it would be possible for a storage peer to obtain k or more chunks and reconstruct the file. As a consequence, at most $k - 1$ availability periods of a storage peer are allowed to belong to a single storage request. To this end, for each storage peer p_i, we sort the availability periods based on the weight $\psi(p_{ij})A(p_{ij})$, for $0 \leq j < n_i$ in decreasing order, where $\psi(p_{ij})$ represents the probability of p_i being online during p_{ij} as explained in Sect. 2.5. Thus, we further restrict G_c by taking the top $k - 1$ availability periods from the sorted list.

Finally, to serve the storage request, PeerVault selects the set of peer paths $\mathcal{X} = \left\{\cup_{j=0}^{n-1} \mathcal{P}_j(\delta_s, \delta_e)\right\}$ so that $\mathcal{P}_i \cap \mathcal{P}_j = \{p_s, p_t\}$, $\forall i \neq j$ and the total cost of the availability periods in the selected peer paths is minimized. Note that this problem can be mapped to the *minimum weight n-node disjoint path problem* in an undirected graph [8] which is well studied in the literature, and can be solved in polynomial time [9].

2.4 Data Dissemination and Retrieval

According to the definition of peer path, at any time instant, a storage peer can be found online. When a source peer intends to backup a file, it creates n encoded chunks and sends a storage request to the tracker. The tracker selects a set of n peer paths and sends it back to the source peer. Now the source peer uploads each chunk to the currently available storage peer from each peer path. After all the chunks are uploaded, the source peer can leave the system. Once a storage peer of a peer path receives a chunk, it transfers the chunk to the next storage peer of the peer path. Thus the chunk is propagated to all the storage peers of the peer path. This dissemination process for a file consisting of two encoded chunks is illustrated in Fig. 3 for the scenario already introduced in Fig. 2a. In this example, the tracker reports two peer paths to the source peer, namely $\mathcal{P}_1 = \{p_0, p_2, p_3\}$ and $\mathcal{P}_2 = \{p_1, p_4, p_5\}$. If the source peer is online at t_0, it can upload chunk 1 to p_0 and chunk 2 to p_1, and may leave the system.

When the source peer decides to retrieve the stored file, it selects k distinct peer paths and proceeds to download the chunks from the currently available storage peer of each peer path. Once k chunks are successfully downloaded, the source peer reconstructs the original file.

2.5 Estimating Available Resources

As the offered backup service is largely dependent on the long-term availability of the storage peers, it is essential to know the relevant parameters of a storage peer – namely, availability periods, bandwidth, and storage space – before it is actually allowed to participate in the system. Unlike some existing approaches [4], we do not rely on the user to define the expected operating parameters. Instead, PeerVault observes the users for a *training period* denoted by σ.

We use the bit vector method similar to [10] to predict the long-term availability of the storage peers. Consistent with that solution, we consider the service time frame of one week, wherein each hour of the week is represented by a bit. For each hour, the corresponding bit is set to 1 if a storage peer is available for more than 55 minutes. The peer is observed for each hour in the entire training period. Let us assume that there are y weeks in σ, and a bit is set for x weeks. Then the *training probability* of the corresponding hour is defined by $\frac{x}{y}$. We consider an hour to include in an availability period if the training probability exceeds a threshold α_b. Finally, the availability periods are obtained by merging the contiguous available hours. The training probability of the availability period is denoted by $\psi(\cdot)$ and computed by taking the average of the probabilities of the constituent hours. A storage peer is considered as eligible if it has at least one availability period with training probability greater than α_b. At the end of the training period, an eligible peer is requested to approve its estimated availability periods and specify the information of the free disk space, bandwidth, and cost it can offer to PeerVault. Subsequently, the associated cost per unit storage, $c(\cdot)$ is derived through a revenue model. These sets of parameters are referred as the advertised resources of the storage peer for the considered availability period. A specific choice of the revenue model is out of the scope of this paper.

3 A Distributed Peer Monitoring Scheme

To ensure the long-term availability of the stored data, peer churns must be detected and the corresponding peers need to be replaced accordingly. In this section, we introduce a distributed algorithm to monitor storage peers and detect churn. In our approach, each storage peer sends a ping message to a set of other peers to monitor whether or not they are maintaining their advertised resources. Our algorithm, called DistMonitor has the following properties: (i) the absence of a storage peer is reported to the tracker with high probability; (ii) the overhead of the monitoring effort is proportional to the number of stored chunk and, hence, is fairly distributed; (iii) newly joined peers can easily be included in the monitoring process, thus making the solution scalable; (iv) most of the monitoring overhead is assigned to the peers themselves, while only limited interactions with the tracker are needed; and (v) overall, the required bandwidth for the monitoring scheme is nominal.

For convenience of discussion, let γ_k denotes a particular availability period. Let $h(\cdot)$ be a one-to-one function that maps an ordered pair of integers $< i, j >$ to a single integer k. Thus, γ_k represents a unique availability period p_{ij}.

Definition 1 (Simultaneous Availability Period List). *Let n chunks of a file be stored among a set of peers with availability periods $P = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$. The* simultaneous availability period list *(SAPL) for a given availability period $\gamma_i \in P$ is the set $\mathcal{S}_i \subseteq P$ such that $T(\gamma_i, \gamma_j) > t_{max}$, for all $\gamma_j \in S_i \setminus \{\gamma_i\}$ and t_{max} is a predefined timeout period greater than zero.*

Definition 2 (Potential Availability Period List). *The* potential availability period list *(PAPL) $\mathcal{N}_i$ of a given availability period γ_i is a randomly selected proper subset of $\mathcal{S}_i$.*

After finding the peer paths for a given file, the tracker computes the PAPL for each of the availability periods. Basically, when a storage peer participates in storing a file (by holding a chunk during a particular availability period), it is also assigned with the PAPL.

Once a storage peer p_k obtains the PAPL for a particular availability period γ_i, it executes the function DistMonitor illustrated in Algorithm 1. Specifically, the storage peer selects q random availability periods from the PAPL of γ_i and assigns it to a set $\mathcal{B}$ (line 1). For each member of $\mathcal{B}$, there is a counter (*count*) initialized with a value l (line 2). For each of the availability periods γ_j from $\mathcal{B}$, the peer p_k selects a random time in the overlapping time $T(\gamma_i, \gamma_j)$ and sends a ping message at that particular time to the storage peer associated with γ_j, namely, $g(\gamma_j)$ (line 11). Note that $g(\cdot)$ is a function that returns the identifier of a peer corresponding to a given availability period. After sending the ping message, p_k waits for the reply for a predefined timeout period t_{max}. If the reply is received within that period, γ_j is removed from the list $\mathcal{B}$, and it is assumed that the corresponding peer is conforming to its commitment. Otherwise, the value of the corresponding *count* is decremented by 1 (lines 14–15). Note that p_k may try to send at most l messages to a particular peer associated with an

Algorithm 1. DistMonitor($\gamma_i, \mathcal{N}_i, q, l$)

```
   output: R                                          // list of reported peers
 1 R ← ∅; B ← q randomly chosen elements from N_i;
 2 foreach γ_j ∈ B do count[γ_j] ← l ;
 3 while B ≠ ∅ do
 4     foreach γ_j ∈ B do
 5         if count[γ_j] = 0 then R ← R ∪ g(γ_j) ; B ← B \ γ_j;
 6         else
 7             t' ← [t_now, ∞) ; t_ol ← T(γ_i, γ_j) ∩ t';
 8             if t_ol = 0 then B ← B \ γ_j;
 9             else
10                 t[γ_j] ← randomly selected value from t_ol;
11                 schedule a message for g(γ_j) at t[γ_j] ;
12                 schedule a thread waiting for γ_j from t[γ_j];
13     foreach γ_j ∈ B do
14         if a reply is received within T[γ_j] + t_max then B ← B \ γ_j;
15         else count[γ_j] ← count[γ_j] − 1;
16 report R to the tracker;
```

availability period. If the value of *count* is 0 for a particular availability period γ_j, then p_k adds the corresponding peer $g(\gamma_j)$ to the set $\mathcal{R}$, (line 5), where $\mathcal{R}$ denotes the set of peers that have not responded to the ping messages. Before γ_i ends, p_k sends $\mathcal{R}$ to the tracker as negative feedback.

3.1 Analysis of DistMonitor

The performance of the monitoring algorithm is measured in terms of two metrics, namely, percentage of peers that were monitored and the associated message overhead. Let the chunks of a particular file is stored among a group of peer availabilities. We denote γ_i^j as the i-th availability period in the peer path j (holding the j-th chunk of the file). Let $|\overline{\mathcal{S}}|$ and $|\overline{\mathcal{N}}|$ denote the average size of SAPL and PAPL of the involved availability periods. The following theorem characterizes q (the number of selected availability periods from the PAPL) and $|\overline{\mathcal{N}}|$ to ensure the desired performance of the DistMonitor algorithm.

Theorem 1. *For a stored file, each storage peer is monitored in its availability period by* DistMonitor *with high probability, for a proper choice of q and $|\overline{\mathcal{N}}|$, i.e., $q = |\overline{\mathcal{N}}| = \log |\overline{\mathcal{S}}|$.*

Proof. Let us assume that a file has n encoded chunks and $C = \{\gamma_{i_1}^{x_1}, \gamma_{i_2}^{x_2}, \ldots, \gamma_{i_m}^{x_m}\}$ is the group of availability periods for a particular chunk. Let $\gamma_i^j \in C$ be an availability period such that all other members in C are in the SAPL of γ_i^j (excluding itself). Without loss of generality, let us assume that the size of the SAPLs and PAPLs of all the availability periods in C are $|\overline{\mathcal{S}}|$ and

$|\overline{\mathcal{N}}|$, respectively. We aim at finding the probability of a peer p_k corresponding to γ_i^j being monitored, that is, the probability of receiving at least one message from any of the peers corresponding to the availability periods of C. Essentially, all the corresponding peers of C (except for p_k) contain p_k in their SAPL, so each of these peers has a probability to send a ping message to p_k.

Let us introduce the following notation first. Let D_k be the event that p_k receives at least one message from any of the peers, and G be the event that p_k receives a message from p_l. By recalling that a peer sends out ping messages to q randomly selected peers from its PAPL $\mathcal{N}$ (for a particular availability period), let us also define M as the event that p_k is in the PAPL of p_l, and Y as the event that p_k is sent a message by p_l. Hence, $P(G) = P(M)P(Y|M)$. Now,

$$P(M) = 1 - \left(1 - \frac{1}{|\overline{\mathcal{S}}|}\right)^{|\overline{\mathcal{N}}|}$$

and $P(Y|M) = q \cdot (|\overline{\mathcal{N}}|)^{-1}$. Therefore,

$$P(G) = \frac{q}{|\overline{\mathcal{N}}|}\left(1 - \left(1 - \frac{1}{|\overline{\mathcal{S}}|}\right)^{|\overline{\mathcal{N}}|}\right) = \frac{q}{|\overline{\mathcal{N}}|}\left(1 - e^{-\frac{|\overline{\mathcal{N}}|}{|\overline{\mathcal{S}}|}}\right)$$

So $P\left(\overline{D}_k\right) = (1 - P(G))^{|\overline{\mathcal{S}}|} = f\left(|\overline{\mathcal{N}}|, q\right)$, for a fixed value of $|\overline{\mathcal{S}}|$. Thus $P(D_k) = 1 - f\left(|\overline{\mathcal{N}}|, q\right)$. Thus, the probability of a peer being monitored depends on $f(\cdot)$ which, in turn, depends on q and $|\overline{\mathcal{N}}|$ for a specific file. For an instance, if we choose both q and $|\overline{\mathcal{N}}|$ as 1, a peer is monitored with a constant probability of around 63% by other storage peers. In the specific case where $\log|\overline{\mathcal{S}}|$ is chosen for both $|\overline{\mathcal{N}}|$ and q, $f(\cdot)$ becomes 0 with high probability asymptotically with increasing values of $|\overline{\mathcal{S}}|$. Therefore, a peer is guaranteed to be monitored with high probability in its availability period for storing a single file chunk when $q = |\overline{\mathcal{N}}| = \log|\overline{\mathcal{S}}|$.

Now, we consider the bandwidth requirements for the monitoring scheme.

Lemma 1. *In an availability period, a peer sends/receives a total of* $O(\xi \log|\overline{\mathcal{S}}|)$ *ping messages, where* ξ *represents the number of file chunks it holds.*

Lemma 1 follows from the following arguments. During an availability period, for each file, a storage peer sends out $O(q)$ ping messages. It can be shown that the expected number of ping messages received by a storage peer is also $O(q)$. In addition, a peer has to send (and receive) $O(q)$ reply messages. In our application, a storage peer can send/receive at most $l\log|\overline{\mathcal{S}}|$ ping messages for a single file. Therefore, in total, a peer can send/receive at most $l\xi\log|\overline{\mathcal{S}}|$ ping messages. Similarly, a peer can send/receive at most $l\xi\log|\overline{\mathcal{S}}|$ reply messages.

If the average size of the ping and reply messages is α, and the availability period is $\overline{A}$, a peer incurs an average download/upload bandwidth of $\frac{4 \cdot l\alpha\xi\log|\overline{\mathcal{S}}|}{\overline{A}}$. When $l = 2$, as in our application, a peer with an availability period of 20 hours contributing 100 GB storage space may incur an average upload/download

bandwidth of less than 0.75 KBps. This assumes the average size of ping/reply message as 100 bytes, and the average chunk size as 10 MB.

Replacement Strategy. The tracker maintains a negative feedback counter for each availability period. If it receives negative feedback for more than 10 times about an availability period in a particular week, it verifies whether the peer is unavailable by sending periodic ping messages for the next 4 weeks. Based on the response, the tracker computes the probability of the associated peer to be available using the bit vector method (recall from Sect. 2.5). If the probability is less than 0.2, the tracker picks a new availability period with similar or longer availability duration and similar or higher bandwidth offering the minimum cost.

4 Performance Evaluation

We simulated the PeerVault system based on the user availability traces of the SETI@home project [6]. SETI@home is a scientific experiment that uses the idle resources of the Internet-connected computers, in the Search for Extraterrestrial Intelligence (SETI).

4.1 Simulation Setup

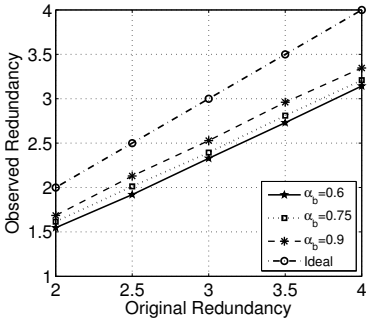
In the following, we will present the details about the traces, the parameters, and the methodology used in the performance evaluation.

SETI@Home Traces. In our experiments, we used the traces corresponding to the CPU availability of the SETI@home project as collected by the Failure Trace Archive [6]. We consider each unique host in the trace as a storage peer. The data reported in the trace spans over a period of a year and nine months that we call the *trace duration*. All the hosts of the trace data are not available for the entire trace duration. Some hosts start contributing after the trace duration starts, while some others leave permanently before the trace duration ends. Thus, each host (or storage peer) has trace data over an interval that we call *host duration*. If the host duration of a storage peer is (t_1, t_2), we define the *prediction interval* as $(t_1 + \sigma, t_2)$ if $t_1 + \sigma < t_2$, and 0 otherwise (in which case, we ignore that particular host). Recall from Sect. 2.5 that σ refers to the training period.

Simulation Details and Relevant Metrics. We carried out the experiments through a custom simulator written in Java to validate the availability and reliability of the backup service as well as the performance of the monitoring scheme, DistMonitor. In order to serve the storage requests, we implemented the minimum weight n-node disjoint path algorithm proposed in [9]. The availability periods were derived by using the method described in Sect. 2.5, with a training period σ of 4 weeks. We considered three different datasets of storage peers for the experiments. For each dataset, we picked a random sample of $10,000$ hosts, from which we extracted storage peers with training probability equal to or greater than $0.6, 0.75$, and 0.9. Throughout this section, they will be referred

Training probability threshold (α_b)	Hosts selected from sampled ones (%)	Average number of availability periods per host	Average length of availability periods (hours)
0.6	71	3.38	31.28
0.75	70	3.4	29.75
0.9	53	3.32	26.84

(a)



(b)

Fig. 4. (a) Availability periods obtained from traces. (b) Effective redundancy against the original redundancy of the files for storage peers with different training probability thresholds.

to as datasets A, B and C, respectively. Table 4a shows the percentage of hosts with the desired training probability and the number of availability periods per host which are considered in the simulation. We performed independent experiments for each of the datasets. In each experiment, 1,000 files are requested and the file sizes were generated from a lognormal distribution with a mean and standard deviation of 100 MB and 20 MB, respectively [11].

We considered the following performance metrics:

- *Observed redundancy*: the ratio of the number of available encoded chunks (n^*) to the minimum number of encoded chunks (k), for a given file.
- *Percentage of available files*: the ratio of the files with greater than or equal to k chunks available to the total number of files initially stored.

4.2 Experimental Results

Figure 4b shows the observed redundancy, averaged over all stored files, against the applied redundancy. In all datasets, the observed redundancy for a single service time frame (i.e., the first week) is summarized in a single plot to assess the availability of the offered service in a short time frame. The figure clearly shows that the observed redundancy increases with the threshold for increasing training probability of the hosts. Therefore, a higher threshold for training probability (e.g., higher than or equal to 90%) can be used to achieve a better performance. The line marked as ideal represents the case wherein all peers are available.

Figure 5 shows the availability of the stored files over a long time period to assess the reliability of the offered service. Specifically, it shows the percentage of accessible files (with $\eta = 2.5$) over a period of 52 weeks for datasets A and[1] C. The figures show that the availability of the files gradually decreases

[1] Results for dataset B are similar to those for dataset A, so we did not report them here due to lack of space.

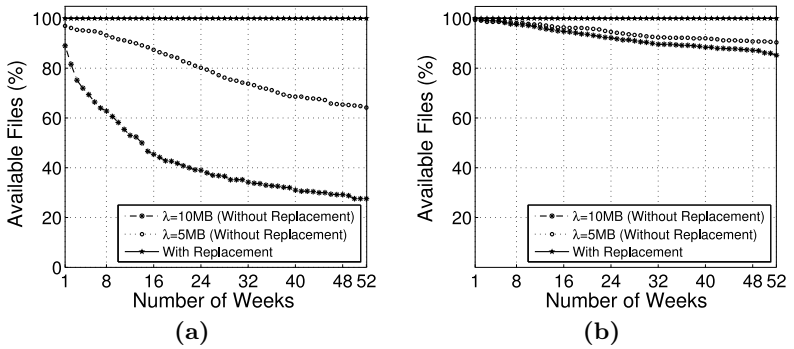


Fig. 5. Available files during 52 weeks for the different datasets: (a) A ($\alpha_b = 0.6$); and (b) C ($\alpha_b = 0.9$).

for all datasets. Even though dataset C shows a much higher availability over time than others (i.e., after 1 year, around 90% files are still accessible), there is no guarantee that all files can be accessed throughout the entire simulated period when no monitoring and replacement are used. This result, in addition to Fig. 4b, suggests that file availability is improved and retained over time when the training probability is high. However, some peers permanently leave the system over time and, thus, the data stored by them become unavailable. The monitoring algorithm and the replacement policy can guarantee that the files are available over the entire simulated period. The results also suggest that the availability of files can be improved by reducing the chunk size. Since the peer paths increase when the chunk size λ decreases, the probability of getting the minimum number of chunks for a file increases as well. On the other hand, very small file chunks result in a higher overhead for both the tracker and the storage peers. After considering all the above-mentioned aspects, 5 MB appears to be a suitable choice for the chunk size.

5 Related Work

In the following, we will summarize relevant literature on P2P systems used for both storage and for monitoring purposes.

The P2P networking paradigm has been exploited in the context of distributed file systems [12–14]. However, none of the proposed approaches exploits the availability pattern and the idle resources of the computer users. In some existing works, P2P networks were used to provide an enhanced online storage service in addition to dedicated servers. FS2You [15] and Amazing Store [16] are examples of such hybrid P2P systems explicitly designed to improve the availability of stored data with efficient bandwidth utilization. Among P2P backup systems, Symform [4] offers up to 200 GB of free storage space. In return, users are required to be online at least 80% of the time and provide at least 1.5 times the storage they receive from the system. Unfortunately, the details on the system design are not publicly available. Wuala [3] is another commercial P2P backup

system that relies on a symmetric service between users and exploits a hybrid architecture. A peer-assisted backup service was also proposed in [17], wherein it was shown that a performance comparable to traditional client-server architecture can be achieved by temporarily using storage space from cloud providers. However, all the above solutions still rely on the presence of special servers or data centers. In contrast, our solution is based on a pure P2P architecture. FriendStore [2] is a backup system where users store data by exploiting their social connection with other peers. Specifically, personal data are backed up on "friend" peers. Thus, availability and reliability depend on the number of friends, which can be rather low in realistic scenarios. In [18], a pricing mechanism for the offered resources in a P2P backup system is investigated. However, the work does not define any specific architecture as for the storage mechanism.

P2P networks employ monitoring schemes to ensure peer participation. A generic monitoring system based on the principles of autonomic computing was presented in [19]. Such a mechanism assumes that the P2P network is structured (i.e., has a logical overlay), thus, it is not directly applicable to our system. Existing P2P backup services use monitoring approaches which assign the monitoring responsibility to either a centralized server [4] or the peer that originated the backup request [3, 20]. On the other hand, our approach is distributed, since a peer is randomly monitored by some other peers, and assigns minimal responsibility to the tracker.

6 Conclusion

In this paper, motivated by the availability of unused disk space of the users and their long-term availability pattern, we introduced PeerVault, a data storage system based on a peer-to-peer architecture which can be used to provide a seamless backup service. PeerVault exploits group availability of participating peers to ensure long-term availability of the stored data. Moreover, to address peer churns, we proposed a distributed monitoring scheme that detects peers deviating from the desired availability pattern. Simulation results based on the traces of the SETI@home computing project demonstrated that the proposed approach efficiently utilizes the available resources and obtains a very high service reliability. In future, we propose to investigate revenue and recommendation models that will enhance the peer selection mechanism.

Acknowledgements. This research was partially supported by the NSF grants CNS-1049652, IIS-1064460, CNS-0916221, IIP-1242521 and CNS-1150192.

References

[1] Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. Trans. Storage 7(4), 14:1–14:20 (2012)
[2] Tran, D.N., Chiang, F., Li, J.: Friendstore: cooperative online backup using trusted nodes. In: Proc. of the 1st Workshop on Social Network Systems, pp. 37–42 (2008)

[3] LaCie AG: Wuala – Secure Online Storage, http://www.wuala.com (retrieved November 16, 2011)
[4] Symform, Inc.: symform – Revolutionary Cloud Storage Network, http://www.symform.com/our-solutions/storage-backup/ (retrieved July 22, 2012)
[5] Turner, D.A., Ross, K.W.: A lightweight currency paradigm for the p2p resource market. In: Proc. 7th ICEC (2004)
[6] Kondo, D., Javadi, B., Iosup, A., Epema, D.: The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp. 398–407 (May 2010)
[7] Fishburn, P.: Interval orders and interval graphs: a study of partially ordered sets. Wiley-Interscience series in discrete mathematics. Wiley (1985)
[8] Bhandari, R.: Survivable Networks: Algorithms for Diverse Routing. Kluwer Academic Publishers, Norwell (1998)
[9] Bhandari, R.: Optimal physical diversity algorithms and survivable networks. In: Proc. of Second IEEE Symposium on Computers and Communications (1997)
[10] Lázaro, D., Kondo, D., Marquès, J.M.: Long-term availability prediction for groups of volunteer resources. JPDC 72(2) (2012)
[11] Downey, A.B.: The structural cause of file size distributions. In: Proc. of the 2001 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems. SIGMETRICS 2001, pp. 328–329 (2001)
[12] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: an architecture for global-scale persistent storage. SIGPLAN 35 (November 2000)
[13] Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer, M., Wattenhofer, R.P.: Farsite: federated, available, and reliable storage for an incompletely trusted environment. In: Proc. of the 5th OSDI (2002)
[14] Hasan, R., Anwar, Z., Yurcik, W., Brumbaugh, L., Campbell, R.: A survey of peer-to-peer storage techniques for distributed file systems. In: Proc. of ITCC (2005)
[15] Sun, Y., Liu, F., Li, B., Li, B., Zhang, X.: Fs2you: Peer-assisted semi-persistent online storage at a large scale. In: INFOCOM 2009, pp. 873–881. IEEE (April 2009)
[16] Yang, Z., Zhao, B.Y., Xing, Y., Ding, S., Xiao, F., Dai, Y.: Amazingstore: available, low-cost online storage service using cloudlets. In: Proc. of the 9th International Conference on Peer-to-peer Systems, IPTPS 2010 (2010)
[17] Toka, L., Dell'Amico, M., Michiardi, P.: Online data backup: A peer-assisted approach. In: 2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P), pp. 1–10 (August 2010)
[18] Seuken, S., Charles, D., Chickering, M., Puri, S.: Market design & analysis for a p2p backup system. In: Proc. of the 11th ACM Conference on Electronic Commerce, EC 2010, pp. 97–108. ACM, New York (2010)
[19] Graffi, K., Stingl, D., Rueckert, J., Kovacevic, A., Steinmetz, R.: Monitoring and management of structured peer-to-peer systems. In: IEEE Ninth International Conference on Peer-to-Peer Computing, P2P 2009, pp. 311–320 (September 2009)
[20] Pamies-Juarez, L., Garcia-Lopez, P., Sanchez-Artigas, M.: Rewarding stability in peer-to-peer backup systems. In: 16th IEEE International Conference on Networks, ICON 2008, pp. 1–6 (December 2008)

Distributed Verification Using Mobile Agents*

Shantanu Das[1,2], Shay Kutten[2,**], and Zvi Lotker[1]

[1] Ben Gurion University of the Negev, Beer-Sheva, Israel
[2] Technion - Israel Institute of Technology, Haifa, Israel

Abstract. We study the problem of distributed verification in the mobile agent model. The problem of distributed verification in a network using local checking has been studied previously. In the local verification model, each node of the network must decide on a yes or no answer based on the knowledge of its immediate neighborhood and the global answer is obtained by a conjunction of the local answers. The efficiency of such a verification process is determined by the sizes of the proofs i.e. labels that must be assigned to the nodes to enable local verification of some global property. On the other hand, in the mobile agent model, verification is performed by an agent that is allowed to move from node to node of the graph, reading the labels of visited nodes in order to verify the required property. In this case, minimizing the memory of the agent is the primary objective. We study the space complexity of performing mobile verification in terms of memory of each agent as well as the number of agents required globally in networks of size n. In the case of a solitary agent, logarithmic memory is both necessary and sufficient for solving certain graph-based verification problems (even in the family of trees). For a team of at least two agents, the space complexity of most verification problems (including the well-studied MST verification) is reduced to $O(\log \log n)$, while a team of at least three agents even with constant size memory each, is sufficient to solve all graph-based verification problems. We also study the effect of randomization and show that one agent with $O(\log \log n)$ bits of memory and the ability to flip coins is as powerful as two deterministic agent having the similar memory limitations.

Keywords: Mobile Agents, Distributed Verification, Proof Labeling, Network Exploration, Finite State Automata.

1 Introduction

Consider a distributed network of processors connected by point to point message-passing channels, where each processor can talk to only its neighbors in the network. The topology is any arbitrary connected (undirected) graph. Suppose we want to verify some global property in the network, e.g. whether the routing

* This work was partially supported by the Israel Ministry of Science and Technology grant (#85387301): "Algorithmic approaches to energy savings".
** Additionally supported by Israel Science Foundation and the TASP center at the Technion.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 330–347, 2013.

subnet used by the network is a minimum spanning tree. Since each node has only local information about the network, verification of any global property would require exchange of information across the network.

The problem of distributed verification has been well investigated by the distributed computing community [1,6,8,9,12]. The problem is of fundamental importance in a decentralized network of autonomous entities where any global structure (e.g. routing tree) may become out-of-date due to changes in the network or failure of some components. So, the solution must be periodically verified in an efficient but robust manner. The capability of local verification of global properties of a network is important also in the context of self-stabilization. Individual components of a network may wake-up in an arbitrary state and must verify whether they are is a correct state with respect to the rest of the network.

The concept of local checking using proof labeling schemes was studied by Korman et al. [8,9]. In this model, each node is assigned a label (also called local proof) that depends on the problem to be verified. Every node is allowed to perform a local checking by reading labels of its neighboring nodes and output a yes or no answer. If the global solution to be verified is correct then all nodes should output "YES" and if the global solution is not correct, at least one node must output "NO". Such a distributed verification method works well in the context of self-stabilization; if the states of some nodes are incorrect this will be eventually detected by the distributed verification method, and a new solution can then be computed. However if one of the nodes is faulty or is under the control of an adversary, then such a distributed verification method may fail to detect the problem.

In this paper, we study distributed verification from a different point of view. We study the problem in the context of the mobile agents model. In this model, the nodes of the graph are merely data repository and the computations are performed by a mobile process (called an agent) that can move from node to node along the edges of the graph. In this context the distributed verification problem can be defined as follows. Each node of the network has a state (in our case, it's same as the label of a node). An agent starts from an arbitrary node of the graph and traverses the graph. The agent can read the state of any node that it visits and store it in its internal memory (but it cannot modify the state of a node i.e. it cannot write on a node). The objective of the agent is to verify whether the configuration of the graph—as defined by the states of the nodes—satisfies some predicate that corresponds to some global property on the graph. For example, we may wish to check whether the states of the nodes induce a rooted spanning tree on the graph. If the agent has enough memory to store the complete graph then the problem becomes trivial, assuming the agent can traverse the graph. Once the agent has traversed the graph and obtained state information from each node, it can decide locally using the complete information. However if the memory of the agent is restricted then it is not clear if the agent can perform verification even in those cases, where distributed verification is possible in the traditional model. This is the case for example when the size of the agent's memory is independent of the graph size, i.e. when the agent is a

finite state automaton. This paper investigates how much memory is needed by an agent, or a team of agents, to perform distributed verification of problems for which solutions exists in the message-passing network model with one round of communication. Further, we discuss how randomization helps in reducing the space complexity of mobile agent based verification.

The verification problem is known to be different for anonymous graphs and for graphs having unique identifiers. The focus of this paper is on the latter scenario. For an anonymous graph it is not always possible to traverse the graph (without knowledge of the size of the graph) unless the agent is allowed to leave marks on the nodes of the graph. On the other hand if an agent is allowed to write on the nodes of the graph it is possible to both verify and correct the labels of the nodes. In fact, an agent can perform the marking part of the local distributed verification process, assigning labels to the nodes, and then perform the verification using the labels. In case the verification process fails, the agent can recompute the solution and reassign appropriate labels. The verification problem for anonymous graphs using mobile agents has been studied only in the recent paper of Fraigniaud et al. [5]. However the focus of that paper is on computability rather than complexity. Since not all problems can be verified in anonymous networks, the paper determines which classes of problems can be verified in such networks (with or without additional information provided to the agents). To the best of our knowledge, there are no other known results on distributed verification in the mobile agent model.

2 The Model and the Problems

We consider an undirected connected graph $G = (V, E)$, with $|V| = n$ nodes and each node has a unique identifier of size $O(\log n)$. The edges incident to a node v are locally oriented with port-numbers from $0, 1, 2, \ldots, deg(v) - 1$. A node may also contain a label (not the same as the identifier) which depends on the problem to be verified. The label $L(v)$ of a node v consists of two parts:

1. Solution(v) denoted by $s(v)$: This is the local part of the solution that needs to be verified. If the solution is the graph G itself, then this field may be empty. Otherwise if the solution is a subgraph H of G then $s(v)$ is a set of some of the edges incident to the node v such that H is the graph defined by the union $\cup s(v)$, $v \in V(G)$. For example, in case of the spanning tree ([SPT]) problem, $s(v)$ is simply the parent pointer v, i.e. the edge connecting v to its parent in the spanning tree.
2. Proof(v) denoted $p(v)$: This is the local part of the proof that the solution is correct. The size of the proof depends on the particular model. For example, in the local verification model, the [SPT] problem can be verified using local proofs of size $O(\log n)$ where $p(v)$ consists of the identifier of the root of the spanning tree T and the distance from v to the root in T. In case of the verification by mobile agents, the size of local proofs depends on the capabilities of the agents as discussed later. In some cases, the proof field may be empty.

We consider the problem of verifying some global property satisfied by the graph and node labels. We investigate the relationship between the following two models for distributed verification—(i) Local verification model and (ii) Mobile Agent verification model.

2.1 Local Verification Model

This model is based on the proof-based verification in local communication model [9]. In this model, each node v in the network can see its own label, the labels of its neighboring nodes and the port numbers leading to any such node. Based on only this information the node has to output either YES or NO. If the global property that we wish to verify is true, then all nodes should output YES. Otherwise at least one node must output NO. There are no restrictions on the memory or computational power of a node. However, the objective is to minimize the size of the largest label assigned to any node in the graph, that still allows local verification. The verification process consists of two parts: (i) Encoding or the process of assigning proofs to the nodes, and (ii) Decoding or checking the proofs and producing the output of YES or NO at every node. We assume that the label assignment (proof assignment) is done by a central authority that has knowledge of the network and the global solution[1]. For every true instance of the problem, there is always a proof assignment that will enable each node to locally verify the solution (e.g. writing the complete map of the graph at every node). However, in most cases, it is possible to find much smaller proofs. In this model, the focus is on reducing the space complexity (memory required at each node for storing the proof). Note that the time taken for verification is always one time unit since the nodes decide on the output by just looking at the local neighborhood.

We define by $LVD(q)$ as the set of all problems that can be locally verified using labels of size at most q bits at each node of the graph.

2.2 Mobile Verification Model

In this model, computations are performed by mobile agents that can traverse the edges of the graph. Nodes of the graph are inactive except when an agent arrives to that node. The agent can be modeled as an automaton with state space S and the following behavior. When an agent is at a node v, the agent can read the label of node v (and the node identifier, if any) and the port number of the edge through which it arrived at node v (unless v is the starting node). The agent can perform local computations based on this information and the value of its own state. The agent then changes states and decides on a port number through which it exits the current node. Note that the agent residing on a node can use the (possibly unlimited) memory and computational power of that node. An agent at a node v is provided with a working memory at node v which can be

[1] Note that the proof assignment process can also be performed in a distributed manner without global knowledge but this is irrelevant for the purpose of this paper.

used for computation. However, any information written by an agent at a node, disappears as soon as the agent leaves that node.

Note that the agent is only responsible for the checking (or 'decoding') part of the verification process, i.e. given a proof-labeling for a particular problem, the job of the agent is to check if the proofs at each node correspond to a globally correct solution to the problem. The label assigned to a node v (and also the node identifier) is written in a read-only memory at node v and the agent is not allowed to change the label of a node. The only information that the agent can carry with it when moving between nodes, is the state information of the agent. The agent can be in k possible states (i.e. $k = |S|$) and the parameter k may or may not depend on the size n of the graph. An agent must output YES or NO within a finite time. If the global property that we wish to verify is true, then all agent should output YES; Otherwise all agents must output NO. An agent may not have any prior information about the graph, e.g. the topology, the size or the diameter. If two or more agents are at the same node, the agents can exchange information among themselves. The agents are distinctly identifiable and they can read from the read-only memory of the node (containing the node label) and can read/write on their own working memory at the current node where they are located.

We define by $MVD(q, k, t)$ the set of all problems that can be verified by k mobile agents, each having a memory of at most q bits, in time t. We define $MVD(q, k) = MVD(q, k, poly(n))$ as the set of problem verifiable in time polynomial in n, using k agents, each having at most q bits of memory.

2.3 Verification Problems

Given any connected undirected graph G, where each node v has a unique identifier ID(v) and also a label $L(v) = (s(v), p(v))$, we consider the following verification problems:

1. [ACYCLICITY] Verify if G is acyclic (i.e. contains no undirected cycles). In other words, G is a tree.
2. [ORIENTED] Given a tree G where for each node v, $s(v)$ is a pointer to the parent of v, verify if all the parent pointers are oriented towards a common root node.
3. [PATH] Verify if G is a simple path.
4. [ELECTED] Verify if there is exactly one node v such that $s(v) = 1$.
5. [AGREEMENT] Verify if all nodes have the same input value i.e. $s(v) = s(u)$, $\forall u, v \in V(G)$.
6. [SIZE] Verify if the number of nodes in G is exactly k, where $k = s(v)$, $\forall v \in V(G)$.
7. [SPT] Verify if the subgraph H induced by the edges in $\cup s(v)$, is a rooted spanning tree of G.
8. [MST] Verify if the subgraph H induced by the edges in $\cup s(v)$, is a minimum spanning tree of G (If the edges of the graph have unit weights, then this is equivalent to SPT problem).

9. [MIS] Verify if the set $\{v \in V(G) : s(v) = 1\}$ forms a maximal independent set of G.
10. [k-CONNECTED] Verify if G is k-edge-connected, i.e. no set of edges of cardinality at most k disconnects the graph G, where $k = s(v)$, $\forall v \in V(G)$.
11. [k-CLIQUE] Verify if G contains a clique of size k, as a subgraph, where $k = s(v)$, $\forall v \in V(G)$.
12. [SYMMETRICITY] Verify if the graph G (ignoring node-identities) is symmetric, i.e. there is an axis of symmetry passing through some of the edges of G, and cutting G into two disjoint, isomorphic components.

We distinguish *graph-based* problems as those problems that depend only on the structure of the graph G. For example, [SPT] is a graph-based problem since the solution depends only on the graph structure, but [MST] is not a graph-based problem since the solution depends also on the weights of the edges (which could potentially be arbitrarily large compared to the size of the graph).

3 Preliminaries

The problem of mobile verification requires the agents to traverse the graph and perform computations on the labels of the nodes. We present here some techniques for graph traversal which will be used in the rest of the paper. (Note that we consider traversal techniques for agents with limited memory so the agents cannot possibly remember the identifiers of all visited nodes during the traversal.) Given a tree T that is rooted at node r and where all other nodes contain a pointer to their parent in the tree, it is easy to traverse T using the depth-first traversal algorithm. Such a traversal begins at the root r and traverses all incident edges one after the other in order of the port numbers. On reaching any node v, the agent recursively performs a depth-first traversal of the subtree rooted at v. Note that the agent does not need to remember anything as the traversal is guided by the port-numbering and the parent pointers at each node. Whenever the agent enters a node v through the non-parent edge having the largest port-numbers, the agent knows that it has traversed the complete subtree rooted at v and must now return to the parent of v to continue the traversal.

A more general traversal technique for any arbitrary connected graph G, is based on the concept of a *Universal Exploration Sequence* (UXS) [10]. For any node $u \in G$, we define the ith successor of u, denoted by $succ(u, i)$ as the node v reached by taking port number i from node u (where $0 \leq i < deg(u)$). Let $(a_1, a_2, \ldots, a_k)$ be a sequence of integers. An *application* of this sequence to a graph G at node u is the sequence of nodes $(u_0, \ldots, u_{k+1})$ obtained as follows: $u_0 = u$, $u_1 = succ(u_0, 0)$; for any $1 \leq i \leq k$, $u_{i+1} = succ(u_i, (p + a_i) \mod deg(u_i))$, where p is the port-number at u_i corresponding to the edge $\{u_{i-1}, u_i\}$. A sequence $(a_1, a_2, \ldots, a_k)$ whose application to a graph G at any node u contains all nodes of this graph is called a UXS for this graph. For any positive integers n, d, $d < n$, there exists a sequence denoted UXS(n,d) whose

application to any graph G having at most n nodes and maximum degree at most d, visits all nodes of the graph G at least once. Moreover, such a sequence can be computed online by an agent having $O(\log n)$ memory (due to a recent result [11]). At each step i of the traversal, the agent needs to remember only the index i to compute the next element of the sequence. If the agent wants to backtrack j steps, the agent can compute the reverse sequence for the j steps. In the following, we will denote by UXS(n) = UXS(n,n), the sequence that can be used to traverse all graphs of size at most n.

We now consider the problem of comparison of labels. We define an artificial problem called EQUALITY$[m]$ for a graph G_2 on two nodes connected by a single edge. The two nodes u and v contains two m-bit numbers a and b, i.e. $L(u) = a$ and $L(v) = b$. The agent must output YES if $a = b$ and NO otherwise. This problem is a special case of the [AGREEMENT] problem defined above. If an agent cannot verify the EQUALITY problem then it cannot compare the labels of two adjacent nodes of the graph and thus most of the verification problems are impossible to solve in that case.

Lemma 1. *An agent with $o(\log m)$ memory cannot verify EQUALITY[m].*

Proof. Consider the minimum value k such that an agent with k states can verify EQUALITY$[m]$. Such an agent starts in the initial state s_0 and terminates in state s_y or s_n (corresponding to YES or NO answer respectively). These three states are necessarily distinct. Without loss of generality, we can assume the algorithm A of the agent, to be as follows: In state s_0 the agent at node u reads a, changes states to some state s_x, moves to node v, reads b, changes states and continues moving back and forth between u and v, alternately reading a and b, until the first time the agent enters state s_y or s_n and then it terminates. Note any other algorithm can be modified to the above algorithm without using any more states than algorithm A.

We denote by $S_i \subset S$ the set of possible states to which the agent may transform after the ith step of algorithm (i.e. the agent transforms to some state $s_x \in S_1$ after reading a for the first time, depending on the value of a). Note that the S_i are mutually disjoint and non-empty, for each $i \leq t$ where t is the maximum number of time steps taken by the agent to decide on some input of size n. Further s_0 does not belong to any S_i (since otherwise there will be a cycle and the algorithm will never terminate). Now let us consider the number of distinct paths from s_0 to s_y in the state transition diagram for algorithm A. This must correspond to the number of YES instances for EQUALITY$[m]$, which is equal to 2^m in our case (i.e. all pairs (a, b) where $a = b$). The number of distinct paths from s_0 to s_y is given by the expression $t + |S_1| * |S_2| * \ldots |S_t|$ where the number of states of the agent is $3 + |S_1| + |S_2| + \cdots + |S_t| = k$. Equating this expression $t + |S_1| * |S_2| * \ldots |S_t|$ to 2^m and minimizing for k, gives us $(3m/2 + 3) \leq k \leq (3m + 3)$. Thus, an agent with at most m states, (i.e. $\log n$ memory) cannot verify EQUALITY$[m]$.

The above result is related to the recognition of palindromes by a turing machine with memory constraints. It was shown [7] that even a probabilistic turing

machine with $o(\log n)$ memory cannot recognize the language consisting of all palindromes of size at most n bits. The result of Lemma 1 can be obtained as a corollary of that result (the original proof is non-trivial and thus is not reproduced here). The stronger result of [7] implies that even randomization does not help in overcoming the lower bound above (unless we make additional assumptions). We return to this issue in Section 5. We now present the following positive results.

Lemma 2. *An agent having* $1 + \log(m)$ *memory can verify EQUALITY[m] in* $O(m)$ *time.*

The algorithm that achieves the above result is simple. The agent compares the two bit strings bit-by-bit, by moving back and forth between u and v. The agent needs to maintain a counter of $\log m$ bits that counts from 0 to $m-1$. At each step, the counter gives the index i of the bit to be compared and the additional bit in memory is used to store the value of the i-th bit from a when the agent moves to the other node to compare it with the i-th bit of b.

The above approach can be used not just for verifying equality, but also for other boolean operations that can be performed bit-wise on the two numbers. For example, the agent can verify if $a > b$, $a < b$, $a = b + 1$, or $a = f(b)$ for any computable function f. We shall call such operations as *comparison-based operations* on two numbers a and b.

Corollary 1. *An agent having* $O(\log m)$ *memory can compare two labels of size at most* m *bits in* $O(m)$ *time.*

If there are multiple agents which start at the same node and have distinct identifiers, then we can overcome the logarithmic bound on memory.

Lemma 3. *For any* $m > 0$*, three (initially colocated) agents having* $O(1)$ *memory each, can verify EQUALITY[m] in* $O(m)$ *time.*

Proof. The three agents can solve the verification problem using the following algorithm. Agent 1 remains at the node u and agent 2 moves moves to the other node v. Agent 3 travels back and forth between the nodes for m steps, comparing bitwise the numbers a and b. Agent 1 (respectively agent-2) maintains a counter in its working memory at node u (resp. node v), containing the index of the bit to be compared. Each time agent-3 arrives at node u, agent-1 communicates the next bit of a to agent 3, which remembers the bit and moves to node v. At node v agent-2 communicates the corresponding bit of b to agent 1. If there is a mismatch, the agent 1 outputs NO and informs the other agents. Otherwise the process continues until there are no more bits to read and the agents output YES. Note that the traveling agent (agent-3) only needs to remember the value of the current bit and the outcome of the last comparison. This requires a constant amount of memory.

The algorithm above can be modified to compute any boolean function on the numbers a and b, instead of the equality function. In this case, the agent 3 could

copy the number b bit-by-bit into the working memory of agent 1 at node u and then agent 1 can perform the computation at node u. A similar approach can be used for a team of two agents having $(1 + \log m)$ bits of memory each. The second agent would copy the number b bit-by-bit from node v to node u, using a counter of $\log m$ bits containing the index of the bit to be copied in the next step, while the first agent would remain stationary at node u and receive the information from the second agent. Thus a team of three $O(1)$ memory agents, or two $O(\log m)$ memory agents can compute any computable function on two m-bit size labels a, b stored at nodes u and v respectively.

4 Space Complexity of Mobile Verification

4.1 Mobile Agents with Logarithmic Memory

We first show that there exists some problems for which a mobile agent needs at least logarithmic memory for verification, even in the family of trees. On the positive side we show that logarithmic memory is sufficient for verifying any graph-based property using a single agent in any arbitrary graph (even if the size of the graph is unknown to the agent).

Theorem 1. *An agent needs $\Omega(\log n)$ memory to verify the [SYMMETRICITY] problem in the family of all trees of at most n nodes.*

Proof. A tree is symmetric only if it has a central edge, say (u, v), and the subtrees rooted at the two end-points u and v are isomorphic (where u maps to v). Even if the map of the complete subtree is provided as label at the nodes u and v, the agent needs to compare these maps (each containing at least $\Omega(n)$ bits of information). The problem is equivalent to the communication complexity problem of checking the equality of two $\Omega(n)$ bit strings given to the two end-points of an edge. An agent with less than n states cannot accomplish this task, as shown in Lemma 1.

Lemma 4. *An agent having $O(\log n)$ memory can explore any graph of size at most n and stop, even without any prior knowledge of n.*

Proof. The agent has enough memory to remember the ID of the starting location (which we call the source node). The idea of the algorithm is that the agent guesses a value N as an upper bound for n and perform a traversal by applying the universal exploration sequence UXS(N). During this traversal whenever it reaches a vertex v, the agent performs a check operation and if this operation returns false, then the agent aborts the current traversal, doubles the value of N and starts the whole procedure again using the new value of N and the current vertex as source. The checking operation at a vertex v is as follows. For each neighbor w of v, the agent stores the ID of w in its memory, returns to the source (by backtracking) and performs another traversal using UXS(N) to check if the traversal visits w (i.e. during this traversal, the agent compares the ID of every visited vertex with the stored value of ID(w)). If the agent does not encounter

node w during the traversal then the checking operation returns false (i.e. the value of N is not correct in this case). Otherwise, as soon as the agent encounters the node w, it aborts the current traversal, returns to node v by backtracking and the checking procedure returns true.

If the agent completes the traversal without having to increase N anymore then this traversal has visited all nodes of G. So the agent can stop.

Theorem 2. *For any constant $c > 0$, if there is a local verification algorithm for a problem that uses labels of size at most $m < n^c$, then there is mobile verification algorithm for the same problem for a single agent having $O(\log n)$ memory. In other words,*

$$LVD(n^c) \subset MVD(O(\log n), 1)$$

Proof. We have seen that an agent with logarithmic memory can traverse the graph visiting each vertex at least once. Such an agent can simulate the local verification algorithm for the message passing model. The simulation works as follows: The agent performs a traversal of the graph and at each vertex v, the agent imitates the local proof-checking procedure, by traveling to each neighbor w of v, reading the label of node w and returning to v to check for consistency with the label at node v. If the label of v is $l'(v)$ in the local verification algorithm, then the label used in our algorithm is $L(v) = l'(v)l'(w_1)l'(w_2)\ldots l'(w_d)$, where $w_1, w_2, \ldots, w_d$ are the neighbors of v.

By assumption, the labels are of size at most m, thus the consistency check for label of a neighbor w can be done by the agent making $O(m)$ trips between v and w, on each occasion remembering a constant number of bits from label(w). Since the agent has enough memory to count up to m, it possible to perform this operation (keeping track of which bits have been read). Once the agent has checked the label of each neighbor w, the agent can execute the local checking algorithm using the label $L(v)$ of vertex v. If the local checking algorithm outputs NO at any vertex v, then the agent outputs NO. Otherwise, when the agent complete the traversal of G, the agent output YES. By the correctness of the local verification algorithm, the agent has correctly performed mobile verification for the same problem.

Note that any graph-based property can be verified locally if a complete map of the graph is available at each node. Such a map can be encoded with labels of $O(n^2)$ bits. Thus all graph-based problems can be verified by agents having logarithmic memory.

In this section, we considered the space complexity of mobile agent based verification and did not discuss the time complexity. The algorithm in Theorem 2 that simulates local checking runs in polynomial time, but is time consuming as the length of known log-space constructible universal sequences are quite large. However it is possible to have more time-efficient algorithms for the traversal and thus the simulation algorithm. One possibility is to use a spanning tree T of G, for the traversal. In this case, the labels at each node v must include a pointer to the parent of v in T and the local proof at each node must include a proof

that T is a spanning tree of G. The agent would first perform [SPT] verification and then use the tree T for visiting each node v of G and simulating the local checking at v as before.

4.2 Agents with $O(\log(\log n))$ Memory

Although we showed in the previous section that verification of certain problems require logarithmic memory, there are problems that can be verified using less memory. In this section we are interested in those problems that can be verified using sub-logarithmic memory or more precisely, using only $O(\log \log n)$ bits of memory per agent. It is known that traversal of an unlabeled graph requires logarithmic memory in the worst case. On the other hand, if the graph is appropriately labelled then even a constant memory agent can traverse the graph, e.g. using a labelled spanning tree of the graph. Thus, solving [SPT] verification is important for solving other verification problems. We first show how an agent can verify whether a given subgraph is a tree.

Theorem 3. *A single mobile agent having $O(\log \log n)$ bits of memory can verify [ACYCLICITY] in any graph of n nodes.*

Proof. We provide a labeling and a verification algorithm for a $O(\log \log n)$ memory agent. Notice that any YES instance of the problem is a tree. Given a tree T, we choose an arbitrary root and orient all edges towards the root. The root is distinctly labeled as the node with no parent and the label of every other node contains a pointer to its parent node, the identifier of the root and distance to the root. Thus, the labels are of size $O(\log n)$. The verification algorithm works as follows. The agent, starting at any node follows the parent pointers to reach the root. At each step, while going from v to parent(v), the agent checks if the distance field decreases by exactly one and is greater than zero. This checking can be performed due to lemma 2. On reaching the root, the agent performs a depth-first traversal of T (using the parent pointers for backtracking). During the traversal the agents checks the following. Whenever going up the tree, from node u to v, the agent checks that the distance-to-root of v is one less than that of u and the root identifiers are same. Similarly, while going down the tree from node u to v, the agent checks that the distance-to-root of v is one more than that of u and the root identifiers are same. Whenever the agent reaches a node with the distance-to-root field equal to zero, the agent checks if the identifier of the node is same as the root identifier.

The above property implies that agents with $O(\log \log n)$ memory can verify whether a subgraph of G is a tree or not. However, such an agent cannot verify if the tree spans the entire graph.

Theorem 4. *A single agent having $O(\log \log n)$ bits of memory cannot verify [SPT] in all graphs of size n.*

We consider a team of two agents of $O(\log \log n)$ memory each and show that [SPT] verification is possible for such a team of 2 agents.

Theorem 5. *Two agents having $O(\log \log n)$ bits of memory each, can verify [SPT] in all graphs of size n.*

Proof. As shown in Theorem 3, the agents can verify if a subgraph T is a tree. To verify whether T is a SPT, it suffices to traverse T and at each node check each incident edge e to see if the other end of the edge also belongs to the tree T. Thus, the two agents can traverse the tree T together and at each non-tree edge $e = (u, v)$, the first agent would wait at node u and the second agent would traverse e to reach v and then follow the parent pointers from v to reach the root. If node v was indeed included in tree then the agent would have reached the root of T. Now, the agent would perform a depth-first traversal of T, starting from the root, until meeting the first agent again. If node v is not included in tree T, then the agent would not be able to reach the root of T by following the parent pointers from v, if any. Thus, in this case, the agent would detect a problem and thus output 'NO'. Otherwise, after the two agents meet, they would continue the traversal and checking process until they have visited all nodes of tree and check all non-tree egdes. In this case, the agents output YES.

When there are at least two agents, the team of two agents can verify [SPT] in any arbitrary graph G as we have seen before. Thus after verification of [SPT], the agents can use the spanning tree to visit every node of the graph and at each such node, perform local verification if the node labels are of size at most $O(\log^c n)$.

Theorem 6. *In arbitrary graphs of size n, a team of two agents with memory size of $O(\log \log n)$ each, can perform mobile verification of any problem P that admits local verification using labels of size $O(\log^c n)$. Thus,*

$$LVD(\log^c n) \subset MVD(O(\log \log n), 2)$$

The above result implies that a team of two agents can verify the [MST] problem provided that the maximum edge weight W is polynomial in n [2]. Such a team of two agents can also verify the problems of [ELECTED], [SIZE], [MIS], [k-CLIQUE] and [AGREEMENT] (For the latter, it is required that the input at each node is not larger than n^c).

4.3 Mobile Agents with Finite Memory

We now consider mobile agents whose memory is bounded by a finite constant independent of the size of the graph and determine which problems can be verified by such agents. For a single agent of $O(1)$ size memory, it is difficult to perform verification of most problems in the arbitrary graphs.

Theorem 7. *For any $k > 0$, a single mobile agent with k bits of memory cannot verify [ACYCLICITY] in graphs of arbitrary size n, even for the family of bounded degree graphs.*

[2] It is known [8] that in the local verification model, [MST] can be verified with proof labels of size $O(\log n \cdot \log W)$.

Proof. Due to Lemma 1 we know that the agent cannot verify EQUALITY[m] for $m = \log n$. Thus the agent cannot compare the identifiers of the nodes of the graph. Suppose for the sake of contradiction that there is an algorithm A for verification of [ACYCLICITY] for an agent having k-bit memory. Consider the execution of the algorithm in a ring of size n, where the agent is supposed to output 'NO' within a finite time. If the agent outputs 'NO' without visiting all edges of the graph, then we can delete one of the unvisited edges and obtain a line graph which is YES instance of the problem and thus, the algorithm fails. Hence assume that the agent visits all the edges and thus it visits one node v at least twice. If the agent visits node v twice in the same state then the algorithm would never terminate. Thus, if A is a correct algorithm, the agent can visit any node at most k times before it outputs 'NO'. Now, consider a line of size $4nk$ and place the agent in the middle of this line. If the agent executes the same algorithm, an adversary can assign labels and identifiers to the nodes of this line in a such a way that the agent would output 'NO' before reaching any of the end-points of the line. Thus, the algorithm fails to verify [ACYCLICITY].

The above result holds for the family of graphs of maximum degree two. Thus, even if an agent has memory size proportional to the maximum degree of the graph, it does not help for the verification of global properties in arbitrary graphs.

Corollary 2. *An agent with memory size of $O(\Delta)$ is not sufficient for mobile verification of [ACYCLICITY] in graphs of maximum degree Δ.*

We know that a single agent having O(1) bits of memory cannot traverse an arbitrary graph. However such an agent can traverse any tree. Moreover the agent can verify the [ORIENTED] problem in any tree, using the techniques we have seen before. Thus, a finite memory agent can verify any problem in trees that can be verified with O(1) bit labels in the local verification model.

Theorem 8. *In the family of all trees, a single agent with memory size of $O(1)$ bits can perform mobile verification of any problem P that admits local verification using labels of size $O(1)$.*

We now focus on teams of multiple agents (each having finite memory) starting from a common node of G, and show that even a team of two agents can verify a large set of problems in arbitrary graphs.

Lemma 5. *A team of two mobile agents having memory size of $O(1)$ each, can perform verification of [ACYCLICITY] in arbitrary graphs.*

Proof. A YES instance of the problem is a tree. We can orient the tree with respect to some root node r and use a labeling where each node has a pointer to its parent in the rooted tree. The two agents can traverse G, starting from any node u, using the following algorithm. Agent-1 stays at u and agent-2 goes down the subtree rooted at u using the reverse of the parent pointers. If at any point the agent reaches a node containing multiple parent pointers, it returns the answer NO. Otherwise the agent would either succeed in visiting the complete

subtree rooted at u or it would discover a cycle (a cycle of reverse parent pointers must lead to u if there are no nodes with multiple parents). In the latter case, the agent answers NO. Otherwise both agents move up to the parent v of u and continue the same procedure. If the agents do not answer NO until they eventually reach the root node r and traverse the subtree under the root (i.e. the complete graph G), then the graph must be a tree. Thus the agents can correctly identify all YES instances.

Lemma 6. *A team of two mobile agents having memory size of $O(1)$ each, can verify [SPT] in any arbitrary graph G.*

Proof. Due to the previous result we know that two mobile agents can verify whether a given subgraph of G is a tree. For verification of the [SPT] problem the agents can first verify whether the solution T is a tree. If so, the agents can traverse T and for each non-tree edge e incident to a node $u \in T$, determine whether the other end-point v of the edge also belongs to T. This can be done as follows. When the agents reach node u having a non-tree edge $e = (u, v)$, one of the agents travels to the other end-point v. The other agent goes up from u to the root of T using the parent pointers. This agent now traverses the tree to find the unique node v that still contains the first agent. If the agents succeed then $v \in T$ and otherwise T is not an SPT of G.

Since it is possible to verify [SPT] for two finite memory agents, it is also possible for them to traverse any G (using the spanning tree that they verified) and visit every node of G. Thus the team of two agents can verify all problems for which labels of constant size are sufficient in the local verification model. These include the problems of [ELECTED], [MIS], [PATH], [k-CLIQUE].

Theorem 9. *A team of two agents having memory size of $O(1)$ bits each can perform mobile verification of any problem P in arbitrary graphs that admits local verification using labels of size $O(1)$. Thus,*

$$LVD(O(1)) \subset MVD(O(1), 2)$$

Note that there are problems which require labels of size larger than O(1) in the local verification model but can be verified by a team of two finite memory agents. One such example is the [SPT] problem which requires labels of size $\Omega(\log n)$ in the local verification model.

We now consider teams of three agents and show that they can verify a larger set of problems. We already know the three agents can verify EQUALITY$[n]$ for any $n > 0$ and thus they can compare any two labels of size at most n in time polynomial in n. This leads us to the following results:

Theorem 10. *A team of three mobile agents having memory size of $O(1)$ can perform verification of any graph-based property in arbitrary graphs. Thus, $LVD(n^c) \subset MVD(O(1), 3)$.*

5 Randomized Algorithms for Mobile Verification

In this section, we assume the nodes of the network provide the agents with some mechanism to generate random bits. An agent located at any node v, can call a subroutine that returns a random bit to the agent. A mobile agent that has this additional capability will be called a probabilistic mobile agent. A probabilistic mobile agent can use a random walk to traverse the graph. During a random walk, an agent at any node v chooses one of the incident edges at v uniformly at random, and traverses that edge. It is well known that a random walk visits all the vertices of the graph of size n in $O(n^3)$ time steps, with high probability [3].

We denote by $MVR(q,k)$ the set of all problems that can be verified with very high probability, by a team of k probabilistic mobile agents, each having a memory of at most q bits, in polynomial time.

Theorem 11. *A probabilistic mobile agent having $O(\log\log n)$ bits of memory can verify, with high probability, any property in graphs of size n that requires at most $\log^c n$ size labels for local verification. Thus,*

$$LVD(\log^c n) \subset MVR(O(\log\log n), 1)$$

Proof. We first show that the agent can traverse the graph and visit every node with high probability, using the method of approximate counting (e.g. see [4]). Each agent carries a counter C (initialized to zero) of size $\log\log n$. The agent follows a random walk and at each node the agent makes a random coin flip, C times; if all the C coin flips return 1, then and only then the agent increments the counter C. When the counter value is $3\log n$, the agent stops. With very high probability the agent would performed a random walk for n^3 steps and thus would have visited all nodes of the graph. During the traversal, the agent performs local checking at each node v, comparing the label of v with its neighbors (assuming that the labels are of size $\log^c n$ bits, this checking can be performed by an agent having $O(\log\log n)$ memory).

The above result shows that one probabilistic agent having $O(\log\log n)$ memory can verify the same set of problems as a team of two deterministic agents having same memory restrictions (see Theorem 6). We now consider a team of two probabilistic agents and show that they can verify problems that two deterministic agents could not.

Lemma 7. *A team of two probabilistic mobile agents having $O(\log\log n)$ bits of memory each, can verify [k-CONNECTED] for k=2, in any arbitrary graph G of size n.*

Proof. Consider a spanning tree T of G. We know that a team of two agents can verify [SPT]. If the node labels of G include proofs for [SPT] then the agents can first verify [SPT] and then use the spanning tree T for traversal. To check for 2-connectivity, it is sufficient to verify that each tree edge belongs to a cycle in the

graph. For each edge $e = (u, v)$ of the tree T, the agents perform the following check. One agent stays at the parent node u and the other agent moves to the child node v and then tries to reach u without using the edge (u, v) (i.e. the agent performs a random walk in $G \setminus e$). If the agent does not succeed in reaching u then G is not 2-connected, and the agent outputs NO. Otherwise, the two agents would meet at u and then continue with the traversal of T, checking each tree edge and finally output YES, if all tree edges satisfy the property.

Lower Bound: As mentioned before, based on the results of [7], we have the lower bound of $\Omega(\log(n))$ for randomized verification of EQUALITY$[n]$ and thus the same lower bound holds for the verification of some graph-based problems (e.g. [SYMMETRICITY]) in graphs of size n.

Theorem 12. *Any probabilistic agent requires at least $\Omega(\log n)$ bits of memory to verify certain graph-based problems in arbitrary graphs of size n.*

This lower bound can be overcome by making an additional assumption. We assume there is a global clock and the agents can access the clock at any stage of the algorithm. This allows an agent having a much smaller memory to verify the EQUALITY$[n]$ problem.

Theorem 13. *In the presence of a global clock, a probabilistic mobile agent having only $O(\log^* n)$ bits of memory can verify any graph-based property in graphs of size n with high probability.*

Proof. As mentioned before, the agent can traverse the graph using a random walk and it only needs to know when to stop traversing the graph. This can be done by using a counter of $O(\log^* n)$ bits that counts up to n^3 with very high probability. To imitate the local verification at each node the agent needs to compare labels of adjacent nodes. Assuming that the labels are of size at most $m \leq n^2$, this requires solving the EQUALITY$[m]$ problem. The agent maintains another counter that counts up to n^2 with very high probability and this counter is used for termination of the process of comparing two labels bit-by-bit. At each step of the comparison, the agent obtains the index of the bit that is being compared, as $(C \mod 2m)/2$ where C is the value of the global clock at that time.

6 Time Complexity of Mobile Verification

Until now we have only considered the space complexity of mobile verification in terms of the size and number of mobile agents needed to perform verification. All algorithms discussed so far run in polynomial time with respect to the size of the graph. We now discuss the time efficiency of the mobile verification algorithms. A trivial lower bound on the time complexity of mobile verification (irrespective of the size of agents) is the following.

Theorem 14. *The verification of any global property by a mobile agent in graphs of size n requires $\Omega(n)$ time. The verification of [SPT] requires $\Omega(m)$ time in graphs with m edges.*

The algorithm for mobile verification of [SPT] using two or three agents takes $O(mn)$ time in graphs with n nodes and m edges. All the other verification algorithms are based on traversal using a spanning tree and thus require the agents to perform [SPT] verification as the first step. The rest of the algorithm requires $O(m + n)$ time and thus the time complexity of the whole algorithm is $O(mn)$. Mobile verification in the family of trees can be performed in $O(n)$ time.

7 Verification in Anonymous Networks

We assumed, in this paper, that the nodes of the network have unique identifiers. One can ask what properties can be verified in an anonymous network (i.e. when the nodes of the network are not labelled with unique identifiers)? First let us suppose that the agents are allowed to leave marks on the nodes. A single agent that can write on the nodes is capable of performing mobile verification of any computation even in an anonymous graph. The agent requires only $O(1)$ bits of memory. Note that there exists labeling schemes for labeling the vertices of the graph such that a finite state agent can traverse the graph [2]. Further such a labeling can be computed online by the agent. A finite state agent traversing the graph, can use the memory of the nodes of the graph to perform verification of any Turing-computable computation.

When the agents are not allowed to leave marks (labels) on the nodes of the graph, one agent is not sufficient for mobile verification in anonymous graphs (it is known that such an agent cannot even traverse the graph). Two co-located agents (with sufficient memory each, e.g. polynomial in the size of the graph) can perform mobile verification. In this case it is possible to traverse the graph (using one agent as pebble or marker) and compute the size of the graph. If the two agents are not initially co-located then randomization can help. Two agents starting at distinct nodes of the graph can use random walk to gather at some node with high probability. The gathered agents can then perform verification without using randomization.

8 Conclusions

We studied the problem of distributed verification using mobile agents that traverse the network. We showed that a few agents with small memory (or, constant memory) are capable of performing verification of global problems in arbitrary graphs. We compared the set of problems verifiable by such agents to those that can verified in the local verification model under constraints on the memory available at each node and the main results are summarized in the table below.

Table 1. Set of problems verifiable by teams of mobile agents

#Agents	Agent Memory	Randomized	Verifiable Problems
1	$O(\log n)$	NO	$LVD(n^c)$
1	$O(\log \log n)$	YES	$LVD(\log^c n)$
2	$O(\log \log n)$	NO	$LVD(\log^c n)$
2	$O(1)$	NO	$LVD(O(1))$
3	$O(1)$	NO	$LVD(n^c)$

References

1. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction. In: Proc. of the 32nd Annual Symposium on Foundations of Computer Science (FOCS), pp. 268–277 (1991)
2. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-guided graph exploration by a finite automaton. ACM Transactions on Algorithms 4(4), 42:1–42:18 (2008)
3. Feige, U.: A Tight Upper Bound on the Cover Time for Random Walks on Graphs. Random Struct. Algorithms 6(1), 51–54 (1995)
4. Flajolet, P.: Approximate Counting: A Detailed Analysis. BIT 25(1), 113–134 (1985)
5. Fraigniaud, P., Pelc, A.: Decidability Classes for Mobile Agents Computing. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 362–374. Springer, Heidelberg (2012)
6. Fraigniaud, P., Korman, A., Peleg, D.: Local Distributed Decision. In: Proc. of 52nd Annual Symposium on Foundations of Computer Science (FOCS), pp. 708–717 (2011)
7. Freivalds, R., Karpinski, M.: Lower Space Bounds for Randomized Computation. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 580–592. Springer, Heidelberg (1994)
8. Korman, A., Kutten, S.: Distributed verification of minimum spanning trees. Distributed Computing (DC) 20(4), 253–266 (2007)
9. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Distributed Computing (DC) 22(4), 215–233 (2010)
10. Koucký, M.: Universal traversal sequences with backtracking. Journal of Comput. Syst. Sci. 65, 717–726 (2002)
11. Reingold, O.: Undirected connectivity in log-space. Journal of the ACM 55, 1–24 (2008)
12. Das Sarma, A., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., Wattenhofer, R.: Distributed verification and hardness of distributed approximation. In: Proc. of the 43rd ACM Symposium on Theory of Computing (STOC), pp. 363–372 (2011)

Sublinear Bounds for Randomized Leader Election

Shay Kutten[1,⋆], Gopal Pandurangan[2,⋆⋆], David Peleg[3,⋆⋆⋆], Peter Robinson[4,†], and Amitabh Trehan[1,⋆]

[1] Information Systems Group, Faculty of Industrial Engineering and Management, Technion - Israel Institute of Technology, Haifa-32000, Israel
kutten@ie.technion.ac.il, amitabh.trehaan@gmail.com

[2] Division of Mathematical Sciences, Nanyang Technological University, Singapore 637371 and Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, USA
gopalpandurangan@gmail.com

[3] Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot-76100 Israel
david.peleg@weizmann.ac.il

[4] Division of Mathematical Sciences, Nanyang Technological University, Singapore 637371
peter.robinson@ntu.edu.sg

Abstract. This paper concerns *randomized* leader election in synchronous distributed networks. A distributed leader election algorithm is presented for complete n-node networks that runs in $O(1)$ rounds and (with high probability) takes only $O(\sqrt{n}\log^{3/2} n)$ messages to elect a unique leader (with high probability). This algorithm is then extended to solve leader election on any connected non-bipartite n-node graph G in $O(\tau(G))$ time and $O(\tau(G)\sqrt{n}\log^{3/2} n)$ messages, where $\tau(G)$ is the mixing time of a random walk on G. The above result implies highly efficient (sublinear running time and messages) leader election algorithms for networks with small mixing times, such as expanders and hypercubes. In contrast, previous leader election algorithms had at least linear message complexity even in complete graphs. Moreover, super-linear message lower bounds are known for time-efficient *deterministic* leader election algorithms. Finally, an almost-tight lower bound is presented for randomized leader election, showing that $\Omega(\sqrt{n})$ messages are needed for any $O(1)$ time leader election algorithm which succeeds with high probability. It is also

⋆ Supported by the Israeli Science Foundation and by the Technion TASP center.

⋆⋆ Research supported in part by the following grants: Nanyang Technological University grant M58110000, Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 2 grant MOE2010-T2-2-082, and a grant from the US-Israel Binational Science Foundation (BSF).

⋆⋆⋆ Supported in part by the Israel Science Foundation (grant 894/09), the United States-Israel Binational Science Foundation (grant 2008348), and the Israel Ministry of Science and Technology (infrastructures grant).

† Research supported in part by the following grants: Nanyang Technological University grant M58110000, Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 2 grant MOE2010-T2-2-082.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 348–362, 2013.

shown that $\Omega(n^{1/3})$ messages are needed by any leader election algorithm that succeeds with high probability, regardless of the number of the rounds. We view our results as a step towards understanding the randomized complexity of leader election in distributed networks.

1 Introduction

Background and motivation. Leader election is a classical and fundamental problem in distributed computing. It originated as the problem of regenerating the "token" in a local area *token ring* network [19] and has since then "starred" in major roles in problems across the spectrum, providing solutions for reliability by replication (or duplicate elimination), for locking, synchronization, load balancing, maintaining group memberships and establishing communication primitives. As an example, the content delivery network giant Akamai uses decentralized and distributed leader election as a subroutine to tolerate machine failure and build fault tolerance in its systems [23]. In many cases, especially with the advent of large scale networks such as peer-to-peer systems [27,28,32], it is desirable to achieve low cost and scalable leader election, even though the guarantees may be probabilistic.

Informally, the problem of distributed leader election requires a group of processors in a distributed network to elect a unique leader among themselves, i.e., exactly one processor must output the decision that it is the leader, say, by changing a special *status* component of its state to the value *leader* [20]. All the rest of the nodes must change their status component to the value *non-leader*. These nodes need not be aware of the identity of the leader. This *implicit* variant of leader election is rather standard (cf. [20]), and is sufficient in many applications, e.g., for token generation in a token ring environment. This paper focuses on implicit leader election.

In the *explicit* variant, all the non-leaders change their status component to the value *non-leader*, and moreover, every node must also know the identity of the unique leader. This formulation may be necessary in problems where nodes coordinate and communicate through a leader, e.g., implementations of Paxos [7,18]. In this variant, there is an obvious lower bound of $\Omega(n)$ messages (throughout, n denotes the number of nodes in the network) since every node must be informed of the leader's identity. This explicit leader election can be achieved by simply executing an (implicit) leader election algorithm and then broadcasting the leader's identity using an additional $O(n)$ messages and $O(D)$ time (where D is the diameter of the graph).

The complexity of the leader election problem and algorithms for it, especially deterministic algorithms (guaranteed to always succeed), have been well-studied. Various algorithms and lower bounds are known in different models with synchronous/asynchronous communication and in networks of varying topologies such as a cycle, a complete graph, or some arbitrary topology (e.g., see [12,20,24,29,31] and the references therein). The problem was first studied in context of a ring network by Le Lann [19] and discussed for general graphs in the influential paper of Gallager, Humblet, and Spira [8]. However, the class of

complete networks has come to occupy a special position of its own and has been extensively studied [1,10,13,15,16].

The study of leader election algorithms is usually concerned with both message and time complexity. For complete graphs, Korach, Moran and Zaks [14] and Humblet [10] presented $O(n \log n)$ message algorithms. Korach, Kutten, and Moran [13] developed a general method decoupling the issue of the graph family from the design of the leader election algorithm, allowing the development of message efficient leader election algorithms for any class of graphs, given an efficient traversal algorithm for that class. When this method was applied to complete graphs, it yielded an improved (but still $\Omega(n \log n)$) message complexity. Afek and Gafni [1] presented asynchronous and synchronous algorithms, as well as a tradeoff between the message and the time complexity of synchronous *deterministic* algorithms for complete graphs in the non-simultaneous wake-up model: the results varied from a $O(1)$-time, $O(n^2)$-messages algorithm to a $O(\log n)$-time, $O(n \log n)$-messages algorithm. Singh [30] showed another trade-off that saved on time, still for algorithms with a super-linear number of messages. (Sublinear time algorithms were shown in [30] even for $O(n \log n)$ messages algorithms, and even lower times for algorithms with higher messages complexities). Afek and Gafni, as well as Korach, Moran, and Zaks [14,16] showed a lower bound of $\Omega(n \log n)$ messages for *deterministic* algorithms in the general case. Multiple studies showed a different case where it was possible to reduce the number of messages to $O(n)$ by using a *sense of direction*–essentially, assuming some kind of a virtual ring, superimposed on the complete graph, such that the order of nodes on a ring is known to the nodes [6]. The above results demonstrate that the number of messages needed for deterministic leader election is at least linear if nodes wake up simultaneously, or even super-linear (i.e., $\Omega(n \log n)$) if nodes are woken up by the adversary. In this paper, we focus on simultaneous wake-up. Nevertheless, in the full paper we show that our algorithms also yield sublinear message complexity even in the case where the adversary can wake up nodes at arbitrary times, which is a significant improvement over the $\Omega(n \log n)$ bound required for deterministic algorithms.

At its core, leader election is a symmetry breaking problem. For anonymous networks under some reasonable assumptions, deterministic leader election was shown to be impossible [3] (using symmetry concerns). Randomization comes to the rescue in this case; random rank assignment is often used to assign unique identifiers, as done herein. Randomization also allows us to beat the lower bounds for deterministic algorithms, albeit at the risk of a small chance of error.

Randomized asynchronous (explicit) leader election algorithms for various networks were presented by Itai and Rodeh, Scheiber and Snir, and Afek and Matias [11,5,2]. In particular, one of the algorithms elects a leader in a complete graph with $O(n)$ messages and $O(\log n)$ time [2]. The probability of error there tends to zero when n grows to infinity but is not given explicitly. A randomized leader election algorithm (for the explicit version) that could err with probability $O(\frac{1}{(\log n)^{\Omega(1)}})$ was presented recently in [26][1], with time $O(\log n)$ and linear

[1] In contrast, the probability of error in the current paper is $O(\frac{1}{n^{\Omega(1)}})$.

message complexity. That paper also surveys some related papers about randomized algorithms in other models that use more messages for performing leader election [9] or related tasks (e.g., probabilistic quorum systems, Malkhi et al [21]). In the context of self-stabilization, a randomized algorithm with $O(n \log n)$ messages and $O(\log n)$ time until stabilization was presented in [33].

1.1 Our Main Results

The main focus of this paper is to study how randomization can help in improving the complexity of leader election, especially message complexity in synchronous networks. We first present a (implicit) leader election algorithm for a complete network that runs in $O(1)$ time and uses only $O(\sqrt{n}\log^{3/2} n)$ messages to elect a unique leader (with high probability[2]). This is a significant improvement over the linear number of messages required by any deterministic algorithm (in the simultaneous wake-up model). In the full paper we show that our algorithm also works in the non-simultaneous wake-up model, which is an even larger gap to the $\Omega(n \log n)$ message complexity bound required by any deterministic algorithm. For the explicit variant of the problem, our algorithm can be extended to use (w.h.p.) $O(n)$ messages and $O(1)$ time.

We then extend this algorithm to solve leader election on any connected (non-bipartite)[3] n-node graph G in $O(\tau(G))$ time and $O(\tau(G)\sqrt{n}\log^{3/2} n)$ messages, where $\tau(G)$ is the mixing time of a random walk on G. The above result implies highly efficient (sublinear running time and messages) leader election algorithms for networks with small mixing time. In particular, for important graph classes such as expanders (used, e.g., in modeling peer-to-peer networks [4]), which have logarithmic mixing time, it implies an $O(\log n)$ time and $O(\sqrt{n}\log^{5/2} n)$ messages algorithm, and for hypercubes, which have a mixing time of $O(\log n \log\log n)$, it implies a sublinear $O(\log n \log\log n)$ time and $O(\sqrt{n}\log^{5/2} n \log\log n)$ messages algorithm.

For our algorithms, we assume that the communication is synchronous and follows the standard $\mathcal{CONGEST}$ model [25], where in each round a node can send at most one message of size $O(\log n)$ bits on a single edge. For our algorithm on general graphs, we also assume that the nodes have an estimate of the mixing time. We do not however assume that the nodes have unique IDs, hence the algorithms in this paper work also for anonymous networks. We assume that all nodes wake up simultaneously at the beginning of the execution. (Additional details on our distributed computing model are given later.)

Finally we show that, in general, it is not possible to improve over our algorithm substantially, by presenting an almost-tight lower bound for randomized leader election. We show that $\Omega(\sqrt{n})$ messages are needed for any $O(1)$ time leader election algorithm in a complete network which succeeds with high probability. It is also shown that $\Omega(n^{1/3})$ messages are needed by any leader election algorithm that succeeds with high probability, regardless of the number of the

[2] Throughout, "with high probability (whp)" means with probability $\geqslant 1 - 1/n^{\Omega(1)}$.
[3] Our algorithm can be modified to work for bipartite graphs as well (cf. Section 3).

rounds. These lower bounds hold even in the $\mathcal{LOCAL}$ model [25], where there is no restriction on the number of bits that can be sent on each edge in each round. To the best of our knowledge, these are the first non-trivial lower bounds for randomized leader election.

1.2 Technical Contributions

The main algorithmic tool used by our randomized algorithm involves reducing the message complexity via random sampling. For general graphs, this sampling is implemented by performing random walks. Informally speaking, a small number of nodes (about $O(\log n)$), which are the candidates for leadership, initiate random walks. We show that if a sufficient number of random walks are initiated (about $\sqrt{n}\log n$), then there is a good probability that random walks originating from different candidates meet (or collide) at some node which acts as a referee. The referee notifies a winner among the colliding random walks. The algorithms use a birthday paradox type argument to show that a unique candidate node wins all competitions (i.e. is elected) with high probability. An interesting feature of that birthday paradox argument (for general graphs) is that it is applied to a setting with non-uniform selection probabilities. See Section 2 for a simple version of the algorithm that works on a complete graph. The algorithm of Section 3 is a generalization of the simple algorithm of Section 2 that works for any connected graph; however the algorithm and analysis are more involved.

The main intuition in our lower bound proof for randomized leader election is that we show that any algorithm which sends less messages than required by our lower bound has a good chance of generating runs where there are multiple potential leader candidates in the network that do not influence each other. In other words, the probability of such "disjoint" parts of the network to elect a leader is the same, which implies that there is a good probability that more than one leader is elected. Although this is conceptually easy to state, it is technically challenging to show formally since our result applies to all randomized algorithms without further restrictions.

1.3 Distributed Computing Model

The model we consider is similar to the models of [1,10,13,15,16], with the main addition of giving processors access to a private unbiased coin. Also, we do not assume unique identities. We consider a system of n nodes, represented as an undirected (not necessarily complete) graph $G = (V, E)$. Each node runs an instance of a distributed algorithm that has knowledge of n. The computation advances in synchronous rounds, where, in every round, nodes can send messages, receive messages that were sent in the same round by neighbors in G, and perform some local computation; every node has access to the outcome of unbiased private coin flips. The messages are the only means of communication; in particular, nodes cannot access the coin flips of other nodes, and do not share any memory. Throughout this paper, we assume that all nodes are awake initially

and simultaneously start executing the algorithm. We discuss some relaxations of this point in the full paper.

Leader Election. We now formally define the leader election problem. Every node u has a special variable $\texttt{status}_u$ that it can set to a value in the set $\{\perp, \text{NON-ELECTED}, \text{ELECTED}\}$; initially we assume $\texttt{status}_u = \perp$. An *algorithm A solves leader election in T rounds* if, from round T on, exactly one node has its status set to ELECTED while all nodes are in state NON-ELECTED. This is the requirement for standard (implicit) leader election.

2 Randomized Leader Election in Complete Networks

To provide the intuition for our general result, let us first illustrate a simpler version of our leader election algorithm, adapted to complete networks. More specifically, this section presents an algorithm that, with high probability, solves leader election in complete networks in $O(1)$ rounds and sends no more than $O(\sqrt{n}\log^{3/2} n)$ messages. Let us first briefly describe the main ideas of Algorithm 1 (see pseudo-code below). Initially, the algorithm attempts to reduce the number of leader candidates as far as possible, while still guaranteeing that there is at least one candidate (with high probability). Non-candidate nodes enter the NON-ELECTED state immediately, and thereafter only reply to messages initiated by other nodes. Every node u becomes a candidate with probability $\frac{2\log n}{n}$ and selects a random rank r_u chosen from some large domain. Each candidate node then randomly selects $2\lceil\sqrt{n\log n}\rceil$ other nodes as *referees* and informs all referees of its rank. The referees compute the maximum (say r_w) of all received ranks, and send a "winner" notification to the node w. If a candidate wins all competitions, i.e., receives "winner" notifications from all of its referees, it enters the ELECTED state and becomes the leader.

Algorithm 1. Randomized Leader Election in Complete Graphs

Round 1 :

1: Every node u decides to become a candidate with probability $\frac{2\log n}{n}$ and generates a random rank r_u from $\{1,\ldots,n^4\}$.
If a node does not become a candidate, it immediately enters the NON-ELECTED state; otherwise it executes.

2: **Choosing Referees:** Node u samples $2\lceil\sqrt{n\log n}\rceil$ neighbors (the *referees*) and sends a message $\langle u, r_u\rangle$ to each referee.

Round 2 :

3: **Winner Notification:** A referee v considers all received messages and sends a winner notification to the node w that satisfies
$r_w \geqslant r_u$ for every message $\langle u, r_u\rangle$.

4: **Decision:** If a node receives winner notifications from all its referees, then it enters the ELECTED state,
otherwise it sets its state to NON-ELECTED.

Theorem 1. *Consider a complete network of n nodes and assume the $\mathcal{CONGEST}$ model of communication. Algorithm 1 solves leader election with high probability, terminates in $O(1)$ rounds, and uses $O(\sqrt{n}\log^{3/2} n)$ messages with high probability.*

Proof. Since all nodes enter either the `elected` or `non-elected` state after two rounds at the latest, we get the runtime bound of $O(1)$.

We now argue the message complexity bound. On expectation, there are $2\log n$ candidate nodes. By using a standard Chernoff bound (cf. Theorem 4.4 in [22]), there are at most $7\log n$ candidate nodes with probability at least $1 - n^{-2}$. In step 3 of the algorithm, each referee only sends messages to the candidate nodes by which it has been contacted. Since there are $O(\log n)$ candidates and each approaches $\Theta(\sqrt{n\log n})$ referees, the total number of messages sent is bounded by $O(\sqrt{n}\log^{3/2} n)$ with high probability.

Finally, we show that Algorithm 1 solves leader election with high probability. With probability $\left(1 - \frac{2\log n}{n}\right)^n \approx \exp(-2\log n) = n^{-2}$, no node becomes candidate. Hence the probability that at least one node is elected as leader is at least $1 - n^{-2}$. Let ℓ be the node that generates the highest random rank r_ℓ among all candidate nodes; with high probability, ℓ is unique. Clearly, node ℓ enters the ELECTED state, since it receives "winner" notifications from all its referees.

Now consider some other candidate node v. This candidate chooses its referees randomly among all nodes. Therefore, the probability that an individual referee selected by v is among the referees chosen by ℓ, is $\frac{2\lceil\sqrt{n\log n}\rceil}{n}$. It follows that the probability that ℓ and v do not choose any common referee node is at most

$$\left(1 - 2\sqrt{\frac{\log n}{n}}\right)^{2\sqrt{n\log n}} \leqslant \exp\left(-4\log n\right) = n^{-4},$$

which means that with high probability, some node x serves as common referee to ℓ and v. By assumption, we have $r_v < r_\ell$, which means that node v does not receive $2\lceil\sqrt{n\log n}\rceil$ "winner" notifications, and thus it subsequently enters the NON-ELECTED state. By taking a union bound over all other candidate nodes, it follows that with probability at least $1 - \frac{1}{n}$, no other node except ℓ wins all of its competitions, and therefore, node ℓ is the only node to become a leader. □

3 Randomized Leader Election in General Graphs

In this section, we present our main algorithm, which elects a unique leader (w.h.p.), and terminates in $O(\tau(G,n))$ rounds while using $O(\tau(G,n)\sqrt{n}\log^{3/2} n)$ messages (w.h.p.), where $\tau(G,n)$ is the mixing time of a random walk on G. Initially, a node u only knows the mixing time (or a constant factor estimate of) $\tau(G,n)$ (defined below in (1)); in particular u does not have any a priori knowledge about the actual topology of G.

The algorithm presented here requires nodes to perform random walks on the network by token forwarding in order to choose sufficiently many referee

nodes at random. Thus essentially random walks perform the role of sampling as done in Algorithm 1 and is conceptually similar. Whereas in the complete graph randomly chosen nodes act as referees, here any intermediate node (in the random walk) that sees tokens from two competing candidates can act as a referee and notify the winner. One slight complication we have to deal with in the general setting is that in the $\mathcal{CONGEST}$ model it is impossible to perform too many walks in parallel along an edge. We solve this issue by sending only the *count* of tokens that need to be sent by a particular candidate, and not the tokens themselves.

While using random walks can be viewed as a generalization of the sampling performed in Algorithm 1, showing that two candidate nodes intersect in at least one referee leads to an interesting balls-into-bins scenario where balls (i.e., random walks) have a *non-uniform* probability to be placed in some bin (i.e., reach a referee node). This non-uniformity of the random walk distribution stems from the fact that G might not be a regular graph. We show that the non-uniform case does not worsen the probability of two candidates reaching a common referee, and hence an analysis similar to the one given for complete graphs goes through.

We now introduce some basic notation for random walks. Suppose that $V = \{u_1, \ldots, u_n\}$ and let d_i denote the degree of node i. The $n \times n$ *transition matrix* $\mathbf{A}$ of G has entries $a_{i,j} = \frac{1}{d_i}$ if there is an edge $(i, j) \in E$, otherwise $a_{i,j} = 0$. Entry $a_{i,j}$ gives the probability that a random walk moves from node u_i to node u_j. The position of a random walk after k steps is represented by a probability distribution π_k determined by $\mathbf{A}$. If some node u_i starts a random walk, the initial distribution π_0 of the walk is an n-dimensional vector having all zeros except at index i where it is 1. Once node u has chosen a random neighbor to forward the token, the distribution of the walk after 1 step is given by $\pi_1 = \mathbf{A}\pi_0$ and in general we have $\pi_k = \mathbf{A}^k\pi_0$. If G is non-bipartite and connected, then the distribution of the walk will eventually converge to the *stationary distribution* $\pi_* = (b_1, \ldots, b_n)$, which has entries $b_i = \frac{d_i}{2|E|}$ and satisfies $\pi_* = \mathbf{A}\pi_*$.

We define the *mixing time* $\tau(G, n)$ of a graph G with n nodes as the minimum k such that, for all starting distributions π_0,

$$\|\mathbf{A}^k\pi_0 - \pi_*\|_\infty \leqslant \frac{1}{2n}, \tag{1}$$

where $\|\cdot\|_\infty$ denotes the usual maximum norm on a vector. Clearly, if G is a complete network, then $\tau(G, n) = 1$. For expander graphs it is well known that $\tau(G, n) \in O(\log n)$. Note that mixing time is well-defined only for non-bipartite graphs; however, by using a lazy random walk strategy (i.e., with probability 1/2 stay at the current node; otherwise proceed as usual) our algorithm will work for bipartite graphs as well.

Theorem 2. *Consider a non-bipartite network G of n nodes with mixing time $\tau(G, n)$, and assume the $\mathcal{CONGEST}$ model of communication. Algorithm 2 solves leader election with high probability, terminates within $O(\tau(G, n))$ rounds, and uses $O(\tau(G, n)\sqrt{n}\log^{3/2} n)$ messages with high probability.*

Algorithm 2. Randomized Leader Election

1: **VAR** `origin` $\leftarrow 0$; `winner-so-far` $\leftarrow \perp$
2: Initially, node u decides to become a candidate with probability $\frac{2\log n}{n}$ and generates a random rank r_u from $\{1, \ldots, n^4\}$.

Initiating Random Walks:

3: Node u creates $2\lceil\sqrt{n \log n}\rceil$ tokens of type $\langle r_u, k\rangle$.
4: Node u starts $2\lceil\sqrt{n \log n}\rceil$ random walks (called *competitions*), each of which is represented by the random walk token $\langle r_u, k\rangle$ (of $O(\log n)$ bits) where r_u represents u's random rank. The counter k is the number (initially 1) of walks that are represented by this token (explained in Line 8).

Disqualifying hopeless candidates (note that any node can be a referee and notify winner/loser):

5: A node v discards every received token $\langle r_u, k\rangle$ if v has received (possibly in the same round) a token r_w with $r_w > r_u$.
6: **if** a received token $\langle r_w, k'\rangle$ is not discarded and `winner-so-far` $\neq r_w$ **then**
7: Node v remembers the port of an arbitrarily chosen neighbor that sent one of the (possibly merged) tokens containing r_w in its variable `origin` and sets its variable `winner-so-far` to r_w.

Token Forwarding:

8: Let $\mu = \langle r_u, k\rangle$ be a token received by v and suppose that μ is not discarded in Line 5. For simplicity, we consider all distinct tokens that arrive in the current round containing the same value r_u at v to be merged into a single token $\langle r_u, k\rangle$ before processing where k holds the accumulated count. Node v randomly samples k times from its neighbors. If a neighbor x was chosen $k_x \leqslant k$ times, v sends a token $\langle r_u, k_x\rangle$ to x.

Notifying a Winner in round $\tau(G, n)$:

9: **if** `winner-so-far` $\neq \perp$ **then**
10: Suppose that node v has not discarded some token generated by w. According to Line 5, w has generated the largest rank among all tokens seen by v.
11: Node v generates a winner notification $\langle \text{WIN}, r_w, cnt\rangle$ for r_w and sends it to the neighbor stored in `origin` (cf. Line 7). The field cnt is set to 1 by v and contains the number of winner notifications represented by this token.
12: If a node u receives (possibly) multiple winner notifications for r_w, it simply forwards a token $\langle \text{WIN}, r_w, cnt'\rangle$ to the neighbor stored in `origin` where cnt' is the accumulated count of all received tokens.

Decision:

13: If a node wins all competitions, i.e., receives $2\lceil\sqrt{n \log n}\rceil$ winner notifications it enters the ELECTED state, otherwise it sets its state to NON-ELECTED.

Proof. We first argue the message complexity bound. On expectation, there are $\Theta(\log n)$ candidate nodes. By using a standard Chernoff bound (cf. Theorem 4.4 in [22]), there are at most $7 \log n$ candidate nodes with probability at least $1 - n^{-2}$. Every candidate node u contacts $\Theta(\sqrt{n \log n})$ referee nodes and initiates a random walk of length $\tau(G, n)$, for each of the $\Theta(\sqrt{n \log n})$ referees. By the description of the algorithm, each referee node only sends messages to the candidate nodes by

which it has been contacted. Since we have $O(\log n)$ candidates, the total number of messages sent is bounded by $O(\tau(G,n)\sqrt{n}\log^{3/2} n)$ with high probability.

The running time bound depends on the time that it takes to complete the $2\lceil\sqrt{n\log n}\rceil$ random walks in parallel and the notification of the winner. By Line 5, it follows that a node only forwards at most one token to any neighbor in a round, thus there is no delay due to congestion. Moreover, for notifying the winner, nodes forward the winner notification for winner w to the neighbor stored in `origin`. According to Line 7, a node sets `origin` to a neighbor from which it has received the first token originated from w. Thus there can be no loops when forwarding the winner notifications, which reach the winner w in at most $\tau(G,n)$ rounds.

We now argue that Algorithm 2 solves leader election with high probability. Similarly to Algorithm 1, it follows that there will be at least one leader with high probability. Let ℓ be the candidate that generated the (unique) highest random rank among all candidates and consider some other candidate node v, i.e., we have that $r_v < r_\ell$ by assumption. By the description of the algorithm, node v chooses its referees by performing $\rho = 2\lceil\sqrt{n\log n}\rceil$ random walks of length $\tau(G,n)$. We cannot argue the same way as in the proof of Algorithm 1, since in general, the stationary distribution of G might not be the uniform distribution vector $(\frac{1}{n},\ldots,\frac{1}{n})$. Let p_i be the i-th entry of the stationary distribution. Let X_i be the indicator random variable that is 1 if there is a collision (of random walks) at referee node i. We have $\mathbb{P}[X_i = 1] = (1-(1-p_i)^\rho)^2$. We want to show that the probability of error (i.e., having no collisions) is small; in other words, we want to upper bound $\mathbb{P}[\bigcap_{i=1}^{n}(X_i = 0)]$. The following lemma (proved in the full paper) shows that it is sufficient to obtain a bound for the case when the stationary distribution is uniform:

Lemma 1. *Consider ρ balls that are placed into n bins according to some probability distribution π and let p_i be the i-th entry of π. Let X_i be the indicator random variable that is* 1 *if there is a collision (of random walks) at referee node i. Then $\mathbb{P}[\bigcap_{i=1}^{n}(X_i = 0)]$ is maximized for the uniform distribution.*

By (1), the probability of such a walk hitting any of the referees chosen by ℓ, is at least $\frac{2\sqrt{n\log n}}{2n}$. It follows that the probability that ℓ and v do not choose a common referee node is at most

$$\left(1-\sqrt{\frac{\log n}{n}}\right)^{2\sqrt{n\log n}} \leqslant \exp\left(-2\log n\right). \tag{2}$$

Therefore, the event that node v does not receive sufficiently many winner notifications, happens with probability $\geqslant 1-n^{-2}$, which requires v to enter the NON-ELECTED state. By taking a union bound over all other candidate nodes, it follows that with high probability no other node except ℓ will win all of its competitions, and therefore, node ℓ is the only node to become a leader with probability at least $1-\frac{1}{n}$. □

4 Lower Bound

In this section we prove a lower bound on the number of messages required by any algorithm that solves leader election with probability at least $1 - 1/n$.

Our model assumes that all processors execute the exact same algorithm and have access to an unbiased private coin. So far we have assumed that nodes are *not* equipped with unique ids. Nevertheless, our lower bound still holds even if the nodes start with unique ids.

Our lower bound applies to all algorithms that send only $o(\sqrt{n})$ messages with probability at least $1 - 1/n$. In other words, the result still holds for algorithms that have small but nonzero probability for producing runs where the number of messages sent is much larger (i.e., $\Omega(\sqrt{n})$). We show the result for the $\mathcal{LOCAL}$ model, which implies the same for the $\mathcal{CONGEST}$ model.

Theorem 3. *Consider any algorithm A that uses $f(n)$ messages (of arbitrary size) with high probability on a complete network of n nodes. If A solves leader election in $O(1)$ rounds with high probability, then $f(n) \in \Omega(\sqrt{n})$. Moreover, $f(n) \in \Omega(n^{1/3})$ for any algorithm A using any number of rounds that solves leader election with high probability. This holds even if nodes are equipped with unique identifiers (chosen by the adversary).*

Proof. We first show the result for the case where nodes are anonymous, i.e., are *not* equipped with unique identifiers, and later on extend the impossibility to the non-anonymous case by an easy reduction.

Assume that there is some algorithm A that solves leader election with high probability but sends only $f(n)$ messages. The remainder of the proof involves showing that this yields a contradiction. Consider a complete network where for every node, the adversary chooses the connections of its ports as a random permutation on $\{1, \ldots, n-1\}$.

For a given run α of an algorithm, define the *communication graph* $\mathcal{C}^r(\alpha)$ to be a directed graph on the given set of n nodes where there is an edge from u to v if and only if u sends a message to v in some round $r' \leqslant r$ of the run α. For any node u, denote the *state of u in round r of the run α* by $\sigma_r(u, \alpha)$. Let Σ be the set of all node states possible in algorithm A. (When α is known, we may simply write $\mathcal{C}^r$ and $\sigma_r(u)$.) With each node $u \in \mathcal{C}^r$, associate its state $\sigma_r(u)$ in $\mathcal{C}^r$, the communication graph of round r. We say that node u *influences node w by round r* if there is a directed path from u to w in $\mathcal{C}^r$. (Our notion of influence is more general than the causality based "happens-before" relation of [17], since a directed path from u to w is necessary but not sufficient for w to be causally influenced by u.) A node u is an *initiator* if it is not influenced before sending its first message. Note that a mute node that never receives any messages is also an initiator. For every initiator u, we define the *influence cloud* $\mathcal{IC}_u^r$ as the pair $\mathcal{IC}_u^r = (C_u^r, S_u^r)$, where $C_u^r = \langle u, w_1, \ldots, w_k \rangle$ is the ordered set of all nodes that are influenced by u, namely, that are reachable along a directed path in $\mathcal{C}^r$ from u, ordered by the time by which they joined the cloud, and $S_u^r = \langle \sigma_r(u, \alpha), \sigma_r(w_1, \alpha), \ldots, \sigma_r(w_k, \alpha) \rangle$ is their configuration after round r, namely, their current tuple of states. (In what follows, we sometimes abuse

notation by referring to the ordered node set C_u^r as the influence cloud of u.) Note that a *passive* (non-initiator) node v does not send any messages before receiving the first message from some other node.

Since we are only interested in algorithms that send a finite number of messages, in every execution α there is some round $\rho = \rho(\alpha)$ by which no more messages are sent.

In general, it is possible that in a given execution, two influence clouds $C_{u_1}^r$ and $C_{u_2}^r$ intersect each other over some common node v, if v happens to be influenced by both u_1 and u_2. The following lemma shows that the low message complexity of algorithm A yields a good probability for all influence clouds to be disjoint from each other.

Hereafter, we fix a run α of algorithm A. Let $\hat{N}_i$ be the event that there is no intersection between (the node sets of) the influence clouds existing at the end of round i, i.e., $C_u^i \cap C_{u'}^i = \emptyset$ for every two initiators u, u'. Let $N_r = \bigwedge_{i=1}^r \hat{N}_i$. Let $N = N_\rho$ be the corresponding event at the end of the run α. Let M be the event that algorithm A sends no more than $f(n)$ messages in the run α.

Lemma 2. *Assume that* $\mathbb{P}[M] \geqslant 1 - \frac{1}{n}$. *Then either of the following two conditions is sufficient to ensure that* $\mathbb{P}[N \wedge M] \geqslant 1 - o(1)$*:*
(a) $f(n) \in o(\sqrt{n})$ *and* A *terminates in* $O(1)$ *rounds, or*
(b) $f(n) \in o(n^{1/3})$ *and* A *terminates (in an arbitrary number of rounds).*

We defer the proof of Lemma 2 to the full paper. The main idea is that, due to the random choice of the port numberings and the assumption that at most $f(n)$ messages are sent, the probability that 2 influence clouds intersect is $o(1)$. Note that $f(n)$ also limits the size of any influence cloud.

We next consider *potential cloud configurations*, namely, $Z = \langle \sigma_0, \sigma_1, \ldots, \sigma_k \rangle$, where $\sigma_i \in \Sigma$ for every i, and more generally, *potential cloud configuration sequences* $\bar{Z}^r = (Z^1, \ldots, Z^r)$, where each Z^i is a potential cloud configuration, which may potentially occur as the configuration tuple of some influence clouds in round i of some execution of Algorithm A (in particular, the lengths of the cloud configurations Z^i are monotonely non-decreasing). We study the occurrence probability of potential cloud configuration sequences.

We say that the potential cloud configuration $Z = \langle \sigma_0, \sigma_1, \ldots, \sigma_k \rangle$ *is realized* by the initiator u in round r of execution α if the influence cloud $\mathcal{IC}_u^r = (C_u^r, S_u^r)$ has the same node states in S_u^r as those of Z, or more formally, $S_u^r = \langle \sigma_r(u, \alpha), \sigma_r(w_1, \alpha), \ldots, \sigma_r(w_k, \alpha) \rangle$, such that $\sigma_r(u, \alpha) = \sigma_0$ and $\sigma_r(w_i, \alpha) = \sigma_i$ for every $i \in [1..k]$. In this case, the influence cloud $\mathcal{IC}_u^r$ is referred to as a *realization* of the potential cloud configuration Z. (Note that a potential cloud configuration may have many different realizations.)

More generally, we say that the potential cloud configuration sequence $\bar{Z}^r = (Z^1, \ldots, Z^r)$ is realized by the initiator u in execution α if for every round $i = 1, \ldots, r$, the influence cloud $\mathcal{IC}_u^i$ is a realization of the potential cloud configuration Z^i. In this case, the sequence of influence clouds of u up to round r, $\bar{\mathcal{IC}}_u^r = \langle \mathcal{IC}_u^1, \ldots, \mathcal{IC}_u^r \rangle$, is referred to as a realization of $\bar{Z}^r$. (Again, a potential cloud configuration sequence may have many different realizations.)

For a potential cloud configuration Z, let $E_u^r(Z)$ be the event that Z is realized by the initiator u in (round r of) the run of algorithm A. For a potential cloud configuration sequence $\bar{Z}^r$, let $E_u(\bar{Z}^r)$ denote the event that $\bar{Z}^r$ is realized by the initiator u in (the first r rounds of) the run of algorithm A.

Lemma 3. *Restrict attention to executions of algorithm A that satisfy event N, namely, in which all stable influence clouds are disjoint. Then $\mathbb{P}\left[E_u(\bar{Z}^r)\right] = \mathbb{P}\left[E_v(\bar{Z}^r)\right]$ for every $r \in [1, \rho]$, every potential cloud configuration sequence $\bar{Z}^r$, and every two initiators u and v.*

The proof (deferred to the full paper) proceeds by induction. For the base case, it is intuitively clear that every initiator has the same probability for some specific action. In the induction step, we generalize this statement to influence clouds, conditioned on all influence clouds being disjoint so far.

We now conclude that for every potential cloud configuration Z, every execution α and every two initiators u and v, the events $E_u^\rho(Z)$ and $E_v^\rho(Z)$ are equally likely. More specifically, we say that the potential cloud configuration Z is *equi-probable for initiators u and v* if $\mathbb{P}\left[E_u^\rho(Z) \mid N\right] = \mathbb{P}\left[E_v^\rho(Z) \mid N\right]$. Although a potential cloud configuration Z may be the end-colud of many different potential cloud configuration sequences, and each such potential cloud configuration sequence may have many different realizations, the above lemma implies the following (integrating over all possible choices).

Corollary 1. *Restrict attention to executions of algorithm A that satisfy event N, namely, in which all (final) stable influence clouds are disjoint. Consider two initiators u and v and a potential cloud configuration Z. Then Z is equi-probable for u and v.*

By assumption, algorithm A errs with probability $p_{\text{err}} \leqslant 1/n$. Let S be the event that A elects exactly one leader. We get

$$\mathbb{P}\left[S \mid M \wedge N\right] \geqslant \mathbb{P}\left[M \wedge N\right] - p_{\text{err}} = 1 - o(1). \tag{3}$$

Conditioning on event $M \wedge N$, let X be the random variable that represents the number of disjoint influence clouds generated by algorithm A. By Cor. 1, each of the initiators has the same probability p of generating a leader cloud. Algorithm A succeeds whenever event S occurs. Its success probability assuming $X = c$ is

$$\mathbb{P}\left[S \mid M \wedge N \wedge (X = c)\right] = cp(1-p)^{c-1}. \tag{4}$$

For any given c, the value of (4) is maximized if $p = \frac{1}{c}$, which yields that $\mathbb{P}\left[S \mid M \wedge N \wedge (X = c)\right] \leqslant 1/e$ for any c. It follows that $\mathbb{P}\left[S \mid M \wedge N\right] \leqslant 1/e$ as well. This, however, is a contradiction to (3) and completes the proof of Theorem 3 for algorithms without unique identifiers.

We now briefly argue why our result holds for any algorithm B that runs in a model where nodes are equipped with unique ids (chosen by the adversary). Suppose that, w.h.p., B succeeds in electing a leader while sending only $f(n)$ messages. Now consider an algorithm B' in our model that is identical to B

with the difference that before performing any other computation, every node generates a random number from the range $[1, \ldots, n^4]$ and uses this value instead of the unique id. Let I be the event that all node ids are distinct; clearly I happens with high probability. Therefore, by the success probability of B, it follows that B' also succeeds with probability $1 - o(1)$ (conditioned on I), which contradicts our result for algorithms without unique ids. This completes the proof of Theorem 3. □

5 Conclusion

We studied the role played by randomization in distributed leader election. Some open questions on randomized leader election are raised by our work: (1) Can we find (universal) upper and lower bounds for general graphs? (2) Is $\Omega(\sqrt{n})$ a lower bound on the number messages needed for a complete graph, *regardless* of the number of rounds?

References

1. Afek, Y., Gafni, E.: Time and message bounds for election in synchronous and asynchronous complete networks. SICOMP 20(2), 376–394 (1991)
2. Afek, Y., Matias, Y.: Elections in anonymous networks. Inf. Comput. 113(2), 312–330 (1994)
3. Angluin, D.: Local and global properties in networks of processors (extended abstract). In: STOC, pp. 82–93 (1980)
4. Augustine, J., Pandurangan, G., Robinson, P., Upfal, E.: Towards robust and efficient distributed computation in dynamic peer-to-peer networks. In: SODA (2012)
5. Snir, M., Scheiber, B.: Calling names on nameless networks. Inf. Comput. 113(1), 80–101 (1994)
6. Loui, M.C., Matsushita, T.A., West, D.B.: Election in a complete network with a sense of direction. Information Processing Letters 22(4), 185–187 (1986)
7. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live - an engineering perspective (2006 invited talk). In: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (2007)
8. Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst. 5(1), 66–77 (1983)
9. Gupta, I., van Renesse, R., Birman, K.P.: A Probabilistically Correct Leader Election Protocol for Large Groups. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 89–103. Springer, Heidelberg (2000)
10. Humblet, P.: Electing a leader in a clique in $O(n \log n)$ messages. Intern. Memo., Laboratory for Information and Decision Systems. M.I.T, Cambridge (1984)
11. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Inf. Comput. 88(1), 60–87 (1990)
12. Khan, M., Kuhn, F., Malkhi, D., Pandurangan, G., Talwar, K.: Efficient distributed approximation algorithms via probabilistic tree embeddings. In: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, PODC 2008, pp. 263–272. ACM, New York (2008)

13. Korach, E., Kutten, S., Moran, S.: A modular technique for the design of efficient distributed leader finding algorithms. ACM Trans. Program. Lang. Syst. 12(1), 84–101 (1990)
14. Korach, E., Moran, S., Zaks, S.: Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In: PODC 1984, pp. 199–207. ACM, New York (1984)
15. Korach, E., Moran, S., Zaks, S.: The optimality of distributive constructions of minimum weight and degree restricted spanning trees in a complete network of processors. SIAM Journal on Computing 16(2), 231–236 (1987)
16. Korach, E., Moran, S., Zaks, S.: Optimal lower bounds for some distributed algorithms for a complete network of processors. Theoretical Computer Science 64(1), 125–132 (1989)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
18. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
19. Le Lann, G.: Distributed systems - towards a formal approach. In: IFIP Congress, pp. 155–160 (1977)
20. Lynch, N.: Distributed Algorithms. Morgan Kaufman Publishers, Inc., San Francisco (1996)
21. Malkhi, D., Reiter, M., Wright, R.: Probabilistic quorum systems. In: PODC 1997, pp. 267–273. ACM, New York (1997)
22. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press (2004)
23. Nygren, E., Sitaraman, R.K., Sun, J.: The akamai network: a platform for high-performance internet applications. SIGOPS Oper. Syst. Rev. 44(3), 2–19 (2010)
24. Peleg, D.: Time-optimal leader election in general networks. Journal of Parallel and Distributed Computing 8(1), 96–99 (1990)
25. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM (2000)
26. Ramanathan, M.K., Ferreira, R.A., Jagannathan, S., Grama, A., Szpankowski, W.: Randomized leader election. Distributed Computing, 403–418 (2007)
27. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: SIGCOMM 2001, pp. 161–172. ACM, New York (2001)
28. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
29. Santoro, N.: Design and Analysis of Distributed Algorithms. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience (2006)
30. Singh, G.: Efficient distributed algorithms for leader election in complete networks. In: ICDCS, pp. 472–479 (1991)
31. Tel, G.: Introduction to distributed algorithms. Cambridge University Press, New York (1994)
32. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: a resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications 22(1), 41–53 (2004)
33. Kutten, S., Zinenko, D.: Low Communication Self-stabilization through Randomization. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 465–479. Springer, Heidelberg (2010)

Linear Space Bootstrap Communication Schemes

Carole Delporte-Gallet[1,⋆], Hugues Fauconnier[1,⋆], Eli Gafni[2], and Sergio Rajsbaum[3,⋆⋆]

[1] LIAFA-Université Paris-Diderot, France
{cd,hf}@liafa.jussieu.fr
[2] Computer Science Department, UCLA, USA
eli@ucla.edu
[3] Instituto de Matemáticas, UNAM, Mexico
rajsbaum@math.unam.mx

Abstract. We consider a system of n processes with ids not a priori known, that are drawn from a large space, potentially unbounded. How can these n processes communicate to solve a task? We show that n a priori allocated Multi-Writer Multi-Reader (MWMR) registers are both needed and sufficient to solve any read-write wait free solvable task. This contrasts with the existing possible solution borrowed from adaptive algorithms that require $\Theta(n^2)$ MWMR registers.

To obtain these results, the paper shows how the processes can *non blocking* emulate a system of n Single-Writer Multi-Reader (SWMR) registers on top of n MWMR registers. It is impossible to do such an emulation with $n-1$ MWMR registers.

Furthermore, we want to solve a sequence of tasks (potentially infinite) that are sequentially dependent (processes need the previous task's outputs in order to proceed to the next task). A non blocking emulation might starve a process forever. By doubling the space complexity, using $2n-1$ rather than just n registers, the computation is wait free rather than non blocking.

Keywords: shared memory, read/write registers, distributed algorithms, wait free, space complexity, renaming.

1 Introduction

In many distributed algorithms it is assumed that the processes, $p_1, \ldots, p_n$, communicate using Single-Writer Multi-Reader (SWMR) registers, $R_1, \ldots, R_n$, so p_i knows it is the i-th process, and can write exclusively to R_i. However, often processes do not know their indexes, they know only their ids, and the number of possible ids N, is much bigger than the number of processes, n. In this situation, preallocating a register for each identifier would lead to a distributed algorithm with a very large space complexity. One would like the processes to run a *renaming* algorithm as a preprocessing stage, to obtain new ids from a

⋆ Supported by DISPLEXITY ANR-11-BS02-014.
⋆⋆ Supported by UNAM-PAPIIT project IN104711.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 363–377, 2013.

smaller range, $M(n)$, that depends only on n, and then use these ids to index only $M(n)$ SWMR registers. Several wait free renaming algorithms e.g. [6,12] are known that reduce the name space to $M(n) = 2n - 1$ (and this is the best that can be done, except for some exceptional values of n [14]). However, if the system did not allocate N SWMR registers to start with, how do the processes communicate to run the renaming algorithm?

Such a question has been considered in the context of *adaptive* computation [7,8,26,2,4]. In these papers, n is unknown, and moreover processes "arrive" and "depart." When first solving adaptive renaming to allocate MWMR registers to processes, they obtain $\Theta(n^2)$ space complexity. The first basic question we address here is: In the most favorable situation, when n is known and we want to solve a *task*, how much space do we need? Is solving renaming first unavoidable?

Recall that a task e.g. [21] is a one shot problem, where processes start with inputs and after communicating with each other, must decide on outputs that satisfy the task's specification. Famous examples of tasks are consensus and set agreement.

We show that indeed it is possible to solve any read-write wait free solvable task with n MWMR registers. To obtain this result, the paper shows how the processes can *non blocking* emulate a system of n SWMR registers on top of n MWMR registers. Moreover, as we explain, it is not hard to prove that no such emulation exists on less than n MWMR registers.

An application of the non blocking emulation is that one can run any SWMR wait free algorithm that solves a task on n MWMR registers. In particular, one can run directly a SWMR $(2n-1)$-renaming algorithm such as the one of [12] on top of n MWMR registers. This is a significant improvement over the previous adaptive $\Theta(n^2)$ space renaming algorithm, when that algorithm is instantiated to our context. Admittedly, the previous renaming algorithm will actually use fewer than n^2 registers but nevertheless since it is apriori unknown how the algorithm will evolve a preallocation of $\Theta(n^2)$ registers is necessary. Notice that as the simulation is non blocking, a SWMR algorithm that solves a task on top of the n MWMR registers may incur some cost in time: a process may not be able to produce an output value, until another process finishes the algorithm and exits the emulation.

As said, with n registers we can solve $(2n - 1)$-renaming. Using these new names, with additional $2n - 1$ registers, each process can obtain a dedicated register, and from there on emulate a simulated write operation in a wait free-manner. Are $3n - 1$ MWMR necessary to have a wait free emulation of a write operation in a non-terminating environment? We show that a total of just $2n-1$ MWMR registers is sufficient, describing how the processes can *wait free* emulate a system of n SWMR registers on top of $2n-1$ MWMR registers. We conjecture that $2n - 1$ registers are the minimum possible for wait free emulation. The wait free emulation allows to solve a sequence of tasks (potentially infinite) that are sequentially dependent (processes need the previous task's outputs in order to proceed to the next task). A non blocking emulation might starve a

process forever. By doubling the space complexity, using $2n-1$ rather than just n registers, the computation is wait free rather than non blocking.

The paper also describes an algorithm to broadcast the value of one of the processes, using $n/2$ MWMR registers. The algorithm, which seems interesting in itself, illustrates how the information propagates in our emulations. It is simple, but the proof is subtle.

At the end of the paper we briefly discuss why we believe our techniques will be useful for dynamic systems of bounded concurrency n e.g. [4]. In such a system, any number of processes arrive, compute and depart. Yet, at any point in time the number of arrivals exceeds the number of departures by at most n.

We stress that our interest is in space complexity, our emulations are not particularly efficient in terms of step complexity. Also, there are various previous papers dealing with a similar problem, but in the adaptive setting. In [10] there is a definition similar to our emulation problem, as "store-collect" of a (key,value) problem. Another definition of store-collect, by equivalence to an array (hence more like our problem specification) appears in [3]. Similarly, solutions without explicitly solving renaming first appear in [9], although not very space efficient.

There is a long history of space complexity results, starting with the mutual exclusion lower bound of [13] and even before (see references herein). In this context, [27] considers renaming with the same motivation that we do, and shows that $2\lceil logn \rceil + 1$ registers are sufficient to solve it, and a corresponding lower bound within a constant factor but their model is stronger than ours. There are various algorithms and lower bounds on the number of registers needed to solve specific problems e.g. [16,17,19,26], but we are not aware of general emulations. In [24] there is a lower bound for non blocking implementations, but it is $n-1$ registers. In special cases it can be beaten [5]. In [18] a $\Omega(\sqrt{n})$ space complexity lower bound for a randomized nonblocking implementation of consensus is presented.

Due to lack of space, we refer the reader to the long version [15] for detailed proofs.

2 Model

We assume a standard asynchronous shared-memory model of computation with n processes communicating by reading and writing to a fixed set of shared registers [11,22]. The processes have unique ids in $\{1,\ldots,N\}$, with $N >> n$. Processes may take a finite or an infinite number of steps but we assume that at least one process takes an infinite number of steps.

The shared memory consists of a set of atomic Multi-Writer Multi-Reader (MWMR) registers. We assume that processes can read and write any MWMR register and these operations are atomic [20]. For short, we usually omit the term atomic. If r is such a MWMR register, a process can write x on r using $write(r,x)$, and read on r using $read(r)$. A process executes its code taking three types of atomic steps: the read of a register, the write of a register, and the modification of its local state.

We consider also more powerful operations to read the registers. A *collect* is an iterative read of all registers. A *scan* returns a snapshot, an instantaneous view of the registers. In [1], there are non blocking and wait free linearizable implementations of scan. A non blocking implementation of scan ($NBScan$) can be obtained by repeating $Collect$ operations until two of them return the same values. A wait free implementation of scan ($Scan$) is more involved, and embeds a snapshot with the write. So in this case instead of *write*, we use the term *update* ($update(r, v)$ updates the register r with the value v).

Two progress conditions that have received much attention are non blocking and wait free. The *non blocking* progress condition states that when there are concurrent operations at least one process terminates its operations. The *wait free* [20] progress condition states that each process terminates its operations in a bounded number of its own steps.

We consider also a model in which each process has its own atomic Single-Writer Multi-Reader register (SWMR). Process p can write a value x in its SWMR register with the operation WRITE(p, x) and all the processes may read the value in the SWMR register of p with READ(p).

We say that we have a non blocking (resp. wait free) *emulation* of the n processes SWMR model using m MWMR registers if we implement WRITE and READ in terms of *write* and *read* such that the implementation is linearizable [23] and the progress condition is non blocking (resp. wait free).

A regular SWMR register [25], is a weaker type of register. A SWMR register is *regular* if when no write is concurrent with a read operation, the read returns the current value in the register. Otherwise the read returns any value written by the concurrent writes or the last value in the register before the concurrent writes. With regular SWMR registers it is possible to wait free free implement atomic SWMR register [22,25]. So to emulate non blocking (resp. wait free) the n processes SWMR register model using m MWMR registers it is enough to emulate non blocking (resp. wait free) n processes model with regular SWMR register model.

3 Preliminaries

We consider algorithms that have the same structure as the algorithm in Figure 1. Processes share a set $\mathcal{R}$ of MWMR registers. Each process maintains a variable $View$. Each process repeatedly reads all registers (function $Collect()$) and updates its variable $View$ by adding all it has just read in $View$ together with some other values in variable *input*, and then writes it in some registers.

For each $R \in \mathcal{R}$, R^τ denotes the value of register R at time τ. Similarly, $View_p^\tau$ denotes the value of variable $View$ of p at time τ.

We say that v is *eventually forever* in R if there is a time τ such that for all time $\tau' > \tau$, $v \in R^{\tau'}$. We have directly from the algorithm:

Observation 1. *For all processes p and for all times τ and τ', if $\tau < \tau'$ then $View_p^\tau \subseteq View_p^{\tau'}$.*

```
function Collect():
1   V = ∅
2   for all R in R
3       V = V ∪ read(R)
4   return V

main loop:
    repeat forever
5       View = Collect() ∪ View ∪ input
6       Let (deterministically) Reg be some register in R
7       write(Reg, View)
```

Fig. 1. Generic algorithm

Following [13], we say that register R is *covered* by process p at some time τ if the next register that p writes after time τ is R: more precisely p *covers* R at time τ if, at time τ, the next writing of p (Line 7) is on register R and p has begun the *Collect* in Line 5. Note that if p does not cover R, before writing R, p reads the value of R (by *Collect* in Line 5) and then it will write in R at least this value.

If some register is covered the value of this register may be lost. So, by extension, we say that R is *V–covered* at time τ if all processes covering R at time τ are going to write sets containing V in register R. If R is V-covered at time τ, next writings contains V (by processes covering R) or contains R^τ (by processes not covering R that read R before writing). We have:

Lemma 1. *If, at some time τ, register R is V–covered at time τ then for all $\tau' \geq \tau$, $(R^\tau \cap V) \subseteq R^{\tau'}$.*

If no process covers R at time τ, then by definition R is V-covered for any set V, then by the previous Lemma after time τ, R contains forever R^τ. In particular:

Lemma 2. *If R is not covered at time τ and $v \in R^\tau$ then v will be eventually forever in R.*

Let $\mathcal{R}_\infty$ be the set of registers infinitely often written:

Lemma 3. *If v is eventually forever in R then v is eventually forever in all registers in $\mathcal{R}_\infty$.*

Lemma 4. *If at time τ, $card\{R \in \mathcal{R} | v \in R^\tau\} \geq n$, then v will be forever in all registers in $\mathcal{R}_\infty$.*

4 Broadcast

We give here a very weak definition of broadcast. Essentially this definition ensures that the value of some process will be known by all processes making an infinity of steps. More precisely, we assume that each process p has a value v_p to broadcast, the broadcast is defined by way of a primitive *Deliver* returning a set of values. The broadcast ensures:

– *(integrity)* if v belongs to the set of values returned by some *deliver* then $v = v_p$ for some process p,
– *(convergence)* there is a value v and a time τ after which every set returned by any *Deliver* contains v.

Shared variable :
`array of` m `MWMR register` : R

CODE FOR PROCESS p
Local variable:
`set of Values` $View = \{v_p\}$
`set of Values` $Deliver = \emptyset$

function $Collect()$:
1 variable: `set of Values` $V = \emptyset$
2 **for** i **from** 1 **to** m **do**
3 $V = V \cup read(R[i])$
4 return V

Task main:
5 **repeat forever**
6 **for** $i = 1$ **to** m
7 $View = View \cup NBScan()$;
8 $write(R[i], View)$

Task Deliver:
9 **forever** $Deliver = Collect()$

Fig. 2. Broadcast with m MWMR registers

Here processes share m MRMW registers $R[i]$, $1 \leq i \leq m$ and write successively each register.

The algorithm of Figure 2 is a simple application of the generic algorithm of Figure 1. But here in, the main loop, we use a non blocking scan $NBScan$ instead of $Collect$. $NBScan()$ returns V, a snapshot of the registers, such that there is a time τ such that $R[i]^\tau = V[i]$. $NBScan()$ is only assumed to be non blocking. Implementation of such non blocking snapshot is easy [1]. As $NBScan$ is a particular form of $Collect$ all previous Lemmata from Section 3 apply. For a snapshot V, $V[i]$ is the value returned for register $R[i]$ and abusing notation, V may denote $\cup_{1 \leq i \leq m} V[i]$.

A *Deliver* reads all the registers and returns the union of the values read in the registers. Note, using $Collect()$ that *Deliver* always terminate. We prove the main property:

Theorem 2. *Algorithm of Figure 2 implements broadcast if* $m > \frac{n}{2}$.

5 Non Blocking Emulation of SWMR Registers

We first describe the emulation, and then the lower bound, in Section 5.2.

5.1 The Algorithm

The algorithm in Figure 3 is a non blocking emulation of regular SWMR registers for n processes using an array R of m MWMR registers, with $m \geq n$.

```
Shared variable :
    array of m MWMR-register : R      To ensure non blocking we assume m ≥ n
CODE FOR PROCESS p
Local variable:
    set of Values View = ∅
    integer k = 0
WRITE(p, x):
1   v = (x, p, k)
2   next = 0
3   View = View ∪ {v}
4   do
5       Snap = NBScan()          NBScan returns a snapshot of the shared memory
6       View = Snap ∪ View
7       write(R[next], View)
8       next = (next + 1) mod m
9   until (card{r|v in Snap[r]} ≥ n)
10  k = k + 1

READ(q):
11  View = Collect()
12  return x such that (x, q, u) ∈ View with maximal u
```

Fig. 3. Non blocking implementation of regular SWMR registers for n processes

In the following to distinguish between the writings of MWMR registers and the emulation of SWMR writings, we denote the first ones using lower case and the second ones with upper case. To make its k-WRITE, a process p adds the value to WRITE (in fact the value, its identity and its timestamp k) to its variable $View$. The WRITE ends when the value to be WRITTEN is in all the m registers. A READ of value WRITTEN by process q collects the values present in registers and returns the value from q with the maximal timestamp among all the values from q.

As in the generic algorithm of Figure 1, each process p maintains a variable $View$ containing all the information it knows. Iteratively, each process reads all the m registers by a non blocking scan and accordingly updates its variable $View$ before writing it in the next register in cyclic order. The WRITE of v terminates as soon as v is in n registers. Here instead of $Collect$, we use $NBScan$, a non blocking $Scan$, as described in Section 2.

The main point here is that as we have n processes and at least n registers then at least one register will not be covered and then all the values contained

in this register will eventually be present in all registers and will remain forever in all registers. Hence as soon as a value is present in all registers the WRITE is terminated because afterwards the *Collect* of every READ will contain this value.

First we prove the safety properties of the implementation of regular register.

We say that the WRITE of v *succeeds* at time τ if there is some register such that after time τ, v is forever in this register. By extension, we say that the WRITE of v succeeds if there is a time at which the WRITE of v succeeds or equivalently if v is eventually forever in R. Directly from the algorithm we get the following lemmata:

Lemma 5. *Let $v = (x, p, k)$ and $v' = (y, p, k')$ such that $k \geq k'$, if v succeeds at time τ, then v' succeeds at time τ too.*

By Lemma 3 and the code of the algorithm:

Lemma 6. *If the* WRITE *of v succeeds at time τ, after this time v is returned by every Collect.*

Lemma 7. *Let S be the set of all values (x, p, k) that succeed at time τ and K the maximal k over all (x, p, k) in S, then* READ *of p returns the value $v = (x, p, k) \in S$ with $k \geq K$.*

From Lemma 4:

Lemma 8. *If at some time τ, v is in n registers, then the* WRITE *of v succeeds at time τ.*

Lemma 9 (safety). *Any* READ(p) *returns the last value x such that* WRITE(p, x) *terminates before the beginning of the* READ*, or a value x such that* WRITE(p, x) *is concurrent with the* READ*.*

Proof. Assume x is the kth WRITE of p, let v be (x, p, k). Consider any READ(p) and let E be the set of values returned by the *Collect* made for this READ. By Lemma 6 all values for which the WRITE has succeeded are in E. WRITE(p, x) returns when, for a $NBScan$ (Line 5), v belongs to n registers (Line 9) at some time. Then by Lemma 8, v succeeds by the time of the $NBScan$ and E contains all values for which the WRITE has terminated. Lemma 7 proves that the value returned by the READ(p) is either the last value for which the WRITE by p has terminated or a value for which the WRITE is concurrent.

Now we prove that the algorithm is non blocking. First as *Collect* is wait free, any READ is wait free too:

Lemma 10. *Any* READ *made by a process that takes an infinite number of steps terminates.*

By Lemma 3, and the fact that the WRITE ends when the value is in all registers:

Lemma 11. *Assume the registers are written infinitely often, if the* WRITE *of v succeeds then v will eventually be forever in all registers and if the process that* WRITES *v takes an infinite number of steps, the* WRITE *terminates.*

Lemma 12. *If the registers are written infinitely often then an infinity of* WRITE *terminate.*

Proof. By contradiction assume the contrary: there is a time τ after which no WRITE terminates. When a process has terminated all its WRITEs it stops writing the registers, then there is at least one process that takes an infinite number of steps and does not terminate the WRITE of some v. By Lemma 11, there is time at which all registers will contain values for which the WRITE is not terminated. By pigeon hole principle one of these registers is not covered and contains a value for which the WRITE is not terminated, by Lemma 2 the WRITE of this value succeeds, and by Lemma 11, the WRITE of this value terminates –a contradiction.

If after some time there is no writing of the registers, being non blocking any $NBScan$ returns:

Lemma 13. *If some process that takes an infinite number of steps is stuck forever on a $NBScan$ then the registers are written infinitely often.*

Lemma 14 (non blocking). *If $m \geq n$, and the* WRITE *of v by a process p that takes an infinite number of steps does not terminate, then infinitely often some* WRITE*s terminate.*

Proof. By contradiction, assume that the WRITE of some process p that takes an infinite number of steps does not terminate and only a finite number of WRITEs occur. Then by Lemma 12, there is a time after which no registers are written and by Lemma 13, p may not be stuck on a $NBScan$ and hence p makes progress in its code and hence writes registers infinitely often. By Lemma 12 an infinity of WRITE terminates —a contradiction.

Lemmata 9 and 14 prove that the algorithm in Figure 3 is a non blocking emulation of regular SWMR registers for n processes from n MWMR registers. We now use the classical wait free transformation [22,25] from regular to atomic registers to conclude:

Theorem 3. *There is a non blocking emulation of SWMR registers for n processes from n MWMR registers.*

5.2 Lower Bound

We prove in this section that we cannot emulate SWMR registers for n processes with less than n MWMR registers.

Lemma 15. *SWMR registers for n processes cannot be emulated with $n-1$ MWMR registers.*

Proof. Consider a set of n processes $p_1, \ldots, p_n$. By contradiction, assume we can emulate the n processes SWMR model with $(n-1)$ MWMR registers. We construct inductively a run e where this assumption is not satisfied. For this we

construct by induction on k a partial run e_k and a set $\mathcal{R}_k$ of k registers each being covered by processes $p_1, \ldots, p_k$.
$(k = 1)$: Consider a run where only process p_1 takes steps and its code is WRITE(p_1, p_1).

Claim: *p_1 has to write in some register.*
Proof: By contradiction assume p_1 does not write in any register and assume the code of p_n is READ(p_1). Once p_1 ends its WRITE, p_n takes steps and does not find any value written by p_1, contradicting the semantics of a register.

Let the partial run e_1, where only p_1 takes steps, its code is WRITE(p_1, p_1) and it stops just before its first writing of a register, say R_1. Then p_1 covers R_1 and $\mathcal{R}_1 = \{R\}$.
$(k < n - 1)$: By induction let e_k be such that $\{p_1, \cdots, p_k\}$ covers each register in the set of k registers $\mathcal{R}_k$.

Let a run that extends partial run e_k by the process p_{k+1} that executes the code WRITE(p_{k+1}, p_{k+1}).

Claim: *p_{k+1} has to write in some register not in $\mathcal{R}_k$.*
Proof: By contradiction assume p_{k+1} does not write any register not in $\mathcal{R}_k$ and assume the code of p_n is READ(p_{k+1}). p_{k+1} ends its WRITE and has only written registers in $\mathcal{R}_k$, then each process in $\{p_1, \ldots, p_k\}$ executes one step and overrides each register in $\mathcal{R}_k$. Then p_n executes the code for READ(p_{k+1}). But this execution is indistinguishable (for all processes different from p_{k+1}) from the one in which p_{k+1} does not make any WRITE and the READ may not return the value WRITTEN by p_{k+1}.

In the partial run e_{k+1}, that extends e_k, p_{k+1} takes steps, executes the code WRITE(p_{k+1}, p_{k+1}). and stops before writing a register that is not in $\mathcal{R}_k$, say R_{k+1}. Then p_{k+1} covers R_{k+1}. Define $\mathcal{R}_{k+1} = \mathcal{R}_k \cup R_{k+1}$.

When $k = n - 1$, $\mathcal{R}_{n-1}$ contains all the MWMR registers. Now, consider process p_n and assume it runs the code WRITE(p_n, p_n). Run e_n is an extension of $e_n - 1$ in which p_n takes steps until it ends its WRITE and takes no other steps. Each process executes one step and overrides each register in $\mathcal{R}_{n-1}$. At this point the value WRITTEN by p_n is not in the local memory of any process (except p_n) and in particular it is not in the local memory of p_1. Then p_1 ends its WRITE(p_1, p_1), at the end of this WRITE, the value WRITTEN by p_n is not in any MWMR registers and the run is indistinguishable (for all processes different from p_n) from the same run in which p_n does not take any WRITE. Now if p_1 runs READ(p_n), p_1 cannot get the value written by p_n.

6 Wait Free Emulation of SWMR Registers

The previous emulation is only non blocking. Using $3n-1$ MWMR registers with help of the simulation of n non blocking SWMR registers it is easy to simulate n *wait free* SWMR registers. For this, the $3n-1$ registers are partitioned into a set W of n registers and a set PR of $2n-1$ registers. The n MWMR registers of W are used to (non blocking) simulate n SWMR registers with the algorithm of

```
Shared variable :
        array of n − 1 MWMR-register : W
        array of n + 1 MWMR-register : PR
CODE FOR PROCESS p
Local variable:
        set of Values View = ∅
        integer k = 0
WRITE(p, x):
1   v = (x, p, k)
2   next = 0
3   nextReg = W[next]
4   View = View ∪ {v}
5   Name = index of p in View.proc
                                  View.proc: set of processes in View ordered by name
6   do
7       Snap = Scan()                                          Scan of W and PR
8       View = Snap ∪ View
9       if (Snap[card(View.proc)] = ⊥) then
10          Name = index of p in View.proc
11          Update(PR[card(View.proc)], View)
12      else
13          if next = n then update(PR[Name], View)
14          else update(W[next], View)
15          next = (next + 1) mod (n + 1)
16  until (card{r|v in Snap[r]} ≥ n)
17  k = k + 1

READ(q):
18  View = Collect()
19  return x such that (x, q, u) ∈ View with maximal u
```

Fig. 4. Wait free emulation with $2n$ MWMR registers for n processes

Figure 3. With these n SWMR registers we can run a renaming algorithm [6,12]. In fact as such an algorithm always terminates in a finite number of steps it is easy to verify that the non blocking simulation is enough to ensure that the renaming terminates. Hence each process p of the n processes gets an unique identity $id(p)$ in the set $\{1, \cdots, 2n-1\}$. To WRITE a value v, p will write in the $id(p)$th register of PR. As p is the unique writer, the simulation is wait free. But it is possible to reduce the number of MWMR registers to $2n-1$. In the algorithm we propose, we eventually get an unique identity in the set $\{1, \ldots, n\}$.

We present first an algorithm with $2n$ MWMR registers. The algorithm of Figure 4 has a structure similar to the previous algorithms. Each process maintains a variable $View$ containing all information it knows. Here the processes access shared variables using wait free snapshot by means of primitives $Update$ and $Scan$. Implementations of such primitives are described for example in [1].

For a set V of values $V.proc$ denotes the list of all processes occurring in V (process q occurs in V if there is any $v = (x, q, s)$ in V). For convenience all the lists of processes are ordered by process identities. The *index* of process p in an ordered list of processes is the rank of p in the list. Indexes begins on 0, for example, for $1, 3, 8, 15$ the index of 1 is 0 and index of 3 is 1.

In the algorithm the processes share an array W (Working registers) of $n-1$ MWMR registers $(W[0], \ldots, W[n-2])$ and an array PR (Personal Registers) of $n+1$ MWMR registers $(PR[0], \ldots, PR[n])$.

Personal registers play a double part: they give the supposed number of participants and give a personal register for each participant. For this, the last cell different from $\perp$ gives the supposed number and list of participants: if $PR[a]$ is this last cell then the supposed number of participants is a and the list of participants is $PR[a].proc$. In the ordered list of assumed participants, process p determines its index (variable $Name$) and will consider $PR[Name]$ as its personal register (Lines 5 and 10).

As usual to WRITE a value x (assuming it is the kth), a process adds $v = (x, p, k)$ to its $View$ and successively writes (by $Update$ and after updating its $View$ by $Scan$ in Line 8) to the $n-1$ registers W (Line 14) and in its personal register $PR[Name]$(Line 13). But it has to check that its $Name$ is correct. After each $Scan$ a process will verify that the number of participants corresponds to the contents of PR registers (Line 9): if the number of participants is a then $PR[a]$ must be the last cell different from $\perp$. If it is the case, its $Name$ is up to date and it writes the next register (Line 13 and 14). If it not the case, the index of the last non $\perp$ cell in PR is less than the number of participants that p has seen, then p writes in the correct cell of PR ($PR[a]$ if a is the number of participants seen by p) (Line 11) and determines its new index $Name$ in its list (Line 10).

As before the WRITE terminates when v is in at least n registers (condition in Line 16): when v is in n registers at least one of these registers is not covered. Roughly speaking, WRITE is wait free because eventually the last cell distinct of $\perp$ will correspond to the actual number of participants, and the list of participants in this cell will be the actual list of all participants, hence processes will have a personal register in which it will be alone to write.

The same arguments proving the safety properties of the non blocking algorithm of Figure 3 apply here (essentially the WRITE of v terminates when v is in at least n registers. By a non covering argument, v will be returned by all $Scan$). Then we restrict ourselves to prove the wait freed progress condition.

For each register r, $r.proc$ is the list of all processes occurring in r.

Lemma 16. *There is a time after which each register r of $PR[0], ..., PR[n-1]$ is written by at most one process.*

Proof. Let max be the greatest i such that $PR[i] \neq \perp$. By definition of the index, max may not be the $Name$ of any register, then $PR[max]$ may only be written by processes in Line 11 for which for the previous $Scan$, $PR[max]$ was $\perp$. Hence this register is written only a finite number of time. Consider time τ

after which only processes that make an infinite number of *Scan/Update* take steps and after which $PR[max]$ is no more written.

Claim: *There are a set Participant of size max and a time $\sigma > \tau$ after which: (1) $PR[max].proc = Participant$ and (2) for every process p that makes an infinite number of Scan/Update, $View_p.proc = Participant$.*
Proof: Let *Participant* be the value of $PR[max].proc$ a time τ. Let p be a process that makes an infinite number of *Scan/Update*, p will make some *Scan* after time τ getting $PR[max]$. Then there is a time τ_p after which $PR[max].proc \subseteq View_p.proc$. After time τ_p, $PR[max].proc \neq View_p.proc$ is equivalent to $max \neq card(View_p.proc)$. If $Participant = PR[max].proc \neq View_p.proc$ then $max < card(View_p.proc)$ and p will write in $PR[card(View_p.proc)]$ contradicting the definition of $PR[max]$. Let time σ be the maximum of τ_p over all p making an infinite number of *Scan/Update*, after time σ conditions (1), and (2) of the claim are satisfied.

Let i such that $0 \leq i \leq n$ and assume that $PR[i]$ is written after time σ, then by definition of max, we can assume that $0 \leq i < max$. If, after time $\sigma > \tau$, some q writes $PR[i]$, by definition of τ, this process makes an infinite number of *Scan/Update*. By the claim, $View_q.proc = Participant$, and if q writes in $PR[i]$ that means that the index of q in *Participant* is i and as each process making an infinity of *Scan/Update* has an unique index in *Participant*, q is the only writer of $PR[i]$.

Lemma 17. *The algorithm in Figure 4 emulates wait free n SWMR registers with 2n MWMR registers for n processes.*

Proof. We prove only the wait freed progress condition. Assume a process p that takes an infinite number of steps tries to WRITE $v = (x, p, k)$ in its register and does not succeed. Then (v, p, k) is inserted in $View_p$ and by Lemma 11, (v, p, k) remains forever in $View_p$.

By the algorithm, p executes an infinite number of *Scan/Update* and writes $View_p$ an infinite number of times in at least one register of PR. By Lemma 16, there is a register $PR[i]$ and a time τ after which p writes infinitely often in $PR[i]$ and p is the only writer of $PR[i]$. Then v will be forever in $PR[i]$. After time τ, every process q that executes *Scan/Update* reads $PR[i]$ and includes it in its $View_q$. Then all processes that write after time τ in some register will write v in this register too. As we assume that the WRITE of p does not terminate, at least p writes in each register of W. Therefore, there is a time after which v will be forever in all the $n-1$ registers W and in register $PR[i]$, then for any *Scan* v will be in at least n registers and the WRITE of v ends.

Notice that the information in $PR[n]$ can be easily integrated to $PR[n-1]$. So we can use only $2n-1$ MWMR registers: The shared MWMR registers array W is of size $n-1$ and array PR is of size n, and we replace Line 9 to Line 11 from algorithm in Figure 4 with Lines in Figure 5 :

Theorem 4. *The algorithm in Figure 5 emulates wait free n SWMR registers with $2n-1$ MWM Rregisters for n processes.*

```
01 if Name ≠ index of p in View.proc then
02     Name = index of p in View.proc
03 if (card(View.proc) = n  and card(PR[n − 1].proc) = n − 1)
       or (card(View.proc) < n  and Snap[card(View.proc)] = ⊥))
04     then
05     if (card(View.proc) = n) then
06         update(PR[n − 1], View)
07         else
11         update(PR[card(View.proc)], View)
```

Fig. 5. Wait free emulation with $2n - 1$ MWMR registers for n processes

7 Concluding Remarks

We have seen how n processes emulate SWMR registers non blocking using n MWMR registers and wait free using $2n - 1$ MWMR registers. What if we have M processes $M >> n$ (M possibly infinite) but we have the notion of arrivals and departures of processes? Processes have to solve the task only under the condition that the number of processes that invoked the task (arrived) minus the number of processes that obtain an output (departed) is at any point of time less or equal to n. Suppose that we know that a task T is read-write non blocking (resp. wait free) solvable in the SWMR model under the assumption of n-concurrency. Can we solve T with just n (resp. $2n - 1$) MWMR registers, without indefinite postponement, i.e. the step complexity of a process until it gets an output will be, like in the SWMR model, a function of n rather than M?

We observe that the non blocking simulation is in fact n-concurrent. Thus, for the non blocking case, the answer is positive. Concerning the wait free simulation it is more tricky, but we conjecture that the answer is positive too.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. Journal of the ACM 40(4), 873–890 (1993)
2. Afek, Y., De Levie, Y.: Efficient adaptive collect algorithms. Distributed Computing 20, 221–238 (2007)
3. Afek, Y., Stupp, G., Touitou, D.: Long-lived adaptive collect with applications. In: Proceedings of FOCS 1999, pp. 262–272. IEEE (1999)
4. Aguilera, M.K.: A pleasant stroll through the land of infinitely many creatures. SIGACT News 35(2), 36–59 (2004)
5. Aspnes, J., Attiya, H., Censor-Hillel, K.: Polylogarithmic concurrent data structures from monotone circuits. J. ACM 59(1), 2:1–2:24 (2012)
6. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. Journal of the ACM 37(3), 524–548 (1990)
7. Attiya, H., Fouren, A.: Polynomial and Adaptive Long-Lived ($2k$ - 1)-Renaming. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 149–163. Springer, Heidelberg (2000)

8. Attiya, H., Fouren, A.: Adaptive and efficient algorithms for lattice agreement and renaming. SIAM J. Comput. 31(2), 642–664 (2002)
9. Attiya, H., Fouren, A.: Algorithms adapting to point contention. J. ACM 50(4), 444–468 (2003)
10. Attiya, H., Fouren, A., Gafni, E.: An adaptive collect algorithm with applications. Distributed Computing 15(2), 87–96 (2002)
11. Attiya, H., Welch, J.: Distributed Computing. Fundamentals, Simulations, and Advanced Topics. John Wiley & Sons (2004)
12. Borowsky, E., Gafni, E.: Immediate atomic snapshots and fast renaming. In: PODC, pp. 41–51. ACM Press (1993)
13. Burns, J.E., Lynch, N.A.: Bounds on shared memory for mutual exclusion. Inf. Comput. 107(2), 171–184 (1993)
14. Castañeda, A., Rajsbaum, S.: New combinatorial topology bounds for renaming: the lower bound. Distributed Computing 22(5-6), 287–301 (2010)
15. Delporte-Gallet, C., Fauconnier, H., Gafni, E., Rajsbaum, S.: Linear space bootstrap communication scheme. Technical Report hal-00717235, LIAFA, Université Paris 7-Denis Diderot, France (2012)
16. Fatourou, P., Fich, F., Ruppert, E.: Space-optimal multi-writer snapshot objects are slow. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, PODC 2002, pp. 13–20. ACM, New York (2002)
17. Fatourou, P., Fich, F.E., Ruppert, E.: Time-space tradeoffs for implementations of snapshots. In: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing, STOC 2006, pp. 169–178. ACM, New York (2006)
18. Fich, F., Herlihy, M., Shavit, N.: On the space complexity of randomized synchronization. J. ACM 45(5), 843–862 (1998)
19. Helmi, M., Higham, L., Pacheco, E., Woelfel, P.: The space complexity of long-lived and one-shot timestamp implementations. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2011, pp. 139–148. ACM, New York (2011)
20. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 123–149 (1991)
21. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM 46(2), 858–923 (1999)
22. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
23. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
24. Jayanti, P., Tan, K., Toueg, S.: Time and space lower bounds for nonblocking implementations. SIAM J. Comput. 30(2), 438–456 (2000)
25. Lamport, L.: On interprocess communication; part I and II. Distributed Computing 1(2), 77–101 (1986)
26. Moir, M.: Fast, long-lived renaming improved and simplified. Sci. Comput. Program. 30(3), 287–308 (1998)
27. Styer, E., Peterson, G.L.: Tight bounds for shared memory symmetric mutual exclusion problems. In: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, PODC 1989, pp. 177–191. ACM, New York (1989)

An Analysis Framework for Distributed Hierarchical Directories

Gokarna Sharma and Costas Busch

School of Electrical Engineering and Computer Science, Louisiana State University
Baton Rouge, LA 70803, USA
{gokarna,busch}@csc.lsu.edu

Abstract. We provide a novel analysis framework for distributed hierarchical directories for an arbitrary set of dynamic (online) requests. We prove a general $\mathcal{O}(\eta \cdot \varphi \cdot \sigma^3 \cdot h)$ competitive ratio for any distributed hierarchical directory, where η is a write set size related parameter, φ and σ are stretch and growth related parameters, and h is the number of levels in the hierarchy. Through this framework, we give bounds for several known distributed directory protocols. In general network topologies, we obtain $\mathcal{O}(\log^2 n \cdot \log D)$ competitive ratio, where n and D are the number of nodes and the diameter, respectively, of the network. Moreover, we obtain $\mathcal{O}(\log D)$ competitive ratio in constant-doubling metric topologies. To the best of our knowledge, this is the first (competitive) dynamic analysis for distributed hierarchical directories.

1 Introduction

Distributed hierarchical directories are data structures that enable one to access shared objects whenever needed. These directories are used to implement fundamental coordination problems in distributed systems, including distributed transactional memory [9,13], distributed queues [5], and mobile object tracking [4]. These directories support access to the shared objects in a network through three basic operations : (i) *publish*, allowing a shared object to be inserted in the directory so that other nodes can find it; (ii) *lookup*, providing a read-only copy of the object to the requesting node; and (iii) *move*, allowing the requesting node to write the object locally after the node gets it.

The hierarchical structure is constructed based on some well-known clustering techniques (e.g., sparse covers, maximal independent sets) which organize the nodes in multiple level clusters and the cluster sizes grow exponentially towards the root level. Hierarchical directories provide a better approach than pre-selected spanning tree based implementations [2,5,12,14] which do not scale well, since the stretch of the spanning tree can be as much as the diameter of the network, e.g. in ring networks.

The three basic operations of distributed directories are applied in different coordination problems. In *distributed transactional memory* [2,9,13,14], each transaction accesses several shared objects for read or write. A shared object ξ originating at some node u is inserted in the directory through a *publish* operation; a *lookup* operation facilitates a requesting node v to locate ξ for read; a *move* operation allows v to move ξ explicitly to its cache for write. A *distributed queue* [5,10] is implemented through a shared object ξ which is the head of the queue. Initially the object (the head) is inserted

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 378–392, 2013.

through a *publish* operation; enqueue requests are realized through *move* operations which insert new requests at the tail of the queue. In *mobile object (user) tracking* [3,4], given a mobile object ξ which is being observed by the network, a search request for the location of the object is implemented through a *lookup* operation. When the object is moved, the hierarchy is updated through a *move* operation issued at its new location.

We present a novel analysis framework for distributed hierarchical directories for an arbitrary set of dynamic (online) requests. In our analysis, the goal is to minimize the *total communication cost* for the request set. Previous dynamic analysis approaches were only for spanning tree based implementations Arrow [10] and Relay [15], and they can not be directly extended to analyze hierarchical directories. To the best of our knowledge, ours is the first formal dynamic performance analysis of distributed hierarchical directories which are designed to implement a large class of fundamental coordination problems in distributed systems.

In order to analyze distributed hierarchical directories, we model the network as a weighted graph, where graph nodes correspond to processors and graph edges correspond to communication links between processors. The network nodes are organized into $h + 1$ levels. In every level, we select a set of leader nodes; higher level leaders coarsen the lower level set of leaders. At the bottom level (level 0) each node is a leader, while in the top level (level h) there is a single special leader node called the $root$.

We consider an execution of an arbitrary set of dynamic (online) requests, e.g. *publish*, *lookup*, and *move*, which arrive at arbitrary moments of time at any (bottom level) node. We bound the *competitive ratio* (i.e., *stretch*), which is the ratio of the total communication cost (measured with respect to the edge weights) of serving the set of dynamic requests in the hierarchy to the optimal communication cost of serving them through some shortest path in the original network.

We prove $\mathcal{O}(\eta \cdot \varphi \cdot \sigma^3 \cdot h)$ competitive ratio for any distributed hierarchical directory for any arbitrary set of (online) *move* requests in dynamic executions, where η is a write size related parameter, φ is a stretch related parameter, and σ is a growth related parameter on the hierarchy, respectively. A node u in each level k has a *write set* of leaders which helps to implement the *move* requests. The parameter η expresses what is the maximum size of the write set of leaders of the node u among all the levels in the hierarchy; the parameter φ expresses how far the leaders in the write set of u can appear beyond a minimum radius around u; and the parameter σ expresses the minimum radius growth ratio. It seems that the linear dependency on η cannot be avoided because possible shortening of the route using sub-linear number of write sets may introduce race conditions in the concurrent executions as noticed in [9,13]. We focused only on *move* requests since they are the most costly operations. (The cost due to a *publish* operation is the fixed initial cost which is compensated by the *move* and *lookup* operations issued thereafter, and the *lookup* operations have always small cost even when considered individually.) Further, we consider only one shared object as in [10,15].

We apply our framework to analyze three variants of distributed hierarchical directory-based protocols, Spiral [13], Ballistic [9], and Awerbuch and Peleg's tracking a mobile user [3,4] (hereafter AP-algorithm), and we obtain the following results.

– Spiral: $\mathcal{O}(\log^2 n \cdot \log D)$ competitive ratio in general networks, where n is the number of nodes and D is the diameter, respectively, of the network. Spiral is

designed for the *data-flow* distributed implementation of software transactional memory in large-scale distributed systems, where transactions are immobile (running at some particular node) and shared objects are moved to those nodes that need them [2,5,9,13,15]. In a previous work [13], we have shown that Spiral is $\mathcal{O}(\log^2 n \cdot \log D)$ competitive in both sequential executions which consists of non-overlapping sequence of requests and one-shot concurrent executions where all requests appear simultaneously. Here, we provide the analysis for arbitrary dynamic requests which subsumes these previous bounds.

- AP-algorithm: $\mathcal{O}(\log^2 n \cdot \log D)$ competitive ratio in general networks. The AP-algorithm is appropriate for a general mobile user tracking problem that arises in many applications in the distributed setting, e.g. sensor networks. It has been proven in [4] that the algorithm is $\mathcal{O}(\log n \cdot \log D + \log^2 D/\log n)$ competitive in sequential executions and $\mathcal{O}(\log^2 n \cdot \log D + \log^2 D/\log n)$ competitive in one-shot concurrent executions. Our analysis subsumes these results, since it considers the more general case of arbitrary dynamic executions.
- Ballistic: $\mathcal{O}(\log D)$ competitive ratio in constant-doubling dimension networks. This protocol is also for the *data-flow* distributed implementation of software transactional memory. It is shown in [9] that Ballistic is $\mathcal{O}(\log D)$ competitive in both sequential and one-shot executions. Again, our analysis subsumes these results.

The logarithmic factors in the competitive ratio are mainly due to the properties of the hierarchical clustering techniques used in the protocols. Utilizing improved clustering techniques and/or considering specific networks may result in better factors in the competitive ratio. The general network bounds for Spiral and AP-algorithm are within a poly-log factor far from optimal, in light of the $\Omega(\log n/\log\log n)$ lower bound proved by Alon *et al.* [1] in certain topologies, for the Awerbuch and Peleg's *mobile user tracking* problem [3,4]. To the best of our knowledge, this is the first (competitive) dynamic analysis for distributed hierarchical directories. (This framework also gives competitive ratio for these protocols when requests can execute concurrently.)

Someone may use the spanning tree T of Gupta [7] (constructed by transforming the randomized tree structure of Fakcharoenphol *et al.* [6]), which guarantees that the expected distance in the tree T for every two nodes in the graph is at most $\mathcal{O}(\log n)$ times their distance in the graph, and run the Arrow protocol on T. An expected bound on the stretch can be proved using the dynamic analysis of Herlihy *et al.* [8] for the Arrow protocol on the spanning tree T. However, the (worst-case) stretch for T can be still as large as D. This is because, for example, in ring networks, the minimum distance is 1 and the maximum distance is $n/2$ between every two nodes in the graph. This results in a competitive ratio of $(D \cdot \log D)$, which is significantly larger than the polylogarithmic competitive ratio of our solution. That is, our solution yields good behavior every time for any arbitrary set of dynamic requests, whereas this solution yields good behavior only in the expected case.

Our analysis framework captures both the time and the distance restrictions in ordering dynamic requests through a notion of *time windows*. For obtaining an upper bound, we consider a synchronous execution where time is divided into windows of appropriate duration for each level. For obtaining a lower bound, given an optimal ordering of the requests, we consider the communication cost provided by a Hamiltonian path that

visits each request node exactly once according to their order. The lower bound holds for any asynchronous execution of the requests. We perform the analysis level by level. The time window notion combined with a Hamiltonian path allows to analyze the competitive ratio for the requests that reach some level. After combining the competitive ratio of all the levels, we obtain the overall competitive ratio.

Related Work. There have been endeavors analyzing the dynamic performance of distributed protocols that are based on pre-selected spanning trees. An analysis of the Arrow protocol [5] given in [8,10] for an arbitrary set of (online) ordering requests generated over a period of time shows that Arrow is $\mathcal{O}(s \cdot \log D)$-competitive, where s and D, respectively, are the stretch and the diameter of the spanning tree on which Arrow operates. Note that s can be as large as D, as for example, in ring networks, giving a competitive ratio $\mathcal{O}(D \cdot \log D)$, which is significantly larger than ours. The Arrow protocol, originally developed for distributed mutual exclusion [12], is one of the simplest distributed directory protocol based on spanning trees. Along the lines of Arrow, an analysis of the Relay protocol [14] is presented in [15], for dynamic (online) *move* requests in the context of distributed transactional memory, and shown that Relay is $\mathcal{O}(s \cdot \log D)$-competitive, for a set of transactions that request the same object.

Outline of the Paper. The rest of the paper is organized as follows. We give a generic distributed hierarchical directory algorithm in Section 2. In Section 3, we present a novel dynamic analysis framework based on time windows. We analyze the generic algorithm of Section 2 in Section 4. Through the framework, we analyze Spiral, Ballistic, and AP-algorithm in Section 5. Proofs are omitted from the paper due to space limitations.

2 An Online Algorithm

Network Model. We model a distributed network as a weighted graph $G = (V, E, \mathfrak{w})$, where the nodes V represent network machines, $|V| = n$, the edges E represent interconnection links between machines, and the weight function $\mathfrak{w} : E \to \mathbb{R}^+$ corresponds to the latency of communication links. The weight of a link is equal to the communication cost of sending a message over the link. We assume that $\mathfrak{w}(u, u) = 0$ for any $u \in V$. We take G to be connected, i.e., there is a path of nodes (with respective sequence of edges connecting the nodes) between any pair of nodes in G. Let $\mathrm{dist}(u, v)$ be the shortest path length (distance) between nodes u and v, with respect to the weight function $\mathfrak{w}$. The k-neighborhood of a node v is the set of nodes which are within distance at most k from v (including v). The *diameter* is $D = \max_{u,v \in V} \mathrm{dist}(u, v)$, which denotes the maximum shortest path distance over all pairs of nodes in G. We assume that nodes and links do not crash and there is FIFO communication between nodes (i.e. no overtaking of messages occurs).

Hierarchy. Algorithm 1 presents a generic distributed hierarchical directory algorithm. It is based on a hierarchy with h levels of leaders $\mathbf{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ of a network $G = (V, E)$, such that $Z_{k+1} \subseteq Z_k$. In other words, the leaders are partitioned recursively such that, at level 0, each node $v \in V$ is a leader by itself, namely, $Z_0 = V$;

and the highest level Z_h contains a single leader root with leader node r (the root of $\mathbf{Z}$). Communication between leader nodes occurs through shortest paths. We consider a single hierarchical structure per object similar to previous directory protocols [9,13].

Each node $v \in V$ has, at level k, a *write* set of leaders, $Write_k(v) \subseteq Z_k$, and a *read* set of leaders $Read_k(v) \subseteq Z_k$ (Lines 1–6 of Algorithm 1). For convenience, $Write_0(v) = Read_0(v) = v$. The write set of leaders are used to route *move* requests from requesting nodes to their predecessor nodes, and the read set of leaders are used to route *lookup* requests from requesting nodes to the current owner node of the object ξ (we provide details on how this is done in Algorithm 1).

We define the following parameters which will be useful later in the analysis.

- ϕ_k: a maximum radius of the farthest node in $Write_k(v)$ from node v. In other words, $\forall u \in Write_k(v), \mathrm{dist}(v,u) \leq \phi_k$.
- ϕ'_k: a minimum radius such that if two nodes are within distance ϕ'_k, then they must have a common leader in their write sets at level k. In other words, $\forall u, v \in V, \mathrm{dist}(v,u) \leq \phi'_k \Longrightarrow Write_k(u) \cap Write_k(v) \neq \emptyset$.
- φ: the stretch of maximum versus minimum radius in the write set, that is, $\varphi = \max_{0 \leq k \leq h} \frac{\phi_k}{\phi'_k}$. Typically, $\varphi \geq 1$, since $\phi_k \geq \phi'_k$.
- σ: is the minimum radius growth ratio, such that $\phi'_k = \sigma^{k-1}$, for $k > 0$. Typically, $\sigma \geq 2$.
- η: the maximum write set size for any node v in any level of the hierarchy, namely, $\eta = \max_{0 \leq i \leq h, v \in V} |Write_i(v)|$.

Shared Object Operations. Let ξ be a shared object which we want to access through the distributed directory. At any time there is an owner node, denoted $Owner(\xi)$, which holds the object and is allowed to modify it. The directory hierarchy $\mathbf{Z}$ is a data structure that enables one to find and modify the object whenever needed. The directory supports the following three operations.

- $publish(\xi)$: when invoked by node s, it sets s to be the current owner of ξ, i.e., $Owner(\xi) \leftarrow s$. This operation is issued when the object is created at s.
- $lookup(\xi)$: when invoked by vertex v, this operation delivers a search query from v to $Owner(\xi)$, and an object copy is delivered to v without updating the directory.
- $move(\xi)$: when invoked at some vertex v, this operation delivers a move request from v to $Owner(\xi)$, which moves ξ to the new location v, and at the same time the directory is updated so that $Owner(\xi) \leftarrow v$.

We now describe how the algorithm $\mathfrak{A}$ supports *publish*, *lookup*, and *move* in $\mathbf{Z}$. Each leader node t at some level k has a $Pointer_t(\xi)$ pointing towards one of the leaders in level $k-1$ (otherwise it is $\perp$ (null)). A downward chain of pointers will lead to the owner of the object at level 0.

Suppose that some node s issues a $publish(\xi)$ operation. Node s initiates an update of pointer directions from level 1 up to level h such that any downward chain leads to s. At each vertex t in $Write_k(s)$ the pointer is set $Pointer_t(\xi)$ to point toward any leader in $Write_{k-1}(s)$ (Lines 7–9 of Algorithm 1). Note that after the publish the $Pointer_r(\xi)$ at the root will not be $\perp$ thereafter.

In order to implement a $lookup(\xi)$ operation, the requesting node v successively queries the vertices in its read set, $Read(v)$, until hitting a vertex t at level k that has

Algorithm 1. A generic distributed hierarchical directory algorithm for an object ξ

1 **Initialization:**
2 On input graph $G = (V, E)$ build a hierarchy of leaders $\mathbf{Z} = \{Z_0, Z_1, \cdots, Z_h\}$, such that:
3 $Z_{k+1} \subseteq Z_k, 0 \leq k < h$;
4 Every node $v \in V$ at Z_0 is a leader by itself;
5 Z_h consists of a single leader with leader node r (the root of the hierarchy);
6 Each node $v \in V$ has a *write* set of leaders at level k, $Write_k(v) \subseteq Z_k$, and a *read* set $Read_k(v) \subseteq Z_k$ (with $Write_0(v) = Read_0(v) = v$);
7 **Publish object ξ by node s:**
8 **For** *all* layers $1 \leq k \leq h$ and for *all* $t \in Write_k(s)$ **do**:
9 Set downward pointer of t, $Pointer_t(\xi)$, to point towards any leader in $Write_{k-1}(s)$;
10 **Lookup object ξ by node v:**
11 $k \leftarrow 1$;
12 **Until** $Pointer_t(\xi) \neq \perp$ for any $t \in Read_k(v)$ **do**
13 $k \leftarrow k + 1$;
14 Go to $Owner(\xi)$ following the chain of downward pointers, and send a copy of ξ to v;
15 **Move object ξ to node v:**
16 $k \leftarrow 1$;
17 **Until** $Pointer_t(\xi) \neq \perp$ for any $t \in Write_k(v)$ **do**
18 Set $Pointer_t(\xi)$ of all $t \in Write_k(v)$ to point to any leader in $Write_{k-1}(v)$;
19 $k \leftarrow k + 1$;
20 $old \leftarrow Pointer_t(\xi)$ (where $Pointer_t(\xi) \neq \perp$ and $t \in Write_k(v)$);
21 Set $Pointer_t(\xi)$ of all $t \in Write_k(v)$ to point to any leader in $Write_{k-1}(v)$;
22 Go to $Owner(\xi)$ following the chain of downward pointers starting from old, and at the same time set older downward pointers to $\perp$;
23 As soon as $Owner(\xi)$ is reached, move ξ to v (hence, $Owner(\xi) \leftarrow v$);

a non-null pointer $Pointer_t(\xi)$, which leads to the current owner of ξ (Lines 10–14 of Algorithm 1). Therefore, following the chain of downward pointers the owner node can be reached, and a copy of the object can be obtained by v.

The execution of a $move(\xi)$ operation, invoked at some requesting node v, consists of: (i) inserting the pointer $Pointer_v(\xi)$ pointing to any leader in $Write_{i-1}(v)$ for all the leaders $t \in Write_i(v)$ at each level $i < k$ (effectively setting $Owner(\xi) \leftarrow v$ through the new chain of downward pointers) until hitting a vertex t at level k that has a pointer $Pointer_t(\xi)$ leading to the current owner s of ξ; and (ii) deleting $Pointer_t(\xi)$ at all the vertices t in the the chain of downward pointers towards s (Lines 15–23 of Algorithm 1). As soon as the current owner is reached, ξ is moved to v.

In concurrent execution scenarios the operations in the algorithm require coordination to avoid deadlocks or blocking. For example, updates to the pointers of the write set of a node should all occur in an atomic manner. The various instantiations of the generic algorithm that we describe in Section 5 take care of this issue by using different distributed coordination techniques.

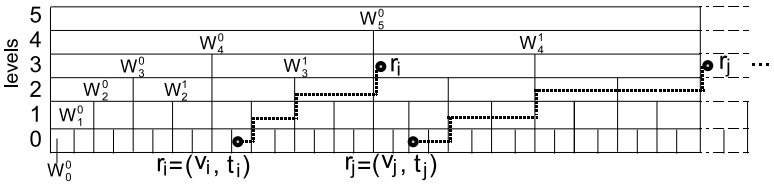


Fig. 1. Illustration of time windows for $\sigma = 2$

3 Analysis Framework

We now proceed with describing the framework to analyze the generic online Algorithm $\mathfrak{A}$ (Algorithm 1) for a set of arbitrary *move* requests. We identify a *move* request r by the tuple $r = (u, t)$ where u is the leaf node in the cluster hierarchy $\mathbf{Z}$ that initiates the *move* request and $t \geq 0$ is the time when the request is initiated. We denote by $\mathbf{R} = \{r_0 = (v_0, t_0), r_1 = (v_1, t_1), \ldots\}$ the arbitrary finite set of dynamic (online) move requests, where the requests $r_i \in \mathbf{R}$ are indexed according to their arrival time, i.e., $i < j \Longrightarrow t_i \leq t_j$.

Since passing the object from one owner to the next can take some time, the effect of Algorithm $\mathfrak{A}$ to the distinct *move* requests is similar to a distributed queue which orders the requests. Each requesting node will eventually receive the object according to the provided order. Suppose that a request $r_1 = (v_1, t_1)$ is ordered by Algorithm 1 after another request $r_2 = (v_2, t_2)$. The ordering of r_1 from a node v_1 is considered complete as soon as v_2 is informed that r_1 is the successor of r_2.

Windows. Time windows is an essential ingredient of our analysis framework. We divide time into fixed duration periods which allows us to obtain upper bounds for the communication cost and also respective lower bounds. At each level k a window represents the time that a node needs to reach and modify the pointers of all the leader nodes in its write set (this is $\mathcal{O}(\eta \cdot \phi_k)$).

In order to define windows we assume a synchronous communication model such that all the network nodes share the same clock. Upon receiving a message, a node is able to perform a local computation and send a message in a single atomic step. We assume that a *time unit* is of duration required for a message sent by a node to reach a destination node that is a unit distance far from it. We make an assumption that message delays respect the triangle inequality. The number of time steps that it takes to deliver the message is equal to the (ceiling rounded integral) distance between the sender and receiver.

We define for level k the *time window* of time duration $W_k = \mathcal{O}(\eta \cdot \phi_k), 1 \leq k \leq h$, and $\mathcal{O}(1)$ for $k = 0$. Assuming an execution starts at time 0, we can have the sequence of windows for each level k, $0 \leq k \leq h$, i.e., $\mathbf{W}_k = \{W_k^0, W_k^1, \ldots\}$, where W_k^0 is the first window at level k, W_k^1 is the second window at level k, and so on. These windows have the property that W_k^{j+1} starts immediately after W_k^j expires. When the notations are clear, we simply denote by W_k one of the windows in $\mathbf{W}_k$.

Hereafter, assume for simplicity the worst value for ϕ_k, namely $\varphi \cdot \sigma^{k-1} = \phi_k$ (this doesn't affect the results of the analysis). We also consider $\sigma \geq 2$, which is the case for the algorithms in Section 5. The windows are aligned in such a way that W_k and one of W_{k-1} start at the same time. For one window at level h, there are σ windows at level $h-1$, σ^2 windows at level $h-2$, and so on, so that there are σ^{h-k} windows at level k. When we consider the windows of all the levels, there are h overlapping aligned windows for one window at the root level as depicted in Fig. 1 for $\sigma = 2$.

The requests that arrive to the system at level 0 are forwarded to level 1 at the end of the window W_0 for level 0. Level 1 leaders forward requests to level 2 at the end of the window W_1. This proceeds at higher levels and in a similar way to the downward direction.

Therefore, at the end of a window, each level k leader node can exchange a message with its leader neighbors at level $k+1$ or level $k-1$. A leader node y_k at level k forwards the request to a leader node y_{k+1} at level $k+1$ at the end of its window W_k (see paths of r_i and r_j in Fig. 1). Similarly, a leader node y_k may forward the request to a leader node y_{k-1} at level $k-1$ at the end of its window W_k. There may be the case that the current window W_{k+1} at level $k+1$ is not yet expired when the window W_k is ready to send the requests at its end. In this case, the requests that need to be sent to level $k+1$ (to level $k-1$) from level k are sent as soon as a new W_{k+1} window (a new W_{k-1} window) starts, respectively (see path of r_j in Fig. 1).

We can show the following basic results on windows. Assume that when in the same window requests update the same pointers then higher priority is given to older requests. In many occasions we will use request r_i to refer to the respective node v_i.

Lemma 1. *Let $r_i = (v_i, t_i) \in \mathbf{R}$ and $r_j = (v_j, t_j) \in \mathbf{R}$ be two requests that reach level k inside respective windows W_k^i and W_k^j, and $j - i - 1 = m$ for some integer $m > 1$. Then the difference in their arrival time is at least $t_j - t_i \geq (m-1) \cdot \eta \cdot \phi_k$.*

Lemma 2. *Suppose $r_i = (v_i, t_i) \in \mathbf{R}$ and $r_j = (v_j, t_j) \in \mathbf{R}, j > i$, are the two requests that reach level k. If they both fall inside the same window W_{k-1}^i at level $k-1$ and also the same window W_k^j at level k, then $\mathrm{dist}(r_i, r_j) \geq \sigma^{k-2}$.*

Lemma 3. *Suppose $r_i = (v_i, t_i) \in \mathbf{R}$ and $r_j = (v_j, t_j) \in \mathbf{R}, j > i$, are the two requests that reach level k. If $\mathrm{dist}(r_i, r_j) < \sigma^{k-1}$ then there must exist some request r_l with arrival time between r_i and r_j, such that either $\mathrm{dist}(r_i, r_l) \geq \sigma^{k-4}$ or $\mathrm{dist}(r_l, r_j) \geq \sigma^{k-4}$.*

4 Analysis of the Online Algorithm

We proceed with necessary definitions. We denote by S_k^j the total count of the number of requests that reach level k inside some window W_k^j. We call the level k windows which have $S_k^j \geq 3$ the *dense windows* and the rest of the level k windows (which have $S_k^j < 3$) the *sparse windows*. The reason behind considering the windows with $S_k^j \geq 3$ and $S_k^j < 3$ separately is that we need always at least $\lceil S_k^j/2 \rceil \geq 2$ requests inside any window that are at least σ^{k-2} far from each other in the graph G (as implied by Lemma

2). This will help to establish a non-trivial lower bound in the communication cost for ordering all the requests in $\mathbf{R}$ that reach level k. For $S_k^j < 3$ windows (i.e. sparse windows), the goal is to transform them into the case of dense windows and apply a similar analysis.

We are interested in obtaining bounds for the communication cost measured as the sum of the distances traversed by all messages. We will bound the competitive ratio $CR_{\mathfrak{A}} = \max_{\mathbf{R}} C(\mathbf{R})/C^*(\mathbf{R})$, where $C(\mathbf{R})$ and $C^*(\mathbf{R})$ are the total communication cost and the optimal cost, respectively, of serving all the requests in $\mathbf{R}$ using the online algorithm $\mathfrak{A}$ and the optimal algorithm. For convenience, we analyze the competitive ratio of $\mathfrak{A}$ for the dense windows and the sparse windows separately. Hence, $CR_{\mathfrak{A}} \leq CR_A(\mathbf{R}) + CR_B(\mathbf{R})$, where $CR_A(\mathbf{R})$ is the competitive ratio of $\mathfrak{A}$ for serving all the requests inside dense windows and $CR_B(\mathbf{R})$ is the competitive ratio of $\mathfrak{A}$ for serving all the requests inside sparse windows.

4.1 Dense Windows

In this section, we analyze the total communication cost $C_A(\mathbf{R})$ and the optimal cost $C_A^*(\mathbf{R})$ for dense windows, and bound the competitive ratio $CR_A(\mathbf{R}) = C_A(\mathbf{R})/C_A^*(\mathbf{R})$. We will first focus on a single dense window W_k^j (i.e., a window with $S_k^j \geq 3$). We give bounds for the total and the optimal communication cost for W_k^j which will be useful when we later analyze the performance for all the dense windows in $\mathbf{W}_k$.

We denote by $C_A(W_k^j(\mathbf{R}))$ the total communication cost of serving requests that reach level k inside a dense window W_k^j by the online algorithm $\mathfrak{A}$, and by $C_A^*(W_k^j(\mathbf{R}))$ the respective optimal communication cost. Note that, for simplicity, we consider only the cost incurred by the up phase of each *move* request. When we consider the down phase of each request the cost increases by a factor of 2 only. We can prove the following lemmas (using Lemma 2).

Lemma 4. $C_A(W_k^j(\mathbf{R})) \leq 2 \cdot S_k^j \cdot \eta \cdot \phi_k$.

Lemma 5. $C_A^*(W_k^j(\mathbf{R})) \geq \lceil (S_k^j - 1)/2 \rceil \cdot \sigma^{k-2}$.

Among all the dense windows $\mathbf{W}_k$ for level k, we define a subsequence of dense windows $\mathfrak{W}_k^\alpha = \{W_k^\alpha, W_k^{\alpha+\lambda_d}, W_k^{\alpha+2\lambda_d}, \cdots\} \subset \mathbf{W}_k$ such that $\alpha \in \{0, 1, \cdots, \lambda_d - 1\}$ for $\lambda_d = 3$. Thus, there will be λ_d *dense subsequences* in $\mathbf{W}_k$. The intuition behind including every third dense window in a dense subsequence is to guarantee that all the requests in window $W_k^{\alpha+i\lambda_d}$ are arrive in the system at least $\eta \cdot \phi_k$ time before any request in window $W_k^{\alpha+(i+1)\lambda_d}, i \geq 0$, arrives in the system (Lemma 1). We prove the following lemma.

Lemma 6. *For any two requests* $r_a = (v_a, t_a) \in W_k^{\alpha+j\lambda_d}$, $r_b = (v_b, t_b) \in W_k^{\alpha+(j+1)\lambda_d}$, $j \geq 0$, *in the dense subsequence* $\mathfrak{W}_k^\alpha$, $t_b - t_a \geq \eta \cdot \phi_k$.

We proceed with giving an upper bound in the total communication cost $C_A(\mathfrak{W}_k^\alpha(\mathbf{R}))$ for all the requests in the dense subsequence $\mathfrak{W}_k^\alpha$. We fix $\mathfrak{S}_k^\alpha = \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} S_k^i$, the total number of requests inside all the windows of the dense subsequence $\mathfrak{W}_k^\alpha$, where $|\mathfrak{W}_k^\alpha|$ is the total number of windows in $\mathfrak{W}_k^\alpha$. The following result follows from Lemma 4.

Lemma 7. *For the requests in a dense subsequence* $\mathfrak{W}_k^\alpha, C_A(\mathfrak{W}_k^\alpha(\mathbf{R})) \leq 2 \cdot \mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k$.

We now bound the optimal cost $C_A^*(\mathfrak{W}_k^\alpha(\mathbf{R}))$ for all the requests in the dense subsequence $\mathfrak{W}_k^\alpha$. The main idea here is to show that $C_A^*(\mathfrak{W}_k^\alpha(\mathbf{R}))$ is at least the cost due to a minimum cost Hamiltonian path that visits each vertex of $\mathfrak{W}_k^\alpha(\mathbf{R})$ exactly once. On our way, we use a notion of directed dependency graph. Note that the lower bound is for any asynchronous execution for the involved requests.

We start with necessary definitions. Let $\mathfrak{R} = \{r_1, r_2, \cdots\} \subset \mathbf{R}$ denote a subset of requests in $\mathbf{R}$. The *directed dependency graph* $H(\mathfrak{R}) = (V', E', \mathfrak{w}')$ has requests as vertices V', where $|V'| = |\mathfrak{R}|$, a directed edge from any vertex $u' \in V'$ to any other vertex $v' \in V'$ such that $(u', v') \in E'$ and $(v', u') \in E'$, and edge weight function $\mathfrak{w}' : E' \rightarrow \mathbb{R}^+$. Note that $H(\mathfrak{R})$ is a *directed complete graph* − there are two directed edges between every pair of vertices. The directed edge weights in $H(\mathfrak{R})$ are assigned as given below:

$$\forall i, j, \mathfrak{w}'(v_i, v_j) = \max\{\text{dist}(v_i, v_j), t_i - t_j\}.$$

Note that $\mathfrak{w}'(v_i, v_j)$ may be different than $\mathfrak{w}'(v_j, v_i)$. We argue that the time in $\mathfrak{w}'(v_i, v_j)$ translates to the communication cost as there is always a request that is searching for the predecessor node as soon as it is initiated in the system until it is ordered behind the predecessor.

Each possible ordering for any algorithm for the requests in $\mathfrak{R}$ is given by a *directed Hamiltonian path*, that visits each vertex exactly once, on the graph $H(\mathfrak{R})$. Out of the possible orderings, the order which minimizes the ordering cost will be the *lowest cost directed Hamiltonian path*. Since the graph $H(\mathfrak{R})$ is a directed complete graph, there is always a Hamiltonian path. An example of a Hamiltonian path is given in Fig. 2 for $\mathfrak{W}_k^\alpha$ with $|\mathfrak{W}_k^\alpha| = 4$, where N_s is the starting node and N_t is the ending node.

Observation 1. *The optimal communication cost* $C^*(\mathfrak{R})$ *for the requests* $\mathfrak{R}$ *is at least the* lowest cost directed Hamiltonian path *in the graph* $H(\mathfrak{R})$.

We now consider the directed dependency graph $H(\mathfrak{W}_k^\alpha(\mathbf{R}))$ for all the requests in the dense subsequence $\mathfrak{W}_k^\alpha$. We divide vertices in $H(\mathfrak{W}_k^\alpha(\mathbf{R}))$ into $|\mathfrak{W}_k^\alpha|$ groups, denoted as $H_i, 1 \leq i \leq |\mathfrak{W}_k^\alpha|$, such that H_i corresponds to a window $W_k^i \in \mathfrak{W}_k^\alpha$, where $|\mathfrak{W}_k^\alpha|$ is the total number of windows in the dense subsequence $\mathfrak{W}_k^\alpha$. In other words, a group constitutes a dense window in $\mathfrak{W}_k^\alpha$. We order the groups H_i from left to right. If we look at a particular group H_i, there are some directed edges between vertices inside H_i, some directed edges going out to the groups in both sides (left and right of H_i), and some directed edge coming into H_i from the groups in its both sides (see Fig. 2). We focus on a subgraph $H^{sub}(\mathfrak{W}_k^\alpha(\mathbf{R}))$ of the graph $H(\mathfrak{W}_k^\alpha(\mathbf{R}))$ such that, for any two vertices $u, v \in H_i$, $\text{dist}(u, v) \geq \sigma^{k-2}$. As argued in Lemma 5, there will be at least $\lceil S_k^i/2 \rceil$ vertices in each group H_i after removing such vertices. Denote by P some directed Hamiltonian path on $H^{sub}(\mathfrak{W}_k^\alpha(\mathbf{R}))$ (see Fig. 2) and by P^* the lowest cost directed Hamiltonian path among all P. We can make the following observations (with the help of Lemma 1).

Observation 2. *For any two requests* $r_a = (v_a, t_a) \in H_i$ *and* $r_b = (v_b, t_b) \in H_i$, $\mathfrak{w}'(v_a, v_b) = \mathfrak{w}'(v_b, v_a) \geq \text{dist}(v_a, v_b) \geq \sigma^{k-1}$.

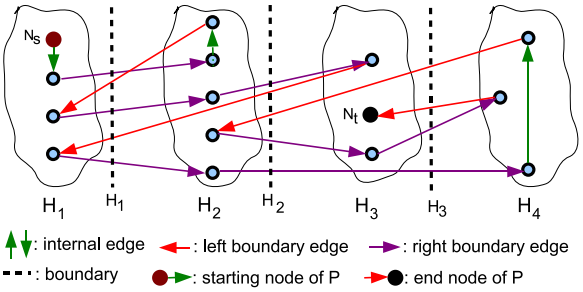


Fig. 2. Illustration of a Hamiltonian path P starting from the node $N_s \in H_1$ and ending in the node $N_t \in H_3$ for the dense subsequence $\mathfrak{W}_k^\alpha$ with $|\mathfrak{W}_k^\alpha| = 4$. The left boundary edges of a group H_3 are $\overline{E}_3^{b,left} = 2$ and the right boundary edges of H_3 are $\overline{E}_3^{b,right} = 2$. Moreover, the left external edges of H_3 are $E_3^{ext,left} = 1$ and the right external edges of H_3 are $E_3^{ext,right} = 1$.

Observation 3. *For any two requests* $r_a = (v_a, t_a) \in H_i$ *and* $r_b = (v_b, t_b) \in H_j$, $j > i$, $\mathfrak{w}'(v_a, v_b) \geq 0$ *and* $\mathfrak{w}'(v_b, v_a) \geq (j - i) \cdot \eta \cdot \phi_k$.

In each group H_i, there are two types of edges, internal and external. The *internal* edges E_i^{int} are all the edges (u', v') from any vertex $u' \in H_i$ to any other vertex $v' \in H_i$. The *external* edges E_i^{ext} are all the edges (u', v') from any vertex $u' \in H_i$ to any other vertex $v' \in H_j, j \neq i$. Moreover, the external edges E_i^{ext} of H_i are of two types, that go to the groups on the left ($H_{<i}$), which we denote by $E_i^{ext,left}$ (the *left external* edges), and that go to the groups on the right ($H_{>i}$), which we denote by $E_i^{ext,right}$ (the *right external* edges). We have that $E_i^{ext} = E_i^{ext,left} + E_i^{ext,right}$.

We define the *boundary* of H_i as a dotted vertical line on its right (see Fig. 2) which shows the interaction between $H_{>i}$ and $H_{\leq i}$. Consider a Hamiltonian path P on $H^{sub}(\mathfrak{W}_k^\alpha(\mathbf{R}))$. We define the *boundary edges* as follows (see Fig. 2). For H_i, let $\overline{E}_i^{b,right}$ (the *right boundary* edges) be the set of edges (u', v') in P which satisfy the condition that $u' \in H_{\leq i}$ and $v' \in H_{>i}$. All the right boundary edges $\overline{E}_i^{b,right}$ will cross the boundary of H_i and point to right groups. Similarly, let $\overline{E}_i^{b,left}$ (the *left boundary* edges) be the set of edges (u', v') in P which satisfy the condition that $u' \in H_{>i}$ and $v' \in H_{\leq i}$. All the left boundary edges $\overline{E}_i^{b,left}$ will cross the boundary of H_i and point to it or the groups on the left of it. We can prove the following relation between $\overline{E}_i^{b,left}$ and $E_i^{ext,left}$ for any group H_i.

Lemma 8. $|\overline{E}_i^{b,left}| \geq |E_i^{ext,right}| - 1$ *for each group* H_i.

The following observation is straightforward.

Observation 4. $\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}|$.

In a directed Hamiltonian path P due to the optimal algorithm, some edges (u, v) are between the vertices of a particular group $H_i, 1 \leq i \leq |\mathfrak{W}_k^i|$ (denoted P_{int}), and

some are between the vertices of groups H_i and $H_j, j \neq i$ (denoted P_{ext}). Thus, the Hamiltonian path $P = P_{int} \cup P_{ext}$ (union of the edges from both groups). Denote by

$$\begin{aligned} C(P) &= C(P_{int}) + C(P_{ext}) \\ &= C(P_{int}) + C(P_{ext,left}) + C(P_{ext,right}) \end{aligned}$$

the total cost of any Hamiltonian path P, where $P_{ext,left}$ is the edges in P due to left external edges $E_i^{ext,left}, 1 \leq i \leq |\mathfrak{W}_k^\alpha|$ and $P_{ext,right}$ is the edges in P due to right external edges $E_i^{ext,left}, 1 \leq i \leq |\mathfrak{W}_k^\alpha|$. We now put bounds on the minimum cost by any Hamiltonian path P. The following observation is straightforward from the way we defined $H^{sub}(\mathfrak{W}_k^\alpha(\mathbf{R}))$.

Observation 5. $C(P_{int}) \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \cdot \sigma^{k-2}$.

Lemma 9. $C(P_{ext,left}) \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \cdot \eta \cdot \phi_k$.

We are now ready to prove the lower bound $C_A^*(\mathfrak{W}_k^\alpha(\mathbf{R}))$ for the requests in the dense subsequence $\mathfrak{W}_k^\alpha$.

Lemma 10. *For the requests in a dense subsequence* $\mathfrak{W}_k^\alpha, C_A^*(\mathfrak{W}_k^\alpha(\mathbf{R})) \geq \frac{1}{8} \cdot \mathfrak{S}_k^\alpha \cdot \sigma^{k-2}$.

We now bound the total communication cost $C_A(\mathbf{R})$ of the online algorithm $\mathfrak{A}$ and the optimal communication cost $C_A^*(\mathbf{R})$ for serving all the requests in $\mathbf{R}$ that are inside dense windows.

Lemma 11. *For the execution* $\mathbf{R}$*, the total communication cost of the online algorithm* $\mathfrak{A}$ *for all the requests inside the dense windows (of all the levels) is* $C_A(\mathbf{R}) \leq 2 \cdot \sum_{k=1}^{h} \sum_{\alpha=0}^{\lambda_d - 1} (\mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k)$.

Lemma 12. *For the execution* $\mathbf{R}$*, the optimal communication cost for all the requests inside dense windows (of all the levels) is* $C_A^*(\mathbf{R}) \geq \frac{1}{8} \cdot \max_{1 \leq k \leq h} \cdot \max_\alpha \left(\mathfrak{S}_k^\alpha \cdot \sigma^{k-2}\right)$.

We are now ready to put bounds on the competitive ratio $CR_A(\mathbf{R})$ for the dense windows.

Theorem 1. $CR_A(\mathbf{R}) = \mathcal{O}(\eta \cdot \varphi \cdot \sigma \cdot h)$.

4.2 Sparse Windows

In this section, we analyze the total communication cost $C_B(\mathbf{R})$ and the optimal communication cost $C_B^*(\mathbf{R})$ for serving requests inside sparse windows, and bound the competitive ratio $CR_B(\mathbf{R}) = C_B(\mathbf{R})/C_B^*(\mathbf{R})$. Recall that a level k window W_k^j is sparse if $S_k^j \leq 2$. We consider a subsequence of sparse windows of $\mathbf{W}_k$ (the set of all windows at level k) for the competitive ratio.

Due to $S_k^j \leq 2$ requests inside each sparse window, it may not always be the case that these (at most) 2 requests satisfy the requirements for the lower bound derivation.

Therefore, our goal in the analysis that follows is to transform each sparse window scenario into a dense window case such that there are exactly two requests in each sparse window that are at least σ^{k-4} far in the graph G. Note that in dense windows the distance lower bound was σ^{k-2}; here however it becomes σ^{k-4} because of Lemma 3.

Similar to the subsequences of dense windows, we consider the subsequence of sparse windows $\mathfrak{W}_k^\beta = \{W_k^\beta, W_k^{\beta+\lambda_s}, W_k^{\beta+2\lambda_s}, \cdots\} \subset \mathbf{W}_k$ such that $\beta \in \{0, 1, \cdots, \lambda_s - 1\}$ for $\lambda_s = 3$. Thus, there will be λ_s *sparse subsequences* in $\mathbf{W}_k$. Similar to Lemma 6, for any two requests $r_a = (v_a, t_a) \in W_k^{\beta+j\lambda_d}$ and $r_b = (v_b, t_b) \in W_k^{\beta+(j+1)\lambda_d}$, $j \geq 0$, of $\mathfrak{W}_k^\beta$, $t_b - t_a \geq \eta \cdot \phi_k$.

Next, we will focus on a sparse subsequence $\mathfrak{W}_k^\beta$. We give bounds on the total and the optimal communication cost for all the requests in the sparse subsequence $\mathfrak{W}_k^\beta$ and these results extend to all sparse windows in $\mathbf{W}_k$.

Denote by $\mathcal{P}_k^\beta = \{r_1, r_2, r_3, \cdots\}$ a sequence of requests in the sparse subsequence $\mathfrak{W}_k^\beta$ such that each window $W_k^i \in \mathfrak{W}_k^\beta$ has one request r_i (chosen arbitrarily among the two it contains). In other words, $|\mathcal{P}_k^\beta| = |\mathfrak{W}_k^\beta|$. As $S_k^j \leq 2$, for each window $W_k^j \in \mathfrak{W}_k^\beta$, the total cost computed via $\mathcal{P}_k^\beta$ for $\mathfrak{W}_k^\beta$ will increase by a factor of 2 only.

We define a notion of *request pair* that is useful later in the discussion. A request pair is defined as a set of two consecutive requests in $\mathcal{P}_k^\beta$. $\mathcal{P}_k^\beta$ can be seen as a collection of request pairs $\mathcal{P}_k^\beta = \{(r_1, r_2), (r_2, r_3), (r_3, r_4), \cdots\}$.

Each request pair $(r_a, r_{a+1}) \in \mathcal{P}_k^\beta$ has the property that $t_{a+1} - t_a \geq \eta \cdot \phi_k$, however there may be the case that $\mathrm{dist}(r_a, r_{a+1}) < \sigma^{k-1}$. We define another sequence of request pairs $\widetilde{\mathcal{P}}_k^\beta = \{(r'_1, r''_1), (r'_2, r''_2), (r'_3, r''_3), \cdots\}$ for the sequence of requests in $\mathcal{P}_k^\beta$ using a transformation given below.

i. If $\mathrm{dist}(r_a, r_{a+1}) \geq \sigma^{k-1}$ in the graph G for any two subsequent requests $r_a \in \mathcal{P}_k^\beta$ and $r_{a+1} \in \mathcal{P}_k^\beta$, we fix $r'_a = r_a$ and $r''_a = r_{a+1}$.
ii. if $\mathrm{dist}(r_a, r_{a+1}) < \sigma^{k-1}$ in the graph G for any two subsequent requests $r_a \in \mathcal{P}_k^\beta$ and $r_{a+1} \in \mathcal{P}_k^\beta$, then according to Lemma 3, there exists an ordering request r_c (it can be from the same level k or the lower) after r_a and before r_{a+1} in time such that either $\mathrm{dist}(r_a, r_c) \geq \sigma^{k-4}$ or $\mathrm{dist}(r_c, r_{a+1}) \geq \sigma^{k-4}$. We fix r'_a and r''_a following the criteria given below:
    a. If there is the case that $\mathrm{dist}(r_a, r_c) \geq \sigma^{k-4}$, then we fix $r'_a = r_a$ and $r''_a = r_c$.
    b. If there is the case that $\mathrm{dist}(r_c, r_{a+1}) \geq \sigma^{k-4}$, then we fix $r'_a = r_c$ and $r''_a = r_{a+1}$.

The transformation from $\mathcal{P}_k^\beta$ to $\widetilde{\mathcal{P}}_k^\beta$ guarantees that $\mathrm{dist}(r'_a, r''_a) \geq \sigma^{k-4}$ for any request pair $(r'_a, r''_a) \in \widetilde{\mathcal{P}}_k^\beta$. However, the timing requirement of at least $\eta \cdot \phi_k$ for the any two requests r_1 and r_2 in the subsequent request pairs of $\widetilde{\mathcal{P}}_k^\beta$ may be violated. We satisfy the timing requirement through special sparse subsequences on $\widetilde{\mathcal{P}}_k^\beta$.

The special sparse subsequence $\widehat{\mathcal{P}}_k^\gamma$ has exactly two requests in each window it contains and the requests in subsequent windows satisfy the timing property. Therefore, each request pair in $\widehat{\mathcal{P}}_k^\gamma$ can be treated as a group H_i in the dense window analysis. From this point on, the analysis proceeds similar to the case of dense windows, where

now each pair corresponds to a "dense window" (note that a pair may not actually reside in the same window, but we will assume it does, without affecting correctness, in order to perform the lower bound analysis). Similar to Theorem 1, we can obtain the following theorem for the competitive ratio $CR_B(\mathbf{R})$ for the sparse windows (the term σ^3 comes from using the σ^{k-4} distance in the pairs).

Theorem 2. $CR_B(\mathbf{R}) = \mathcal{O}(\eta \cdot \varphi \cdot \sigma^3 \cdot h)$.

4.3 Complexity of the Online Algorithm

We now prove the main theorem of the analysis. Since the execution $\mathbf{R}$ is arbitrary, we obtain from Theorem 1 of the dense window analysis (Section 4.1) and Theorem 2 of the sparse window analysis (Section 4.2), the competitive ratio of the online algorithm $\mathfrak{A}$ bounded by $CR_{\mathfrak{A}} \leq CR_A(\mathbf{R}) + CR_B(\mathbf{R})$.

Theorem 3. *The competitive ratio of the online algorithm* $\mathfrak{A}$ *is* $CR_{\mathfrak{A}} = \mathcal{O}(\eta \cdot \varphi \cdot \sigma^3 \cdot h)$ *for any arbitrary set of (online) move requests in dynamic executions.*

5 Analysis of Existing Directories

In this section, we analyze three existing distributed hierarchical directory based protocols: Spiral [13], Ballistic [9], and AP-algorithm [3,4].

The Spiral *Protocol:* it uses a hierarchical sparse cover (details in [13]). It has $h + 1 = \mathcal{O}(\log D)$ levels. The sparse cover hierarchy of Spiral can be converted to the hierarchy of leaders $\mathbf{Z}$ by considering only the leader nodes of clusters that include a node. The Spiral hierarchy has the property that $\eta = \mathcal{O}(\log n)$, $\phi_k = \mathcal{O}(2^k \log n)$, and $\phi'_k = 2^{k-1}$ for any level $0 \leq k \leq h$, since $\sigma = 2$. Therefore, $\varphi = \phi_k/\phi'_k = \mathcal{O}(2^k \log n)/2^{k-1} = \mathcal{O}(\log n)$. We note that distributed coordination is achieved by performing the η pointer accesses per level separately at sub-levels according to a labelling of the clusters. Hence, from Theorem 3, we obtain:

Theorem 4. $CR_{\mathsf{Spiral}} = \mathcal{O}(\log^2 n \cdot \log D)$ *in dynamic executions.*

The Ballistic *Protocol:* it uses a sequence of connectivity graphs as a directory hierarchy (details in [9]), obtained using a distributed maximal independent set algorithm (e.g. [11]), on a constant-doubling metric network. It has $h+1 = \mathcal{O}(\log D)$ levels. Due to the use of maximal independent set of leaders, the Ballistic hierarchy directly translates to the hierarchy of leaders $\mathbf{Z}$. Since $\sigma = 2$ in Ballistic, $\eta = \mathcal{O}(1)$, $\phi_k = \mathcal{O}(2^k)$, and $\phi'_k = 2^{k-1}$ for any level $0 \leq k \leq h$. Hence, $\varphi = \mathcal{O}(1)$. Therefore, from Theorem 3 we obtain:

Theorem 5. $CR_{\mathsf{Ballistic}} = \mathcal{O}(\log D)$ *in dynamic executions.*

The AP-algorithm*:* it uses a hierarchical directory composed of a hierarchy of $h = \lceil \log D \rceil + 1$ *regional directories* $RD_i, 1 \leq i \leq h$ (details in [4]). The regional directory construction is based on the concept of *regional matching*. This matching concept is based on a read set $Read(v) \subseteq V$ and a write set $Write(v) \subseteq V$, defined for every vertex v, similar to the one given in Section 2. In AP-algorithm, we have that $\eta = \mathcal{O}(\log n)$ and $\varphi = \mathcal{O}(\log n)$ in dynamic executions, and $\sigma = 2$. Therefore, from Theorem 3, we obtain:

Theorem 6. $CR_{\mathsf{AP-algorithm}} = \mathcal{O}(\log^2 n \cdot \log D)$ *in dynamic executions.*

References

1. Alon, N., Kalai, G., Ricklin, M., Stockmeyer, L.J.: Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. Theor. Comput. Sci. 130(1), 175–201 (1994)
2. Attiya, H., Gramoli, V., Milani, A.: A Provably Starvation-Free Distributed Directory Protocol. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 405–419. Springer, Heidelberg (2010)
3. Awerbuch, B., Peleg, D.: Sparse partitions. In: Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS), vol. 2, pp. 503–513 (1990)
4. Awerbuch, B., Peleg, D.: Concurrent online tracking of mobile users. SIGCOMM Comput. Commun. Rev. 21(4), 221–233 (1991)
5. Demmer, M.J., Herlihy, M.P.: The Arrow Distributed Directory Protocol. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 119–133. Springer, Heidelberg (1998)
6. Fakcharoenphol, J., Rao, S., Talwar, K.: A tight bound on approximating arbitrary metrics by tree metrics. J. Comput. Syst. Sci. 69(3), 485–497 (2004)
7. Gupta, A.: Steiner points in tree metrics don't (really) help. In: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 220–227 (2001)
8. Herlihy, M., Kuhn, F., Tirthapura, S., Wattenhofer, R.: Dynamic analysis of the arrow distributed protocol. Theor. Comp. Sys. 39(6), 875–901 (2006)
9. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)
10. Kuhn, F., Wattenhofer, R.: Dynamic analysis of the arrow distributed protocol. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 294–301 (2004)
11. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. 15(4), 1036–1053 (1986)
12. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst. 7(1), 61–77 (1989)
13. Sharma, G., Busch, C., Srinivasagopalan, S.: Distributed transactional memory for general networks. In: Proceedings of the 2012 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1045–1056 (2012)
14. Zhang, B., Ravindran, B.: Brief Announcement: Relay: A Cache-Coherence Protocol for Distributed Transactional Memory. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 48–53. Springer, Heidelberg (2009)
15. Zhang, B., Ravindran, B.: Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory. In: Proceedings of the 2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–11 (2010)

SMT-Based Model Checking for Stabilizing Programs*,**

Jingshu Chen and Sandeep Kulkarni

Department of Computer Science and Engineering,
Michigan State University, East Lansing, MI, 48824, US

Abstract. We focus on the verification of stabilizing programs using SMT solvers. SMT solvers have the potential to convert the verification problem into a satisfiability problem of a Boolean formula and utilize efficient techniques to determine whether it is satisfiable. We focus on utilizing techniques from bounded model checking to determine whether the given program is stabilizing. We illustrate our approach using three case studies. We also identify tradeoffs between verification with SMT solvers and existing approaches.

Keywords: Verification, Stabilization, Model checking.

1 Introduction

A program is stabilizing if it is guaranteed to recover to a legitimate state even if its initial state is arbitrary. Hence, stabilization is very helpful in providing fault tolerance, especially in coping with the case where the program is perturbed by unpredictable transient faults. For this reason, several distributed algorithms (e.g., leader election, mutual exclusion, routing, spanning tree maintenance etc. [9,10,17]) are designed to be stabilizing.

Although the ability to recover from an arbitrary state is desirable from the perspective of a user of a stabilizing program, it is challenging from the perspective of the designer of that program. Moreover, if stabilization is used to provide assurance, e.g., for ensuring that the program does not stay outside legitimate states forever no matter what the (transient) fault does, it is important to verify this property. Hence, it is valuable to use automated techniques for verifying this property.

One of the successful automated approaches is model checking [13]. Model checking is a technique to automatically verify whether a given model meets a given property. If the program does not meet the given property, the process of model checking typically produces a counterexample. However, using model checking techniques to verify stabilization is exacerbated by the fact that one needs to consider all possible states as opposed to reachable states.

* This work is sponsored in part by AFOSR FA9550-10-1-0178 and NSF CNS Grant 0914913.

** Part of this work is accepted as a brief announcement in SSS 2012.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 393–407, 2013.

Techniques such as symbolic model checking have the potential to mitigate the state space explosion. Examples of such techniques include binary decision diagrams (BDDs) [3], SAT solvers [14,18,24] and SMT solvers [8,12]. These techniques allow one to model programs and specifications in terms of Boolean formulae. Moreover, efficient techniques are developed to manage state space exploration. Finally, the verification property itself is expressed in terms of a Boolean formula.

Previously, model checking of stabilization using BDDs is considered in [4,29]. Specifically, in these approaches, the program and the specification is modeled using Boolean formulae. Subsequently, they utilize SMV [22,23] for verification.

In this paper, we evaluate the effectiveness of SMT solvers in verifying stabilization with the use of bounded model checking [2]. The process of using bounded model checking to verify stabilization consists of two parts, (1) verification of *closure* and (2) verification of *convergence*. Specifically, the former requires that if the program begins in a legitimate state then it remains in legitimate states. And, the latter requires that if the program starts in a state outside its set of legitimate states then it eventually reaches a legitimate state. The proof of closure requirement only requires one to analyze execution of one step of the program. However, for convergence, the number of steps is unknown. We consider the verification of the convergence property with parameter k that identifies the permissible number of steps in the convergence. Specifically, we first encode the program behavior within k (a given parameter) steps into a propositional formula Ψ_v and utilize an SMT solver to determine its satisfiability. Depending upon the answer, one can either determine that the given program is stabilizing or that there exists a program computation where the program does not reach a legitimate state even after executing k program steps. We also utilize a simple cycle detection approach to determine if the given program may not reach a legitimate state.

Contributions of the Paper

- We describe an approach for verifying stabilization with SMT solvers.
- We evaluate the approach with three case studies, namely Dijkstra's token ring [9], Ghosh's Mutual Exclusion [15] and Stabilizing tolerant version of tree-based mutual exclusion algorithm [28]. One of these case studies demonstrates the feasibility of verifying a stabilizing program with infinite state space.
- We argue that SMT solvers are more likely to be effective when one considers synchronous execution of the given program where each process executes (if it can) in each step.
- We evaluate the effectiveness of decomposition and convergence stair [19] in expediting the verification of stabilizing programs. Due to space limitations, we present the details of this approach in [5].

Organization of the Paper. In Section 2, we give the formal definition of the program, state space, computations and stabilization. In Section 3, we present the approach for verifying stabilization with SMT solvers by utilizing techniques

from bounded model checking. Section 4 presents the experimental results for our case studies. Section 5 identifies how program computations in synchronous semantics can be used to reduce the time for verifying stabilization. Section 6 presents the related work. Finally, Section 7 identifies concluding remarks and future work.

2 Preliminaries

In this section, we present the formal definition of programs, state space, computations and stabilization. These definitions are based on previous work in [1,11,16].

Definition 1. ***(Program)*** *A program, $\mathcal{P}$, is described as $\langle V_{\mathcal{P}}, A_{\mathcal{P}} \rangle$, where $V_{\mathcal{P}} = \{v_0, v_1, \ldots, v_n\}$, $n \geq 0$ is a finite set of variables and $A_{\mathcal{P}} = \{a_0, a_1, \ldots, a_m\}$ is a finite set of program actions. Each variable, $v_i \in V_{\mathcal{P}}$, is associated with a finite or infinite domain of values, D_i. Each action, $a_i \in A_{\mathcal{P}}$, is defined as follows: $a_i :: g_i \longrightarrow st_i$; where g_i is a Boolean formula involving program variables and st_i is a statement that updates a subset of program variables.*

For such a program, we define the notion of state, state space and state predicate.

Definition 2. ***(State)*** *A state, s, of program $\mathcal{P}$ is identified by assigning each variable in $V_{\mathcal{P}}$ a value from its respective domain.* □

Definition 3. ***(State space)*** *The state space, $S_{\mathcal{P}}$, of $\mathcal{P}$ is the set of all possible states of $\mathcal{P}$.* □

Definition 4. ***(State predicate)*** *A state predicate of $\mathcal{P}$ is a Boolean expression defined over the program variables $V_{\mathcal{P}}$. Thus, a state predicate C of $\mathcal{P}$ identifies the subset, $S_C \subseteq S_{\mathcal{P}}$, where C is* true *in a state s iff $s \in S_C$.* □

Let $\mathcal{X}$ be a state predicate of program $\mathcal{P}$ and let s be a state of program $\mathcal{P}$. We use the notation $\mathcal{X}(s)$ to denote a predicate that is true iff $\mathcal{X}$ is true in state s, i.e., the boolean expression corresponding to $\mathcal{X}$ evaluates to *true* when it is instatiated with variable values in state s.

Remark. For compactness of the formulae, we do not include the name of the program (e.g., $\mathcal{X}_{\mathcal{P}}$) in specifying its state predicates. In this paper, the program corresponding to the predicate should be clear from the context. We use similar approach for other formulae as well.

Definition 5. ***(Enabled)*** *The action $a_i :: g_i \rightarrow st_i$, is enabled in a state s iff g_i is true in s.* □

Observe that action in a program corresponds to a set of transitions $(s_0,\ s_1)$ where s_0 is the initial state and s_1 is the next state that is obtained by executing the statement of the action that is enabled in s_0. Thus, program transitions are defined as follows:

Definition 6. ***(Transitions)*** *Transitions of $\mathcal{P}$ are defined by the following set:* $\{(s_0, s_1) \mid s_0,\ s_1\ \in S_{\mathcal{P}}, \wedge(\exists a_i \in A_{\mathcal{P}} :: g_i$ *is true in* s_0 *and* s_1 *is obtained by executing* st_i *from* $s_0)\ \} \cup$
$\{(s_0, s_0)\ | s_0 \in S_{\mathcal{P}} \wedge (\forall a_i \in A_{\mathcal{P}} :: g_i$ *is false in state* $s_0)\}$.

Let $\delta_{\mathcal{P}}$ be the set representing transitions of program $\mathcal{P}$. We use the notation $\mathcal{T}(s_0, s_1)$ to denote a predicate that is true iff $(s_0, s_1) \in \delta_{\mathcal{P}}$.

Definition 7. ***(Closed)*** *Let S_c be a state predicate, then S_c is closed in a program $\mathcal{P}$ iff* $(\forall\ (s_0, s_1) : (s_0, s_1) \in \delta_{\mathcal{P}} : (s_0 \in S_c \Rightarrow s_1 \in S_c))$. □

Definition 8. ***(Computation)*** *An infinite sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$ is a computation of $\mathcal{P}$ iff $\forall j : j > 0 : (s_{j-1}, s_j)$, is a transition of $\mathcal{P}$.*

Definition 9. ***(Stabilization)*** *Let $\mathcal{P}$ be a program and let $\mathcal{I}$ be a state predicate of $\mathcal{P}$. We say that $\mathcal{P}$ is stabilizing for $\mathcal{I}$ iff:*

1. *closure: if (s_0, s_1) is a transition of $\mathcal{P}$ and $s_0 \in \mathcal{I}$, then $s_1 \in \mathcal{I}$;*
2. *convergence: every computation of $\mathcal{P}$ reaches $\mathcal{I}$, i.e., $\forall \sigma : \sigma$ is of the form $\langle s_0, s_1, s_2, \ldots\rangle$ and σ is computation of $\mathcal{P} : (\exists j :: s_j \in \mathcal{I})$.* □

Remark. The predicate $\mathcal{I}$ used in Definition 9 is called the legitimate state predicate (invariant) of $\mathcal{P}$.

3 Approach for Verifying Stabilization with SMT Solvers

In this section, we present the approach of verifying self-stabilization properties with SMT solvers by utilizing techniques from bounded model checking. Verification of stabilization consists of two parts: (1) verification of *closure* and (2) verification of *convergence*. In Section 3.1, we identify the formula whose satisfiability can be used to determine whether closure property is satisfied. In Section 3.4, we identify an algorithm for verifying convergence by using the formulae developed in Sections 3.2 and 3.3.

3.1 Verification of Closure

Let $\mathcal{P}$ be the given program and let $\mathcal{I}$ be the legitimate state predicate used in Definition 9 to conclude that $\mathcal{P}$ is stabilizing. Let $\mathcal{T}$ be the predicate that characterizes transitions of $\mathcal{P}$. Observe that the closure property requires that if (s_0, s_1) is a transition of program $\mathcal{P}$ and state s_0 is a legitimate state then state s_1 is also a legitimate state. Thus, this can be captured by formula $\neg\Psi_l$, where

$$\Psi_l \quad = \quad (\mathcal{I}(s_0) \wedge \mathcal{T}(s_0, s_1) \wedge \neg\mathcal{I}(s_1))$$

Remark. For compactness, the formula Ψ_l does not explicitly specify the program or the set of legitimate states that are inputs in deciding closure. In this paper, these two parameters can be determined based on the context. We use similar approach for other formulae as well.

Based on whether Ψ_l is satisfiable or not, we have two scenarios, SC_1 and SC_2:

1. SC_1 : if Ψ_l is satisfiable then it proves that it is possible to begin in a legitimate state, execute a program transition and be in a state that is not a legitimate state. This implies that the closure property is not satisfied. Moreover, in this case, assignment to s_0 and s_1 (which in turn includes values of variables of the program in state s_0 and s_1) provides a counterexample.
2. SC_2 : if Ψ_l is unsatisfiable then this implies that the closure property is satisfied.

3.2 Verification of Convergence

To verify convergence, we use approach from bounded model checking [2]. We verify convergence by checking that starting from an arbitrary state, the program, say $\mathcal{P}$, reaches a legitimate state (in $\mathcal{I}$) in k steps, where k is a given parameter used in the verification. Observe that the convergence property requires us to consider a sequence of states, $s_0, s_1, \cdots, s_k$ such that each successive transitions are program transitions. Moreover, to verify (negation of) convergence requirement, we require that $\mathcal{I}(s_k)$ should be false. Additionally, in this verification, we can utilize the closure requirement to add additional constraints requiring that $\mathcal{I}(s_j)$, $0 \leq j \leq k$, should be false. Additionally, in bounded model checking, one typically adds constraint about what the initial state should be. However, in convergence, the initial state can be arbitrary and, hence, there is no corresponding constraint for the initial state. Thus, the formula used for verifying convergence is as follows:

$$\begin{aligned}\Psi_v = \; & \mathcal{T}(s_0, s_1) \wedge \mathcal{T}(s_1, s_2) \wedge \cdots \wedge \mathcal{T}(s_{k-1}, s_k) \\ & \neg\mathcal{I}(s_0) \wedge \neg\mathcal{I}(s_1) \wedge \cdots \wedge \neg\mathcal{I}(s_k)\end{aligned}$$

Based on whether Ψ_v is satisfiable or not, we have the following two scenarios:

1. SC_3 : if Ψ_v is satisfiable, convergence cannot be achieved in k steps. In this case, the number of steps needs to be increased. If the state space of the program is finite and k equals the number of states in the program then this implies that the convergence property is not satisfied. However, a simple cycle detection algorithm (discussed next) can be used to conclude that the program is not stabilizing for smaller values of k.
2. SC_4 : if Ψ_v is unsatisfiable, then it proves that even if we begin in an arbitrary state, it is impossible for the program to be in an illegitimate state if it executes for k steps. In other words, the convergence property is satisfied.

3.3 Resolving Ambiguity by Cycles Detection

As discussed in Section 3, when Ψ_v is satisfiable, either the given program is not stabilizing or the value of k is too small. To distinguish between these scenarios, we use an approach of resolving ambiguity by checking for an existence of a cycle outside legitimate states. The main idea of this approach is to check whether the given program can run into a cycle that is outside legitimate states.

To check whether the given program can execute a cycle that is outside legitimate states, we consider the behavior of the given program for k steps. Hence, we construct a formula similar to that of Ψ_v. Additionally, the computation created by Ψ_v creates a cycle iff some state is repeated in this path. This can be checked by adding another constraint that two of the states visited are identical. In other words, the added constraint is that there exists two states s_j and s_k, where $j < k$ and $s_j = s_k$.

Note that in case of stabilization, the initial state is arbitrary. Hence, if there exists a cycle where $s_j = s_k$ then there exists a suffix of the given computation that begins with s_j. In other words, it suffices to check whether state s_0 is repeated in the given computation. Hence, the formula used for detecting cycle is as follows:

$$\Psi_y = \ \Psi_v \wedge (s_0 \cong s_1) \vee (s_0 \cong s_2) \vee \cdots \vee (s_0 \cong s_k)$$

In the above formula, we use $(s_0 \cong s_1)$. One implementation of this is to require $s_0 = s_1$. However, in certain cases, it may be sufficient to reach a state that is *similar* to the initial state. Such similarity is application dependent. One example of this is based on symmetry of processes and values.

Based on whether Ψ_y is satisfiable or not, we have two scenarios: SC_5 and SC_6:

1. SC_5 : if Ψ_y is satisfiable then this implies that there is a computation of the given program that starts in state s_0 and revisits state s_0 without reaching a legitimate state in between. This implies that there is a possibility that the program may never reach a legitimate state. In other words, the given program is not stabilizing.
2. SC_6 : if Ψ_y is unsatisfiable then there are two possibilities: either the given program is stabilizing and, hence, such a cycle cannot exist or the number of steps is insufficient to create a cycle.

3.4 Combining Verification of Convergence and Cycle Detection

Depending upon the satisfiabilty of Ψ_v and Ψ_y, we have four possibilities. Considering these four possibilities, we can determine whether the given program is stabilizing or not. We illustrate this checking process by Algorithm 1. Line 1 first constructs formula $\Psi_v$and Ψ_y. Then, the algorithm utilizes bounded model checking techniques to check the satisfiability of the two formulas in the loop starting from Line 2 and ending at Line 12. If the condition identified in Line 3 is satisfied, the algorithm concludes that the program is stabilizing. If the condition identified in Line 6 is satisfied, the algorithm concludes that the program is not stabilizing. And if the condition identified in Line 9 is satisfied, the algorithm simply increases the value of k and repeat checking the conditions identified in Line 3, 6 and 9.

Algorithm 1. Checking Whether a Program is Stabilizing.

Input: $\mathcal{P}$: program to be verified; $\mathcal{I}$: set of legitimate states.

1: Construct $\Psi_v$and $\Psi_y$for $\mathcal{P}\&\mathcal{I}$.
2: **for** $k = 1 \rightarrow \cdots$ **do**
3: **if** $\Psi_v$is unsatisfiable **then**
4: **print** given program is stabilizing.
5: **end if**
6: **if** $\Psi_y$is satisfiable **then**
7: **print** given program is not stabilizing.
8: **end if**
{Since $\Psi_y \Rightarrow \Psi_l$ it is impossible for $\Psi_y$to be satisfiable and $\Psi_l$to be unsatisfiable.
}
9: **if** $\Psi_v$is satisfiable and $\Psi_y$is unsatisfiable **then**
10: increase the value of k;
11: **end if**
12: **end for**

Note that the above algorithm begins with $k = 1$. However, a better approach is to begin with k to be the expected number of steps for convergence. Also, the algorithm can be tuned in terms of how k is increased. Note that for finite state program, the above program is guaranteed to terminate. For infinite state programs, however, it may not. This is expected given that the halting problem can be trivially reduced to verification of stabilization.

4 Experimental Results

In this section, we present our case studies. These case studies include K-state token ring program [9] and Ghosh's mutual exclusion program [15] and Stabilizing Tree based mutual exclusion [28]. We use the SMT solver Yices [12] to verify the stabilization property. We note that we have also used Z3 [8]. While the exact numbers associated with Z3 are different, the observations and conclusions from this section still hold. Hence, the results for Z3 are not presented in this paper.

4.1 K-State Token Ring Program

In this section, we first present the K-state program. Then, we present the results for verifying the K-state program with an SMT solver. Although the algorithm in Section 3.4 attempts different values of k to decide whether the program is stabilizing, in this section, we only focus on the value of k for which we can conclude that the program is stabilizing (respectively, not stabilizing). This is due to the fact that several heuristics (e.g., analysis of the program to evaluate expected number of steps) can be used to limit the values of k used in Section 3.4. Hence, we only focus on the value of k for which the algorithm terminates.

The token ring program is as follows: The program consists of $N+1$ processes, numbered from 0 to N. Each process $p.i$, $0 \leq i \leq N$, has one variable $x.i$. The domain of $x.i$ is $\{0, 1, \ldots, K-1\}$. These processes are organized in a unidirectional ring. The program consists of two types of actions. The first type is for process 0. This action is enabled when $x.0$ equals $x.N$. When $p.0$ executes its action, it increments $x.0$ by 1 in modulo K arithmetic. The second type of action is for process $p.i$, $i \neq 0$. This action is enabled when $x.i$ is not equal to $x.(i-1)$. When $p.i$ executes its action, it copies $x.(i-1)$. Thus, the actions are as follows:

$$\begin{array}{lllll} K_0:: & x.0 = x.N & \longrightarrow & x.0 = (x.0+1) \ mod \ K; \\ K_i:: & x.i \neq x.(i-1) & \longrightarrow & x.i = x.(i-1); \end{array}$$

Remark 1. We consider three variations of this program: In the first variation, K is set to $N+1$. In the second variation, value of x is unbounded and, hence, K_0 simply increments $x.0$. Finally, we consider the value of $K = 2$ in Section 5.

Legitimate states. The state where x values of all processes is 0 is a legitimate state. In this state, only process 0 is enabled. After process 0 is executed, $x.0$ is assigned 1 and all other x values are still 0. In this state, only process 1 is enabled. Hence, it can be executed and change $x.1$ to 1. Continuing executing the enabled processes further, eventually, we reach a state where all x values are 1 where process 0 is the only enabled process and process 0 will increment $x.0$ to 2. The legitimate states of the K-state program are equal to all the states reached in such subsequent execution.

Performance evaluation. We evaluate the performance of the token ring program for both bounded and unbounded setting. Tables 1 and 2 respectively illustrate the time for verifying the closure and the convergence property for the bounded and unbounded version of the token ring.

As we can observe, the verification time is significantly lower for unbounded version of the token ring. In particular, for the case where x values are unbounded, it is possible to verify the convergence property of a ring with 5 processes in less than a second. However, the corresponding time for program with bounded x value is 214 seconds.

One of the reasons for this is that the bounded version utilizes a modulo operation. One can attempt to revise the token ring program to simplify the mod operation to gain a substantial benefit. Specifically, Table 3 considers the case where action K_0 is split into two actions: The first action executes only if $x.0$ is not equal to $K-1$. And, it increments the value of $x.0$. The second action executes only if $x.0$ equals $K-1$. And, it resets $x.0$ to 0. With this change, the verification time for five processes reduces from 214 seconds to 33 seconds.

4.2 Ghosh's Binary Mutual Exclusion Protocol

Our second case study is Ghosh's binary mutual exclusion protocol [15]. This protocol considers a program of $2m (m \geq 2)$ nodes, numbered from 0 to $2m-1$. The neighbor relation is defined as follows:

Table 1. Verification Time for Ψ_v for Token Ring with Unbounded Variables

Number of nodes	Number of steps for convergence	Execution time(s) for convergence	Execution time(s) for closure
3	4	0.0044	0.003928
4	14	0.01229	0.004257
5	25	0.209468	0.005399
6	39	154.1079279	0.004608

Table 2. Verification Time for Ψ_v for Token Ring with Bounded Variables

Number of nodes	state space	Number of steps for convergence	Execution time(s) for convergence	Execution time(s) for closure
3	10^1	4	0.008944	0.005617
4	10^2	14	0.494496	0.005979
5	10^3	25	214.0957	0.013349

Table 3. Verification Time for Ψ_v for Token Ring with Split Actions for K_0

Number of nodes	State space	Number of steps for convergence	Execution time(s) for convergence	Execution time(s) for closure
3	10^1	4	0.005855	0.005387
4	10^2	14	0.090116	0.006480
5	10^3	25	33.526028	0.006716

- n_0 has one neighbor n_1;
- $n_{2i-1}(1 \leq i \leq m-1)$ has three neighbors n_{2i-2}, n_{2i}, and n_{2i+1};
- $n_{2i}(1 \leq i \leq m-1)$ has three neighbors n_{2i-2}, n_{2i-1}, and n_{2i+1};
- n_{2m-1} has one neighbor n_{2m-2}.

The state s_i of each node n_i can be either 0 or 1. Each node can read its own state and the state of its neighbor nodes. The protocol defines the four types of actions as follows:

$$
\begin{aligned}
&G_{n_0} :: \\
&\quad s_0 \neq s_1 \longrightarrow s_0 = 1 - s_0; \\
&G_{n_{2m-1}} :: \\
&\quad s_{2m-1} = s_{2m-2} \longrightarrow s_{2m-1} = 1 - s_{2m-1}; \\
&G_{n_{2i-1}} (1 \leq i \leq m-1) :: \\
&\quad s_{2i-2} = s_{2i-1} = s_{2i} \ \wedge \ s_{2i-1} \neq s_{2i+1} \longrightarrow s_{2i-1} = 1 - s_{2i-1}; \\
&G_{n_{2i}} (1 \leq i \leq m-1) :: \\
&\quad s_{2i-2} = s_{2i-1} = s_{2i+1} \ \wedge \ s_{2i} \neq s_{2i+1} \longrightarrow s_{2i} = 1 - s_{2i};
\end{aligned}
$$

Performance evaluation. Table 4 gives the performance results of Ghosh's program. In this program, the time for verification is very small for upto 12 processes. After this, the time for verification (along with number of steps necessary for convergence) increases substantially.

Table 4. Verification Results for Ghosh's Program Using SMT Solver

Number of nodes	State space	Number of steps for convergence	Execution time(s) for convergence	Execution time(s) for closure
8	10^2	9	0.040316	0.007433
10	10^3	16	0.470958	0.012232
12	10^3	25	11.166705	0.009348
14	10^4	36	314.851055	0.015544

4.3 Stabilizing Tolerant Version of Tree-Based Mutual Exclusion Algorithm

Our third case study is a stabilizing tolerant version of tree-based mutual exclusion algorithm [28]. In this program, the processes are arranged in a *fixed*[1] tree, called the parent tree. A holder tree is created by assigning directions to each tree edge. We denote the holder variable for process j by $h.j$. Thus, for process j, if $h.j$ is k then this implies that j should request k for the token when it needs it. In turn, if $h.j$ equals j then it implies that j has the token and can access critical section.

In this example, we only focus on actions for correcting the holder relation if it is corrupted to an arbitrary state. We denote neighbors of each process j by $NBR.j$ and parent node of each proces j by $prt.j$. Thus, the program has three actions. The first action ensures that the holder of a process is a tree neighbor. The second action ensures that on any edge between j and $(prt.j)$, either holder of j is same as $prt.j$ or the holder of $prt.j$ is j. And, the third action ensures that holder relation does not have cycles. Thus, the convergence actions of this program are as follows:

$$
\begin{aligned}
&R_1 :: \\
&\quad h.j \neq NBR.j \cup j \\
&\qquad\qquad \longrightarrow h.j = prt.j; \\
&R_2 :: \\
&\quad j \neq prt.j \wedge h.j \neq prt.j \wedge h.(prt.j) \neq j \\
&\qquad\qquad \longrightarrow h.j = prt.j; \\
&R_3 :: \\
&\quad j \neq prt.j \wedge h.j = prt.j \wedge h.(prt.j) = j \\
&\qquad\qquad \longrightarrow h.(prt.j) = prt.(prt.j);
\end{aligned}
$$

Performance evaluation. Table 5 gives the performance results for verifying the stabilizing tolerant version of tree-based mutual exclusion algorithm. In this program, the time for verification is very small for upto 9 processes. After this, the time for verification (along with number of steps necessary for convergence) increases substantially.

[1] By fixed, we mean that the parent of j $(prt.j)$ is fixed and hard coded in the actions themselves and, hence, cannot be corrupted.

Table 5. Verification Results for Stabilizing Tolerant Version of Tree-based Mutual Exclusion Algorithm Using SMT Solver

Number of nodes	State space	Number of steps for convergence	Execution time(s) for convergence	Execution time(s) for closure
3	10^1	5	0.007162	0.005046
7	10^5	8	1.377709	0.004943
9	10^8	9	125.644908	0.005432

5 Verification of Token Ring in Synchronous Semantics

The computation model we considered in Definition 8 corresponds to interleaving semantics where in each step one of the actions is executed. Another computation model uses synchronous semantics. Here, the program actions are partitioned into groups and for every group there is a corresponding *process* responsible for executing those actions. Furthermore, in each step, every process executes one of its enabled actions (unless it has no enabled action in that state). Since the number of steps needed for convergence affects the verification time with SMT solvers significantly, in this section, we consider execution of the program in synchronous semantics and evaluate its effect on verification.

Verification of the program under synchronous semantics can assist in two scenarios: One scenario is that one can verify the stabilization property under synchronous semantics. In this case, the program is guaranteed to reach a legitimate state under synchronous semantics. One can utilize a program that is correct under synchronous semantics and compose it with the alternator in [20]. This alternator ensures that given a program that is stabilizing under synchronous semantics, it transforms into a program that is correct under read/write model. Specifically, in this case, the process in the transformed program either reads the state of its neighbor or writes its own state. In other words, the transformed program guarantees that it will reach a legitimate state even if one process executes at a time, i.e., the transformed program is stabilizing under interleaving semantics. Hence, in this scenario, we can obtain a program that is stabilizing. Another scenario is that one can identify a counterexample (illustrating the lack of stabilization) for the synchronous program. This counterexample can in turn be transformed into a counterexample for the original program.

Assisting Verification by Using Synchronous Semantics. We begin with the first scenario. We consider the execution of the token ring protocol under synchronous semantics. Results from Table 6 show the verification cost under synchronous semantics. These results show that the verification under synchronous semantics is substantially faster for both bounded and unbounded token ring. As discussed above, this can allow us to obtain a stabilizing program that ensures that a legitimate state is reached even if only one process executes at a time.

Table 6. Verification Results for Token Ring under Synchronous Semantics

No. of Nodes	Number of Steps for Convergence	Execution time(s) for Convergence (for Bounded Variables)	Execution time(s) for Convergence (for UnBounded Variables)
3	3	0.006714	0.005945
4	5	0.049422	0.008152
5	7	0.306879	0.012003
6	9	1.150117	0.018864
7	11	11.837923	0.030548
8	13	7.741610	0.049720
9	15	18.389602	0.077605
10	17	42.7424	0.130185
11	19	789.75117	0.241754
12	21	324.921817	0.907359
17	23	N/A	22.63948

Identifying Cyclic Computations by Using Synchronous Semantics. To illustrate the second scenario, we consider the case where the given program works in read/write model where in each step, a process can read the state of its neighbor or write its own state but not both. In such a program, the variables can be partitioned into a public variables (variables that can be read by more than one process) and private variables (variables that can be read by only one process. Thus, in read/write model, the read action corresponds to the case where a process reads public variable(s) of one of its neighbors and saves a copy of it in its private variable(s). In write action, the process utilizes its own public/private variables to update them.

Observe that if $\mathcal{P}$ is a program in read/write model then for each computation of $\mathcal{P}$ in synchronous semantics there is a corresponding computation in interleaving semantics. Intuitively, in the computation in interleaving semantics, one transition of the program in synchronous semantics is split into several steps. Hence, if we can find a counterexample to show that $\mathcal{P}$ is not stabilizing under synchronous semantics then it implies that $\mathcal{P}$ is not stabilizing under interleaving semantics either.

To exploit this observation, we consider the execution of the token ring protocol under read/write model. In this case, $x.j$ is a public variable of process j. The action K_0 in Section 4.1 is not in read/write model since it reads $x.N$ and updates $x.0$. Read/write model requires that these two tasks be separated into one read action (of $x.N$) and one write action (of $x.0$). To obtain the corresponding program in read/write model, we introduce $y.j$ that maintains a copy of the x value of the predecessor. Furthermore, each action is split into a read action to read the value of the predecessor and a write action that utilizes the local copy. Thus, the actions of the token ring protocol in read/write model are as follows:

$$
\begin{array}{lllll}
K_{0_r}:: & y.0 \neq x.N & \longrightarrow & y.0 = x.N; & // \text{ read } x.N \\
K_{0_w}:: & x.0 = y.0 & \longrightarrow & x.0 = (x.0 + 1) \ mod \ K; & \\
K_{j_r}:: & y.j \neq x.(j-1) & \longrightarrow & y.j = x.(j-1); & //\text{read } x.(j-1) \\
K_{j_w}:: & x.j \neq y.j & \longrightarrow & x.j = y.j; &
\end{array}
$$

It is well-known that the above protocol can be thought of as the original token ring protocol with $2(N+1)$ processes, where the variables of these processes are $x.0, y.1, x.1, y.2, \cdots, y.N, x.N, y.0$.

Now, we consider the execution of the token ring protocol with N processes under interleaving semantics. If we choose $K = 2$ this program is not stabilizing. We can identify this by checking that Ψ_y is satisfiable when it is used in the context of the token ring program with N processes using interleaving semantics. Alternatively, the lack of stabilization can also be proved by considering execution of the token ring program with $2N$ processes under synchronous semantics.

With this intuition, we evaluate the time for verifying satisfiability of Ψ_y for different processes. Table 7 shows the verification time with interleaving and synchronous semantics respectively. We observe that for 20 processes under interleaving semantics, it took 40 steps to detect a cycle and the time was 100.605236. However, the same property can also be verified under synchronous semantics with 30 processes in 4 steps and the time was 0.040200. Moreover, as discussed above, the latter verification suffices to conclude that the 20 process token ring program is not stabilizing under interleaving semantics if $K = 2$.

Table 7. Verification Result for Cycle Detection under Interleaving/Synchronous Semantics

	Under Interleaving Semantics		Under Synchronous Semantics	
No. of Nodes	No. of Steps	Execution time(s)	No. of Steps	Execution time(s)
10	20	0.253376	4	0.009663
20	40	100.605236	8	0.048594
30	N/A	N/A	4	0.040200
50	N/A	N/A	4	0.103919
100	N/A	N/A	8	0.849282
200	N/A	N/A	16	9.9811778

6 Related Work

Automated verification of stabilization has been studied both in the context of theorem proving and model checking. In the context of theorem proving existing literature includes [21,25,26]. Specifically, [21,26] focus on using theorem proving K-State token ring protocol using the theorem prover PVS. In [25], authors present an approach for verifying stabilizing programs via theorem proving.

In the context of model checking, in [4,29], the problem has been studied with BDDs. Specifically, in [29] authors focus on modeling stabilizing programs in SMV. In [4], authors present the tradeoff between verification of stabilization with and without fairness.

To the best of our knowledge, ours is the first work that focuses on verification of stabilization with the help of SMT solvers. We utilized yices [12] in presenting results in this paper. Several of these results were also conducted with Z3. Although the results with Z3 are not included in this paper, the observations from Sections 4 and 5 hold true with Z3.

Our approach for verifying stabilization is based on the use of bounded model checking. Bounded model checking has been useful in the context of verifying many programs [6,7,27]. Our work builds upon this existing work.

7 Conclusion

We investigated the effectiveness of SMT solvers in verification of stabilization in this paper. We found that the effectivenss of SMT solvers in this context is mixed. Specifically, compared with existing approaches [4,29] that utilize BDD based model checkers to verify stabilization, the time for verification is larger with SMT solvers. However, BDD based tools require one to identify the order of program variables in the BDD. An incorrect ordering of variables can increase the verification time by orders of magnitude making it significantly worse than the corresponding verification time with SMT solvers. Also, the results in [4,29] apply only for verifying finite state programs. By contrast, the results in this paper demonstrate the feasibility of verifying infinite state program.

We also considered execution of the given program under synchronous semantics. We argued that this has a potential to reduce the cost of verification and utilize a transformation approach to achieve a program that is stabilizing under interleaving semantics and/or read/write model. We showed that execution under synchronous semantics can reduce the time for identifying a counterexample illustrating that the given program is not stabilizing.

References

1. Arora, A., Gouda, M.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering 19(11) (1993)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (1999)
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35(8) (1986)
4. Chen, J., Abujarad, F., Kulkarni, S.: Effect of Fairness in Model Checking of Self-stabilizing Programs. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 135–138. Springer, Heidelberg (2010)
5. Chen, J., Kulkarni, S.: Smt-based model checking for stabilizing programs. Technical Report MSU-CSE-12-13, Computer Science and Engineering, Michigan State University, East Lansing, Michigan (October 2012)

6. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Journal of Form. Methods Syst. Des. (2001)
7. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of Bounded Model Checking at an Industrial Setting. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 436–453. Springer, Heidelberg (2001)
8. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Dijkstra, E.W.: Self stabilizing systems in spite of distributed control. Communications of the ACM 17(11) (1974)
10. Dolev, S.: Self-stabilizing routing and related protocols. Journal of Parallel and Distributed Computing 42(2) (1997)
11. Dolev, S.: Self-Stabilization. MIT Press (2000)
12. Dutertre, B., De Moura, L.: The yices smt solver. Technical report, Computer Science Laboratory, SRI International (2006)
13. Grumberg, O., Clarke, E.M., Peled, D.A.: Model Checking. The MIT Press (2000)
14. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
15. Ghosh, S.: Binary self-stabilization in distributed systems. Information Processing Letter 40(3) (1991)
16. Ghosh, S.: Distributed Systems: An Algorithmic Approach. CRC Press (2006)
17. Ghosh, S., Gupta, A.: An exercise in fault-containment: Self-stabilizing leader election. Information Processing Letters (1996)
18. Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2002 (2002)
19. Gouda, M.G., Multari, N.: Stabilizing communication protocols. IEEE Trans. Comput. 40(4), 448–458 (1991)
20. Kulkarni, S.S., Bolen, C., Oleszkiewicz, J., Robinson, A.: Alternator in read/write model. Information Processing Letters (2005)
21. Kulkarni, S.S., Rushby, J.M., Natarajan, S.: A case-study in component-based mechanical verification of fault-tolerant programs. In: Workshop on Self-stabilizing System, pp. 33–40 (1999)
22. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
23. McMillan, K.L.: The smv system for smv version 2.5.4. Technical report, Carnegie Mellon University (2000)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Proceedings of the 38th Annual Design Automation Conference, DAC 2001 (2001)
25. Prasetya, I.S.W.B.: Mechanically verified self-stabilizing hierarchical algorithms. In: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems, pp. 399–415 (1997)
26. Qadeer, S., Shankar, N.: Verifying a self-stabilizing mutual exclusion algorithm. In: IFIP International Conference on Programming Concepts and Methods, PROCOMET 1998 (1998)
27. Rabinovitz, I., Grumberg, O.: Bounded Model Checking of Concurrent Programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005)
28. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Transactions on Computer Systems (TOCS) 7, 61–77 (1989)
29. Tsuchiya, T., Nagano, S., Paidi, R.B., Kikuno, T.: Symbolic model checking for self-stabilizing algorithms. IEEE Trans. Parallel Distrib. Syst. 12, 81–95 (2001)

Deployment and Evaluation of a Decentralised Runtime for Concurrent Rule-Based Programming Models

Marko Obrovac and Cédric Tedeschi

IRISA. Université de Rennes 1 / INRIA, France
{firstname.lastname}@inria.fr

Abstract. With the emergence of highly heterogeneous, dynamic and large distributed platforms, declarative programming, whose goal is to ease the programmer's task by separating the control from the logic of a computation, has regained a lot of interest recently, as a means of *programming* such platforms. In particular, rule-based programming is regarded as a promising model in this quest for adequate programming abstractions for these platforms. However, while these models are gaining a lot of attention, there is a demand for generic tools able to run such models at large scale.

The chemical programming model, which was designed following the chemical metaphor, is a rule-based programming model, with a non-deterministic execution model, where rules are applied concurrently on a multiset of data. In this paper, we explore the experimental side of concurrent rule-based models, by deploying a distributed chemical runtime at large scale.

The architecture proposed combines a peer-to-peer communication layer with an adaptive protocol for atomically capturing objects on which rules should be applied, and an efficient termination-detection scheme. We describe the software prototype implementing this architecture. Based on its deployment over a real-world test-bed, we present its performance results, which confirm analytically obtained complexities, and experimentally show the sustainability of such a programming model.

1 Introduction

One challenge of distributed systems stands in finding the right abstractions to *program* them. The emergence of a novel distributed computing platform calls for adequate programming models able to simply leverage its computing capacities. The global computing platform which is today emerging on top of the Internet allows to interconnect a virtually infinite number of computing devices, which, aggregated, represent a tremendous computing power. However, due to the scale, dynamics and heterogeneity of such a platform, actually leveraging this power remains a wide open issue.

Abstracting out the technical details of the low-level machinery of the platform appears to be a prerequisite to actually being able to efficiently compute over it.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 408–422, 2013.

In other words, the logic of the computation (which does not change, whatever the underlying platform characteristics are) should be separated from its low-level implementation.

This situation advocates the use of declarative programming [1], whose goal is to separate the logic of a computation (*"what we want to do"*) from its control (*"how to achieve it"*). More precisely, while the "what" is to be defined by the programmer, the "how" becomes implicit, hidden in the system. In particular, rule-based programming, where the logic is expressed as a set of *rules*, is very attractive for parallel and distributed systems, as the parallelism and distribution, and their intrinsic difficulties, are hidden from the programmer. Recently, some work has gone into showing how to concretely apply rule-based programming to the specification of distributed systems. For instance, in [2], it has been shown how communication protocols and peer-to-peer applications can be specified using a rule-based language. In [3], the same programming style is applied to web-based data management. On the computing side, rule-based programming was also used as a building block for workflow management systems [4, 5].

In this paper, we focus on the chemical programming model, which is a chemistry-inspired rule-based model. It associates rule-based programming with an implicitly-parallel runtime, and has been advocated as a promising paradigm for the high-level specification of emerging platforms [6–9].

Metaphorically speaking, in such a model, a program is envisioned as a *chemical solution* where molecules of data float and react according to some *reaction rules* specifying the program, to produce new data (the products of reactions). More formally speaking, it relies on *concurrent multiset rewriting* : the solution is a multiset of objects (*molecules*), and reactions are rewriting rules to be applied on it. At run time, these reactions can be triggered concurrently and reactions are carried out until the state of *inertia* — a stable state in which no more reactions are possible — has been reached. The order in which rules are to be triggered is not specified. In other words, it is left to the implementer of the runtime. In this area, the Higher-Order Chemical Language (HOCL) is a full-featured rule-based language [10], providing the higher order: rules themselves are molecules in the multiset and can be consumed or produced at runtime. Hence, one can model programs able to evolve at run time. In HOCL, reaction rules are of the form **replace** P **by** M **if** V where P is the pattern of reactants, V is a condition on them and M is the product of the reaction. Note that the applied rule itself is not deleted in the reaction. An HOCL program is a solution of molecules, that is to say, a multiset of non-ordered atoms (A_1, ..., A_n) which can be constants (integers, booleans, *etc.*), sub-solutions (denoted $< M_i >$), tuples (denoted $M_1 : M_2 : \cdots : M_n$) or reaction rules. Let us illustrate the paradigm with a simple HOCL program couting the characters in a multiset of words:

$$
\begin{array}{l}
\textbf{let}\ \ \text{count} \quad = \ \textbf{replace}\ s :: \mathit{string}\ \textbf{by}\ \mathit{len}(s)\ \textbf{in} \\
\textbf{let}\ \ \text{aggregate} \ = \ \textbf{replace}\ x :: \mathit{int}, y :: \mathit{int}\ \textbf{by}\ x + y\ \textbf{in} \\
< \text{“}\mathit{nel}\text{”}, \text{“}\mathit{mezzo}\text{”}, \text{“}\mathit{del}\text{”}, \text{“}\mathit{cammin}\text{”}, \text{“}\mathit{di}\text{”}, \text{“}\mathit{nostra}\text{”}, \text{“}\mathit{vita}\text{”}, \text{count}, \text{aggregate} >
\end{array}
$$

The rule *count* replaces a string by its length. The *aggregate* rule produces the sum of two consumed integers. At run time, these rules are triggered repeatedly and concurrently, the first one producing inputs for the second one. Note that the order in which rules are triggered is not deterministic; only the atomic capture of reactants is ensured. A possible succession of states is the following, the last one being inert and "$\downarrow *$" denoting the concurrent application of the rules:

$$< \text{“}nel\text{”}, \text{“}mezzo\text{”}, \text{“}del\text{”}, \text{“}cammin\text{”}, \text{“}di\text{”}, \text{“}nostra\text{”}, \text{“}vita\text{”}, \text{count}, \text{aggregate} >$$
$$\downarrow *$$
$$< 3, 3, 4, 6, \text{“}mezzo\text{”}, \text{“}di\text{”}, \text{“}nostra\text{”}, \text{count}, \text{aggregate} >$$
$$\downarrow *$$
$$< 10, 6, 5, 2, 6, \text{count}, \text{aggregate} >$$
$$\downarrow *$$
$$< 29, \text{count}, \text{aggregate} >$$

1.1 Motivation Example

Let us now illustrate the model in context by providing an example of an *autonomic* server, *i.e.,* a server able to run in the most efficient and reliable way given the characteristics of the targeted platform exposed above. More concretely speaking, considering a simple task continuously requested by some clients, and for which several implementations (or *services*) exist on the platform, we want to: (1) select the best service implementation according to some predefined policy, (2) change the optimisation policy dynamically if the criterion changes, and (3) recover automatically after the failure of the service implementation currently in use.

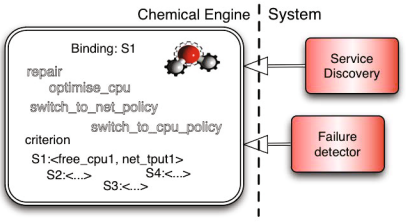


Fig. 1. An HOCL-based autonomic service

Figure 1 illustrates an HOCL implementation of such a self-adaptive service. The multiset (on the left-hand side of the figure) interfaces with two system components that achieve two respective tasks. The first one discovers available service implementations able to perform the desired task and injects them in the solution as S_i:<$free\_cpu_i$, net_tput_i> molecules. This molecule contains an *id* and a sub-solution indicating the service's performance. The list of indicators can be arbitrarily extended. The second connected system service acts as a failure detector of the currently active service (denoted by the *Binding:*S_i molecule) and introduces a *"failure_detected"* molecule upon detection. Let us now review the rules that drive the execution. First, the *repair* rule reacts to the presence of the molecule indicating the current service bounded failed. Once triggered, it binds the task to another service implementation:

$$\textbf{let}\ \ repair\ =\ \textbf{replace}\ \ Binding{:}S_i,\ S_i{:}<\omega_i>,\ S_j{:}<\omega_j>,\ \text{“}failure\_detected\text{”}$$
$$\textbf{by}\ \ \ Binding{:}S_j,\ S_j{:}<\omega_j>$$

Let us now review an optimising rule, named *optimise_cpu*. A reaction following the *optimise_cpu* rule is triggered when a service molecule S_j with a better CPU availability is found in the multiset. This rule corresponds to the decision taken by the system to select services based on their CPU availability:

$$\begin{aligned}\textbf{let}\ \ optimise\_cpu\ \ =\ &\textbf{replace}\ \ Binding{:}S_i,\\ &\ S_i{:}\langle free\_cpu_i,\ net\_tput_i\rangle,\ \ S_j{:}\langle free\_cpu_j,\ net\_tput_j\rangle\\ &\textbf{by}\ \ Binding{:}S_j,\\ &\ S_i{:}\langle free\_cpu_i,\ net\_tput_i\rangle,\ S_j{:}\langle free\_cpu_j,\ net\_tput_j\rangle\\ &\textbf{if}\ \ (free\_cpu_j > free\_cpu_i)\end{aligned}$$

Similarly, an *optimise_net* rule may consider the services' network capabilities. Having different policies brings more flexibility to the adaptation but creates the need for dynamic switching from one to the other based on a criterion, in this case meaning *optimise_cpu* might have to be put aside in favour of *optimise_net*. This can be achieved through the higher order, using the following rule, which is triggered when the criterion changes:

$$\begin{aligned}\textbf{let}\ \ switch\_to\_net\_policy\ \ =\ \ &\textbf{replace}\ \ optimise\_cpu,\ criterion{::}string\\ &\textbf{by}\ \ optimise\_net\\ &\textbf{if}\ \ (criterion = \text{“}Net\text{”})\end{aligned}$$

Note that, as illustrated on Figure 1, other rules, for instance *switch_to_cpu_policy*, can be similarly constructed and introduced in the solution concurrently. They can coexist smoothly in the solution, as the criterion can take only one value at a time, preventing concurrent reactions of contradictory switching rules. Finally, note that for the sake of simplicity, the example deals with only one service. However, it can be easily extended so as to deal with many services distributed over the nodes of a large scale platforms, each area of the platform having its own criteria and policies changing concurrently.

1.2 Contribution and Organisation of the Paper

Our goal is to provide a generic distributed platform dedicated to the execution of chemical programs. We envision a high number of nodes willing to collaborate, with each collaborating node equipped with an engine executing rules on the molecules they hold. The four following issues need to be tackled: (i) *communication abstraction*: each node has to be able to communicate with every other node; (ii) *molecule discovery*: molecules are now dispatched over the network, meaning suitable reaction candidates have to be found efficiently in spite of the scale of the platform; (iii) *atomic capture*: once the appropriate molecules have been located, a node must grab *all of them* atomically, as other nodes may try to fetch them as well at the same time; and (iv) *detection of termination*: to secure the termination of a program, we need to ensure to detect the fact that no more reactions are possible. This detection, when done in a centralised way, has a combinatorial complexity, as every combination of molecules has to be checked

against the rules (yielding $m!$ tests for a program containing m molecules). This suggests that relying on intelligent information retrieval techniques is mandatory in order to circumvent the problem.

This paper builds upon the preliminary work in [11], which gave a conceptual view of our proposal solving the four previously mentioned issues, and which is summarised in Section 2. The resolution of the first two items highly relies on the presence of a distributed hash table (DHT), while the resolution of the third one relies on a recently proposed protocol to capture several objects atomically in concurrent settings. Finally, the inertia detection is based on a second information retrieval layer built on top of the DHT. In [11], the validation of such a platform was only based on analysis. The present paper, in contrast, focuses on its experimental validation. Firstly, a software prototype implementing the concepts is detailed. Secondly, real deployments of chemical programs undertaken are presented and their results are discussed. To sum up, this paper provides the *"how"* of distributed concurrent rule-based programming, allowing programmers to concentrate on the *"what"*.

Section 2 summarises the global architecture, its data structures and algorithms. The software prototype is presented in Section 3, and the conducted experiments are discussed in Section 4. Section 5 concludes.

2 Platform Overview

We will now give a short overview of the platform. For a complete description and details about the exact algorithms and their complexities, we refer the reader to [11]. The conceptual view of the distributed execution platform is depicted on Figure 2. The platform itself is built on top of an overlay network, organising the participants in a ring-like structure. While we have chosen to rely on Pastry [12], DHTs with different topologies [13] could be used. The DHT assures that nodes are able to communicate efficiently regardless of their number while preserving the communication pattern in spite of joins, leaves and failures. An *external application* holding the program to be executed may transfer it to a node of the platform, or may itself join the execution.

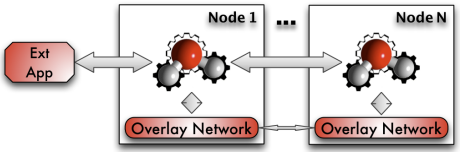


Fig. 2. The platform

The remainder of this section details the initialisation, execution and termination of a program running on this platform.

2.1 Initialisation

The node contacted by the external application is called the *source node* since it represents the data's entry point in the system. Additionally, upon termination, the inert solution, *i.e.* the result, is transferred from the source node to the external application.

Data Distribution. After receiving the data from the application, the source node scatters the data molecules uniformly across the system according to their hash values, in this way globally load-balancing access to molecules. Each node holds a subset of the program's data molecules with high probability, if the number of molecules is high enough, and all of the rules, enabling a high level of parallelism and concurrency in performing reactions. By tracing the molecules' paths, a tree, rooted at the source node, is created. The source node uses this *multicast tree* to diffuse the rules contained in the program. Furthermore, this multicast tree will be used during the termination phase to collect the resulting inert solution.

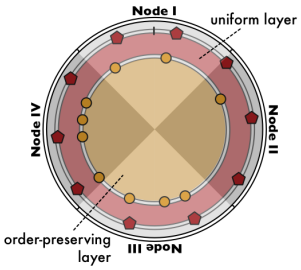


Fig. 3. Double layer: the key space of the uniform layer coincides with that of the order-preserving layer

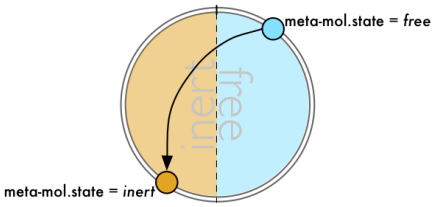


Fig. 4. Order-preserving layer: as a meta-molecule's state changes, it repositions itself in the second layer

Meta-molecules. Since molecules are spread throughout the system, nodes must be able to find suitable candidates for reactions. In order to increase the platform's scalability on top of the existing DHT layer (referred to as the *uniform layer*), we place a second DHT layer (referred to as the *order-preserving layer*), serving a storage of *meta-molecules* arranged around the key space in an order-preserving manner (Figure 3). This allows participants to use range-query techniques [14, 15] to search for the existence of a particular molecule, or a set thereof, during the execution phase.

Meta-molecules offer a lightweight indexing and look-up mechanism because instead of exchanging heavier objects like molecules, nodes are able to query and exchange only their lighter counterparts — the meta-molecules. Each molecule is associated a state in its meta-molecule. Initially, a meta-molecule's state is set to *free*, indicating that nodes can freely take the molecule it describes and combine it with other molecules to perform reactions. At a later stage, during execution, a meta-molecule's state may be set to *inert*, which denotes that a suitable combination for its molecule has not been found thus far.

The order-preserving layer is split in two parts: the one containing only *free* meta-molecules, within the id range $[0, \frac{ks}{2} - 1]$, and the other consisting of only *inert* meta-molecules, within the id range $[\frac{ks}{2}, ks - 1]$, where ks is the size of the key space. The position of a meta-molecule is based on the molecules's value in the total ordering of values of a specific molecule type and its state — *free* or *inert*. As a consequence, when a meta-molecule's state changes, its identifier is recalculated, relocating it to the *other* half of the key space (Figure 4).

2.2 Execution

The distributed platform presented adopts intelligent reactant searching, in which the system is explored for molecules with specific properties matching a rule's pattern and condition, such as *an integer greater than 3*, allowing it to efficiently detect inertia.

Algorithm 1. Main execution loop.

```
1  while not inert do
2      meta_mol1 = random_mol(state = free);
3      if meta_mol1 = null then
4          break;
5      meta_mol2 = find_candidate(meta_mol1, rule);
6      if meta_mol2 = null then
7          meta_mol1.state = inert;
8          store(meta_mol1);
9          continue;
10     if grab_molecules(meta_mol1,meta_mol2) then
11         execute_reaction(rule, mol1, mol2);
12         store(new_mol1, new_mol2);
13         store_ack(meta_new_mol1, meta_new_mol2);
14         remove(meta_mol1, meta_mol2);
```

The main execution loop, executed by every node, is described in Algorithm 1. For the sake of clarity, the algorithm is presented in a simplified form, in which only one rule involving a pair of molecules is considered. Nevertheless, it is easily expandable to multiple rules and multiple molecules per rule. It consists of three steps: (i) getting a random meta-molecule and testing inertia (lines 2—4), (ii) finding a candidate it can react with (lines 5—9) and (iii) atomically grabbing the corresponding molecules and performing the reaction (lines 10—14).

Random Meta-molecule Fetch. During the first step, a node tries to obtain a random meta-molecule, the state of which is set to *free*. `random_mol` guarantees a *free* meta-molecule will be returned, in case one exists. If, on the other hand, no meta-molecule can be found, it means the system could not find any candidate for the currently present molecules, implying their states are set to *inert*. This signals to the requesting node that inertia has been reached. It then stops executing the main loop (line 4).

Search for Candidates. Obtaining a free meta-molecule triggers the second execution step (lines 5—9). The node now asks the system to find it a suitable meta-molecule by supplying the meta-molecule found in step one and the rule which needs to be applied on the molecules to the `find_candidate` routine (line 5). This routine searches in parallel for a meta-molecule matching the provided rule's pattern and reaction condition (Figure 5).

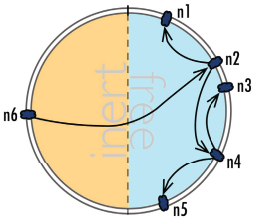


Fig. 5. Meta-molecule fetch example: $n6$ sends a request to $n2$, which propagates it in parallel

Molecule Capture and Reaction Execution. Step three (lines 10—14) concludes an execution loop iteration. The node tries to *capture* the molecules described by the previously obtained meta-molecules. Given the fact that a molecule can be consumed only once, *i.e.* it can be used in at most one reaction during its lifetime, it is imperative that nodes grab all of

the molecules needed in an atomic fashion. For this task, we rely on a capture protocol able to dynamically self-adapt in regard to the probability of conflicts when trying to capture the molecules. As the capture is not our primary concern here, we refer the reader to [16] for more information about the protocol. If the `grab_molecules` routine succeeds, it ensures no other node can obtain these molecules, triggering the actual execution of the reaction, after which the meta-molecules describing the newly created molecules are produced.

2.3 Termination

Inertia has been detected once `random_mol` (Algorithm 1, line 2) can no longer find a *free* meta-molecule in the system[1]. This marks the end of execution and the beginning of the termination phase. Upon inertia detection, each node sends its molecules back up the multicast tree, after which the source node transfers the now inert solution to the external application.

3 Software Prototype

Following the model of the platform laid out in Section 2, we developed a fully-functional software prototype in Java[2]. Figure 6 shows its logical view.

3.1 Entities

Overlay Network. The abstraction from the underlying physical network is handled by this entity. Its main component is FreePastry [17], an open-source DHT coded in Java and developed and maintained by the authors of Pastry.

Molecule Holder. This entity is the implementation of the uniform DHT layer and as such it serves as a container for molecules held by the node. In order to store, index and retrieve molecules more easily, they are grouped by their molecule types and sorted based on their hash identifiers. The molecule holder is contacted during the atomic capture step and is in charge of deciding whether and to which node a molecule it holds will be given, according to the capture protocol.

Meta-molecule Holder. Analogously, this entity represents the implementation of the order-preserving DHT layer and is, thus, a repository of meta-molecules. It manages the insertion, retrieval and deletion of meta-molecules requested by other nodes. Note that when a retrieval request is received, the meta-molecule is not removed. Instead, its copy is returned to the requesting node. Moreover, it handles random meta-molecule fetches and candidate requests. If it cannot satisfy the request, it communicates with meta-molecule

[1] The fact that being unable to find a *free* meta-molecule ensures that the inertia has been globally reached was formally established in [11].

[2] The sources are available in the *branches/devel-distrib* directory of the *svn* repository located at `http://gforge.inria.fr/scm/?group_id=2125`.

holders on other nodes to complete it, as specified by the algorithm devised previously.

Tree Manager. The multicast tree created during the initialisation phase is constructed by this entity. It maintains the node's *local state* (consisting of its parent and children) and uses it to spread the rules down the tree and to send its and its children's remaining molecules to its parent.

Central Unit. This is the main entity in the prototype. It communicates with the application (taking the program to execute from it and returning the inert result to it) and executes the main execution loop (Algorithm 1).

3.2 Execution Cycle

Initialisation. A first step is for the application to transfer the program to execute to the central unit. It then hashes the molecules and dispatches them to the overlay network, which spreads them in the uniform layer. During this period, the tree manager monitors the overlay network traffic and when it stumbles upon a molecule, it adds the destination node to its local state. Once the molecules have been disseminated, the central unit hands the rules over to the tree manager, which sends them to its children in the local state.

On the receiving end, when a node receives a molecule, it stores it in the molecule holder. This entity creates a meta-molecule for each held molecule and routes it in the order-preserving layer through the overlay network. The node then receives the rules to execute, upon which the tree manager completes its local state by assigning the node's parent in the tree (the node which has sent it the rules). Now the nodes are ready to start the execution.

Execution. At this point, the central unit on each node starts the main execution loop. It asks the meta-molecule holder to find it a random meta-molecule in the network. A rule is randomly chosen from which the type of one of its reactants is extracted. The meta-molecule holder then sends out requests in the *free* half of the order-preserving layer to find such a meta-molecule. This process is repeated for each rule until a meta-molecule has been returned. Then, the central unit *translates* the pair $(\texttt{rule}, \texttt{meta}-\texttt{molecule})$ into a range query request by injecting the meta-molecule's identifier in the rule. The request is, again, handed over to the meta-molecule holder which tries to find a candidate meta-molecule satisfying the range query in the order-preserving layer. The process of searching for a candidate is repeated for as many reactants the rule needs, each time introducing the newly acquired meta-molecule into the rule and constructing a new range query for the next candidate to be located. If the candidates cannot

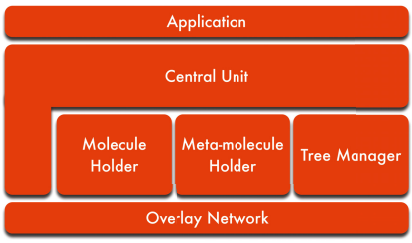


Fig. 6. Logical view of the entities forming the prototype

be found, a delete request is sent to the random meta-molecule's holder. Its state is then changed to *inert* (changing its identifier) and stored in the second DHT layer.

Following Algorithm 1, the final step of the execution phase consists in grabbing atomically the molecules and performing the reaction. For capturing the molecules, the capture protocol in [16] was implemented. In this protocol, the central unit plays the role of the molecule requester. It extracts the identifiers of the molecules to grab and sends the fetch requests to the corresponding nodes. Their molecule holders then evaluate each its own request and decide whether to send back the molecules. Only in case all of the molecules have been received the node performs the reaction. Then, as per Algorithm 1, the products of the reaction and their meta-molecules are stored in the DHT (each in their respective layer) and delete requests for the consumed molecules' meta-molecules are sent.

Termination. Once there are no more *free* meta-molecules in the order-preserving layer, the nodes enter the final, termination step of the execution. A node starts this phase when its meta-molecule holder is not able to find a random meta-molecule. At that point, the tree manager awaits the node's children's molecules. These are then combined with the molecules held by the molecule holder and sent to the parent up the tree. Finally, the central unit of the source node delivers the inert solution to the requesting application.

3.3 Optimisations

Even though the prototype follows the description of the system model discussed in earlier sections, it carries two slight improvements dealing with local meta-molecule search and meta-molecule retrieval. Both of the enhancements are implemented in the meta-molecule holder.

The first optimisation exploits the principle of *locality*: whenever the central unit requests a meta-molecule, the meta-molecule holder first checks whether it can satisfy the request right away without querying other nodes. This method is applied to both random meta-molecule and candidate search requests. At the beginning of the execution, nodes which are located in the *free* half of the key space will be able to benefit directly from it, seeing that during that time most of the meta-molecules' states are set to *free*, enabling nodes to pick a random meta-molecule from their local meta-molecule holders. Towards the end of the execution, on the other hand, nodes located in the other half of the key space can benefit from the principle during the candidate search step, since most meta-molecules will be labelled as *inert* at that point.

The second improvement is introducing a small decision-making mechanism into the meta-molecule holder. Whenever it receives a retrieval request, it tries to return the meta-molecule closest to the requested identifier. It is thus possible for the same meta-molecule to be sent to more than one node. Even though the capture protocol assures a molecule is going to be consumed in only one reaction, giving the same meta-molecule to different nodes generates superfluous network

traffic as some grab requests will be aborted. Therefore, the meta-molecule holder keeps track of the number of times each meta-molecule has been handed out to nodes. Doing so, it is able to return the meta-molecule which satisfies the request criteria but has been handed out less times than other meta-molecules. Such a slight refinement ultimately minimises the number of conflicts between nodes over molecules and consequently the network overhead due to capture aborts.

4 Evaluation

In this section we present the evaluation of the software prototype described above. To better capture the viability of the proposed platform we tested it on two programs with different properties. They are presented in Section 4.1, while the results obtained are detailed in Section 4.2.

4.1 Test Programs

We present now two distinct classes of programs on which we tested the proposed platform: highly-parallel and producer/consumer ones.

Highly-Parallel Programs. In such applications, the same operation is applied to the whole of the input data. It is thus interesting to investigate the behaviour of a decentralised execution environment when faced with such programs. We chose to represent them with `GetMax`. It is a simple single-rule program containing only integer molecules. The rule consumes two such molecules and produces a new one holding the higher value of the two. In addition to being highly parallel, this program resembles most data-processing applications, where multiple input variables are processed to give an output.

From the point of view of the execution of chemical programs, the interest of `GetMax` stands in that it represents a program with a decreasing complexity — the number of molecules declines with every reaction. Furthermore, regardless of the course of the execution the total number of reactions performed during its execution is always constant.

Producer/Consumer Programs. The second category of applications is the producer/consumer one, whose interest stands in that it imposes the sequentiality of events — a producer has to produce the input for the consumer — on an implicitly parallel paradigm executing in decentralised settings. While highly-parallel programs designate data-processing applications, producer/consumer ones can be seen as their temporal compositions — *workflows*.

The experiments were conducted with `StringManip` — a program comprised of two rules manipulating string molecules. The logic of `StringManip` consists in splitting and packing together string molecules in such a way that the resulting string molecules have a predefined length, denoted λ. The first rule, *SplitStr*, consumes one molecule whose string's length is greater than λ and produces two molecules; one is composed of the first λ characters of the original molecule's

string, while the other contains its remainder. The second rule, *ConcatStr*, takes two molecules as input and outputs one which is their concatenation. Thus, *SplitStr* produces the molecules which are going to be consumed by *ConcatStr*.

The course of the execution of `StringManip`, as well its outcome, is non-deterministic. While it is known that at the end of the execution the molecules' strings are going to have a length of λ, their contents depend on the succession of reactions performed by the system, which is influenced by the asynchronous nature of the platform. In other words, the outcome is conditioned by the reactions performed by each node, their input molecules, and the order in which they actually take place. Hence, the number of reactions done throughout the execution varies from run to run. Furthermore, the two rules are *circularly dependent* on each other — *ConcatStr* might produce molecules which are going to be consumed by *SplitStr* —, in this way bringing a partial *sequentiality* into the program.

4.2 Experimental Results

We conducted experiments using Grid'5000[3], the French national grid test-bed. The nodes were spread across nine geographically-distant sites. Each experiment was run ten times, and here we present the average of the values obtained. For every run the nodes were randomly chosen to obtain different network topologies.

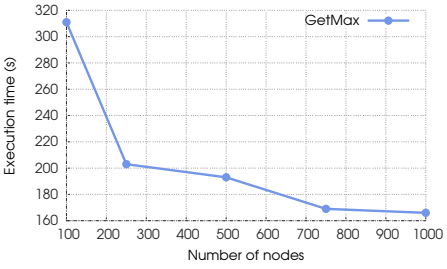


Fig. 7. Execution time of `GetMax` containing 50,000 molecules

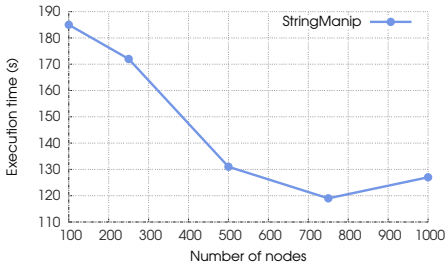


Fig. 8. Execution time of `StringManip` with 20,000 molecules

Experiment 1. Firstly, we evaluated the viability of the platform by executing the two programs while varying the number of nodes participating in the execution. Figures 7 and 8 show the execution times obtained for `GetMax` and `StringManip`, respectively.

In both cases there is a decrease in execution time when increasing the number of nodes carrying out the computation, which is in compliance with the results of the complexity analysis in [11]. Moreover, significant speed-ups were obtained. However, one can notice that the speed-up obtained for `GetMax` is greater than

[3] `http://www.grid5000.fr`

that for `StringManip`. This is due to the difference of the programs' characteristics. On the one hand, the number of molecules in the system strictly decreases after each reaction when executing the `GetMax` program, while the trend is not known for `StringManip` — it may stay constant, decrease or increase. On the other, the execution time of `StringManip` depends on the sequentiality of events: certain reactions cannot be carried out before others are. In contrast, `GetMax` is a highly-parallel program where the maximum possible number of reactions can be performed in any given point in time. Finally, the execution takes more time to complete for 1000 nodes than for 750 when executing `StringManip`. This is the result of the program's sequentiality: more nodes are in conflict over a subset molecules since not all available molecules can be used straight away, in this way prolonging the execution.

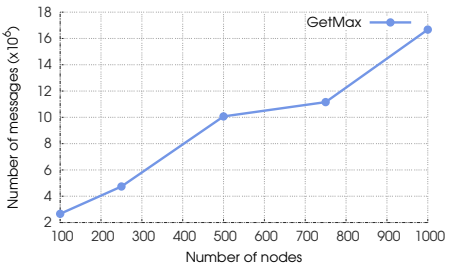


Fig. 9. Number of messages sent during the execution of `GetMax`

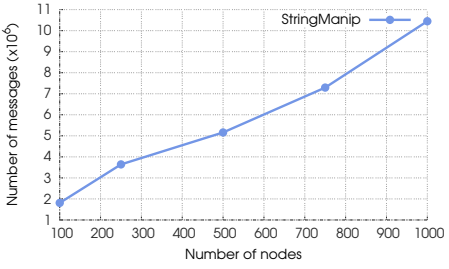


Fig. 10. Number of messages sent during the execution of `StringManip`

Experiment 2. During the execution of the programs we also monitored the generated network traffic. Figures 9 and 10 depict the total number of messages sent. Note that the number of messages in the case of `GetMax` is higher than that of `StringManip` due to the fact that there are more molecules in the initial solution of `GetMax`. Both of them show a linear augmentation in the number of messages, which conforms to the findings of the complexity analysis. We can see, however, that the curve for `GetMax` is steeper than that of `StringManip`. This is due to the fact that, because of the constant number of reactions, when there are more nodes involved in the computation there are more conflicts over molecules during the capture phase. In spite of this effect, one can notice that the actual number of messages per node declines with the growth of the network, which leads to the conclusion that the platform is scalable in terms of network load. We can thus conclude that the platform scales well.

Experiment 3. In this experiment we fixed the number of nodes to 500 while varying the number of molecules contained in the `GetMax` program. Figure 11 shows that the execution time linearly grows with the increase of the size of the problem. The same effect can be observed when looking at the network traffic, depicted in Figure 12. Both figures confirm the analysis' findings: the system is

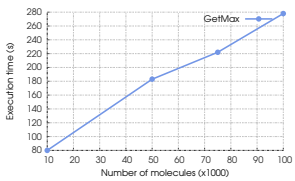


Fig. 11. Execution time of `GetMax` on 500 nodes

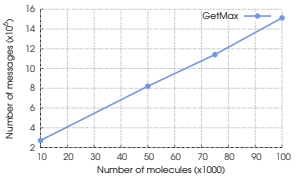


Fig. 12. Number of messages (`GetMax`)

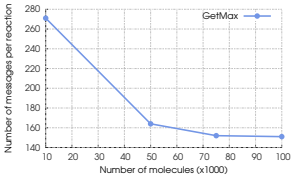


Fig. 13. Number of messages per reaction (`GetMax`)

scalable with regard to the size of the problem to be solved (*i.e.* the number of molecules). It is interesting to note that the number of messages needed to perform one reaction, illustrated in Figure 13, decreases. Indeed, when the number of molecules increases while keeping the number of nodes constant, there are less conflicts between nodes over molecules, and thus less communication is needed to carry out a reaction.

5 Conclusion

Declarative programming has been recently identified as a promising, high-level model to develop distributed systems in a simple manner. However, this calls for mechanisms able to make it real over large scale platforms. In the area of declarative programming, concurrent rule-based programming (and its chemistry-inspired representative) appears to offer an adequate level of abstractions in different areas of distributed computing.

Thus, the large-scale execution of chemical programs has to be tackled in order to put the attractive characteristics of declarative programming into practice over large scale platforms. Note that previous works attempting at running the chemical model at large scale only considered some specific parallel architectures [18–20]. More information about these works can be found in [11].

This paper proposes a generic framework to solve this issue over a distributed platform. On top of a two-layer DHT, the framework relies on two distributed protocols, the first one capturing molecules in a highly concurrent system, the second one efficiently detecting *inertia*. These concepts have been used to implement a software prototype, which was tested on a real-world test-bed. The experiments conducted corroborate the findings of the theoretical analysis about the sustainability of the proposed execution runtime.

References

1. Lloyd, J.W.: Practical advtanages of declarative programming. In: Joint Conference on Declarative Programming (GULP-PRODE 1994), pp. 18–30 (1994)
2. Grumbach, S., Wang, F.: Netlog, a Rule-Based Language for Distributed Programming. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 88–103. Springer, Heidelberg (2010)

3. Abiteboul, S., Bienvenu, M., Galland, A., Antoine, E.: A rule-based language for web data management. In: Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, pp. 293–304. ACM, New York (2011)
4. Wang, Y., Li, M., Cao, J., Tang, F., Chen, L., Cao, L.: An ECA-Rule-Based Workflow Management Approach for Web Services Composition. In: Zhuge, H., Fox, G.C. (eds.) GCC 2005. LNCS, vol. 3795, pp. 143–148. Springer, Heidelberg (2005)
5. Laliwala, Z., Khosla, R., Majumdar, P., Chaudhary, S.: Semantic and rules based Event-Driven dynamic web services composition for automation of business processes. In: Services Computing Workshops, SCW 2006, pp. 175–182. IEEE (2006)
6. Banâtre, J.-P., Radenac, Y., Fradet, P.: Chemical specification of autonomic systems. In: 13th ISCA International Conference on Intelligent and Adaptive Systems and Software Engineering, pp. 72–79 (2004)
7. Fernandez, H., Priol, T., Tedeschi, C.: Decentralized Approach for Execution of Composite Web Services Using the Chemical Paradigm. In: 17th IEEE International Conference on Web Services (2010)
8. Mostefaoui, A.: Towards a Computing Model for Open Distributed Systems. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 74–79. Springer, Heidelberg (2007)
9. Di Napoli, C., Giordano, M., Pazat, J.-L., Wang, C.: A Chemical Based Middleware for Workflow Instantiation and Execution. In: Di Nitto, E., Yahyapour, R. (eds.) ServiceWave 2010. LNCS, vol. 6481, pp. 100–111. Springer, Heidelberg (2010)
10. Banâtre, J.-P., Fradet, P., Radenac, Y.: Generalised Multisets for Chemical Programming. Mathematical Structures in Computer Science 16 (2006)
11. Obrovac, M., Tedeschi, C.: When Distributed Hash Tables Meet Chemical Programming for Autonomic Computing. In: 15th International Workshop on Nature Inspired Distributed Computing (NIDisC 2012), in conjunction with IPDPS 2012, May 21. IEEE, Shanghai (to appear, 2012)
12. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
13. Milojicic, D., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z.: Peer-to-peer computing (2002)
14. Candan, K., Tatemura, J., Agrawal, D., Cavendish, D.: On overlay schemes to support point-in-range queries for scalable grid resource discovery. In: Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005 (2005)
15. Schmidt, C., Parashar, M.: Squid: Enabling search in DHT-based systems. Journal of Parallel and Distributed Computing 68(7) (2008)
16. Bertier, M., Obrovac, M., Tedeschi, C.: A Protocol for the Atomic Capture of Multiple Molecules on Large Scale Platforms. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 1–15. Springer, Heidelberg (2012)
17. Freepastry (June 2012), http://www.freepastry.org
18. Huang, L., Tong, W., Kam, W., Sun, Y.: Implementation of GAMMA on a Massively Parallel Computer. Journal of Computer Science and Technology 12 (1997)
19. Lin, H., Kemp, J., Gilbert, P.: Computing Gamma Calculus on Computer Cluster. International Journal of Training and Development 1(4) (2010)
20. Gladitz, K., Kuchen, H.: Shared memory implementation of the Gamma-operation. Journal of Symbolic Computation 21(4-6), 4–6 (1996)

Weak Read/Write Registers

Gadi Taubenfeld

The Interdisciplinary Center, P.O. Box 167, Herzliya 46150, Israel
tgadi@idc.ac.il

Abstract. In [2], Lamport has defined three classes of shared registers which support read and write operations, called —safe, regular and atomic—depending on their properties when several reads and/or writes are executed concurrently. We consider generalizations of Lamport's notions, called k-safe, k-regular and k-atomic. First, we provide constructions for implementing 1-atomic registers (the strongest type) in terms of k-safe registers (the weakest type). Then, we demonstrate how the constructions enable to easily and efficiently solve classical synchronization problems, such as mutual exclusion and ℓ-exclusion, using single-writer multi-reader k-safe bits, for any $k \geq 1$. We also explain how, by using k-registers, it is possible to provide some level of resiliency against memory reordering.

Keywords: k-safe, k-regular and k-atomic registers, shared memory, memory ordering, memory barriers, synchronization, mutual exclusion, ℓ-exclusion.

1 Introduction

It is common to assume that operations on the same memory location are atomic – they occur in some definite order. However, this assumption can be relaxed allowing the possibility of concurrent operation on the same memory location. In [2], Lamport has defined three classes of shared registers which support read and write operations, called —safe, regular and atomic—depending on their properties when several reads and/or writes are executed concurrently. Below we consider natural generalizations of Lamport's notions, motivate their use and investigate their properties. Unless otherwise stated, it is assumed that each register is a single-writer multi-reader register. Such a register can be written by one predefined process and can be read by all the processes. Let k be a positive integer.

- The weakest possibility is a *k-safe* register, in which it is assumed that a read not concurrent with any write obtains one of the k most recently written values. No assumption is made about the value obtained by a read that overlaps a write, except that it must obtain one of the possible values of the register. We consider the initial value as the first written value.
- The next stronger possibility is a *k-regular* register, in which it is assumed that a read not concurrent with any write obtains one of the k most recently written values. A read that overlaps a write obtains either the new value or one of the k most recently written values. That is, a read that overlaps any series of writes obtains either one of the values being written or one of the k most recently written values before the first of the writes.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 423–427, 2013.

– The final possibility is a k-*atomic* register, in which the reads and writes behave as if they occur in some definite order, and a read obtains one of the k most recently written values. In other words, for any execution, there is some way of totally ordering the overlapping reads and writes so that the value returned by the each read is one of the k most recently written values in the execution which has no overlapping. (Operations that do not overlap should take effect in their "real-time" order.)

We observe that Lamport's familiar notions of safe, regular and atomic registers are equivalent to the notions of 1-safe, 1-regular and 1-atomic registers, respectively. We will use the notion k-register as an abbreviation for k-safe, k-regular and k-atomic registers, when the exact type of a register is not important.

Our study is of both theoretical and practical interest. Various optimizations enable reordering memory references as it allows much better performance. When a correct operation depends on ordered memory references, memory barriers are used to prevent reordering. Memory barriers are required to enable good performance and scalability. The reason for that is the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access [3].

Without using memory barriers, as a result of reordering, a read from an atomic register may obtain some older value when compared to the value this read would return in the in order execution of the original program code. Suppose that in some setting where reordering is possible, a read may obtain, in the worst case, one of the 5 most recently written values when compared to the value it would return in the in order execution. In such a case, no harm done, if the program which uses 1-atomic registers, was designed in the first place to work correctly assuming that communication is done via 5-atomic registers.

Consider the following design strategy: Design your algorithms to be correct when k-atomic registers are used for some $k > 1$. Now replace the k-atomic registers with the stronger 1-atomic registers. In such algorithms the use of memory barriers may not be necessary in some cases, even when reordering is possible. Thus, there is a tradeoff between the number of memory barriers needed to ensure correctness and the type of k-registers used. Put another way, proving correctness w.r.t. k-registers while actually using 1-registers provides some level of resiliency against memory reordering. Finding the exact level of resiliency provided using such a design strategy, as a function of k, is an interesting research topic which is not covered in this paper.

Our results are about computability and complexity of using k-registers. We show that for any $k \geq 1$, k-safe registers and 1-atomic registers have the same computational power. More precisely, it is possible to wait-free implement multi-writer multi-reader multi-valued 1-atomic registers using single-writer single-reader k-safe bits, for any $k \geq 1$. We present simple and efficient constructions that enable to easily and efficiently solve classical synchronization problems, such as mutual exclusion and ℓ-exclusion [4], using single-writer multi-reader k-safe registers, for any $k \geq 1$.

2 Preliminaries

We focus on an architecture in which n processes, denoted $p_1, ..., p_n$, communicate asynchronously via shared registers. A register can be either a single-writer single-reader

(SWSR) register, a single-writer multi-reader (SWMR) register or a multi-writer multi-reader (MWMR) register. Unless explicitly stated, we assume that a register is a SWMR register. Asynchrony means that there is no assumption on the relative speeds of the processes. Processes may fail by crashing, which means that a failed process stops taking steps forever. We require that the constructions presented in this paper satisfy the wait-freedom progress condition. Wait-freedom guarantees that every process will always be able to complete its pending operations in a finite number of its own steps.

3 The Constructions

We present two constructions of registers, by indicating how write operations and read operations are performed. The first construction implements a single-writer *multi-reader* multi-valued k-safe, k-regular or $(k+1)$-atomic register, denoted r, from single-writer *single-reader* multi-valued k-safe, k-regular or k-atomic registers, respectively.

Construction 1. *Let k be an arbitrary natural number, and let $r_1, ..., r_n$ be SWSR multi-valued k-registers, where each r_i ($i \in \{1, ..., n\}$) can be written by the same single process and read by process p_i. We construct a SWMR multi-valued k-register r as follows:*

- *The write operation $r := value$ is performed as follows:* **for** $i = 1$ **to** n **do** $r_i := value$*;*
- *The read operation of r by process p_i is performed by letting p_i read the value of r_i.*

The above construction is similar to Construction 1 from [2] which was designed for 1-registers. We prove the following theorem for the general case of k-registers.

Theorem 1. *The following claims are correct w.r.t. Construction 1, for any $k \geq 1$,*

1. *If $r_1, ..., r_n$ are SWSR k-safe registers or $r_1, ..., r_n$ are SWSR k-regular registers then r is a SWMR k-safe register or a SWMR k-regular register, respectively.*
2. *If $r_1, ..., r_n$ are SWSR k-atomic registers then r is a SWMR $(k+1)$-atomic register.*
3. *If $r_1, ..., r_n$ are SWSR k-atomic registers then r is not a SWMR k-atomic register.*

Proof. A read of r by process p_i that does not overlap a write of r, also does not overlap a write of r_i. If r_i is k-register (i.e., if r_i is k-safe, k-regular or k-atomic), then this read must obtain one of the k most recently written values into r. This is enough to show that if r_i is k-safe then r is k-safe. If a read of r_i by process p_i overlaps a write of r_i, then it overlaps a write of the same value to r. In such a case, if r_i is k-regular then this read must obtain either the last value written or one of the k most recently written values into r_i (and hence into r). This implies that if r_i is k-regular then r is k-regular.

Now, assume that r_i is k-atomic, and that a read of r_i by process p_i overlaps a write of the value v into r. Then (1) if v was already written into r_i, this read must obtain either the value v or one of the $k-1$ most recently written values into r_i before v; or (2) if v was not written into r_i yet, this read must one of the k most recently written values into r_i. Since the linearization point of write of v into r might be before the linearization

point of the read of v, in both cases above, the returned value is one of the $k+1$ most recently written values into r. This implies that if r_i is a SWSR k-atomic register then r is a SWMR $k+1$-atomic register.

Assume that $r_1, ..., r_n$ are k-registers. If a read of r by two different processes p_i and p_j both overlap the same write of value v into r, it is possible for p_i to get the new value v and for p_j to get the kth written value into r before the value v was written. This is possible even in the case where the read by p_i precedes the read by p_j. This possibility is not allowed by a k-atomic register. Thus, r is not a k-atomic register. □

The second construction implements a SWMR multi-valued 1-safe or 1-regular register from SWMR multi-valued k-safe or k-regular registers, respectively.

Construction 2. *Let k be an arbitrary natural number, and let r' be a SWMR multi-valued k-registers. We construct a SWMR multi-valued 1-register r as follows:*

- *The write operation $r := value$ is performed as follows:* **for** $i = 1$ **to** k **do** $r' := value$*;*
- *The read operation of r by process p is performed by letting p read the value of r'.*

Theorem 2. *The following claims are correct w.r.t. Construction 2, for any $k \geq 1$,*

1. *If r' is a SWMR k-safe register or a SWMR k-regular register then r is a SWMR 1-safe register or a SWMR 1-regular register, respectively.*
2. *If r' is a SWMR k-atomic register then r is not a SWMR 1-atomic register.*

Proof. A read of r by process p that does not overlap a write of r, also does not overlap any of the latest k writes of r'. Thus, all the k most recently written values into r' are identical and equal the most recent value written into r'. Since r' is k-register, in the case of no overlap, a read of r must obtain one of the k most recently written values into r', and thus it must obtain the most recent value written into r'. This is enough to show that if r' is k-safe then r is 1-safe.

If a read of r' by process p overlaps a write of r', then it overlaps a write of the same value to r. In such a case, if r' is k-regular then this read must obtain either the new value or one of the k most recently written values into r' (and hence into r). However, since each value is written k times, each of the k most recently written values equals either the new value or the most recent value written before the new value. This implies that if r' is k-regular then r is 1-regular.

We have assumed that r' is a k-register. Thus, during a write of r, the k most recently written values into r' equals either the new value or the most recent value written before the new value. If a read of r by two different processes p_i and p_j both overlap the same write of value v into r, it is possible for p_i to get the new value v and p_j the old value. This is possible even in the case where the read by p_i precedes the read by p_j. This possibility is not allowed by a 1-atomic register. Thus, r is not a 1-atomic register. □

We notice that Construction 2 can be used for implementing a 2-atomic register from k-atomic registers. It would be interesting to find a similar simple and efficient construction also for implementing 1-atomic register from k-atomic registers.

Theorem 3. *It is possible to construct a MWMR 1-atomic register using SWSR k-safe bits.*

Proof. It follows immediately from Construction 1 and Construction 2 that it is possible to implement a SWMR 1-safe bit using SWSR k-safe bits. A well known result is that it is possible to implement a MWMR multi-valued 1-atomic register from SWMR 1-safe bits (see Chapter 4 of [1]). The result follows. □

The known constructions of a MWMR multi-valued 1-atomic register from SWMR 1-safe bits, are complicated and are not practically useful for transforming algorithms that use strong type of registers into algorithms that use weak type of registers.

4 Algorithms Using k-Safe Bits

There are several classical synchronization algorithms that only use SWMR 1-safe registers, for interprocess communication [4]. Using Construction 2, such algorithms can be easily and efficiently modified to use only k-safe registers, for any $k \geq 1$. In the full version of the paper ([5]), we demonstrate how this idea is used for solving the mutual problem and the ℓ-exclusion problem, using SWMR k-safe registers, for any $k \geq 1$.

5 Discussion

We have introduced the new notions of k-safe, k-regular and k-atomic registers, and showed how to implement 1-atomic registers (the strongest type) in terms of k-safe registers (the weakest type). We presented simple and efficient constructions that enabled us to solve classical synchronization problems, such as mutual exclusion and ℓ-exclusion, using single-writer multi-reader k-safe bits, for any $k \geq 1$. On most modern microprocessors memory operations are not executed in the order specified by the program code. Memory reordering is used to fully utilize the different caches installed in such machines. Using k-registers provides some level of resiliency against memory reordering. The idea is to design an algorithm using k-registers and then (after proving its correctness w.r.t. the k-registers) to replace the k-registers with 1-registers. During run time, as a result of memory reordering, the 1-registers may exhibit a behavior of k-registers (w.r.t. the in order execution of the original program code), but that should not cause a problem as the algorithm was designed in advance to be correct when using k-registers. Exploring how the use of weak objects (like k-registers) can provide some level of resiliency against memory reordering and reduce the number of memory barriers required, is an interesting research topic.

References

1. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, 508 pages. Morgan Kaufmann Publishers (2008)
2. Lamport, L.: On interprocess communication, parts I and II. Distributed Computing 1(2), 77–101 (1986)
3. Mckenney, P.E.: Memory barriers: a hardware view for software hackers (2009)
4. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming, 423 pages. Pearson/Prentice-Hall (2006) ISBN 0-131-97259-6
5. Taubenfeld, G.: Weak read/write registers (2012), The full version is avaliable at http://www.faculty.idc.ac.il/gadi/Publications.htm

Fast Leader (Full) Recovery Despite Dynamic Faults*

Ajoy K. Datta[1], Stéphane Devismes[2], Lawrence L. Larmore[1], and Sébastien Tixeuil[3]

[1] Department of Computer Science, UNLV, USA
{Firstname.Lastname}@unlv.edu
[2] VERIMAG, Université de Grenoble, France
Stephane.Devismes@imag.fr
[3] LIP6, UPMC Sorbonne Universités & Inria, France
Sebastien.Tixeuil@lip6.fr

Abstract. We give a leader recovery protocol that recovers a legitimate configuration where a single leader exists, after at most k arbitrary memory corruptions hit the system. That is, if a leader is elected before state corruptions, the *same* leader is elected after recovery. Our protocol works in any anonymous bidirectional, yet oriented, ring of size n, and does *not* require that processes know n, although the knowledge of k is assumed. If $n \geq 18k + 1$, our protocol recovers the leader in $O(k^2)$ rounds using $O(\log k)$ bits per process, assuming unfair scheduling. Our protocol handles *dynamic* faults in the sense that memory corruption may still occur while the network has started recovering the leader.

1 Introduction

Self-stabilization [1] is often regarded as a strong *forward recovery* mechanism that recovers from any transient failure. Informally, a self-stabilizing protocol is able to recover correct behavior in finite time after arbitrary faults and attacks placed the system in some arbitrary initial state. Its generality comes at a price: extra memory could be needed in order to crosscheck inconsistencies; symmetries occurring in the initial state could cause a given problem (*e.g.* leader election or mutual exclusion) to be impossible to solve deterministically, and when few faults hit the network, "classic" self-stabilization does not generally guarantee a smaller recovery time.

The intuition that when few faults hit the system, it should be possible to impose more stringent constraints on the recovery than just a basic "eventual" correctness has proven to be a fertile area in recent research [2–5]. Defining the number of faults hitting a network using some kind of Hamming distance,[1] variants of the self-stabilization paradigm have been given, *e.g.*, k-stabilization [2] guarantees that the system recovers when the initial configuration is at distance at most k from a legitimate configuration. This notion is weaker than self-stabilization, as this latter permits recovering from any configuration. In the literature, weakened forms of self-stabilization have been used for (1) circumventing impossibility results in self-stabilization (*e.g.* deterministic leader election or recovery in anonymous networks) and (2) obtaining recovery times that only

* See www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2012-18 for the full version.

[1] The minimal number of processes whose state must be changed to recover a correct configuration.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 428–433, 2013.

depend on the number of faults k (as opposed to n or D, the network size or diameter). The algorithm given here recovers in $O(k^2)$ rounds, and satisfies both conditions.

The concept of only-k-dependent recovery time has been refined under the name of *time adaptivity* (or fault locality) [3–5], when the recovery time depends on the *actual* distance f to a legitimate configuration in the initial state. Initial work on time adaptivity required the initial distance to be not greater than k (that is, they are k-stabilizing), but the latest work [3] does not have this limitation and is thus also self-stabilizing. However, it is important to note that it distinguishes between "output" stabilization (which considers only the output variables of each process that are mentioned in the problem specification) and the "state" stabilization (which considers the global state, *i.e.,* all variables used by the protocol). In all aforementioned work, only the output is corrected quickly (that is, depending on f or k), while the global state is recovered more slowly (that is, the recover time depends on D or n). Output *vs.* state stabilization has an important practical consequence: if a *new* fault occurs after output stabilization yet before state stabilization, output complexity guarantees are not maintained after the new fault. For networks that are subject to intermittent failures, protocols should strive to provide state stabilization. As a consequence, the "fault gap" (defined as the minimum time between consecutive faults that can be handled by the protocol [6]) remains large.

The problem of correcting global states quickly using self-stabilizing algorithms was investigated for the purpose of *fault containment* [6–9] (that is, preventing local memory corruptions from propagating to the whole network). The state of the art in this matter nevertheless requires that only a single process is corrupted [6], faulty processes are surrounded by many correct ones so that few faults can be caught quickly [8], the network is fully synchronous [7], or that the recovery guarantee is only probabilistic [9]. The "fault gap" that results from those approaches is significantly reduced, as only a delay that depends on the fault span must separate consecutive faults.

Our Contribution. We give a leader recovery protocol, LE(k), that recovers a legitimate configuration where a single leader exists, after at most k arbitrary memory corruptions hit the system. That is, if a leader is elected before state corruption, the *same* leader is elected after recovery. Our protocol works for an anonymous bidirectional, yet oriented, ring of size n, and does *not* require that processes know n, although the knowledge of k is assumed. If $n \geq 18k + 1$, our protocol recovers the leader in $O(k^2)$ rounds using $O(\log k)$ bits per process, assuming unfair scheduling.

With respect to "output stabilization", our protocol recovers the full correct state quickly ($O(k^2)$ rounds). With respect to fault-containment, LE(k) can handle up to k faults, faults can be arbitrarily spread, the network is fully asynchronous, and the scheduling is unfair, and finally the recovery property is deterministic.

LE(k) also exhibits an interesting property with respect to the "fault gap" metric. In our approach, the k tolerated memory corruptions need not occur in the initial state. In fact, they may occur in a dynamic way after the network has started recovering the leader. In other words, faults that can be handled by our protocol are not only arbitrarily placed, but also arbitrarily timed. For a particular set of k faults, the fault gap between those faults is optimal, that is, zero. However, a delay, that depends on k, still must be observed between sets of k faults in a computation.

2 Preliminaries

Model. We consider *distributed systems* of n *deterministic anonymous processes* organized into an *oriented ring* : each process p distinguishes one of its neighbors as its *successor*, and the other its *predecessor*. The orientation is consistent: the successor of the predecessor of any process p is p.

Communication between neighboring processes is carried out using a finite number of *locally shared variables*. Each process has its own set of shared variables which it can write and which its two neighbors can read, *i.e.*, the ring is *bidirectional*.

The *state* of a process is defined to be the vector of values of its variables. A *configuration* of the system consists of a state for each process. A process can change its state by executing its *local algorithm*. We assume uniformity, that is, all processes have the same local algorithm. The set of local algorithms defines a *distributed algorithm* on the ring. The local algorithm executed by each process is described using a finite set of *guarded actions* of the form: If $\langle guard \rangle$ then $\langle statement \rangle$. The *guard* of an action at process p is a Boolean expression involving only variables of p and its neighbors. The *statement* of an action of p updates some variables of p. An action can be executed only if its guard is true. An action of a process p is *enabled* in a configuration γ if its guard is true in γ, and p is said to be enabled in γ if at least one of its actions is enabled in γ.

k-Stabilization. Let $\mathcal{A}$ be a distributed algorithm. An ordered pair (γ, γ') is a *step* of $\mathcal{A}$ if there exist a non-empty subset S of processes enabled in γ such that γ' is the result of the atomic execution one enabled action per process of S on γ. An ordered pair (γ, γ') is a *fault* of $\mathcal{A}$ if there is exactly one process of the network which has a different state in γ' than in γ, and if γ' does not follow from γ by any step of $\mathcal{A}$. A *k-fault computation* of $\mathcal{A}$ is a sequence of configurations $\gamma_0\gamma_1\cdots$ such that: (1) there are at most k choices of i for which (γ_i, γ_{i+1}) is a fault of $\mathcal{A}$, (2) for all other i, (γ_i, γ_{i+1}) is a step of $\mathcal{A}$, and (3) the sequence is either infinite, or ends at a final configuration, where no process is enabled. $\mathcal{A}$ is *silent* if all its k-fault computations end at a final configuration.

k-fault computations are driven by a *daemon* that chooses when the faults occur and which processes execute an action when there is a step. We assume the *unfair distributed daemon*, which is otherwise unconstrained. In particular, it can choose to never select an enabled process in any step, unless it is the only enabled process.

Let $\mathcal{L}$ be a non-empty set of final configurations of $\mathcal{A}$. For a given integer $k > 0$, $\mathcal{A}$ is said to be *k-stabilizing w.r.t.* $\mathcal{L}$ if every k-fault computation of $\mathcal{A}$ which begins at some configuration $\lambda \in \mathcal{L}$ is finite and ends at λ. $\mathcal{L}$ is called the set of *legitimate* configurations of $\mathcal{A}$. In the problem we address $\mathcal{L}$ has n members; for each process ℓ, there is exactly one legitimate configuration in which ℓ is the leader.

3 Algorithm LE(k)

In a legitimate configuration of LE(k), there is one leader process ℓ, and no action of LE(k) is enabled. Once a fault occurs, LE(k) starts. If at most k faults occur, the computation will end, and the last configuration will be the same as the first.

Define the *interval of relevance* of a process p to be the set of all processes within distance $3k$ of p, which has $6k + 1$ processes in all. Every process has a *vote*, and in a

legitimate configuration, every process within ℓ's interval of relevance votes for ℓ, while every other process' vote is $\perp$. Since the system is anonymous, a process p's vote for a process q is a *relative address*, namely i where q is i steps to the right of p, or $-i$ if q is i steps to the left of p. In particular, in a legitimate state, ℓ will be the unique process whose vote is 0.

Since a fault can change any variable, it can change the vote of a process. A single fault can cause up to three processes to change their votes, but not more. Thus, throughout any k-fault computation of LE(k), there will be at least $3k + 1$ votes for ℓ, and at most $3k$ votes for any process other than ℓ.

Every process p has a *rumor* field as well, which is either $\perp$, or is the "rumor" that some process, say q, is the leader. In a legitimate configuration the rumor fields of all processes are the same as their votes.

Processes do not change their votes easily, but rumors spread rapidly. If the rumor field of a process p is different from its vote, it must decide whether to change its vote to match the rumor. To make this decision, p initiates a *query* to count votes for the rumored leader. A rumored leader is called a *candidate*. If the rumor field is $\perp$, p can initiate a query where the candidate is the process that p is voting for.

A *query* has a *home process* and a *candidate process*. The *home process* is the one that initiated the query, and the candidate of the query is the one of its home process.

A query traverses a path of query variables called its *query path*. During that traversal, the query visits every process within the interval of relevance of its candidate process, say q, and counts all votes for q. Upon returning to p, it reports the count of votes. If q receives at least $3k+1$ votes, p concludes that q is the leader; otherwise, p concludes q is not the leader.

There are three *query tracks*, which span the entire ring, each intersecting each process at a *query variable*. A *query* consists of one *live query token*, which is located in one of the query tracks. The process at which the live token is located is called the *host* of the query. The traversal of a query consists of (1) moving along the first query track toward the leftmost process in the interval of relevance of its candidate, then (2) crossing to the second query track and traversing (rightward) the whole interval of relevance of the candidate to count the votes for it, and finally (3) crossing to the third query track and moving leftward along that track until returning to its home process to report the total number of votes for the candidate.

A query moves by forward copying and rear deletion. When a live token is copied to the next query variable in the path, the old copy is designated *dead* and must be deleted before the live token can be copied forward.

During the time the query is outstanding, its home process p will not change its vote (unless it faults) but its rumor field might change. If the candidate of the query differs from p's vote, and if the query reports that the candidate has at least $3k + 1$ votes in the interval of relevance, then p changes its vote to be for that candidate and initiates a new rumor that the candidate is the leader, unless its rumor variable is already for that candidate. Otherwise, *i.e.,* the query reports no more than $3k$ votes for the candidate, p does not change its vote (or changes it to $\perp$ if the vote was already for the candidate) and initiates a *denial*, which floods the interval of relevance with the information that the candidate is not the leader, and then self-deletes. That denial wave

(unless it is interrupted by a fault or a higher priority denial wave) causes all rumors for the candidate to be deleted.

If p's rumor field is $\perp$, but p is voting for a process q, then p initiates a query where q is the candidate. If the query counts at least $3k+1$ votes for q, then p changes its rumor to q; but if the query counts at most $3k$ votes, p changes its vote to $\perp$ and also issues a denial for q.

If a process p is voting for a false leader, it will eventually change to vote to be for true leader, ℓ. If another process, say q, is voting for ℓ but has a rumor supporting some other candidate, say m, it initiates a query with m as the candidate. When q discovers that m is not the leader, it issues a denial of the rumor. If another false rumor spreads to q, it will again send out a query. Eventually, q will send a query whose candidate is ℓ. When this query returns with the information that ℓ has at least $3k+1$ votes, q will issue the rumor that ℓ is the leader. Processes voting for false leaders will see this rumor, and will then initiate their own queries, confirming that ℓ is the leader.

Rogue Queries. Faults can create *rogue queries*. A query is rogue if its home is a process p but p did not initiate it. One fault can cause up to nine rogue queries to be created. In the worst case, there is no way to distinguish a rogue query from one that was initialized normally. Thus, LE(k) cannot specifically delete rogue queries.

Lost Queries. If a process p initializes a query and that query is deleted due to a fault, then p could, in principle, wait forever for the query to return. If p suspects that its query has been deleted, it sends out a *probe wave*, either to the left or the right, whichever is the direction of the missing query, and if it receives back the *report* that there is no query, it returns to the resting state, allowing it to initiate a new query if necessary.

We use two additional variables to count the number of consecutive processes to the left (resp. right) of a process, including the process itself, which have no query, probe, or report token. The value of these variables are only eventually correct, this is why we cannot directly used them to decide that a query is missing. Rather, we use them to stop generating probe waves: while the count is less or equal to $6k+1$ in some direction, the process p does not generate a probe in that direction, because there could exist a token up to $6k+1$ hops away from p in that direction, and its home could be p.

Deadlock Prevention. As with denials, the rumors, probes, and reports can overwrite others with lower priority. This ensures that these waves cannot be deadlocked.

However, the query tracks should be carefully addressed. To avoid congestion in the query tracks, LE(k) never allows two neighboring processes to be querying simultaneously. There is a resource between each pair of adjacent processes, (think of the chopstick between two philosophers in the classic Dining Philosophers problem), and a process must have both adjacent resources to initiate a query, and must hold onto both while it is querying. A resource can be held by only one of its two neighboring processes. It is implemented using two flags, one at each node. To prevent contention, we allow a process to pass a token query flag to its neighbor, but not to seize the token.

The number of outstanding queries never exceeds the number of legitimate queries plus the number of rogue queries. Because of the flags, no more than half of the processes can have legitimately initiated outstanding queries, and there are no more than $9k$ rogue queries. So, the number of outstanding queries never exceeds $\frac{n}{2} + 9k < n$

Thus, assuming $n \geq 18k + 1$, the third query track cannot be deadlocked because there is always some empty place in that track. Similarly, the other query tracks also cannot be deadlocked.

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
2. Beauquier, J., Genolini, C., Kutten, S.: k-stabilization of reactive tasks. In: PODC, p. 318 (1998)
3. Burman, J., Herman, T., Kutten, S., Patt-Shamir, B.: Asynchronous and Fully Self-stabilizing Time-Adaptive Majority Consensus. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 146–160. Springer, Heidelberg (2006)
4. Kutten, S., Patt-Shamir, B.: Stabilizing time-adaptive protocols. TCS 220(1), 93–111 (1999)
5. Kutten, S., Peleg, D.: Fault-local distributed mending. J. Algorithms 30(1), 144–165 (1999)
6. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing 20(1), 53–73 (2007)
7. Ghosh, S., He, X.: Scalable self-stabilization. JPDC 62(5), 945–960 (2002)
8. Beauquier, J., Delaët, S., Haddad, S.: Necessary and sufficient conditions for 1-adaptivity. In: IPDPS (April 2006)
9. Dasgupta, A., Ghosh, S., Xiao, X.: Probabilistic Fault-Containment. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 189–203. Springer, Heidelberg (2007)

Addressing the ZooKeeper Synchronization Inefficiency

Babak Kalantari and André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL)
{babak.kalantari,andre.schiper}@epfl.ch

Abstract. ZooKeeper provides an event like synchronization mechanism, which notifies the clients upon state change on the server. This mechanism leads to very inefficient implementation of synchronization objects. We propose a new solution to this problem. The solution is to handle a sequence of client operations completely on the server through a generic API. We have developed a prototype that allows very efficient implementation of synchronization objects. The solution requires a deterministic multi-threaded server. Experiments show the significant gain in efficiency of our solution on producer-consumer queues and synchronization barriers.

1 Introduction

A *coordination* service is a middleware that provides fault-tolerant high level primitives for distributed applications to achieve a complex task in a coordinated fashion. Examples of coordination are: reliable state management, locks and queues, leader election, etc. ZooKeeper [1] is the most recent, promising development providing coordination. In ZooKeeper coordination is facilitated by allowing client processes to access a shared hierarchical name space, similar to file system, called *znodes*. Znodes can store application states/data and have children nodes. Fault tolerance in ZooKeeper is achieved by replicating this name space using state machine replication. The basic operations allow the client to add/delete znodes, to read/write the data stored in znodes and to obtain the children of a znode. These operations are all non-blocking which releases ZooKeeper from having to address the problem of faulty clients. Instead, a *watch* mechanism is provided to notify clients upon state change of znodes. As we discuss in Section 2, this mechanism causes serious efficiency problems that cannot be solved easily within ZooKeeper. In Section 3 we address this problem by introducing a new coordination framework, which shares some concepts with ZooKeeper. In our framework, servers handle the clients' requests in a minimal kernel with a deterministic scheduler, providing semaphores to allow clients synchronization in an elegant and efficient manner.

2 Limitations of Synchronization in ZooKeeper

Synchronization in ZooKeeper can be only achieved by using a watch primitive. Since a watch is just a notification facility, the resulting synchronization

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 434–438, 2013.

is rather weak.[1] Watches also introduce herd-effect. In order to better understand the deficiencies of ZooKeeper, we discuss the implementation of a very basic coordination object, namely a producer-consumer queue with operations *enqueue()* and *dequeue()*.

Queue example: A queue in ZooKeeper can be represented by a regular znode, say *zn*, and its children. The name *zn* is the queue name and each child of *zn* represents a queue item. Therefore, creating or deleting a child to/from *zn* is in fact adding or removing a queue item. To produce an item, a znode child with a unique sequence number appended to its name is created under *zn*. This sequence number is incremented atomically by ZooKeeper whenever a child is created. To consume an item, the client has to read all the children of *zn* using the *getChildren* operation, and sort them according to their sequence numbers. The child with the smallest sequence number is the queue head. This child is then removed using the *delete* operation. Note that the *delete* may fail if in the meantime this child was removed by another client. If this happens, removing the next (smallest) child in the list has to be tried. If the queue is empty (*zn* has no child), then a *watch* is left on *zn* to get notified whenever *zn* is modified (child added). After notification, the clients execute again the above procedure (*getChildren*, *sorting* and *deleting*). If several consumers are blocked by an empty queue, all of them concurrently execute the procedure but only one will succeed.[2]

Discussion: From the above description, we see several sources of inefficiency in the queue implementation. When consuming an item, requiring the client to read all the queue items plus to order them is highly inefficient. When several consumer clients are blocked because the queue is empty, awaking all of them when an item is added to the queue is inefficient: all consumers — except one — will have to block again. Why not add locks or semaphores to ZooKeeper? To answer this question, assume for a while that ZooKeeper provides locks with *lock/unlock* operations. A client could typically, using a *mutex* lock, execute the following operations: (1) *lock(mutex)*, (2) *read/write* znodes, (3) *unlock(mutex)*. The problem is the crash of the client between (1) and (3): ZooKeeper would have to handle the problem. For this reason, by design, ZooKeeper excludes providing locking mechanisms. ZooKeeper calls the lock operation a non *wait-free* operation, and requires that all operations provided to clients are wait-free.

3 Addressing the ZooKeeper Inefficiency

To explain our solution, we start from the example introduced above, where a client has to execute the following sequence of operations:

lock(mutex); *read/write znodes*; *unlock(mutex)*.

[1] This is because: (a) clients can miss state updates since watch event is one-time, (b) notifications do not contain the state, (c) upon a change all watchers are notified.

[2] This can be avoided using "lock without herd effect" of [1], however, it is more complex and slower method because of its linear search and extra znode create/delete.

The atomicity property is violated if the client crashes before executing *unlock(mutex)*. The problem is not solved using transactions: transactions also would require ZooKeeper to repair partial state changes after the crash of clients, but this is excluded by design. The solution we propose is to *send the whole sequence of operations, including blocking and unblocking operations, to the server.* Even if the client crashes afterwards, the server is able to execute the full sequence of operations. In [2] we explain how this is done. The server executes the following tasks:

1. Reception of client requests;
2. Ordering of client requests (atomic broadcast);
3. Execution of client requests;
4. Sending results to clients.

Step 3 is usually done by one single thread to ensure deterministic execution. In our server, this is done by several threads that are scheduled deterministically, using a kernel based on a coroutine library. [3] This kernel implements also semaphores, which allows the synchronization of client processes. Several threads are needed in our server, precisely because threads, handling client requests, can be blocked by semaphores. Revisiting the queue example, the *produce* operation becomes:

$$P(semaphore_1);\quad deposit\ the\ item;\quad V(semaphore_2),$$

where $semaphore_1$ is initialized to the size of the queue, and $semaphore_2$ is initialized to 0. The *consume* operation becomes:

$$P(semaphore_2);\quad consume\ an\ item;\quad V(semaphore_1).$$

4 Performance Evaluation

We evaluate experimentally the cost of the ZooKeeper synchronization and the alternate implementation we have described, called *GESMAS* (GEneric State Machine and Application Service). Here we present only our results for the *queue* object. Discussion of results as well as the experiments for the *barrier* object and the *herd-effect* are presented in [2].

Our server consists of three machines [4] each running one replica. We have used ZooKeeper release *3.3.4*. Our GESMAS state machine is implemented using JPaxos [4].

Latency: We measured the latency of the *produce* and the *consume* operations, including the latency as a function of the number of items [5] in the queue. Figure 1 shows the latency measurements for the *produce* operation. To measure the cost of N *produce* operations, the client invokes the operation on the queue, waits until the response is received, performs the next *produce* operation, etc. This is

[3] We use Java *Continuations* provided by *JavaFlow*, an Apache Common library [3].
[4] Single-core 2.8 GHz, 1GB RAM running Linux-2.6.18 connected via 1 Gbps links.
[5] Each item in the queue stores only one integer, hence very few bytes.

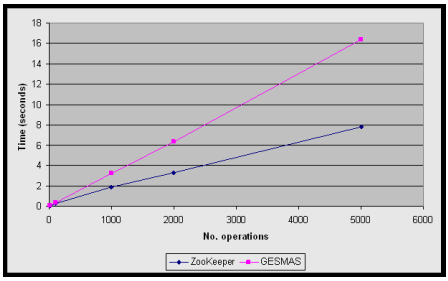


Fig. 1. Latency of *produce* operation

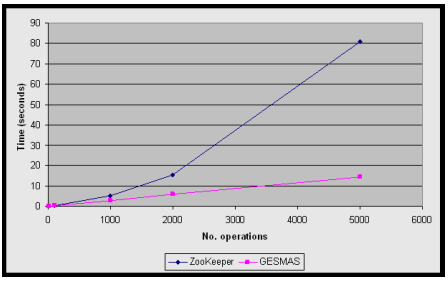


Fig. 2. Latency of *consume* operation

repeated N times and the total duration is measured. We can see in Figure 1 that ZooKeeper has lower latency compared to GESMAS. This can be explained by the fact that the *produce* operation in ZooKeeper requires only the creation of one file, while in GESMAS it involves additionally to write to the *headTail* file.[6] To measure the latency of N consecutive *consume* operations, the queue was first filled with N items. The result is shown in Figure 2. The figure shows that the latency of the *consume* operation in ZooKeeper is much higher than the latency of the *produce* operation, and also much higher than the latency of the *consume* operation with GESMAS.

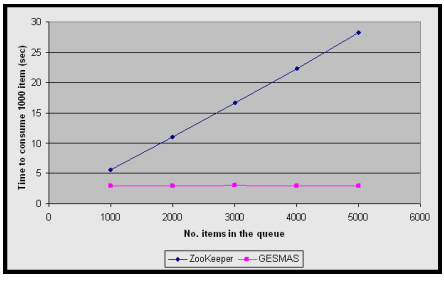


Fig. 3. Dependency to no. items in the queue. Latency of 1000 *consume* operation.

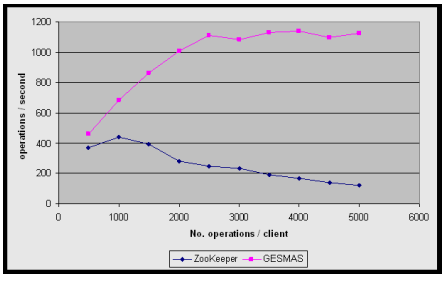


Fig. 4. Throughput of combined *produce*/*consume* operations

This result is not surprising, considering the discussion in Section 2. This is in contrast to GESMAS, where the *consume* operation involves steps similar to those of the *produce*. Finally, we studied the latency of the *consume* operations as a function of the number of items in the queue. We measured the latency of 1000 *consume* operations, for different initial queue sizes, starting with 1000. The results appear in Figure 3: in GESMAS the latency is constant, while in ZooKeeper it increases linearly with the queue size.

Throughput: We measured the maximum number of operations per second that can be executed on a queue. In order to generate high loads, we had to

[6] This file keeps track of the head and tail indices.

consider more than one client. For *produce* operation, we measured the number of operations executed on the server during 1 second. The load was continuously increased by adding clients, and the number of operations per second was measured. The maximum throughputs of 1260 for ZooKeeper and 1041 for GESMAS were reached. For *consume* operations, we could not proceed in the same way for ZooKeeper and GESMAS. For GESMAS, before running the experiments, we initialized the queue with "enough" elements such that *consume* would never block. Then, we did the measurements in the same way as for *produce* operations. This resulted in 1265 consume operations per seconds. For ZooKeeper, the same procedure would lead to very bad results.[7] Therefore, we did the measurements differently. We considered four clients, two producers and two consumers, and a queue initially empty. All clients were started at the same time: each client sends a request, waits for the response, and then sends the next request immediately. We ran the same experiment also for GESMAS to show the relevance of the experiment. The results in Figure 4 show the number of operations per second, as a function of the total number of operations executed by each client. For ZooKeeper the maximum of 442 at 1000 operations per client is reached. Excluding the contribution of the produce operation results in 270 *consume* operations per second, to be compared with the 1265 *consume* per second in GESMAS.[8]

5 Conclusion

In the paper we have addressed an important source of inefficiency in the ZooKeeper coordination service with a radically different approach. The way we provide synchronization in one hand and the openness of the client API on the other hand, enables rather easy building of efficient and sophisticated coordination objects. The open API however raises security and resource usage issues. A simple restrictive solution would be to allow sending the object implementation (i.e., sequence of operations) to the server by well identified, trusted, expert, code developers. Sandboxing techniques can be used in addition for more protection of server's resources.

References

1. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference, USENIXATC 2010, p. 11. USENIX Association, Berkeley (2010)
2. Kalantari, B., Schiper, A.: Addressing the Zookeeper Synchronization Inefficiency. Technical Report 181690, EPFL (October 2012)
3. http://commons.apache.org/sandbox/javaflow/
4. Santos, N., Konczak, J., Zurkowski, T., Wojciechowski, P., Schiper, A.: JPaxos: State machine replication based on the Paxos protocol. Technical Report 167765, EPFL (July 2011)

[7] About 40 operations per second with initial queue size of 10000 and four clients.
[8] A similar derivation for GESMAS, gives 1269 *consume* operations per second [2].

Compact TCAM: Flow Entry Compaction in TCAM for Power Aware SDN

Kalapriya Kannan[1] and Subhasis Banerjee[2]

[1] IBM Research, India
kalapriya@in.ibm.com
[2] IIIT-Delhi, New Delhi, India
subhasis@iiitd.ac.in

Abstract. Low latency lookup (typically single cycle) has made Ternary Content Addressable Memory (TCAM) indispensable in high performance network switching devices. However, high power dissipation of TCAM makes it incongruous in switches for today's power sensitive emerging network framework, viz., Software Defined Network (SDN). In this paper we propose *Compact TCAM*, an approach that reduces the size of the flow entries in TCAM. We use shorter tags for identifying flows than the original number of bits used to store the flow entries for SDN switches. We leverage the dynamic programming capability of SDN to route the packets using these tags. We show that our approach can be easily implemented using the new SDN framework while optimizing the TCAM space. Our experiments with real world and synthetic traffic show average reduction of TCAM power by 80% in SDN switching devices for a given number of flows.

1 Introduction

TCAMs due to their fast lookup are part of every high performance network switch. With Emerging SDN frameworks, the volume of flow entries is expected to grow several orders higher than the traditional L2/L3 networks (about 72,000/min estimated in [4] in SDN's). It requires TCAMs of much larger size (upto 18Mbit are commercially available), but the notoriously high power dissipation and extreme complexity of circuit structure makes is infeasible to architect large sized TCAM. The key challenge in designing TCAM in high speed switching devices is to optimize its size to fit within the power budget [2].

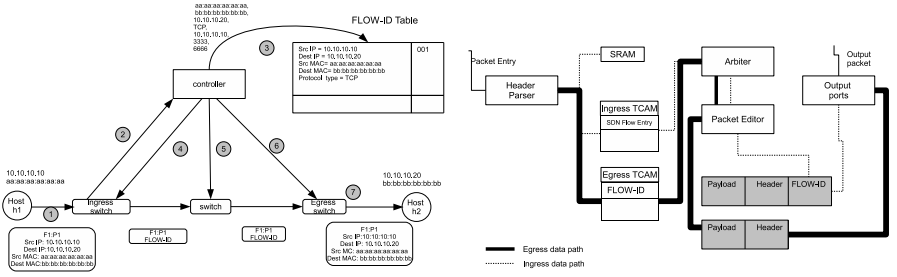
In this paper, we propose **Compact TCAM**, an approach that exploits the SDN's features such as programming interface to the switches and dynamic determination of actions for each flow at the switches. We reduce the size of bits to store information that are essential to classify packets to a flow. To do so, we assign a *Flow-ID* for each flow that can uniquely identify packets in a given flow. The packets headers are modified at the ingress switches to carry the *Flow-ID* that can be used by other switches on the path for classifying the packets. We show that the flow tuple size of 15 fields defined in SDN (15 fields tuple is stored in 356 bits) can be effectively reduced to 16 bits, an entry that is compact compared to the original 356 bits. By reducing this to 16 bits, our experiments indicate savings of about 80% power dissipation and optimizes cost by increasing the number of flows that can now be stored in the same flow tables.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 439–444, 2013.

The rest of the paper is organized as follows: Our approach of *Flow-ID* based flow entry condensation is presented in section 2. We present assessment and evaluation of our approach in section 3. We describe the prior work in section 4. Finally we conclude in section 5.

2 Compacting Flow Entries Using *Flow-ID*

The flow entries in a table classify incoming packets into different flows. Once a flow entry is identified, subsequent packets belonging to the flow need not match the entire flow entry. The size of the entry can now be compacted and replaced with a shorter ID for classification. Hence we introduce the notion of "*Flow-ID*". *Flow-ID* is a numeric identifier used to identify the flows uniquely. In the switch flow table, the flow entry can be reduced to the size of the *Flow-ID* and associated actions. All routings are then performed based on the *Flow-ID*.



(a) Components and Steps (b) Compact TCAM Hardware

Fig. 1. (a) System overview, (b) Compact TCAM hardware design

Fig. 1(a) shows the SDN architecture consisting of the controller, OpenFlow switches and host machines. In OpenFlow the first packet of a new flow arriving at an interface of a switch (Step 1)is forwarded to the controller (Step 2). The controller generates a *Flow-ID* for the flow and stores it in its local reference table (Step 3). The controller responds back to the switch with an OpenFlow action message consisting of the actions to be performed on the packets of the flow as demonstrated in Step 4 of the Figure. We augment the 'action' part of the rule with a boolean 'COMPACT' flag. If the flag is set the switch utilizes the *Flow-ID* and performs operations on the packets corresponding to the *Flow-ID* and output it to specific port as given in the action part. Compact operation consists of insertion of the *Flow-ID* into the header of the packet after the delimiter field of the L2 layer. In OpenFlow enabled switches, this operation can be performed by the 'PUSH' tag (this is supported in OpenFlow specification).

The controller sends a flow insert operation to all the switches along the path of the packet (Step 5 of Fig. 1(a)). This flow entry consists of a *Flow-ID* and associated actions to it. The intermediate switches just perform the normal lookup operations on *Flow-ID*.

For the egress switches, the controller sends a flow insert operation that contains the *Flow-ID* and actions to remove the *Flow-ID* from the header of the packets. Step 6 in Figure illustrates the flow insert operation specified by the controller to the egress switch. The header that was in the original packet without the *Flow-ID* is forwarded to the output port to deliver to the end hosts (Step 7).

Fig. 1(b) illustrates the design with Compact TCAM as part of switching device (refer to the pipeline reference architecture in [6]). The hardware consists of a buffer to hold the packet for assigning *Flow-ID*. *Flow-ID* is copied at the selected bits of the buffer. Packet header parser which will now extract the *Flow-ID* bits is instructed through the bit selection logic. An edge switch acts both as ingress for one set of flow and egress for another set of flows. When a packet arrives, the header either contains a *Flow-ID* (as an egress) or a full header (as an ingress). This will require two flow tables, one with *Flow-ID*s and another with unmodified flow entries. We propose to use two separate TCAMs (Egress TCAM and Ingress TCAM). One TCAM stores the *Flow-ID* and associated action and the other stores the complete flow entry with complete header along with COMPACT flag. Due to this separate TCAM's power gains on the edge switches are lesser than 50% than those obtained from the core switches.

3 Assessment and Evaluation

Our objective is to measure the following: (1) Power gains and (2) Overall cost reduction. We consider three configurations for our experiments: (a) L2 based switching device consisting of 60 bit (48 bit for source MAC + 12 bit for VLAN tag) flow entry, (b) OpenFlow standard based switching device consisting of 356 bits flow entries (15 tuple flow entries) and (c) Compact TCAM based switching device consisting of 16 bits flow entries. We have considered three different data center (DC) topologies: Fat tree (cloud), 2-tier multi rooted tree (Enterprise), multi-tiered multi rooted tree (University). Flow characteristics are obtained from existing studies [3]. We develop a discrete event simulator in Java to simulate the network behavior following these traffic characteristics. Flows are generated with the header, tuple, duration, length, size and *Flow-ID*. We collect the statistics of bin granularity of one sec.

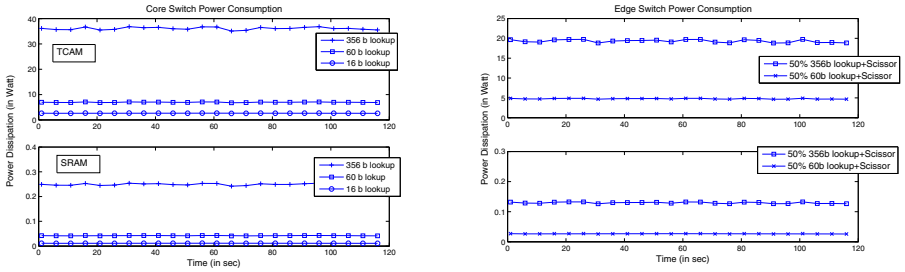
3.1 Power Gain

We study two effects: (a) power saving in core and edge switches and (b) power saving per flow. In the former, we generate flows and study the power consumption per unit time. In the latter case power is measured by generating certain number of flows and associating a path for it. Power on the switches is the function of the read and the write operations on the memory subsystems such as TCAMs/SRAMs. As the lookup in memory subsystem contributes the most in the total power for switch fabric, the following equation 1 is used to calculate the power consumption:

$$P_{TOTAL} \simeq P_{LOOKUP} = E_{SRAM} N_{SRAM\_Write} + E_{SRAM} N_{SRAM\_Read} + E_{TCAM} N_{TCAM\_Write} + E_{TCAM} N_{TCAM\_Read} \quad (1)$$

$E_{TCAM/SRAM}$ denotes the energy to access TCAM/SRAM memory and $N_{*\_Read/Write}$ denotes the number of read/write per second. We use TCAM modeling tool available in [1] to obtain the power consumption for different sizes.

Fig. 2 (a) and (b) presents the power consumption in Watts on the core switches and at edge switches for Enterprise and Univ DC's respectively. At the core switches, a L2 switching fabric would consume about 2.5 times more power and a OpenFlow enabled switch will consume about 12 times more compared to a *Compact TCAM* based switch when the number of flows are observed in the range of 1200-1300 flows/sec. Therefore, in core switches Compact TCAM based design gives best power saving. This gain is lower in the edge switch as the power saving is obtained only for the egress flows. Thus about 80% overall power saving can be observed using *Compact TCAM*.



(a) Power consumption in core switches) (b) Power consumption in edge switches

Fig. 2. Power consumption – core and edge switches, for Enterprise and University DC

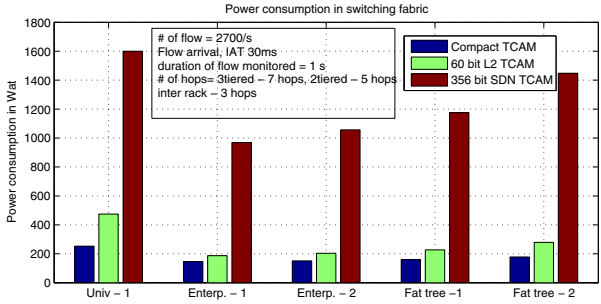


Fig. 3. Power consumption in switching fabric with different configuration.

Fig. 3 shows the average power consumption for the flows considering different topology. On an average about $2700 flows$ are generated every bin. In the university topology, savings can be achieved up to 30-40% compared to a 60 bit L2 switching device and about 87% compared to a 356 bit line SDN switching. This is due to the

fact that in university data Centers the amount of traffic that leaves the rack and travels through the network is about 80% ie., increasing the number of hops visited by the flows. The saving are lesser in a enterprise DC setting as majority of the traffic (about 80%) is intra rack.

3.2 Cost Saving

An important fall out of this optimization is the reduction in the size of the flow entry. This implies that by applying Compact TCAM one can accommodate more number of flow entries in the table compared to the one that does not apply the COMPACT operations. For instance, a 2Mbit TCAM can accommodate around 33,000 flow entries of size of 60 bit while it can accommodate 125,000 entries of 16 bits. Thus either smaller sized TCAM's can be used for the same number of flows to be accommodated as in 60 bit TCAM or the number of entries can be increased with the same TCAM size.

4 Related Work

We classify the existing literature along three broad categories and compare our work. They are: (a) algorithmic optimizations (b) reducing the amount of information stored in the high performance memory(c)architectural solutions. We differentiate our work from these existing work along two dimensions. Firstly, these algorithms assume prefixes. In SDN networks, there is lesser incentive (although not ruled out completely) for storing prefixes. SDN is designed to provide the flexibility of fine grained traffic management by defining the concept of 'micro-flows' by increasing the number of tuples. Secondly, these algorithms suffer from unacceptably slow updating (insert/delete) of the forwarding table. In SDN as insert operations are likely to be dominant (1 in every three packets can result in a insert operation [4]) these algorithms are less scalable. Finally, Our work is along the lines of reducing the information stored in the flow table and can be used to complement other compression techniques (which has limitations due to fields required for routing). Further detail of our work is given in the technical report [5].

5 Conclusions

In an attempt to have combined gain on power and cost we have identified TCAMS as an optimization target. We propose *Compact TCAM* that deals with the redundant information stored in the TCAM required for the flow processing. *Compact TCAM* exploits the emerging SDN framework and utilizes OpenFlow a standard for SDN to eliminate the redundant information. We show that the switch fabric power gain can be about 2.5 times of a standard L2 switch and 80% gain in power compared to SDN switches.

References

1. Agrawal, B., Sherwood, T.: TCAM Delay and Power Model, http://www.cs.ucsb.edu/~arch/mem-model/
2. Agrawal, B., Sherwood, T.: Modeling TCAM power for next generation network devices. In: of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX (March 2006)
3. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: Proceedings of the 10th Annual Conference on Internet Measurement, pp. 267–280 (2010)
4. Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S.: Devoflow: scaling flow management for high-performance networks. SIGCOMM Comput. Commun. Rev. 41(4), 254–265 (2011)
5. Kannan, K., Banerjee, S.: Compact TCAM: Flow entry compaction in tcam for power aware SDN. IIIT-Delhi Technical Report (2012), www.iiitd.edu.in/~subhasisb/researchpapers/compact-tcam.pdf
6. Naous, J., Erickson, D., Covington, G.A., Appenzeller, G., McKeown, N.: Implementing an openflow switch on the netfpga platform. In: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2008, pp. 1–9. ACM, New York (2008)

A Media Access and Feedback Protocol for Reliable Multicast over Wireless Channel

Ashutosh Bhatia and R.C. Hansdah

Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore
{ashutosh.b,hansdah}@csa.iisc.ernet.in

Abstract. A link level reliable multicast requires a channel access protocol to resolve the collision of feedback messages sent by multicast data receivers. In this paper, we propose a virtual token based channel access and feedback protocol (VTCAF), which can trade off between reliability and access delay. The protocol uses the virtual (implicit) token passing mechanism based on carrier sensing to avoid the collision of feedback messages. We have simulated our protocol using Castalia network simulator to evaluate the performance parameters. Simulation results show that our protocol is able to considerably reduce average access delay while ensuring very high reliability at the same time.

Keywords: Wireless Multicast, Reliable Multicast, Media Access Control (MAC).

1 Introduction

Multicast has been used effectively by several applications including multimedia conferencing, multi-party games, distributed computing and many more, over the IP network. New set of wireless network applications have emerged in the last couple of years with the rapid growth of Wireless Sensor Network (WSN) and Mobile Ad-Hoc NETwork (MANET). Many of these applications require reliable and efficient multicast, which includes periodical beacons, alarm signals, route discovery in on-demand routing protocols, clock synchronization and multicast video streams. However, the degree of required reliability may be different for different multicast applications. The execution of these applications relies heavily on reliable and efficient MAC layer multicast. Moreover, for wireless networks, ensuring reliability of multicast data at MAC layer is necessary to realize end-to-end reliability across multiple hops because Bit Error Rate (BER) in wireless networks is high in comparison to that in wired networks. In this paper, we focus on reliable multicast transmission at MAC layer in wireless networks.

Many reliable multicast protocols have been proposed in literature at the MAC layer. In [1], Broadcast Support Multiple Access (BSMA) protocol uses NACK based approach to reduce the delay required for reliable multicast. The protocol does not provide any solution for NACK collisions. In broadcast protocol, Broadcast Medium Window [2], the source sends multicast frame to individual

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 445–449, 2013.

group members on a peer-to-peer basis. The protocol provides high reliability with a very large overhead. In [3], after sending the multicast data to all receiver nodes using a single transmission, the source sends request to ACK (RAK) message to individual nodes one by one to confirm the status of transmission. This approach imposes less overhead as compared to that in [2]. The protocol in [4] uses time division multiple access (TDMA) to transmit feedback messages and thereby eliminates RAK messages, and reduces access delay. The broadcast protocols proposed in [2,3] and [4] could avoid the collision problem and ensure high reliability because of their deterministic nature. But, they still suffer from large access delay and feedback explosion, and are not suitable when the group size is large. Kuri et al. [5] have proposed a novel mechanism, called a leader based protocol (LBP), for reliable multicast with a very less access delay. In LBP, one of the multicast receivers works as a leader which replies with an ACK frame. The LBP only provides probabilistic reliability because reception of data at leader does not always guarantee that the data has been received successfully at all the receives.

In the design of the above protocols, there is a general perception that access delay needs to be reduced as much as possible while ensuring reliability at the same time. It is also to be noted that 100% reliability is not required by all multicast applications. In other words, different multicast applications may have different reliability requirements. For example, in military applications, reliability requirement is very high, but access delay of the order of a second may be tolerated. However, in case of clock synchronization protocol or multimedia applications, reliability requirement is high, but not necessarily 100%. On the other hand, delay requirement is of the order of millisecond. Hence, there is a case for designing multicast protocols which can trade-off reliability for reduced access delay, and vice versa. In this paper, we propose such a protocol for reliable multicast using a novel approach. The approach is based on virtual token rotating among the receivers, and we refer to this protocol as the virtual token-based channel access and feedback (VTCAF) protocol.

2 Informal Description of the Protocol

Essentially, we have used a combination of ACK-based and NACK-based approach together with TDMA and virtual token based channel access mechanism to design our protocol. The transmission of ACK messages by the receivers using ACK approach, triggers the frame loss detection at those receivers which did not receive the data. Increasing the number of receivers, which transmit ACK message would lead to more number of ACK transmission. Hence, it would result in larger access delay, but would ensure guaranteed frame loss detection at individual nodes, and therefore, would provide higher reliability. The number of receivers that use ACK approach can be conveniently varied to tune the reliability of our protocol. The number of receivers transmitting NACK can also affect the reliability and access delay of our protocol. If only a single node transmits the NACK, and the NACK is lost, the reliability would be compromised. On the other hand, transmission of NACK message by all the receivers which received ACK, but not data,

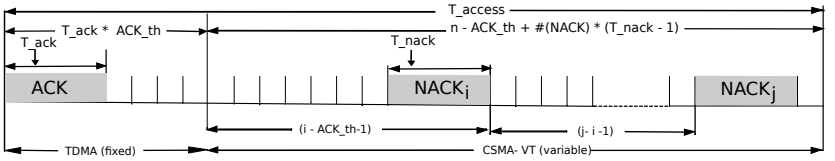


Fig. 1. MAC frame structure

may lead to NACK explosion. In our protocol, to improve reliability, if a receiver using NACK approach receives a threshold number of NACK messages, then it does not transmit NACK even though it has not received data. The threshold for NACK transmission can be selected according to the reliability requirement of the application, and the channel condition. The proposed channel access mechanism not only avoids collision between ACK/NACK messages, but also reduces the access delay. Access delay can be reduced if we can ensure that only those nodes which have transmitted the NACK message should contribute to the major part of access delay. The receivers with ACK approach use TDMA channel access mechanism whereas the receivers using NACK approach use proposed virtual token based channel access mechanism, as given below.

When it is the turn of a receiver X to transmit the feedback message, it can either transmit NACK message or does not transmit anything. If the receiver X does not transmit anything, other receivers can detect this fact using Clear Channel Assessment (CCA) operation in much less time than the time required to transmit ACK/NACK message. In such a case, the next node Y in the order after node X will get its turn immediately after it finishes CCA operation. If the node X transmits NACK message, the next node Y in the order after X has to wait to get its turn until X finishes transmitting NACK. The above transfer of turn from node X to node Y can be viewed as transfer of virtual token from node X to node Y, and we shall refer to the time for which a receiver node holds the virtual token as the token-holding period.

3 Channel Access and Feedback Protocol

In the proposed protocol, the receivers of multicast data are ordered from 1 to n. Timeline is considered slotted with slot size equal to the time required to perform CCA operation. All the receivers with order less than or equal to ACK_{th} use ACK approach, and the others use NACK approach. A receiver node with ACK approach transmits ACK if it has received the data. On the other hand, a receiver node using NACK approach only transmits NACK if it has not received the data, received at least one ACK, and received less than $NACK_{th}$ number of NACKs. The Figure 1 shows the Media Access Control (MAC) frame structure. The MAC frame is divided into two parts namely Time Division Multiple Access (TDMA) and Carrier Sense Multiple Access with Virtual Token (CSMA-VT). The length of TDMA portion is fixed and it is used by the nodes with ACK approach. A node i can transmit its ACK message at slot id $(i-1) * T_{ack} + 1$.

The length of CSMA-VT section is variable and is used by the receiver nodes using NACK approach. The duration of the token holding period is one slot if the node remains idle, otherwise it consists of a fixed number of slots T_{nack}, required to transmit a NACK message. The first receiver using NACK approach gets the token at slot id $(ACK_{th} * T_{ack}) + 1$. All other receiver nodes using NACK approach and the sender node perform CCA operation in the first slot of the token-holding period to figure out when the token holding period of the first receiver gets over. Note that, the CCA operation may fail in the presence of hidden nodes, and therefore, proposed channel access mechanism does not completely avoid the collision between NACK messages.

The steps of the protocol to be executed by a receiver node i using NACK approach while it is waiting for the token, are as follows.

1. Initialize the token-holding period index with 1 at the slot $(ACK_{th} * T_{ack}) + 1$, and the number of NACKs received to 0.
2. Perform the following actions until token-holding period index becomes i.
   - Perform CCA at the first slot of every token-holding period.
   - If the channel is free, the duration of current token-holding period is considered as 1 slot, otherwise equal to number of slots required for NACK transmission.
   - At the end of current token-holding period, increment token-holding period index by 1 and also update the number of NACKs received.

The steps of the protocol to be executed by the source node are as follows.

1. Note down the ACK received from each of the receiver node with device order i such that $i \leq ACK_{th}$.
2. Initialize the token-holding period index with 1 at the slot $(ACK_{th} * T_{ack}) + 1$, and the number of NACKs received to 0.
3. Perform the three actions executed by a receiver in step 2 of its protocol until the token-holding period index becomes $n - ACK_{th}$.
4. At the end of $n - ACK_{th}$ token-holding period, retransmit the data if received at least one NACK or not received ACK from any of the node with device order i, such that $i \leq ACK_{th}$.

4 Simulation Results

We have used Castalia Simulator [6] to study the performance of VTCAF protocol. All receivers are distributed randomly within 100mX100m area. The size of ACK/NACK and duration of clear channel assessment are 100 bits and 128μs respectively. The simulation results for the proposed protocol have been compared with those of TDMA, BMMM [3] and LBP [5]. In figure 2, T_{avg_access} with respect to p for different ACK_{th} values has been plotted. T_{avg_access} increases with the increase of p for all the cases. As we can see, the LBP incurs minimum delay since only leader node transmits ACK. The VTCAF protocol always performs better than the BMW and BMMM. Figure 3 shows the effect of p_{mean} on F_p for various ACK_{th} and $NACK_{th}$ values and the comparison of VTCAF protocol with LBP [5]. The VTCAF protocol outperforms LBP in every case.

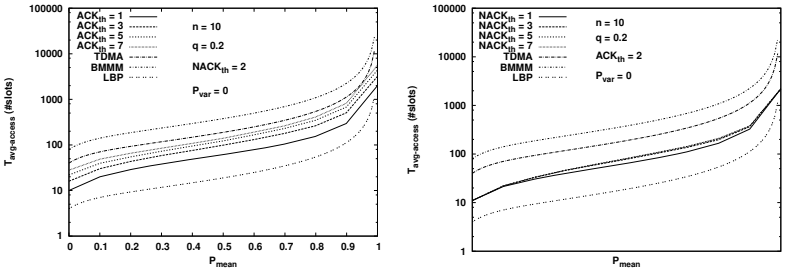


Fig. 2. T_{avg_access} vs p_{mean}

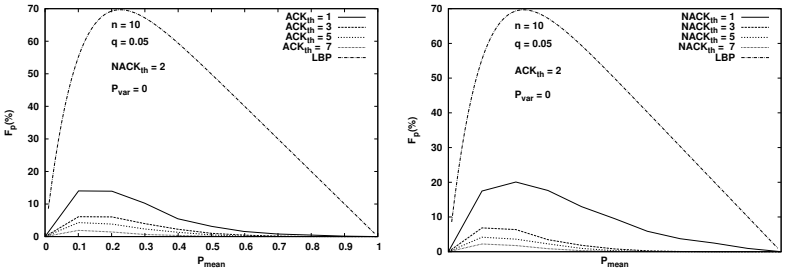


Fig. 3. F_p vs p

5 Conclusions and Future Work

The VTCAF protocol has unique feature that given the reliability and access delay requirement, parameters can be tuned to achieve the desired performance for a given channel condition. The simulations results show that very high reliability can be achieved by appropriately setting ACK_{th} and $NACK_{th}$ values while ensuring reasonably low access delay at the same time. Future works include avoiding the collision of NACK messages due to hidden node problem and extending the protocol for multicast reliability in multihop wireless networks.

References

1. Tang, K., Gerla, M.: Random access mac for efficient broadcast support in ad hoc networks. In: Wireless Communications and Networking Conference, WCNC 2000, vol. 1, pp. 454–459. IEEE (2000)
2. Tang, K., Gerla, M.: Mac reliable broadcast in ad hoc networks. In: Military Communications Conference, MILCOM 2001, vol. 2, pp. 1008–1013 (2001)
3. Sun, M., Huang, L., Arora, A., Hwang Lai, T.: Reliable mac layer multicast in ieee 802.11 wireless networks. In: ICPP 2002, pp. 527–536. IEEE Computer Society (2002)
4. Peng, J.: A new arq scheme for reliable broadcasting in wireless lans. IEEE Communications Letters 12(2) (February 2008)
5. Kuri, J.: Reliable multicast in multi-access wireless lans. Wireless Networks, 359–369 (1999)
6. http://castalia.npc.nicta.com.au/pdfs/CastaliaUserManual.pdf

POSTER: Distributed Lagrangean Clustering Protocol

Ravi Tandon, Biswanath Dey, and Sukumar Nandi

Indian Institute of Technology Guwahati
r.tandon@alumni.iitg.ernet.in,
{bdey,sukumar}@iitg.ernet.in

Abstract. Heterogeneity in sensor nodes may be caused because of differences in transmission capabilities, different terrains or different distribution of event occurrences. A distributed, energy efficient clustering protocol, *Distributed Lagrangean Clustering Protocol (DLCP)* based on Lagrangean Surrogate optimization is proposed for heterogenous networks. DLCP uses residual energy and position of the sensor nodes for the election of cluster heads. Cluster head election is modeled as a facility location problem. Simulation study reveals that DLCP forms better clusters than HEED (Hybrid Energy Efficient Distributed Clustering) and LEACH. DLCP prolongs network lifetime by distributing energy usage in a more uniform manner.

1 Introduction

Election of cluster heads has been modeled as a service placement problem. Facility providers (cluster heads) provide data forwarding and data aggregation facilities to demand points (member nodes). The problem of electing cluster heads can be modeled as a *p-median* facility location problem. The p-median problem is the problem of locating p medians so as to minimize the transmission cost from each of the demand points to its nearest facility. Lagrangean Surrogate heuristic is an optimization technique that is used to find an approximate solution [2] to the p-median problem.

2 Related Work

Heterogeneous clustering protocols [1,3,4] elect cluster heads based on residual energy of sensor nodes. HEED [4] elects cluster heads based on a secondary parameter such as node proximity or node degree. LEACH [1] elects cluster heads in a stochastic and periodic manner. Heterogeneous version of LEACH requires global knowledge of sensor nodes' energy which incurs excessive overheads. SEP [3] assigns probability of becoming a cluster head to each sensor node based up on its initial energy. Stochastic election of cluster heads [1,3,4] may lead to low energy sensor nodes becoming cluster heads. Thus, network life is reduced.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 450–451, 2013.

3 Contribution

Protocol Description: DLCP is a two phase distributed clustering protocol. The first phase (referred to as Centralized phase) lasts only a single round and is initiated by the Base Station. In the centralized phase Base Station acts as the central authority which elects cluster heads using Lagrangean clustering. The second phase (referred to as Distributed phase) lasts up to network death. Clustering is initiated at the advent of each round. Clustering phase is divided into two stages. Stage I consists of cluster formation by Lagrangean Scheme and stage II consists of cluster formation by a fully distributed weight based clustering scheme. In the second phase, Lagrangean clustering is applied on each cluster by the respective cluster head. Lagrangean scheme reduces the problem of clustering to that of optimizing a cost factor. The cost factor takes into consideration position and residual energy sensor nodes.

Results: A simulation study was performed, which compared the performance of DLCP with LEACH and HEED. DLCP improves *network stability* (period before the death of first sensor node) by 61% over a variation of heterogeneity. DLCP aggregates 85% more *reliable data* (data sent during the stability period) than HEED and LEACH. DLCP improves the standard deviation of residual energy by 32% over other protocols.

4 Conclusion

We have proposed an energy-efficient distributed clustering approach for ad-hoc sensor networks. Our approach is a hybrid approach. The Base Station elects cluster heads according to a Lagrangean Surrogate heuristic. The Lagrangean clustering scheme uses residual energy of sensor nodes as a cost metric. Sensor nodes which do not lie within the transmission range of any cluster head broadcast a weighted metric. Each sensor node that has the highest weight amongst its neighborhood becomes a cluster head. This protocol is referred to as Distributed Lagrangean Clustering Protocol (DLCP).

References

1. Cambridge, M.A., Chandrakasan, A.P., Heinzelman, W.B., Balakrishnan, H.: An application-specific protocol architecture for wireless microsensor networks 1, 660–670 (2002)
2. Senne, E.L.F., Lorena, L.A.N.: A Lagrangean/Surrogate Approach To p-Median Problems. In: 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003), pp. 2842–2852 (2003)
3. Smaragdakis, G., Matta, I., Bestavros, A.: SEP: A Stable Election Protocol for clustered heterogeneous wireless sensor networks. In: Second International Workshop on Sensor and Actor Network Protocols and Applications, SANPA 2004 (August 2004)
4. Younis, O.: HEED: A Hybrid, Energy-Efficient, Distributed Clustering Approach for Ad Hoc Sensor Networks 3, 366–379 (2004)

POSTER: Broadcasting in Delay Tolerant Networks Using Periodic Contacts

Prosenjit Dhole, Arobinda Gupta, and Arindam Sharma

Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur - 721302, India

Delay Tolerant Networks (DTN) are characterized by intermittent connectivity between nodes causing parts of an end-to-end path to be formed at different times, though a complete end-to-end path may not exist at any time. In such networks, messages are buffered at nodes, and forwarded as and when a contact with the next hop is made. Typically, broadcasting in DTNs uses some variation of flooding. Such schemes generate a large number of extra messages that may need to be buffered and may not work well in resource-constrained scenarios such as low buffer size at nodes or nodes with energy constraints.

In many mobility patterns seen in practice, the contacts between nodes occur in some time sequence and are repeated over certain periods. As an example, a person will usually go to the office at certain time everyday, coming in contact with similar officegoers in the bus, train, carpool etc. After reaching the office he/she will come in contact with certain people at around the same time every day. In this paper, we propose and evaluate a broadcast algorithm called *P-PREF* (*Probabilistic Path Restricted Flood*) that effectively utilizes the knowledge about this sequencing and the periodicity of the contacts to improve broadcast performance in resource-constrained DTNs.

Periodic contacts can be represented using a *Probabilistic Contact Graph* [3], where nodes correspond to nodes in the network, and edges between two nodes contain the time and probability of contact between the nodes. In order to send a message from one node to another along a path in the probabilistic contact graph, it is required to find a sequence of contacts that will occur one after another in increasing time sequence in the future. Path probability of such a time sequenced path is calculated by multiplying the probability of all edges in the path. Two nodes in such a probabilistic contact graph can have multiple time sequenced paths with different path probabilities. When a message originates in a certain node at a certain time, *P-PREF* calculates the maximum probability paths from the source node to all other nodes in the network occurring within the next one period, and joins these paths efficiently to form a structure called a *broadcast subgraph*. To do this, *P-PREF* assumes that every node has the probabilistic contact graph apriori. Messages are forwarded only along the edges of this subgraph as and when these contacts happen. However, since the contacts are probabilistic and the contact for an edge in the broadcast subgraph may not occur sometimes, *P-PREF* also performs a probabilistic flooding along other edges of the broadcast subgraph to explore a small number of redundant

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 452–453, 2013.

alternate paths to ensure that the message reaches all nodes and to reduce the average message delivery time.

We have simulated *P-PREF* and compared average delivery ratio (fraction of total messages delivered to all nodes), average delivery time, and total number of messages generated of *P-PREF* with that of *Flood* [1] and *Hypergossiping* [2] with 1000 randomly generated messages. The mobility models used are Random Waypoint Model (RWP), Map Based Mobility Model (MBM) [4], and an Educational Campus Model (ECM) [3]. The buffer policies used are FIFO (default), LIFO and random. The buffer size at each node is varied from 25 to 250 (no. of messsages).

In general, it is seen that *P-PREF* performs the best in terms of delivery ratio in the ECM model, with delivery ratio much higher than both *Flood* and *Hypergossiping* for all buffer sizes. This is due to the highly periodic nature of the contacts in the ECM model, which models the movements of faculty and students in a campus, that *P-PREF* is able to exploit effectively. On the other extreme, RWP shows complete random connection pattern and hence *P-PREF* does not perform well in this case, with delivery ratio lower than *Flood* and comparable with *Hypergossiping*. The MBM model, which models movements in a city, exhibits a behavior somewhat in between the two extremes, with delivery ratio of *P-PREF* comparable to that of *Flood* and much higher than that of *Hypergossiping* for all buffer sizes. The total number of message generated by *P-PREF* is always significantly less than both *Flood* and *Hypergossiping* for any mobility model and for any buffer size. However, the average delivery time of *P-PREF* is usually higher than that in the other two protocols, which can be reduced by increasing the flood probability. As far as different buffer policies are concerned, *Flood* and *P-PREF* show their best results with FIFO and random buffer policy, and perform the worst with LIFO policy for any mobility model. For ECM and MBM model, *P-PREF* performs better than *Flood* and *Hypergossiping* for all buffer policies, but the scenario is exactly the opposite for the RWP model.

The results indicate that for mobility models that show some degree of periodicity, *P-PREF* will perform very well, achieving a high delivery ratio at a fraction of total number of messages generated. However, the average delivery time may be higher. Since DTN applications are not real time, *P-PREF* can be an efficient broadcasting strategy in DTNs.

References

1. Vahdat, A., Becker, D.: Epidemic routing for partially connected ad hoc networks. Technical Report 200006, Duke University (2000)
2. Khelil, A., Marrón, P.J., Becker, C., Rothermel, K.: Hypergossiping: A generalized broadcast strategy for mobile ad hoc networks. Ad Hoc Networks 5(5), 531–546 (2007)
3. Gupta, A., Jathar, R.: Probabilistic routing protocol using contact sequencing in delay tolerant networks. In: Second International Conference on Communication Systems and Networks, COMSNETS 2010, pp. 1–10 (2010)
4. Keränen, A., Ott, J., Kärkkäinen, T.: The ONE Simulator for DTN Protocol Evaluation. In: Second International Conference on Simulation Tools and Techniques, SIMUTools 2009, New York, NY, USA. ICST (2009)

POSTER: Cryptanalysis and Security Enhancement of Anil K Sarje's Authentication Scheme Using Smart Cards

Chandra Sekhar Vorugunti[1] and Mrudula Sarvabhatla[2]

[1] Dhirubhai Ambani Institute of Information and Communication Technology, Gandhi Nagar, India
Vorugunti_Chandra_Sekhar@daiict.ac.in
[2] Sri Venkateswara University, Tirupati, India
mrudula.s911@gmail.com

Abstract. In 2010 Anil k Sarje et al. proposed an improved remote user authentication scheme based on Wang et al. authentication scheme using smart cards. In this paper, we will show that in Anil k Sarje scheme, ID and Password can be computed by adversary. Hence it is vulnerable to all traditional attacks. We propose an efficient and secure authentication scheme with smart cards which requires minimum computational cost.

1 Authentication Scheme Proposed by Anil K Sarje et al.

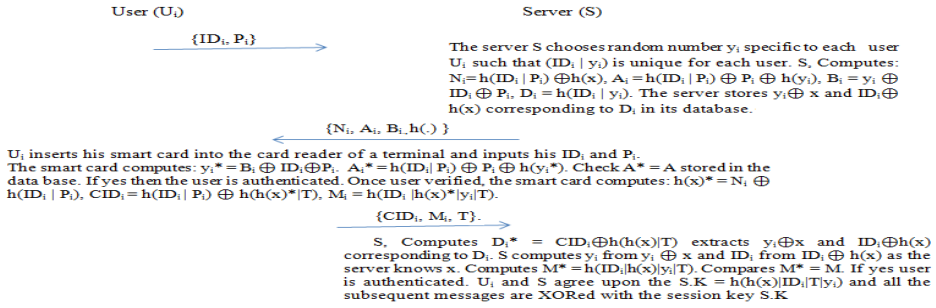


Fig. 1. Graphical view of our Anil K Sarjee et al. authentication scheme

1.1 Revealing User ID and Password U_i to Adversary

E can frame $N_i \oplus A_i \oplus B_i = h(x) \oplus ID_i \oplus h(y_i) \oplus y_i$. (4), $CID_i = h(ID_i | y_i) \oplus h(h(x)|t)$ (7). From (7), 'E' can frame $h(ID_i \mid y_i) = CID_i \oplus h(h(x)|t)$. (8), $M_i = h(ID_i | h(x) | y_i | t)$. (9) ($h(ID_i|y_i)$ is unique for each user), From (4) 'E' can frame: $N_i \oplus A_i \oplus B_i \oplus h(x) = ID_i \oplus h(y_i) \oplus y_i$. (5). From (5) 'E'can frame: $N_i \oplus A_i \oplus B_i \oplus h(x) \oplus h(y_i) \oplus y_i = ID_i$. (6). from (8), the adversary gets an equation which relates ID_i and y_i. Substituting (6) in (8) 'E' gets: $h(N_i \oplus A_i \oplus B_i \oplus h(x) \oplus h(y_i) \oplus y_i | y_i) = CID_i \oplus h(h(x)|T)$ (10). Adversary

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 454–455, 2013.

can perform dictionary guessing attack for 'y_i' on (10) guesses a secret value y_i* and checks $h(N_i \oplus A_i \oplus B_i \oplus h(x) \oplus h(y_i^*) \oplus y_i^* | y_i^*) = CID_i \oplus h(h(x)|T)$. As the value $CID_i \oplus h(h(x)|T)$ is unique because it is equal to $h(ID_i | y_i)$. If they are equal then the secret value chosen by server S for the user U_i is y_i*. Once the adversary calculates y_i of user U_i, the adversary can calculate ID_i and P_i.

2 Proposed Remote User Authentication Scheme

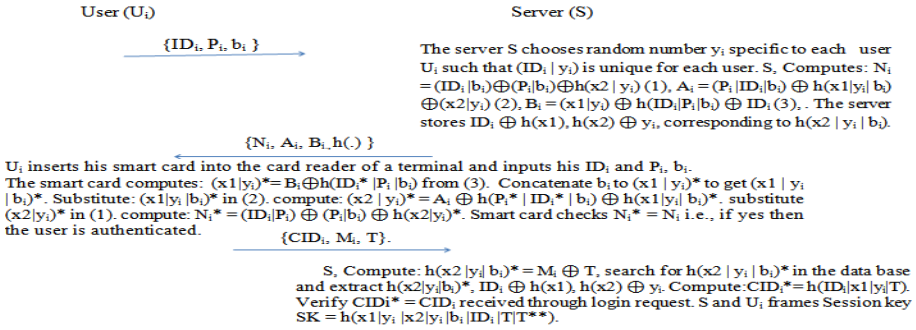


Fig. 1. Graphical view of our proposed authentication scheme using smart cards

2.1 Security Analysis of the Proposed Scheme

As legal user knows ID_i, P_i, b_i. can do following operations. Compute $(x1|y_i) = B_i \oplus h(ID_i| p_i |b_i)$. Concatenate b_i to $(x1| y_i)$ to get $(x1|y_i |b_i)$. Substitute $(x1| y_i |b_i)$ in (2). Compute $(x2|y_i) = A_i \oplus h(P_i| ID_i | b_i) \oplus h(x1|y_i|b_i)$. The values $(x1|y_i)$ and $(x2|y_i)$ are unique to each legal user. It is not possible for the legal user to guess x1, y_i, x2 in a real polynomial time. In our scheme the legal adversary E must guess ID_i, b_i, P_i, x1, x2, y_i of U_i and its not possible for E to frame a single equation with single unknown variable. Therefore in our scheme no secret value of either server or legal user is revealed to others. Hence it's impossible for E to perform vulnerability attacks.

References

1. Wang, Y., Liu, J., Xiao, F., Dan, J.: A more efficient and secure Dynamic ID-based Remote User Authentication scheme. Computer Communications 32(4), 583–585 (2009)
2. Sood, S.K., Sarje, A.K., Singh, K.: An Improvement of Wang et al.'s Authentication scheme Using Smart Cards. In: National Conference on Communications (NCC 2010), IIT-Madras, pp. 1–5 (2010)

POSTER: A New Approach to Impairment-Aware Static RWA in Optical WDM Networks

Sebastian Zawada, Shrestharth Ghosh, Fangyun Luo, Sriharsha Varanasi, Arunita Jaekel, and Subir Bandyopadhyay

School of Computer Science, University of Windsor, Canada N9B 3P4

In order to set up a number of lightpaths to satisfy user requirements for data communication, the *static* (also called *offline*) *Route and Wavelength Assignment* (RWA) problem must be solved. The quality of transmission (QoT) of an optical signal propagating through an optical network degrades, due to physical layer considerations such as *optical noise*, *chromatic and polarization mode dispersion*, *four wave mixing*, *cross-phase modulation* and *cross-talk* [4]. This leads to an increase in the *Bit Error Rate* (BER) of the optical signal and the corresponding lightpath becomes infeasible for communication if the BER value crosses a certain threshold limit. We have used an analytical model proposed by Pereira et al [3] to estimate the BER. The interdependence between the physical and the network layers makes the RWA problem in the presence of impairments a *cross-layer optimization* problem [1]. To address this problem, a number of approaches are emerging, usually referred to as *impairment-aware*-RWA (or IA-RWA) algorithms that take into account the interaction between the network and the physical layers. Our objective is to design a *transparent network*, where the IA-RWA algorithm must provision lightpaths so that the BER value of each lightpath never exceeds a given threshold.

The objective of this heuristic is to carry out static RWA for a set $\mathcal{R}$ of requests for data communication, taking into consideration both class 1 and class 2 physical layer impairments [1]. Here each request is denoted by a pair of nodes (s, d), meaning that a source node s wishes to communicate with a destination node d. If RWA is successful for a pair of nodes $(s, d) \in \mathcal{R}$, it means that a transparent lightpath may be deployed from node s to d, using some path Ψ_d^s from node s to node d, and a channel c that is *not used* by any lightpath using any edge on the path Ψ_d^s. The heuristic takes an iterative approach to RWA where, in a given iteration, lightpaths are assigned to as many source, destination pairs in set $\mathcal{R}$ as possible. In a given iteration, the heuristic considers each request in $\mathcal{R}$ that has not yet been assigned a lightpath successfully and i) determines, if possible, an appropriate path Ψ_d^s on the physical layer, for the request being considered, ii) determines, if possible, the best channel c for the request, iii) determines whether the impairments on all the lightpaths, including this new one, meet the QoT requirements. These impairments we consider are due to a) class 1 impairments on this new lightpath, b) class 2 impairments on existing lightpaths, if this new proposed lightpath is set up, iii) class 2 impairments on this new proposed lightpath due to existing lightpaths. If all these steps are successful, the request is deemed to be "handled" in this iteration. We include the pair (s, d), the path Ψ_d^s and the channel number c in the list of established

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 456–457, 2013.

lightpaths and remove the pair (s, d) from the set of requests $\mathcal{R}$. Otherwise, the request is retained in $\mathcal{R}$ for subsequent iterations. The process terminates either when all requests have been handled or when no other paths remain for the requests currently under consideration. The heuristic consists of 3 phases as follows. Using some relatively large value for k, phase 1 generates a set P_d^s of k or fewer shortest paths from s to d, for each request (s, d) in set $\mathcal{R}$. For each remaining request (s, d) in set $\mathcal{R}$, phase 2 selects, if possible, a path $\Psi_d^s \in P_d^s$ from s to d. For each route Ψ_d^s found in phase 2, phase 3 assigns a valid channel to set up a transparent lightpath from s to d, whenever possible.

In order to evaluate the proposed heuristic, we have compared it to the following approaches i) Classical RWA, ii) Shortest Path First (SPF) [2], iii) Longest Path First (LPF) [2]. Classical RWA assumes an ideal physical layer, with no impairments and the corresponding results provide an upper bound on the number of successful connections. Fig. 1 shows the percentage of demands that could be successfully handled for different network sizes, when presented with a demand set of 50 demands. We see that our proposed heuristic consistently outperforms both SPF and LPF. The amount of improvement varies with the network size and ranges from about 10.33% (13.42%) for a 10-node network to 4.41% (5.56%) for a 30-node network compared to SPF (LPF) and is also quite close to cRWA (which provides the upper bound).

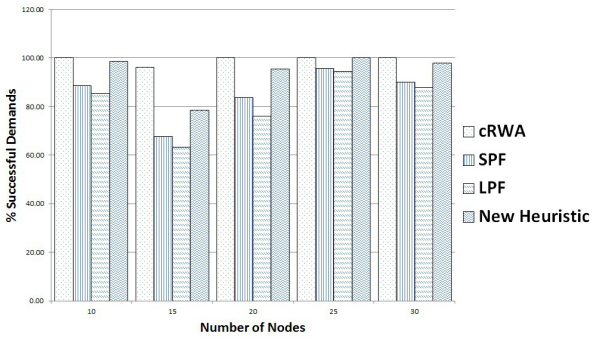


Fig. 1. Comparison of successfully routed demands for different network sizes

References

1. Christodoulopoulos, K., Manousakis, K., Varvarigos, E.: Offline routing and wavelength assignment in transparent wdm networks. IEEE/ACM Transactions on Networking 18(5), 1557–1570 (2010)
2. Ezzahdi, M.A., Al Zahr, S., Koubaa, M., Puech, N., Gagnaire, M.: Lerp: a quality of transmission dependent heuristic for routing and wavelength assignment in hybrid wdm networks. In: Proceedings of the15th International Conference on Computer Communications and Networks, ICCCN 2006, pp. 125–136 (October 2006)
3. Pereira, H.A., Chaves, D.A.R., Bastos-Filho, C.J.A., Martins-Filho, J.F.: Osnr model to consider physical layer impairments in transparent optical networks. Journal of Photonic Network Communications 18, 137–149 (2009)
4. Shen, G., Tucker, R.S.: Translucent optical networks: the way forward. IEEE Communications Magazine 45, 48–54 (2007)

POSTER: Using Directional Antennas for Epidemic Routing in DTNs in Practice

Rajib Ranjan Maiti, Niloy Ganguly, and Arobinda Gupta

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur-721602, India
{rajibrm,niloy,agupta}@cse.iitkgp.ernet.in

Introduction: In this paper, we investigate the effect of using directional antennas (DA) for routing in Delay Tolerant Networks (DTNs) where agents move following some realistic mobility patterns. In particular, the performance of classical SIRS epidemic dynamics [1] for routing in DTNs using a combination of omnidirectional antennas (OAs) and DAs is investigated using the *SLAW (Self-similar Least Action Walk)* model [2] for human daily life mobility patterns. We analyze the performance by placing DAs on randomly chosen agents and orient the DAs in randomly chosen directions. The broad goal of this paper is to initiate a study of using directional antennas for routing in DTN in practice.

Directional Antenna: Unlike OA which transmits its signal to all the directions equally, a DA [3] transmits its signal maximally in the directions of focus with a beamwidth γ. Compared to an OA, the range of a DA with the same power as an OA is much larger in the directions of γ; however, the range is much smaller in other directions. Also, the direction of focus can be changed by rotating the antenna.

Mobility Model: We choose SLAW mobility model [2] which can produce a variety of mobility patterns (primarily varying *Hurst* parameter H) similar to the mobility for different real sites (such as several university campuses and several social gatherings). In this paper, two different sites, *State Fair*, (SF) and *NCSU campus*, (NCSU), are chosen to reflect two dissimilar means of gathering.

Performance Metrics: We use two measures to evaluate the performance of the epidemic protocol- *message delivery delay, t,* (the time when a destination receives a particular message generated by a particular source for the first time) and *hop count, h* (the number of intermediate agents required to reach the destination for the first time from a source).

Simulation Setup: A fraction of the agents ρ_{DA} are chosen randomly to carry DA with the same γ. The antenna is rotated at each time step with a probability p_{rot}. A message is generated from a randomly chosen agent for a destination located at a distance D_{init} from the source at time of message creation. The results are compared with that of a similar setting where all the agents carry OA, we measure the delay improvement factors I_t and hop improvement factor

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 458–460, 2013.

I_h defined as follows. Let t_p and t_q are the delays by using some DA and by using only OA respectively, then $I_t = (t_p - t_q)/t_p$; I_h is defined similarly.

Representative Results: As a representative result, Figure 1 shows how the delay and the hop count vary with D_{init}. The results show that the use of DA can certainly reduce both the delay and the hop count when D_{init} is beyond a threshold, below which both are actually increased. The use of DAs helps to spread information to larger distances faster because of the larger range in the direction of γ. However, many nearby neighbors which would have got the message in one hop in case of an OA-only system are now not covered by the DA; they get the message via other agents increasing both the delay and the hop count.

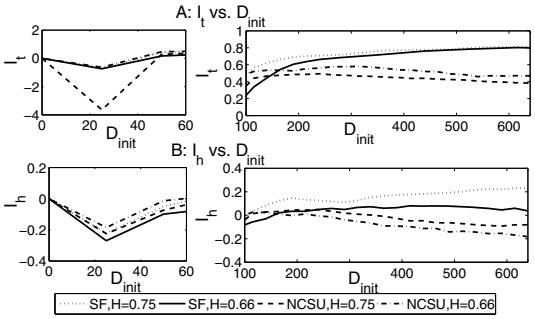


Fig. 1. Variations of I_t and I_h with initial distance D_{init} for the sites SF and NCSU with alternate Hurst values H={0.66,0.75} when ρ_{DA}=0.2 with p_{rot}=0.3

Effect of Other Parameters: We have studied the effect of other parameters such as p_{rot}, ρ_{DA}, γ, number of agents and H. With a fixed ρ_{DA}, it is seen that I_t is higher with higher p_{rot} irrespective of agent density. However, the rate of increase in I_t diminishes beyond a p_{rot} (>0.5). A similar result is seen with ρ_{DA} in case of I_t. Though higher agent density helps reducing the delay, the reduction is much lower at a low agent density. As a DA with smaller γ can throw a signal to a larger distance, I_t is higher with smaller γ. Analyzing the impact of H, it turns out that I_t is positive for all practical values of H. This indicates that the use of DA can be helpful to reduce the routing delay in presence of human daily life mobility patterns.

However, analyzing performance in terms of the hop count, it is seen that I_h is much more sensitive to the parameters used. For example. at a low agent density, either smaller p_{rot} or smaller ρ_{DA} actually increases the hop count. This indicates that the use of DA might incur more cost than an OA-only. A detailed investigation of the performance of the protocol using DA in practice is necessary to properly understand it impact on a variety of metrics before employing them in DTNs in practice; this can help in devising a learning algorithm to efficiently place and orient the DAs among the agents.

References

1. Diekmann, O., Heesterbeek, J.A.P.: Mathematical Epidemiology of Infectious Diseases: Model Building, Analysis and Interpretation. Wiley (2000)
2. Lee, K., Hong, S., Kim, S.J., Rhee, I., Chong, S.: Slaw: A mobility model for human walks. In: Proceedings of INFOCOM, Rio de Janeiro, Brazil (April 2009)
3. Peruani, F., Maiti, A., Sadhu, S., Chat, H., Roy, R., Choudhury, Ganguly, N.: Modeling broadcasting using omnidirectional and directional antenna in delay tolerant networks as an epidemic dynamic. IEEE JSAC 28(4), 524–531 (2010)

POSTER: A Secure and Efficient Cross Authentication Protocol in VANET Hierarchical Model

Chandra Sekhar Vorugunti[1] and Mrudula Sarvabhatla[2]

[1] Dhirubhai Ambani Institute of Information and Communication Technology, Gandhi Nagar, India
Vorugunti_Chandra_Sekhar@daiict.ac.in
[2] Sri Venkateswara University, Tirupati, India
mrudula.s911@gmail.com

Abstract. In 2011, Abhijith Das et al. [1] proposed a protocol based on hierarchical model for node authentication in group communication in VANETs and claimed that their protocol is robust against conventional security attacks. In this paper we will show that Abhijith Das et al. [1] scheme cannot withstand to various conventional security attacks and fails to provide authentication. We then present our improved scheme.

1 Authentication Scheme Proposed by Abhijith Das et al.

Abhijith Das et al. [1] scheme is based on VANET hierarchical Model and Polynomial Interpolation Scheme. (for PIS reading Shamir et al. and Mounita et al.)

Step 1. U_i-> U_0 : E_{pu0} (x_i || T) where T is the time stamp. Every vehicle (U_i) will send its own contribution to the powerful node, encrypting with the public key of U_0. Where U is set of invited CA's , U_0 Head CA.
Step 2. U_0 decrypts all the contributions using its private key i.e., D_{pr0} (x_i || T).
Step 3. U_0 assumes all the IDs of the invited CAs.
Step 4. Generates the session key 'K' with the contributions received as discussed with n+1 points.
Step 5. U_0 -> U_i: E*pui* ($H(ID_i||K||ID_0)$) where ID_0 is the ID of the nominated CA.
Step 6. When a vehicle enters in the supervision area of any CA, it detects the vehicle and gets the ID of the vehicle. Then CA broad casts a message which is encrypted with the ID of the vehicle. CA- >Vehicle: E_{IDi} ($ID_i||pu0||N$) where the serial number N is only for the supervising CA to identify the vehicle but other vehicles in the group just ignore the serial number.
Step 7. Every U_i -> U_0: $E_{pu0}(ID_i||x_i||T)$,where T is the time stamp.
Step 8. Session key is built using polynomial generated using PIS scheme.
Step 9. Session key K = a0||a1||a2||a3||...||an. U_0 calculates P_i for each U_i. $P_i = K \oplus (ID_i||x_i)$. All the P_i are concatenated and a secret is generated by U_0. The secret that is broadcasted is P1||P2||P3||….||Pn. U_i extracts his his P_i automatically. $P_i \oplus (ID_i||x_i) = K$.
Step 10. A->B: E_K (M||T||NA) Where, M: Message to be sent to B. T: Timestamp NA: The sequence number which is automatically assigned to the vehicle A.

1.1 Analysis of Weakness of Abhijith Das et al. Scheme

'E' can decrypt the message $D_K(M||T||N_A)$ and can do following actions.

W1. Alter the identity NA i.e $E_K(M||T||N_R)$. and sends the altered message to B.
W2. Alter the message and broadcast an altered safety message $E_K(M'||T||N_R)$.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 461–462, 2013.

W3. Create arbitrary new false identities and frame safety messages like $E_K(M1||T||N_{R1})$, $E_K(M2||T||N_{R2})$ etc.
W4. Add delay into the message $E_K(M||T+\Delta t||N_R)$. These actions by adversary leads to timing attack, node impersonation attack, sybil attack, failure of assuring message integrity and entity authentication.

2 Our Proposed Authentication Protocol

In our scheme the session key is shared in very secure manner by the message $E_{PubKeyB}\{M3|SK||T3||GK\}$. The decryption must be done with private key of B only. In our scheme A sends the message $Sig_{PriKeyA}\{MAC(E_{SK}(M4),SK)\}$ to B. Signing the message with the private key of A ensures that the message is from A only. Without providing the valid certificate issued by supervision CA, no intended receiver will respond to the invitation message. In the first place it's not possible for A to create fake certificates. Hence in our scheme sybil attack is not possible. The certificate $CertV[VPubKey] = VPubKey||\ Sig_{PriKeyCA}[VPubKey||ID_{CA}]$ are digitally signed by the CA with his private key. In our scheme $E_{SK}[M4||T], Sig_{PriKeyA}\{MAC(E_{SK}(M4||T),SK)$, the time stamp is concatenated to safety message. It's not possible for any insider other than the intended recipient to decrypt the message using the session key SK. Hence our scheme resists all the major attacks. Once a new vehicle enters, the Abhijith Das et al. [1] scheme executes complete key generation algorithm which requires $O(n^2)$ operations where n is number of vehicles.

Step 1. CA->U_i: $E_{IDi}(ID_i,PubKeyCA,PubKeyi,PriKeyi)$.CertUi . where CertUi = {PubKeyi||$Sig_{PriKeyCA}$ [PubKeyCA||IDCA]}
Step 2. CA calculates the Group Key GK(picks up any random number).
Step3.CA->U: Message=H(PubKeyA),$E_{PubKeyA}$(GK),H(PubKeyB),$E_{PubKeyB}$(GK) etc.
$Sig_{PriKeyCA}${Message} (Where A and B are vehicles in the group)
When a member leaves the group a new GK is generated and broadcasted the supervising CA forwards the message H(PubKeyN),$E_{PubKeyN}$(GK),$Sig_{PriKeyCA}\{E_{PubKeyN}\{GK\},h(PubKeyN)\}$ to the newly joined vehicle in the group.

Two vehicles A and B which belongs to same group can communicate by establishing the session key as follows.
Step 1. A->B: M1, $Sig_{PriKeyA}$[M1|T1|GK], CertA
Step 2. B->A: M2, $Sig_{PriKeyB}$[M2|T2|GK],CertB
Step 3. A frames the session key SK.
Step 4. A->B: $E_{PubKeyB}\{M3|SK||T3||GK\}$, $Sig_{PriKeyA}\{MAC(M3||T3||GK,SK)\}$
Step 5. A->B: $E_{SK}[M4||T4]$, $Sig_{PriKeyA}\{MAC(E_{SK}(M4||T4),SK)\}$

Reference

1. Abhijith, D., Dipanwitha, R.C., Anshul, R.: An Efficient Cross Authentication Protocol in VANET Hierarchical Model. International Journal of Mobile & Adhoc Network 1(1), 128–136 (2011)

POSTER: Approximation Algorithm for Minimizing the Size of Coverage Hole in Wireless Sensor Networks

Barun Gorain, Partha Sarathi Mandal, and Sandip Das

[1] Indian Institute of Technology, Guwahati, India
[2] Indian Statistical Institute, Kolkata, India

Abstract. Covering a bounded region with minimum number of homogeneous sensor nodes is an NP-complete problem [1]. In this article we have introduced a variation of area coverage problem in which the boundary sensor nodes of a coverage hole are allowed to move towards the hole for minimizing the size of the hole. We have shown that this problem is NP-hard. A $\rho-$approximation algorithm is proposed to solve this problem, where $2 \leq \rho \leq 3$ and $O(\Delta \log \Delta + k^2)$ is the time complexity.

1 Introduction

Random deployment of static sensor nodes on an area of interest (AoI) does not guarantee complete coverage. Introducing limited mobility over the static sensor nodes, it is possible to improve the coverage by reducing overlaps with neighboring nodes and allowing them to move towards the uncovered region. In literature some heuristics [1,2] are proposed for coverage improvement. To the best of our knowledge, no approximation algorithm has been found in literature in the area coverage problem by mobile sensor nodes. In this article we have considered homogeneous sensor nodes, which are the sensing disks with radius r, centering at the nodes. The *coverage hole* is an area bounded by sensing disks or combination of sensing disks and part of boundary of the AoI, and the area is not in the range of any sensor node. We denote $\mathcal{H}$ for coverage hole or simple 'hole'. We denote $\mathcal{B}(\mathcal{H})$ as the set of sensor nodes which are on the boundary of $\mathcal{H}$. Let $\mathcal{L}(\mathcal{H})$ be the hole after removal of $\mathcal{B}(\mathcal{H})$. We define *extreme points* of $\mathcal{H}$ are the points of intersection of the boundary sensing disks of $\mathcal{L}(\mathcal{H})$, which are on the boundary of $\mathcal{L}(\mathcal{H})$.

Problem Definition: Given a set of sensor nodes with a random deployment over an AoI, find the movements of the boundary nodes of a hole, if exists, such that the area of the hole is minimized without creating any further hole.

Theorem 1. *The above problem is NP-hard.*

A hole is called a *simple hole* if it has the following properties: (*i.*) No sensor node on the boundary of the hole is a part of any other hole, (*ii.*) Every node on the boundary of the hole must have at least one non-boundary neighbor and at most two boundary neighbors.

D. Frey et al. (Eds.): ICDCN 2013, LNCS 7730, pp. 463–464, 2013.

Let $\mathcal{H}$ be a *simple hole* surrounded by k boundary nodes S_{h_1}, S_{h_2}, $\cdots$, S_{h_k}, where $S_{h_{i-1}}$ and $S_{h_{i+1}}$ are the boundary neighbors of S_{h_i}. The area of $\mathcal{H}$ will be minimized if S_{h_i} moves towards $\mathcal{H}$ and places on the perpendicular bisector of $S_{h_{i-1}}$ and $S_{h_{i+1}}$. The final position of S_{h_i} is the point on the bisector for which at least one extreme point located on the perimeter of the sensing disk of S_{h_i} and other extreme points are inside the disk. The line joining between the initial and the final positions of the node is called the *optimal movement vector* (OMV). We have proposed following approximation algorithm for reducing area of a simple hole.

The Algorithm: Suppose k nodes are located on the boundary of a hole $\mathcal{H}$. There are following two cases:

1. k **is even or boundary of $\mathcal{H}$ is union of nodes and part of AoI boundary:** In this case we partition the boundary nodes by taking alternative nodes into two disjoint sets X_1 and X_2. The OMV for a node depends only on the position of its two boundary neighbors. Therefore, in the above partition, OMV of a node does not depend on the movement of the other nodes in the same set. Hence, all nodes of a particular set can move together along their individual OMV. The set of nodes which reduce maximum area of the hole after movement will move first. After that each node in the other set updates OMV and then moves.
2. k **is odd and $\mathcal{H}$ is a hole surrounded by only sensor nodes:** In this case it is not possible to partition into two disjoint sets X_1 and X_2 like Case 1 as there are odd number of nodes. One node gives trouble to move independently along their OMV for all possible choice of two sets. If one particular node keeps aside then the other nodes can be partitioned into two disjoint sets like Case 1. One partition is possible for a choice of one particular node, total k partitions are possible. We choose the partition corresponding to the set for which the reduced area of the hole is maximum. Then execute the node movements of this partition as Case 1. Then move the node which corresponds this partition to cover the hole after updating its OMV.

Theorem 2. *The proposed algorithm is 2 (or $2(1 + \frac{1}{k-1})$)-approximation for covering a hole if the hole is surrounded by even (or odd) number of boundary nodes or the boundary is combination of nodes and part of boundary of the AoI.*

The time complexity of the proposed algorithm is $O(\Delta \log \Delta + k^2)$, where Δ is maximum degree of nodes and k is the number of nodes on the boundary.

References

1. Li, J., Wang, R., Huang, H., Sun, L.: Voronoi-based coverage optimization for directional sensor networks. Wireless Sensor Network 1(5), 417–424 (2009)
2. Wang, G., Cao, G., LaPorta, T.F.: Movement-assisted sensor deployment. IEEE Trans. Mob. Comput. 5(6), 640–652 (2006)

Author Index